Universität Stuttgart

Eine Infrastruktur für die dezentrale Ausführung von BPEL-Prozessen

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Daniel Wutke

aus Ostfildern-Ruit

Hauptberichter:	Prof. Dr. Frank Leymann
Mitberichter:	A.o. Univ. Prof. Dr. eva Kühn
Tag der mündlichen Prüfung:	04. Mai 2010

Institut für Architektur von Anwendungssystemen

der Universität Stuttgart

2010

INHALTSVERZEICHNIS

Zusammenfassung

Die *Web Service Business Process Execution Language (WS-BPEL)* erlaubt die Entwicklung von Anwendungen als maschinell ausführbare Orchestrierungen einzelner, lose gekoppelter Geschäftsfunktionen in Form von Web-Services. Basierend auf den Konzepten der Zwei-Schichten-Programmierung hat die prozessgestützte Anwendungsentwicklung mit BPEL unter anderem die Erhöhung der Wiederverwendbarkeit, sowohl der orchestrierten Dienste als auch der Prozesslogik selbst, zum Ziel.

Gegenwärtig erfolgt die Ausführung von BPEL-Prozessen logisch zentral, d.h. ein Workflow-Management-System interpretiert das Prozessmodell und interagiert daraufhin mit den verschiedenen vom Prozess verwendeten Diensten. In komplexen Prozessen, deren Dienste über eine große Zahl unterschiedlicher Partner verteilt sind, ist diese logisch zentrale Ausführung allerdings nicht immer sinnvoll; vielmehr ist in diesen Szenarien oftmals eine verteilte Ausführung der Orchestrierungslogik des Prozesses wünschenswert. Soll, unter Verwendung existierender Technologien, ein BPEL-Prozess verteilt über eine Reihe unterschiedlicher Ausführungsteilnehmer dezentral ausgeführt werden, so bedingt dies gegenwärtig im Allgemeinen eine Anpassung des Prozessmodells und / oder der verwendeten Dienste, was dem Ziel maximaler Wiederverwendbarkeit entgegensteht.

Um das Problem des Verlusts der Wiederverwendbarkeit bei verteilter Ausfüh-

rung der Orchestrierungslogik von BPEL-Prozessen zu lösen, ist das Ziel dieser Dissertation die Entwicklung eines Ansatzes zur dezentralen Ausführung von BPEL-Prozessen durch nicht-invasive Anpassung der Prozesse an ihre jeweilige Ausführungsumgebung, unter Beibehaltung sowohl ihrer Orchestrierungslogik als auch der Schnittstellen der verwendeten Dienste.

Der entwickelte Ansatz umfasst eine Vorgehensweise zur (Vor-) Verarbeitung dezentral ausgeführter Prozesse, ein Verfahren zu deren automatischer Partitionierung auf die einzelnen Ausführungsteilnehmer sowie die Architektur und prototypische Implementierung eines verteilten Workflow-Management-Systems als Laufzeitumgebung für deren Ausführung.

Es folgt eine Zusammenfassung der einzelnen Kapitel.

Grundlagen und verwandte Arbeiten

Die in dieser Dissertation entwickelten Konzepte stützten sich soweit möglich – aus Gründen der Interoperabilität und der Wiederverwendbarkeit existierender Werkzeuge – auf existierende etablierte Technologien und Standards aus dem Bereich der Web-Service-Technologien. Als Grundlage für die Erläuterung der erarbeiteten Konzepte werden die relevanten Technologien und Standards einführend vorgestellt.

Da sowohl “klassische” als auch “verteilte” Workflow Management Systeme seit Jahren Gegenstand von Forschung und Entwicklung sind, werden einige existierende Produkte sowie Forschungsansätze zur Einordnung des entwickelten Ansatzes in den Kontext bestehender Arbeiten sowohl auf konzeptueller Ebene als auch anhand konkreter Implementierungsbeispiele vorgestellt. Neben diesen Laufzeitumgebungen für die Prozessausführung und deren Vorgehensweise zur Prozessnavigation werden auch unterschiedliche Beispiele von Verfahren zur Partitionierung von Prozessen auf die jeweils vorliegende Ausführungsumgebung erläutert.

Dezentrale Ausführung von BPEL-Prozessen

Grundgedanke des entwickelten Ansatzes zur verteilten Prozessausführung

ist die Dekomposition eines Prozesses in einzelne funktionale Einheiten und die explizite Beschreibung der Interaktionen, die zwischen diesen erfolgen müssen, um ein zur operationalen Semantik von BPEL entsprechendes Ausführungverhalten zu erhalten. Die Repräsentation dieser Beschreibung erfordert die Definition eines geeigneten Metamodells, den sogenannten *Executable Workflow Nets (EWFN)*. Als Grundlage für die Definition des EWFN-Metamodells werden (i) die für dieses geltenden Anforderungen anhand eines Beispiels zusammengetragen, (ii) die für die Diskussion im weiteren Verlauf der Arbeit unmittelbar notwendigen Konzepte des Metamodells einführend vorgestellt, (iii) die Vorgehensweise zur Erzeugung der EWFN-Repräsentation eines BPEL-Prozesses erläutert sowie (iv) das durch den entwickelten Ansatz erreichbare Verteilungsspektrum der Prozessausführung diskutiert. Die Detailbeschreibung der formalen Grundlagen des EWFN-Metamodells ist Gegenstand von [Mar10].

Die dezentrale Ausführung von BPEL-Prozessen resultiert in speziellen Anforderungen an den Prozesslebenszyklus und die während diesem durchlaufenen Lebenszyklusphasen. Als Teil der Dissertation wird ein erweiterter Prozesslebenszyklus definiert, der diesen Anforderungen Rechnung trägt.

Ein Verfahren für die automatische Partitionierung von BPEL-Prozessen

Nach der Dekomposition eines Prozesses in seine funktionalen Einheiten ist es erforderlich, diese auf die Ausführungsumgebung des Prozesses zu verteilen. Das der Arbeit zugrunde liegende Szenario der Ausführung von Geschäftsprozessen in einer Service-orientierten Anwendungsumgebung ist gekennzeichnet durch eine Reihe von Eigenschaften, die durch das Partitionierungsverfahren zu berücksichtigen sind. Dies sind beispielsweise: (i) eine umfangreiche Dienstlandschaft, in welcher funktional äquivalente Dienste von unterschiedlichen Dienstanbietern mit potentiell unterschiedlichen nicht-funktionalen Eigenschaften angeboten werden, (ii) die Berücksichtigung bestehender Verträge oder Geschäftsbeziehungen zwischen Unternehmen sowie (iii) die Struktur der zu partitionierenden Geschäftsprozesse selbst. Zugleich soll der Partitionierer eines Prozesses die Möglichkeit haben,

eventuell vorhandenes Wissen über die erwartete Prozessausführung in den Partitionierungsvorgang einfließen zu lassen. Dabei soll gewährleistet sein, dass diese "Anreicherung" der zu partitionierenden Prozessmodelle selektiv erfolgen kann, d. h. auch eine nur partiell vorgenommene Annotation eines Prozesses bereits zu einer Verbesserung der bestimmten Prozesspartitionierung beiträgt.

Da existierende Prozesspartitionierungsverfahren die oben genannten speziellen Eigenschaften und Anforderungen des Einsatzszenarios nicht oder nur teilweise erfüllen, ist die Entwicklung eines entsprechenden Partitionierungsverfahrens ein Teilaspekt dieser Dissertation.

Eine Infrastruktur zur verteilten Ausführung von BPEL-Prozessen

Der dezentrale Navigationsvorgang eines Prozesses auf Grundlage des EWFN-Modells unterscheidet sich substantiell von existierenden Vorgehensweisen zur Ausführung von BPEL-Prozessen. Dies erfordert die Entwicklung einer entsprechenden Laufzeitumgebung, die sowohl den funktionalen als auch den nicht-funktionalen Anforderungen der Ausführung von Produktionsprozessen in BPEL genügt.

Im Rahmen dieser Dissertation wird die Realisierung der *Process-Space*-Infrastruktur, als eine derartige Laufzeitumgebung auf Grundlage der *Tuplespace*-Konzepte, vorgestellt. Die Beschreibung des Systems umfasst die Vorstellung seiner Architektur, der Funktionsweise seiner einzelnen Komponenten auf konzeptueller Ebene sowie einiger Aspekte seiner prototypischen Umsetzung.

Tuplespace-basierte Kommunikation zwischen Web-Services

Der Mechanismus der sogenannten Web-Service-Bindings erlaubt es, dass Interaktionen zwischen Web-Service-Anbieter und -Nutzer über unterschiedliche Transportprotokolle abgewickelt werden können. Die Realisierung der transportprotokollspezifischen Aspekte erfolgt dabei als Teil der verwendeten Web-Service-Laufzeitumgebung, unabhängig von der Anwendungslogik

bei Dienstnutzer und Dienstanbieter.

Aufgrund der funktionalen und nicht-funktionalen Eigenschaften der *Process-Space*-Infrastruktur bietet diese, neben der dezentralen Prozessausführung, auch eine Plattform für die Interaktion von Web-Services. Im Rahmen dieser Dissertation wird ein entsprechendes Binding vorgestellt, welches die Ausführung von Web-Service-Interaktionen auf der *Process-Space*-Infrastruktur erlaubt.

Prototypische Umsetzung

Zur Demonstration der Umsetzbarkeit der entwickelten Konzepte werden abschließend unterschiedliche Aspekte der prototypischen Realisierungen der *Process-Space*-Infrastruktur, des definierten Partitionierungsverfahrens sowie einer erweiterten Web-Service-Laufzeitumgebung vorgestellt.

Die Arbeit schließt mit einer Zusammenfassung der erzielten Ergebnisse und einem Ausblick auf mögliche Bereiche anknüpfender Forschungsarbeiten.

Abstract

The *Web Service Business Process Execution Language (WS-BPEL)* enables the development of composite applications following the two-level-programming paradigm, where an application is built as an orchestration of individual, loosely coupled business functions represented by Web services.

State-of-the-art in execution of WS-BPEL processes is a logically centralized workflow management system which interprets the composition logic defined by the process model and interacts with the orchestrated services. In complex workflows, whose orchestrated services are distributed among a large number of different partners, logically centralized process execution is not always feasible. Instead, process execution based on decentralized coordination of process participants is desired as it more closely reflects the nature of the process models when compared to execution based on a single, central coordinator.

Decentralized process execution using technologies available today however requires changes to the processes composition logic as well as to the interfaces of the orchestrated services. This is in contrast to the original goal of the two-level-programming paradigm where the definition of process models and service interfaces are motivated by the application's business goal, reusability and loose coupling. Changes to any of these that are motivated by infrastructural reasons only are thus not desireable.

In this thesis a non-invasive approach to execution of WS-BPEL processes is

presented, which allows for decentralized enactment of unmodified WS-BPEL process models without the need for a central workflow management system, while retaining BPEL's execution semantics. The contributions of the thesis are: (i) a method for developing processes executed in a decentralized manner, (ii) algorithms for automatically partitioning processes among the partners participating in their execution and (iii) the architecture and prototypical implementation of a corresponding runtime environment.

Danksagung

Eine ganze Reihe von Personen haben mich während meiner Arbeit an dieser Dissertation in den letzten Jahren begleitet und damit maßgeblich zu ihrer Entstehung beigetragen. Ihnen möchte ich an dieser Stelle gerne danken.

Zunächst gilt mein Dank meinem Doktorvater Prof. Dr. Frank Leymann für seine Betreuung und Unterstützung in den letzten Jahren sowie Prof. Dr. eva Kühn für den thematischen Impuls, der die Arbeit ins Rollen brachte, und für ihre Anregungen und konstruktiven Verbesserungsvorschläge in deren Verlauf.

Weiterhin danke ich meinen Kollegen und Freunden am IAAS, im Speziellen Jörg Nitzsche, Tammo van Lessen, Dimka Karastoyanova, Oliver Kopp, Tobias Unger, Ralph Mietzner, Thorsten Scheibler und Branimir Wetzstein für die vielen Diskussionen, gemeinsamen Reisen und Feiern sowie ihre Unterstützung bei der Korrektur der Dissertation und der Vorbereitung der Verteidigung.

Mein ganz besonderer Dank gilt meinem Projekt- und Promotionsmitstreiter Daniel Martin, ohne dessen geradezu ansteckend optimistische Einstellung und konstruktiven Widerworte während unzähliger Diskussionen diese Dissertation heute nicht in dieser Form vorliegen würde und mir die Arbeit in den letzten Jahren sicherlich deutlich weniger Freude bereitet hätte.

Nicht zuletzt danke ich meiner Familie und meinen Freunden für ihre Unterstützung und ihr Verständnis, besonders in der Schlussphase der Dissertation.

KAPITEL 1

EINFÜHRUNG

Thema der vorliegenden Arbeit ist die Architektur und die Entwicklung eines Middleware-Systems, welches die dezentrale, verteilte Ausführung von WS-BPEL-Prozessen ermöglicht. Gegenstand dieses Kapitels ist die Diskussion der Motivation der Arbeit (Abschnitt 1.2) und die Vorstellung der in ihrem Verlauf bearbeiteten Forschungsprobleme (Abschnitt 1.3). Als Grundlage hierfür wird in der Folge zunächst eine Einführung in die der Arbeit zugrunde liegenden Themengebiete der *Workflow Technologien* sowie der *Service-orientierten Architekturen* gegeben (Abschnitt 1.1). Abschnitt 1.4 schließt das Kapitel mit einer Vorstellung des Aufbaus der Arbeit ab.

1.1 Anwendungsfeld

Die Integration heterogener Anwendungen ist eine häufig auftretende Problemstellung und bis heute – geprägt durch die Merkmale der jeweiligen Anwendungsszenarien – Gegenstand von Forschung und Entwicklung. Der Begriff *Service-orientierte Architektur (SOA)* [KBS04, Erl05] bezeichnet einen relativ jungen Lösungsansatz für diese Problemstellung, welchem die Idee der Anwendungsentwicklung durch Komposition oder Interaktion einzelner, lose

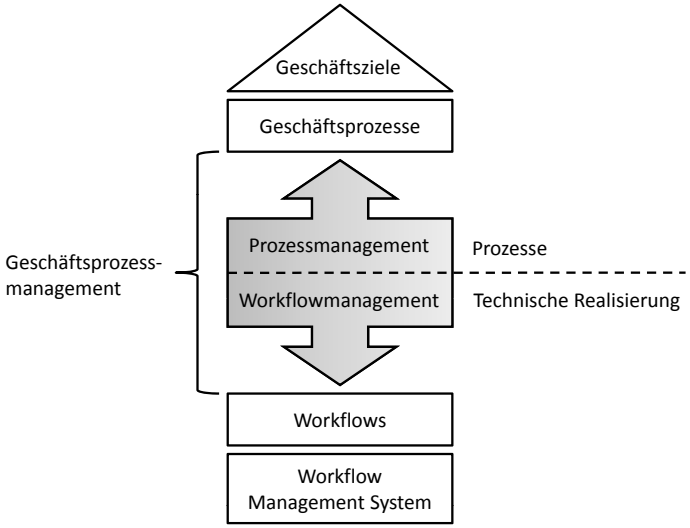


Abbildung 1.1: Geschäftsprozessmanagement als Kombination von Prozess- und Workflow-Management

gekoppelter funktionaler Elemente [Kay03], den sogenannten *Diensten* (engl. *services*), zugrunde liegt.

Ein wichtiges Konzept Service-orientierter Architekturen ist Separation: zum einen werden die Beschreibungen der Schnittstellen von Diensten von deren Implementierung getrennt; zum anderen wird die Logik einer Anwendung getrennt in *Geschäftsfunktionen* (engl. *business functions*) und *Geschäftsprozesse* (engl. *business processes*) [LR99]. Separation von Schnittstellen und Implementierungen wird durch Sprachen zur expliziten Beschreibung von Kontrakten zwischen Diensten und Dienstnutzern erreicht (vgl. Abschnitt 2.1), Separation von Geschäftsfunktionen und Geschäftsprozessen durch die Nutzung von Verfahrensweisen und Technologien aus dem Bereich des *Geschäftsprozessmanagements*.

Ziel des Geschäftsprozessmanagements (engl. *Business Process Management (BPM)*) ist die Flexibilisierung von Geschäftsprozessen und die Erhöhung von

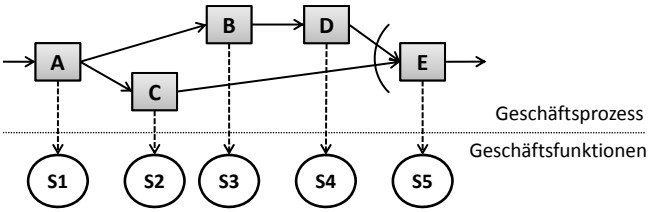


Abbildung 1.2: Separation von Prozesslogik und den verwendeten Geschäftsfunktionen durch das sogenannte *Two-Level-Programming*

deren Wiederverwendbarkeit, um eine effiziente und möglichst einfache Anpassung an sich ändernde Gegebenheiten zu erreichen. Ein wesentlicher Aspekt dieser Bestrebungen ist die Einhaltung einer Trennung der Dokumentation, Überwachung und Optimierung von Geschäftsprozessen auf der Ebene der durch den Prozess realisierten Anwendung, dem sogenannten *Prozess-Management*, und die zur technischen Umsetzung dieser Prozesse notwendigen Technologien als Teil des sogenannten *Workflow-Managements*; Abbildung 1.1 verdeutlicht den Zusammenhang zwischen Prozess- und Workflow-Management graphisch.

Der Begriff *Geschäftsprozess*, wie er in dieser Arbeit verwendet wird, beschreibt einen Vorgang zur Erreichung eines bestimmten Geschäftsziels und die Nutzung von Geschäftsfunktionen zu diesem Zweck. Infolge der unterschiedlichen Ebenen von Geschäftsfunktionen und Geschäftsprozessen spricht man bei der zur Entwicklung von entsprechenden Anwendungen angewandten Methode auch vom sogenannten *Two-Level-Programming* [LR97, Ley03] (Abbildung 1.2), demzufolge die einzelnen Geschäftsfunktionen einer Anwendung durch Geschäftsprozesse zu Anwendungen verbunden werden. Begrifflich wird, analog der Begriffe Prozess- und Workflow-Management, zwischen dem eher anwendungsbezogenen *Geschäftsprozess* und dem *Workflow* als seine technische Realisierung unterschieden; allerdings werden die Begriffe Prozess und Workflow üblicherweise synonym verwendet, wenn aus dem Zusammenhang erkennbar ist, ob der technische oder geschäftliche Aspekt gemeint ist. Im weiteren Verlauf dieser Dissertation wird dies entsprechend gehandhabt. Die

Literatur unterscheidet verschiedene Arten von Prozessen, welche sich hinsichtlich der Anzahl ihrer Ausführungen, der Realisierung ihrer Geschäftsfunktionen oder ihrer Änderungshäufigkeit unterscheiden. Fokus dieser Arbeit sind langlebige Prozesse, deren Geschäftsfunktionen ganz (oder zum überwiegenden Teil) maschinell ausgeführt werden; sogenannte *Workflow-Management-Systeme (WfMS)* [LR99] bieten eine Plattform für die maschinelle Ausführung von derartigen Geschäftsprozessen.

Eine technische Ausprägung der oben aufgeführten Konzepte Service-orientierter Architekturen und des Workflow-Managements sind die sogenannten *Web-Service-Technologien* [ACKM04, WCL+05], die durch einen geschichteten Satz modularer Spezifikationen geprägt sind. Diese Spezifikationen beschreiben unterschiedliche Kommunikationstechnologien, Nachrichtenformate, funktionale und nicht-funktionale Diensteigenschaften, transportunabhängige Umsetzung von Dienstgütegarantien, wie Verlässlichkeit des Nachrichtenaustauschs und Sicherstellung von Integrität und Vertraulichkeit der übertragenen Nachrichten, sowie – auf höchster Ebene – die Orchestrierung von Diensten zu (potentiell komplexen) Workflows. Die Beschreibung einiger grundlegender und für den weiteren Verlauf dieser Arbeit relevanter Web-Service-Technologien ist Gegenstand von Abschnitt 2.1.

Die Web-Service-Spezifikation zur Beschreibung von Dienst-Orchestrierungen ist die *Web Service Business Process Execution Language (WS-BPEL)* (kurz *BPEL*) [Org07]. Sie beschreibt eine Reihe von Funktionalitäten zur Integration von Web Services und verfügt über eine klar definierte Ausführungssemantik, welche eine automatische Ausführung in BPEL beschriebener Prozesse durch ein WfMS erlaubt.

1.2 Motivation

Ein wesentliches Ziel Service-orientierter Architekturen und des Two-Level-Programming ist die Erhöhung der Wiederverwendbarkeit, sowohl auf Ebene der Geschäftsfunktionen als auch auf Ebene der Geschäftsprozesse.

Infolgedessen werden bereits als Teil der BPEL-Spezifikation unterschiedliche Aspekte der Erhöhung der Wiederverwendbarkeit von Prozessen adressiert.

Ein Beispiel hierfür, auf Ebene der Geschäftsprozesse, ist die Verlagerung von Deployment-Informationen benötigter Geschäftsfunktionen aus der Prozessmodellierungsphase in die separate, nachfolgende Deployment-Phase. Dies eliminiert die ansonsten notwendige Anpassung des Modells eines Prozesses, wenn dieser in einer anderen Dienstumgebung eingesetzt werden soll. Ein anderes Beispiel für die Unterstützung der Wiederverwendbarkeit auf Prozessebene sind die sogenannten *abstrakten Prozesse*. Dies sind Prozesse, die Teile ihrer Prozessbeschreibung durch sogenannte opake Aktivitäten verdecken und damit ein Prozessgerüst definieren, das, bedingt durch das Fehlen der opaken Prozessteile, nicht automatisiert ausführbar ist, allerdings als Vorlage für eine ausführbare Vervollständigung des Prozesses dienen kann.

Wie verschiedene andere Arbeiten im Umfeld von BPEL, z. B. [KLN+06, NLKL07, Kha08], hat auch die vorliegende Arbeit die Erhöhung der Wiederverwendbarkeit von BPEL Prozessen in unterschiedlichen Ausführungsumgebungen zum Ziel.

Während des oben genannten Prozess-Deployment-Schritts wird ein Prozess gegenwärtig typischerweise an ein einzelnes, zwar potentiell z. B. mittels Clustering-Technologien verteiltes, aber dennoch logisch zentrales[1], WfMS übergeben und durch dieses zur Ausführung angeboten. In einer Reihe von Anwendungsszenarien ist allerdings die Zuordnung einzelner Prozessteile zu mehreren unterschiedlichen WfMS wünschenswert, welche durch unterschiedliche Ausführungsteilnehmer bereitgestellt werden. Beispiele hierfür sind unter anderem das sogenannte *Process Outsourcing* [Kha08] oder *Scientific Workflows* [SGK+10].

Mit dem in dieser Dissertation entwickelten Ansatz wird dies erreicht, indem die Ausführung der Orchestrierungslogik eines Prozesses auf mehrere Ausführungsteilnehmer in einer hier erstmals vorgeschlagenen Art und Weise verteilt wird. Der entwickelte Ansatz stützt sich dabei auf (i) ein Verfahren zur Parametrisierung von Prozessen zur Anpassung an ihre Ausführungsumgebung

[1] Der Begriff *logisch zentral* bezeichnet hierbei ein WfMS, welches über eine *Naviator*-Komponente (siehe unten) verfügt. Demhingegen bezeichnet ein *logisch dezentrales* WfMS ein WfMS, in welchem der Vorgang der Prozessausführung nicht durch die Navigator-Komponente getrieben wird, sondern durch direkte Koordination der einzelnen Ausführungsteilnehmer erfolgt.

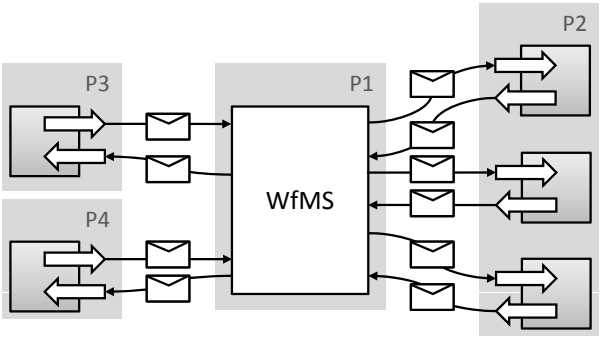


Abbildung 1.3: Sternförmige (in der Folge auch “zentrale”) Interaktion eines zentralen WfMS mit einer Menge orchestrierter Dienste bei unterschiedlichen Ausführungsteilnehmern

und (ii) eine neuartige Laufzeitinfrastruktur für die dezentrale Navigation von BPEL-Prozessen. Ein wesentlicher Aspekt der Arbeit ist dabei, dass diese Anpassung des Prozesses an seine Ausführungsumgebung, analog der oben formulierten Forderung nach Wiederverwendbarkeit, keine Änderungen an seinem Prozessmodell bedingt.

Als Grundlage für eine detaillierte Beschreibung der Motivation des entwickelten Ansatzes veranschaulicht Abbildung 1.3 die Sicht auf ein existierendes zentrales WfMS zur Ausführung von BPEL-Prozessen. Die Abbildung zeigt eine Menge von Diensten, die durch einen Prozess, welcher in einem WfMS ausgeführt wird, zu einer Anwendung integriert werden. Das WfMS selbst stellt hierbei einen aktiven (zentralen) Teilnehmer der Anwendung dar und interagiert durch den Austausch von Nachrichten sternförmig mit den orchestrierten Diensten.

Ein “traditionelles” WfMS zur Ausführung von BPEL beinhaltet als zentrale Komponente für die Prozessausführung einen sogenannten *Navigator*. Dieser evaluiert den aktuellen Zustand einer Instanz eines Prozesses und entscheidet daraufhin, welche Aktivitäten des Prozesses als nächste auszuführen sind. Sind die auszuführenden Aktivitäten bestimmt, führt das WfMS den durch die

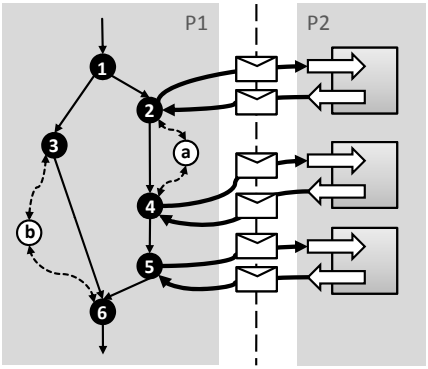


Abbildung 1.4: Zentrale Ausführung eines BPEL Prozesses

Aktivität repräsentierten Verarbeitungsschritt aus. Repräsentiert die jeweilige Aktivität die Interaktion mit einem Dienst (im Gegensatz zu einer internen, vom WfMS selbst bereitgestellten Aktivitätsimplementierung), so wird vom Navigator die Interaktion des WfMS mit dem externen Dienst ausgelöst. Im dargestellten Beispiel (Abbildung 1.3) werden die vom Prozess orchestrierten Dienste von drei unterschiedlichen Partnern P_2, P_3, P_4 bereitgestellt; das WfMS vermittelt als zusätzlicher Partner P_1 zwischen den einzelnen Diensten.

Reduziert man diese Darstellung auf ein Beispiel einer Interaktion eines WfMS mit mehreren, durch denselben Partner bereitgestellten, Diensten, fokussiert sich also beispielsweise auf die Interaktion des WfMS bei P_1 mit den Diensten bei P_2, so zeigt sich das in Abbildung 1.4 dargestellte Szenario.

Analog dem in Abbildung 1.3 beschriebenen Beispiel betreibt Prozessteilnehmer P_1 das WfMS, auf dem der dargestellte Prozess ausgeführt wird. Die in Abbildung 1.4 verwendete graphische Notation zeigt die Aktivitäten des Prozessmodells als schwarz gefärbte Kreise $1, \ldots, 6$. Kontrollflussabhängigkeiten zwischen den Aktivitäten definieren ihre Ausführungsreihenfolge und werden durch schwarze, durchgezogene Pfeile dargestellt. Variablen zur Ablage von im Rahmen der Prozessausführung anfallenden Daten werden durch die weiß gefärbte Kreise a und b repräsentiert; Variablenzugriff wird durch gestrichelte

Pfeile zwischen Aktivitäten und Variablen dargestellt, wobei in der hier verwendeten Notation nicht zwischen lesendem und schreibendem Variablenzugriff unterschieden wird.

Die Aktivitäten 2, 4 und 5 sind sogenannte *Interaktionsaktivitäten*, d. h. Aktivitäten, die während ihrer Verarbeitung eine Interaktion mit WfMS-externen Diensten, in diesem Fall angeboten durch P_2, zur Folge haben. Dienstaufrufe werden durch breite Pfeile zwischen Aktivitäten und Diensten dargestellt. Teil der Dienstaufrufe ist die Übermittlung von Prozessvariablen; im dargestellten Fall dient beispielsweise das Resultat des durch Aktivität 2 repräsentierten Dienstaufrufs, zwischengelagert in Variable a, als Eingabe für den Dienstaufruf durch Aktivität 4.

Zur Verdeutlichung der Motivation des entwickelten Ansatzes zur dezentralen Navigation (und damit zur verteilten Ausführung von BPEL-Prozessen) sollen für das dargestellte Szenario weiterhin folgende Annahmen gelten: (i) Kommunikation zwischen P_1 und P_2 ist "teuer" (beispielsweise infolge der Verwendung eines Kommunikationskanals geringer Bandbreite und/oder hoher Latenz). (ii) Die Datenmenge, die durch die Übermittlung des Werts der Variable a zwischen den durch Aktivitäten 2 und 4 repräsentierten Diensten kommuniziert wird, ist groß. (iii) Die Möglichkeit, die durch P_2 angebotenen Dienste bei P_1 zu betreiben besteht nicht, da P_2 nicht gewillt ist, die Dienste an P_1 zu übergeben.

Wird der Prozess, wie in Abbildung 1.4 dargestellt, zentral ausgeführt, dann resultiert jede der dargestellten Interaktionsaktivitäten in einer (bidirektionalen) partnerübergreifenden Interaktion zwischen P_1 und P_2. Dies wiederum hat zur Folge, dass nach Ausführung der Aktivität 2 der Rückgabewert dieser Interaktion von P_2 an das WfMS bei P_1 übermittelt werden muss; unter den getroffenen Annahmen ist dies eine "teure" Operation. Infolge der wiederholten Interaktion zwischen P_1 und P_2 während der Verarbeitung von Aktivität 4 müssen dieselben Daten erneut partnerübergreifend kommuniziert werden. Legt man die oben getroffenen Annahmen zugrunde, so führt dies hinsichtlich des partnerübergreifenden Kommunikationsaufwands, d. h. Anzahl der notwendigen Interaktionen zwischen unterschiedlichen Ausführungsteilnehmern und der Menge der entlang dieser zu kommunizierenden Daten, zu einem nicht

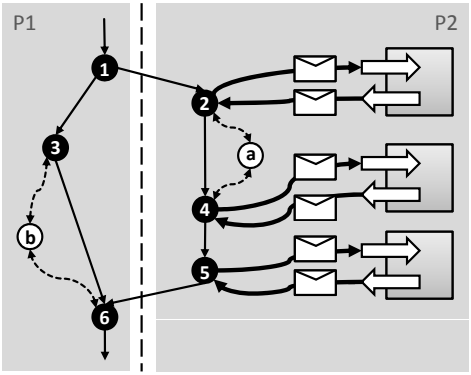


Abbildung 1.5: Dezentrale Ausführung eines BPEL Prozesses

optimalen Laufzeitverhalten.

Abbildung 1.5 zeigt denselben Prozess, welcher mit denselben WfMS-externen Diensten interagiert. In diesem Fall ist allerdings die Ausführung des Prozesses (also seine Navigation) selbst über die am Prozess teilnehmenden Partner P_1 und P_2 verteilt. Aktivitäten $2, 4, 5$ sowie die Variable a wurden hier von P_1 zu P_2 verschoben, also dem Partner, der auch die zu den Aktivitäten $2, 4, 5$ gehörenden Geschäftsfunktionen bereitstellt. Die Ausführungssemantik des Prozesses, d. h. die Abfolge der Ausführung der einzelnen Aktivitäten, entspricht auch in diesem dezentral navigierten und verteilt ausgeführten Fall genau der in Abbildung 1.4 dargestellten zentralen Prozessausführung. Dieser Ausführungsabfolge entsprechend werden nach erfolgreicher Ausführung von Aktivität 1 die Aktivitäten 2 und 3 ausgeführt. Im Gegensatz zur zentralen Prozessausführung werden die Aktivitäten 2 und 3 allerdings von WfMS unterschiedlicher Ausführungsteilnehmer ausgeführt. Die weitere Ausführung der Prozessteile erfolgt nach Beendigung von Aktivität 1 auf beiden an der Ausführung teilnehmenden WfMS parallel. Ausgelöst durch die Aktivitäten $2, 4, 5$ interagiert das WfMS bei P_2 mit den vom Prozess orchestrierten Diensten. Infolge der Verschiebung eines Teils des Prozesses von P_1 zu P_2 werden diese Interaktionen nun allerdings lokal, d. h. ausschließlich auf der Seite des Partners P_2, ausgeführt. Gleiches gilt

für die Zugriffe auf den Wert der Prozessvariable a, welche nun auf Seite von P_2 verwaltet und direkt zwischen den verwendeten Diensten kommuniziert wird. Aufgrund des Fehlens von Kontroll- und Datenflussabhängkeiten zwischen den Aktivitäten 3 und $2, 4, 5$ kann die Verarbeitung der Aktivitäten auf den einzelnen WfMS bei P_1 und P_2 ohne gegenseitige Beeinflussung vollständig parallel erfolgen. Nach erfolgreicher Ausführung von Aktivität 5 wird der Prozess von P_1 fortgeführt. Wurde weiterhin Aktivität 3 erfolgreich ausgeführt, sind die notwendigen Vorbedingungen für eine Ausführung von Aktivität 6 erfüllt und die Ausführung des Prozesses wird auf Seite von P_1 fortgesetzt.

Zusammenfassend kann festgehalten werden, dass im hier dargestellten Szenario ein verbessertes Laufzeitverhalten hinsichtlich des beispielhaft gewählten Kriteriums der Minimierung des Kommunikationsaufwands zwischen den an der Ausführung des Prozesses teilnehmenden Partnern erreicht wurde. Im Gegensatz zur sechsfachen partnerübergreifenden Kommunikation zwischen P_1 und P_2 in Abbildung 1.4 und der Notwendigkeit der partnerübergreifenden Übermittlung der Prozessvariablen mit den Dienstaufrufen, verlangt die dargestellte dezentrale Ausführung lediglich die zweimalige partnerübergreifende Weitergabe von Informationen, die die Navigation der Prozessinstanz beeinflussen; die Notwendigkeit der Übermittlung von Prozessvariablen zwischen den an der Ausführung teilnehmenden Partnern besteht nicht. Es ist anzumerken, dass die Grundlage für das in Abbildungen 1.4 und 1.5 gezeigte Prozess-Deployment derselbe Prozess ist. Eine Änderung des Prozessmodells ist für eine Anpassung an seine Ausführungsumgebung im Fall dezentraler Prozessausführung nicht notwendig.

Soll, ausgehend von der in Abbildung 1.4 dargestellten Ausgangssituation, das in Abbildung 1.5 gezeigte Szenario unter Verwendung existierender Web-Service-Technologien realisiert werden, so kann dies durch eine Änderung der Granularität der vom Prozess orchestrierten Dienste geschehen. Ziel dieser Änderung der Dienstgranularität ist eine manuelle Nachbildung der in Abbildung 1.5 dargestellten Aufteilung des Prozesses, indem die Aktivitäten $2, 4, 5$, dem sogenannten *Remote Façade Pattern* [Fow03] folgend, hinter einer Schnittstelle verborgen werden. Dies kann beispielsweise erreicht werden, indem die durch die Aktivitäten $2, 4, 5$ repräsentierte Orchestrierungslogik selbst,

wiederum in Form eines BPEL-Prozesses (z. B. als sogenannter *Sub-Prozess* [KKL+05]), modelliert und dieser auf einem regulären BPEL-fähigen WfMS bei P_2 zur Ausführung gebracht wird. Um nach dieser Aufspaltung ein zum Original äquivalentes Verhalten zu erreichen, muss der auf Seite von P_1 verbliebene Anteil des ursprünglichen Prozesses um den Aufruf des nun von P_2 bereitgestellten neuen Diensts ergänzt werden. Obwohl diese Änderung des Prozesses zu einem Resultat, ähnlich wie in Abbildung 1.5 dargestellt, führt, ist diese Umsetzung mit einer Reihe von Nachteilen verbunden, von denen nachfolgend zwei beschrieben werden.

Die Notwendigkeit der Weitergabe von Kontrollflussinformationen zwischen den an der Ausführung des Prozesses teilnehmenden WfMS erfordert das Ergänzen der einzelnen Prozessteile um zusätzliche Interaktionsaktivitäten (oder Sub-Prozess-Aufrufe, wenn die Aufteilung des Prozesses mittels Sub-Prozessen [KKL+05] erfolgt) und Koordinationslogik (z. B. in Form zusätzlicher Middleware). Bedingt durch das Fehlen zusätzlicher Kontrollfluss- und Datenabhängigkeiten zwischen den bei P_1 und P_2 parallel ausgeführten Prozesssträngen, ist eine derartige Änderung des Prozesses im dargestellten Beispiel relativ einfach möglich. Im Fall komplexerer Prozesse mit einem höheren Anteil partnerübergreifender Kommunikation ist dies jedoch deutlich aufwändiger.

Ein weiterer Nachteil ist die Definition eines neuen Diensts auf Seite von P_2. Dieser kapselt, im Gegensatz zu den ursprünglichen Diensten, nicht mehr Geschäftslogik in möglichst wiederverwendbarer Weise, sondern realisiert einen Teil des in Abbildung 1.4 dargestellten Gesamtprozesses. Infolgedessen ist die Wiederverwendbarkeit dieses Diensts auf Prozesse beschränkt, welche genau den durch diesen realisierten Prozessteil nutzen. Weiterhin verlangt die Einführung dieses zusätzlichen Diensts seine Entwicklung und Verwaltung über sämtliche Phasen seines Lebenszyklus hinweg. Wird eine Änderung des ursprünglichen Prozesses notwendig, so bedingt dies gleichermaßen eine Änderung des zusätzlich eingeführten Diensts.

Zusammenfassend gilt für die dargestellten Nachteile existierender Ansätze zur Realisierung der dezentralen Prozessausführung, dass diese eine Änderung des ursprünglichen, in Abbildung 1.4 dargestellten, zentralen Prozesses zur Folge haben. Dabei ist die Motivation dieser Änderung nicht im Geschäftsziel

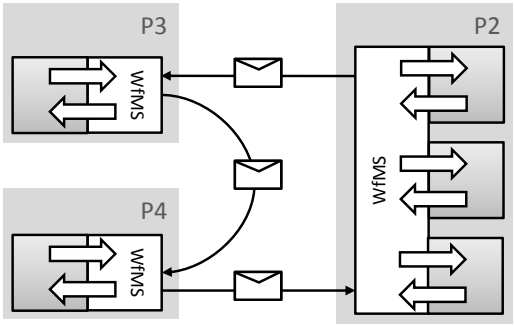


Abbildung 1.6: Koordination durch direkte Kommunikation.

des Prozesses begründet, sondern vielmehr in der Anpassung an seine Umgebung. Auf Basis dieser Erkenntnis lässt sich das Ziel der vorliegenden Arbeit formulieren.

> *Ziel der Arbeit ist die Entwicklung eines nicht-invasiven Ansatzes zur Anpassung eines BPEL-Prozesses an eine geänderte Ausführungsumgebung unter Beibehaltung seiner Orchestrierungslogik. Erreicht wird das Ziel lediglich durch Anpassung von Prozess-externen Deployment-Informationen, die die Elemente eines Prozesses auf dessen Ausführungsumgebung abbilden.*

Der in der Arbeit verfolgte Ansatz lässt sich wie folgt veranschaulichen: Überträgt man die erläuterte Vorgehensweise auf sämtliche, durch das WfMS von P_1 ausgeführten Interaktionen, so zeigt sich das in Abbildung 1.6 dargestellte Resultat. Die durch den Prozess beschriebene Orchestrierungslogik ist dabei vollständig auf die an seiner Ausführung teilnehmenden Partner verteilt. Die Notwendigkeit eines WfMS als zentralem Koordinator entfällt, vielmehr

koordinieren[1] sich die an der Prozessausführung teilnehmenden WfMS der einzelnen Partner direkt.

1.3 Forschungsbeiträge der Arbeit

Wie oben erwähnt bildet die Realisierung eines WfMS zur dezentralen Ausführung von BPEL-Prozessen das Thema der vorliegenden Arbeit. Die wissenschaftlichen Beiträge der vorliegenden Dissertation (Abbildung 1.7) sind in die folgenden Kernbereiche eingeordnet:

Beitrag 1: Ein Verfahren zur dezentralen Ausführung von BPEL-Prozessen

Im Rahmen dieses Forschungsbeitrags wird ein Verfahren vorgestellt, das es erlaubt, einen regulären BPEL-Prozess dezentral auszuführen. Es umfasst: (i) die Erläuterung des dezentralen Navigationsvorgangs, (ii) die Vorstellung des durch den entwickelten Ansatz realisierbaren Verteilungsspektrums der Prozessausführung, (iii) die Definition eines erweiterten Lebenszyklus, der den besonderen Anforderungen einer dezentralen Ausführung Rechnung trägt sowie (iv) die Beschreibung der beteiligten Rollen und die von diesen auszuführenden Arbeitsschritte und hierfür verwendete Werkzeuge.

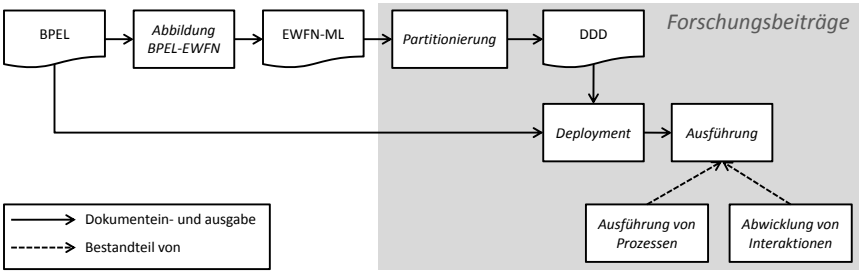


Abbildung 1.7: Forschungsbeiträge der Arbeit

[1] Der Begriff "Koordination" ist hier nicht im Sinne von *WS-Coordination* [Org09], sondern in seiner allgemeinen Bedeutung zu verstehen.

Beitrag 2: Ein Verfahren zur automatischen Partitionierung von BPEL-Prozessen

Die dezentrale Ausführung von BPEL-Prozessen bedingt die Notwendigkeit eines Verfahrens zur Bestimmung, wie ein Prozess vor seiner Ausführung auf seine jeweilige Ausführungsumgebung abgebildet wird; im weiteren Verlauf der Arbeit wird hierfür der Begriff *Partitionierung* verwendet.

Als Grundlage der Beschreibung des entwickelten Verfahrens wird eine Übersicht erstellt, welche Elemente eines Prozesses "Kandidaten" für die Partitionierung sind und welche Faktoren den Partitionierungsvorgang beeinflussen. Weiterhin werden geeignete Mechanismen entwickelt, die die identifizierten Einflussgrößen in wiederverwendbarer Weise potentiellen Nutzern zugänglich machen.

Auf dieser Grundlage wird ein Verfahren beschrieben, welches eine automatische Partitionierung eines BPEL-Prozesses gemäß der identifizierten Einflussgrößen erlaubt. Resultat der Partitionierung ist ein *Verteilter Deployment-Deskriptor* (engl. *Distributed Deployment Descriptor (DDD)*), welcher die Verteilung eines Prozesses auf seine Ausführungsumgebung beschreibt.

Beitrag 3: Entwicklung der Architektur eines verteilten Workflow Management Systems

Der verteilte Deployment-Deskriptor dient in Verbindung mit dem Prozess als Eingabe für das Prozess-Deployment, in welchem den an der Ausführung des Prozesses teilnehmenden Partnern Informationen über die von ihnen jeweils auszuführenden Prozessteile übermittelt werden. Ist der Deployment-Schritt abgeschlossen, so kann der Prozess ausgeführt werden. Die Prozessausführung umfasst sowohl die Ausführung seiner Aktivitäten als auch die damit verbundenen Interaktionen des Prozesses mit den vom diesem orchestrierten Diensten. Die Erstellung der Architektur einer entsprechenden Middleware, die die hierfür notwendigen Funktionen bereitstellt, ist ein weiterer Beitrag dieser Arbeit.

Der Beitrag umfasst im Detail (i) die Realisierung des verteilten Prozess-Deployment-Vorgangs zur Konfiguration der an der Ausführung eines Prozesses teilnehmenden Partner, (ii) die Umsetzung des dezentralen Navigationsvorgangs sowie (iii) die Protokollierung der verteilten Ausführung von Prozessinstanzen zu deren Überwachung.

Geschäftsprozesse stellen für ihre Ausführung eine Reihe nicht-funktionaler Anforderungen hinsichtlich Verlässlichkeit, Verfügbarkeit und transaktionalen Verhaltens; die Umsetzung dieser Anforderungen ist ein wesentlicher Bestandteil der Beschreibung der Architektur des WfMS, dessen Plattform eine auf den Konzepten der *Tuplespaces* basierende Middleware bildet.

Beitrag 4: Bereitstellung von Mechanismen zum Aufruf von Diensten direkt über Tuplespaces

Ein wesentlicher Aspekt der Interaktion zwischen Web-Services ist die Möglichkeit zur flexiblen Wahl der für die Übermittlung der ausgetauschten Nachrichten verwendeten Transportprotokolle (abhängig von den Anforderungen des jeweiligen Szenarios); der hierfür verwendete Mechanismus sind die sogenannten *Web-Service-Bindings*.

Verschiedene funktionale sowie nicht-funktionale Eigenschaften der in Forschungsbeitrag 3 entwickelten Infrastruktur motivieren ihre Verwendung als Kommunikationsplattform für Interaktionen zwischen Web-Services. Im Rahmen dieses Forschungsbeitrags werden diese Eigenschaften erläutert und die Konzepte eines entsprechenden Binding für die Tuplespace-gestützte Kommunikation von Web-Services werden vorgestellt.

Beitrag 5: Prototypische Umsetzung

Um die Umsetzbarkeit der vorgeschlagenen Konzepte zur dezentralen Ausführung von BPEL-Prozessen zu demonstrieren, wurde ein entsprechender WfMS-Prototyp entwickelt. Die Beschreibung des Prototyps diskutiert die wesentlichen Implementierungsaspekte sowohl der entwickelten Middleware als auch des entwickelten Partitionierungswerkzeugs und der Binding-Umsetzung.

1.4 Aufbau der Arbeit

In Kapitel 2 werden die Technologien, die für die Realisierung der in der Dissertation entwickelten Konzepte verwendet werden, zum besseren Verständnis der Arbeit im notwendigen Rahmen erläutert. Weiterhin werden relevante Arbeiten aus den berührten Forschungsgebieten vorgestellt: Dabei handelt es sich sowohl um Vorstellungen von Architekturen existierender WfMS mit zentraler Navigation, als auch Arbeiten aus dem Bereich der verteilten Ausführung von Prozessen und deren Partitionierung.

Im Anschluss an diese Einführung wird in Kapitel 3 das dem entwickelten WfMS zugrunde liegende verteilte Prozessmodell, die sogenannten *Executable Workflow Nets (EWFN)*, kurz vorgestellt; die vollständige Darstellung des EWFN-Modells ist Gegenstand von [Mar10]. Fokus des Kapitels ist das Zusammentragen all jener Anforderungen an ein Prozessmetamodell, die für die Unterstützung einer verteilten Ausführung von BPEL-Prozessen notwendig sind und die Erläuterung der Grundkonzepte des EWFN-Formalismus. Eine kurze exemplarische Erläuterung der Abbildung von BPEL-Prozessmodellen auf EWFN-Modelle und Vorstellung des Verteilungsspektrums, das durch die Navigation von EWFN-Modellen realisierbar ist, schließen diesen Teil der Arbeit ab. Im Anschluss wird die dem entwickelten Ansatz zugrunde liegende Verfahrensweise der (Vor-) Verarbeitung von BPEL-Prozessen für deren dezentrale Ausführung anhand eines erweiterten Prozesslebenszyklus vorgestellt.

An die Darstellung der Methode anknüpfend beschreibt Kapitel 4, als ein Schwerpunkt der Arbeit, ein Verfahren zur automatischen Partitionierung von BPEL-Prozessen, welches die speziellen Anforderungen (i) einer Service-orientierten Anwendungsumgebung, (ii) der Erhaltung der operationalen Semantik von BPEL auch bei dezentraler Ausführung sowie (iii) eines guten Laufzeitverhaltens während der Prozessausführung adressiert.

Kapitel 5 beschreibt die entwickelte Laufzeitinfrastruktur zur dezentralen Prozessausführung und bildet damit einen zweiten Schwerpunkt der vorliegenden Arbeit. Die Beschreibung der Infrastruktur umfasst eine Erläuterung der Gesamtarchitektur des Systems, deren Komponenten und eine detaillierte Darstellung, wie diese sowohl die funktionalen als auch die nicht-funktionalen

Eigenschaften gewährleisten, die für eine Ausführung (verteilter) Geschäftsprozesse notwendig sind. Abschluss des Kapitels bildet eine qualitative Bewertung der Eigenschaften der verteilten Prozessausführung auf Basis der entwickelten Infrastruktur.

In Kapitel 6 wird dargelegt, wie eine auf den Konzepten eines Tuplespace basierende Middleware (wie z. B. die hier entwickelte Infrastruktur) durch Definition eines entsprechenden *Web-Service-Bindings* für die Kommunikation zwischen den Partnern einer Web-Service-Interaktion verwendet werden kann und welche Vorteile dies gegenüber existierenden Kommunikationsmechanismen bietet.

Aufbauend auf der konzeptionellen Infrastrukturbeschreibung ist die prototypische Umsetzung der in den vorangehenden Kapiteln dargelegten Konzepte Gegenstand von Kapitel 7.

Kapitel 8 schließt die Arbeit mit einer Zusammenfassung der erarbeiteten Konzepte und einem Ausblick auf mögliche anknüpfende Arbeiten ab.

KAPITEL 2

GRUNDLAGEN UND VERWANDTE ARBEITEN

Thema des Kapitels ist die Einführung unterschiedlicher Technologien, Standards und verwandter Arbeiten aus den folgenden Themenbereichen, die für den Verlauf der Dissertation relevant sind:

Web Services BPEL-Prozesse erlauben die Orchestrierung von Diensten, die ihren Nutzern durch die Familie der Web-Service-Standards angeboten werden. Web-Service-Technologien finden dementsprechend an verschiedenen Stellen der Arbeit Verwendung. Im Besonderen relevant für die Erläuterungen in den folgenden Kapiteln sind die Standards SOAP, WSDL, WS-Policy, UDDI und BPEL (Abschnitt 2.1): (i) SOAP bildet beispielsweise die Grundlage des in Kapitel 6 vorgestellten *Web-Service-Binding für Tuplespaces*. (ii) WSDL- und WS-Policy-Dokumente finden Verwendung in der Beschreibung funktionaler bzw. nicht-funktionaler Eigenschaften verwendeter Dienste und bilden damit eine Eingabe des in Kapitel 4 vorgestellten Partitionierungsverfahrens für BPEL-Prozesse. (iii) Die Aggregation erfasster Dienstbeschreibungen erfolgt durch eine UDDI-

Registry. (iv) Da BPEL die Sprache zur Beschreibung der Prozesse ist, die auf der PS-Infrastruktur ausgeführt werden, dient die Vorstellung von BPEL als Grundlage für die Beschreibung der Kernkonzepte des EWFN-Metamodells, des Partitionierungsverfahrens sowie der Architektur der entwickelten *Process-Space-Infrastruktur (PS-Infrastruktur)* in Kapitel 5.

Workflow-Management-Systeme Zentrales Thema der Arbeit ist die Entwicklung einer Infrastruktur für die verteilte Ausführung von BPEL-Prozessen. Grundlage hierfür bildet eine Vorstellung existierender Referenzmodelle und Implementierungen "traditioneller" zentraler WfMS (Abschnitt 2.3) sowie eine Übersicht verwandter Arbeiten, die die verteilte Prozessausführung zum Gegenstand haben (Abschnitt 2.4). Da viele Vorgehensweisen, die in der verteilten Prozessausführung Verwendung finden, nicht an ein bestimmtes Prozessmetamodell gebunden sind, umfasst das Spektrum der betrachteten Ansätze auch WfMS für die Ausführung von Prozessen, die nicht mit den Mitteln des BPEL-Metamodells beschrieben werden.

Verfahren zur Prozesspartitionierung Aufgrund der Eigenschaften und Anforderungen einer Service-orientierten Anwendungsumgebung und der Möglichkeit zur weitgehend dezentralen Navigation durch die entwickelte PS-Infrastruktur, unterscheidet sich das im Rahmen dieser Arbeit entwickelte Verfahren zur automatischen Prozesspartitionierung (Kapitel 4) wesentlich von existierenden Ansätzen. In Abschnitt 2.5 werden eine Reihe existierender Partitionierungsverfahren mit ihren jeweiligen Eigenschaften vorgestellt. Um ein möglichst breites Spektrum unterschiedlicher Verfahren abdecken zu können, werden auch hier – analog zur Erläuterung der Grundlagen und der verwandten Arbeiten aus dem Bereich der WfMS – Verfahren betrachtet, die eine Partitionierung von Prozessen anderer Metamodelle als BPEL zum Gegenstand haben.

Tuplespaces Grundlage für die Realisierung der PS-Infrastruktur bilden die Konzepte der sogenannten *Tuplespaces*. Abschnitt 2.6 stellt diese Konzepte vor und verweist auf existierende Implementierungen und Erweiterungen des *Tuplespace*-Konzepts.

2.1 Web-Services

Web-Services sind – ähnlich zu beispielsweise der *Common Object Request Broker Architecture (CORBA)* [OMG04], *Enterprise Java Beans* [BM06] oder dem *Distributed Component Model (DCOM)* [HK97] – eine Technologie für die Realisierung von verteilten Anwendungen. Ihre Eigenschaften werden durch eine Menge unterschiedlicher Standards [Pap07, Erl05] definiert.

Web-Services bilden eine mögliche technologische Plattform für die Realisierung Service-orientierter Anwendungen [KBS04]. Nach [HBN+04] gilt: Ein Web-Service ist eine logische Einheit von Anwendungsfunktionalität, die in maschinenlesbarer Form in einer Weise beschrieben ist, so dass seine Dienstbeschreibung lediglich die Details eines Diensts preis gibt, die für ein Auffinden und eine zweckgemäße Verwendung des Diensts erforderlich sind [Kay03]. Aus Gründen angestrebter Plattformneutralität und Interoperabilität sind Kenntnisse über Implementierungsdetails eines Diensts für seine Interaktionspartner nicht erforderlich. Die Schnittstelle eines Diensts ist gekennzeichnet durch die *Nachrichten* (und deren *Typen*), die der Dienst mit seinen Interaktionspartnern austauscht. Typischerweise erfolgt der Austausch dieser Nachrichten zwischen Interaktionspartnern unter Verwendung von Netzwerkprotokollen und standardisierten Nachrichtenformaten.

Abbildung 2.1 verdeutlicht die an einer Web-Service-Interaktion beteiligten Rollen und setzt diese in Beziehung zu einigen wesentlichen hierfür verwendeter Web-Service-Technologien. An der dargestellten Interaktion sind die Rollen *Dienstnutzer* (engl. *Service Requester*), *Dienstanbieter* (engl. *Service Provider*) und *Dienstverzeichnis* (engl. *Service Registry*) beteiligt. Ein Dienstanbieter beschreibt die von ihm angebotenen Dienste in maschinenlesbarer Form unter Verwendung der Web-Service-Standards *WSDL* (Abschnitt 2.1.1) und *WS-Policy* (Abschnitt 2.1.2). Die Dienstbeschreibungen veröffentlicht ein Dienstanbieter in einem Dienstverzeichnis und macht sie damit potentiellen Dienstnutzern zugänglich; ein Beispiel einer etablierten Technologie für die Realisierung eines Dienstverzeichnisses ist UDDI (Abschnitt 2.1.3). Dienstnutzer befragen das Dienstverzeichnis nach Diensten, die ihre funktionalen und nicht-funktionalen Eigenschaften erfüllen. Hat ein Dienstnutzer einen entsprechenden Dienst

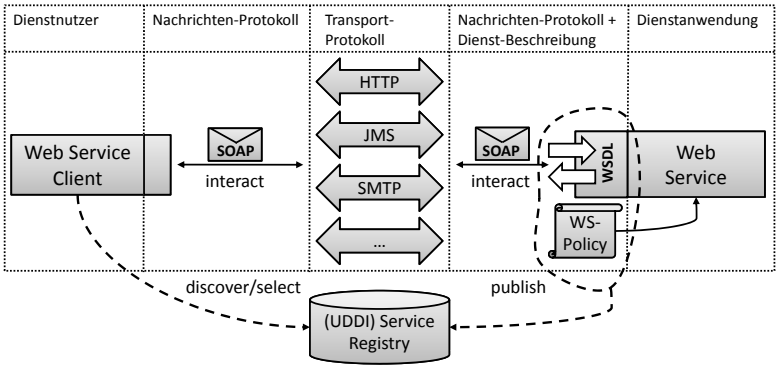


Abbildung 2.1: Web-Service-Interaktion und verwendete -Technologien und -Standards.

gefunden, so kann er – unter Verwendung der technischen Zugriffsinformationen in der WSDL-Beschreibung des Diensts – mit diesem über potentiell unterschiedliche Transportprotokolle in Interaktion treten. Da der eigentliche Nachrichtenaustausch zwischen Dienstnutzer und Dienstanbieter unter Verwendung der Nachrichtenarchitektur SOAP (Abschnitt 2.1.4) durch die, für die Interaktion verwendete, Web-Service-Laufzeitumgebung abgewickelt wird, ist eine Anpassung der Anwendungslogik des jeweiligen Diensts beispielsweise im Fall eines Wechsels des verwendeten Transportprotokolls nicht erforderlich.

2.1.1 Web Service Description Language (WSDL)

Die *Web Service Description Language (WSDL)* [CGM+04, CHL+07] ist eine XML-basierte Sprache zur Beschreibung von Web-Services in Form sogenannter Endpunkte, welche Nachrichten eines bestimmten definierten Aufbaus verarbeiten können.

Ein WSDL-Dokument ist eine formale technische Beschreibung der funktionalen Eigenschaften eines Web-Service. Als solche beschreibt ein WSDL-Dokument, (i) welche Operationen ein Dienst seinen Nutzern durch seine Schnittstelle anbietet, (ii) welche Nachrichtentypen (einschließlich möglicher-

weise auftretender Fehler) der Dienst in seinen unterschiedlichen Operationen konsumiert und produziert und (iii) in welcher Form Dienstnutzer mit dem Dienst interagieren können (z. B. die Adresse einer bestimmten Implementierung des Diensts oder die zur Kommunikation mit diesem zu verwendenden Nachrichten- und Transportprotokolle). Zur Abbildung dieser Informationen umfasst ein WSDL-Dokument drei Teile: eine *abstrakte Schnittstellenbeschreibung* des Diensts, *konkrete Dienstbeschreibungsinformationen*, die sich auf eine bestimmte Implementierung der abstrakten Schnittstelle beziehen und sogenannte *Binding-Informationen*, die die konkrete Beschreibung des Diensts mit seiner abstrakten Schnittstellenbeschreibung in Verbindung setzen und zusätzlich all jene Informationen bereitstellen, die ein Dienstnutzer für die Interaktion mit dem Dienst unter Verwendung eines bestimmten Binding benötigt.

Der konkrete Teil einer WSDL-Beschreibung, repräsentiert durch das SERVICE-Element, ist gekennzeichnet durch einen sogenannten ENDPOINT, welcher eine Implementierung eines Web-Service repräsentiert[1]. Ein ENDPOINT beinhaltet einen eindeutigen Namen, einen Verweis auf das BINDING-Element, welches den SERVICE mit seiner abstrakten Schnittstellenbeschreibung in Verbindung setzt, sowie die notwendige Adressierungsinformation für dessen Implementierung(en).

Der abstrakte Teil einer WSDL-Beschreibung eines Web-Service umfasst die folgenden Informationen: Die von dem Web-Service angebotenen Operationen (engl. OPERATION) werden – gemeinsam mit zugehörigen Fehlerinformationen (engl. FAULT) – im sogenannten INTERFACE (bzw. PORTTYPE in WSDL 1.1 [CCMW01]) der Dienstbeschreibung gruppiert, welche durch einen eindeutigen Namen identifiziert ist. Die Operationen selbst sind ebenfalls durch Namen eindeutig identifiziert und werden durch die Typen der konsumierten (INPUT) und produzierten (OUTPUT) Nachrichten und deren Abfolge in Form eines sogenannten *Message Exchange Pattern (MEP)* näher charakterisiert. WSDL erlaubt die Verwendung unterschiedlicher Typsysteme zur Typisierung der kommunizierten Nachrichten; ein häufig verwendetes Typsystem ist XML-Schema [FW04].

[1] Dieser und die in der Folge der Beschreibung von WSDL verwendeten Bezeichner beziehen sich auf die WSDL-2.0-Spezifikation [CGM+04].

Ein MEP identifiziert die Anzahl und Abfolge der Nachrichten, die Teil der mit dem Aufruf einer Operation eines Diensts verbundenen Interaktion sind. Weiterhin beschreibt es, an welche Teilnehmer der Interaktion eine Nachricht gesendet bzw. von welchem diese empfangen werden soll. WSDL 2.0 definiert eine Reihe standardisierter MEP, wie beispielsweise *In-Only* für Dienstaufrufe, für die keine Antwortnachricht erwartet wird, oder *In-Out* für bidirektionale Dienstaufrufe mit erwarteter Antwortnachricht. Darüber hinaus erlaubt WSDL 2.0 (im Gegensatz zu seinem Vorgänger WSDL 1.1 [CCMW01]) die Erweiterung dieser Menge an standardisierten MEP um zusätzliche anwendungsspezifische MEP.

Der abstrakte Teil einer WSDL-Beschreibung ist mit dessen konkreter Beschreibung durch das sogenannte BINDING-Element verbunden. Diese Indirektion ermöglicht die Wiederverwendung derselben abstrakten Dienstbeschreibung in unterschiedlichen Implementierungen, von denen potentiell jede eigene Transport- und Nachrichtenprotokolle verwenden kann.

WSDL ist neben SOAP ein Basis-Standard aus der Familie der Web-Service-Standards und findet im weiteren Verlauf der Arbeit an unterschiedlichen Stellen Verwendung. In Abschnitt 4.2.2.1 wird WSDL beispielsweise für die Beschreibung der funktionalen Aspekte der, von unterschiedlichen Dienstanbietern angebotenen, Dienste als Grundlage für die Prozesspartitionierung verwendet. In Abschnitt 6.4 wird erläutert, wie die Adresse eines Web-Service-Endpunkts, welcher über das Web-Service-Binding für Tuplespaces angeboten wird, in seiner WSDL-Beschreibung repräsentiert werden kann.

2.1.2 Web Service Policy (WS-Policy)

Während das Ziel von WSDL (Abschnitt 2.1.1) die Beschreibung der funktionalen Charakteristika eines Diensts ist, haben die Standards *WS-Policy* [OHV+] und *WS-Policy-Attachment* [YBV+07] die Beschreibung nicht-funktionaler Eigenschaften von Diensten und Anforderungen von Dienstnutzern zum Gegenstand.

Ein WS-Policy-Dokument ist eine Menge sogenannter *Policy-Alternatives*, welche selbst jeweils aus einer Menge sogenannter *Policy-Assertions* beste-

hen. Eine Policy-Assertion kann beispielsweise eine Eigenschaft eines Diensts, eine Anforderung eines Dienstnutzers oder andere Eigenschaften oder Verhaltensweisen beschreiben. Policy-Assertions beinhalten domänenspezifische Informationen; die Beschreibung dieser domänenspezifischen Informationen selbst ist nicht Gegenstand der WS-Policy-Spezifikation, sondern muss als domänenspezifische Erweiterung spezifiziert werden. WS-Policy beschreibt lediglich das Rahmenwerk und das zum Verbund von Policy-Assertions zu Policies notwendige Vokabular in Form der Operatoren ExactlyOne und All, sowie ein Verfahren für den Schnitt (engl. *Intersection*) von Policy-Dokumenten, bestehend aus: (i) einem domänenunabhängigen und (ii) einem domänenabhängigen Vergleichsschritt. Als Teil des domänenunabhängigen Vergleichs werden Policy-Dokumente im Hinblick auf das verwendete Vokabular ihrer Policy-Assertion-Elemente verglichen. Das Resultat dieses Vergleichsschritts ist ein Policy-Dokument, welches sämtliche, hinsichtlich ihres Vokabulars kompatible, Policy-Assertion-Elemente enthält. Die Überprüfung, ob die so bestimmte Policy tatsächlich den Anforderungen des Dienstnutzers entspricht, verlangt abschließend die domänenspezifische Auswertung ihrer Assertion-Elemente.

Die Herstellung der Verbindung einer Policy zu beispielsweise einem Web-Service erfordert geeignete Mechanismen zur Einbettung von Policy-Informationen in die Dienstbeschreibungen oder das Einfügen entsprechender Verweise. Auf welche Weise dies erreicht werden kann, ist Gegenstand der Spezifikation *WS-Policy-Attachment*. Diese beschreibt beispielsweise, wie ein Policy-Dokument an die Elemente einer WSDL-Beschreibung[1] eines Diensts (im Besonderen dessen Service-, Endpoint-, Operation- oder Message-Elemente) gebunden werden können. Weiterhin wird spezifiziert, wie Policies in Verbindung zu einer Web-Service-Beschreibung in einer UDDI-Registry (vgl. Abschnitt 2.1.3) in Verbindung gesetzt werden kann.

WS-Policy findet in der Arbeit zur Beschreibung nicht-funktionaler Diensteigenschaften und -anforderungen Verwendung, welche als eine Eingabe für das in Kapitel 4 beschriebene Partitionierungsverfahren für BPEL-Prozesse

[1] Obwohl die WS-Policy-Attachment-Spezifikation die Vorgehensweise der Annotation lediglich für WSDL 1.1-Elemente beschreibt, kann dieselbe Vorgehensweise auch auf WSDL 2.0-Elemente angewendet werden.

dienen, sowie zur Parametrisierung von Prozessmodellen im Hinblick auf den Partitionierungsvorgang.

2.1.3 Universal Description, Discovery and Integration (UDDI)

Dienstverzeichnisse, sogenannte *Service-Registries*, bilden eine Schnittstelle zwischen Dienstanbieter und Dienstnutzer und sind damit ein wesentlicher Bestandteil jeder SOA [WCL+05]. Dienstanbieter nutzen sie zur Veröffentlichung von Beschreibungen der von ihnen angebotenen Dienste; potentielle Dienstnutzer befragen sie auf der Suche nach Diensten, die ihren Anforderungen genügen.

Universal Description, Discovery and Integration (UDDI) [oas04], ist eine etablierte Implementierung des *Service-Registry*-Konzepts für das Anwendungsfeld Service-orientierter Architekturen. Die zentralen Bestandteile der UDDI-Spezifikation sind (i) ein generisches Datenmodell zur Beschreibung von Dienstmerkmalen, bezeichnet als *Registry-Information-Model (RIM)*, und (ii) eine maschinell nutzbare Web-Service-Schnittstelle zum Ablegen von Daten und deren Abfrage.

Das UDDI-Datenmodell ist generisch, d. h. es ist an keine spezielle Domäne gebunden, sondern erlaubt eine Beschreibung beliebiger Dienste. Erreicht wird die Domänenunabhängigkeit durch das Konzept der Kategorisierung: jedes in einer UDDI-Registry veröffentlichte Datum kann durch Schlüssel-Wert-Paare kategorisiert werden, welche von Klienten in Abfragen an die Registry als Teil von Suchanfragen verwendet werden können. Zur Kategorisierung von Daten definiert die UDDI-Spezifikation eine Menge standardisierter Kategorien, welche direkt zur Dienstbeschreibung verwendet werden können, wie beispielsweise *UNSPSC* [1]zur Kategorisierung von Produkten und Dienstleistungen und *ISO 3166* [2] zur geographischen Zuordnung von Diensten. Für Fälle, in denen die vorgegebenen Kategorisierungen nicht ausreichen, erlaubt UDDI die Definition beliebiger anwendungsspezifischer Kategorisierungsschemata.

Die Kernelemente des UDDI-Datenmodells sind:

[1] http://www.unspsc.org/
[2] http://www.iso.org/iso/country_codes.htm

BusinessEntity Eine BusinessEntity beschreibt den Anbieter eines Diensts mit Kontaktinformationen, dessen Kategorisierung und Verweisen auf vom Anbieter angebotene Dienste.

BusinessService Ein BusinessService repräsentiert einen, von einer BusinessEntity angebotenen, Dienst, ordnet diesen in Dienst-Kategorien ein und verweist auf beliebig viele BindingTemplate-Elemente.

BindingTemplate Ein BindingTemplate beschreibt eine konkrete Implementierung eines Diensts, den Endpunkt, an dem mit diesem interagiert werden kann (der sogenannte AccessPoint) und weitere, für die Interaktion mit dem Dienst notwendige technische Informationen durch Referenzierung entsprechender TModel-Elemente. Jedes BindingTemplate eines BusinessService repräsentiert also eine, über eine bestimmte Kombination aus Kommunikations- und Transportprotokoll ansprechbare, Implementierung desselben (abstrakten) Diensts. Weiterhin kann das BindingTemplate durch die OverviewURL im OverviewDoc auf das WSDL-Dokument des Diensts verweisen.

TModel Ein TModel ist ein Platzhalter für ein, in einer UDDI-Registry abgelegtes, Konzept. Durch die Verknüpfung eines Datums mit einem TModel wird das Datum entsprechend der Bedeutung des TModel kategorisiert [WCL+05]. Die Zuordnung bzw. Referenzierung eines TModel erfolgt mittels einer sogenannten KeyedReference.

Für die Interaktion zwischen einer Registry und ihren Nutzern bietet eine UDDI-Registry mehrere, in Form eines Web-Service angebotene, Schnittstellen; die für die funktionalen Aspekte der Registry zentralen Schnittstellen sind die PublicationAPI und die InquiryAPI. Die PublicationAPI wird von Dienstanbietern verwendet, um Informationen über von ihnen angebotene Dienste in die Registry einzubringen. Die InquiryAPI wird von potentiellen Dienstnutzern zur Suche und Abfrage von Daten verwendet. Weitere standardisierte Schnittstellen adressieren Sicherheit und Replikation von Registry-Servern.

Im weiteren Verlauf der Arbeit findet UDDI zur Aggregation und Veröffentlichung der Beschreibungen der von den ausgeführten Prozessen verwendeten

Web-Services (Abschnitt 4.2.2.1) sowie der PS-Infrastruktur-Dienste (Abschnitt 5.2.2) Verwendung, die von den einzelnen Ausführungsteilnehmern zur Verwendung als Teil einer dezentralen Prozessausführung eingebracht werden.

2.1.4 SOAP

Das *SOAP-Messaging-Framework* [GHM+07a] definiert ein standardisiertes Nachrichtenformat sowie Regeln, die die Verarbeitung von Nachrichten steuern. Eine Nachricht, ein sogenannter *SOAP-Envelope*, umfasst dabei beliebig viele *Header* und genau einen *Body*. Während der Body einer SOAP-Nachricht die Nutzdaten (die auch einen anwendungsspezifischen Fehler in Form eines sogenannten *SOAP-Fault* repräsentieren können), die zwischen Dienstnutzer und Dienstanbieter ausgetauscht werden sollen, enthält, beschreiben die Header einer SOAP-Nachricht Informationen, die während der Übermittlung der Nachricht verarbeitet werden. Nachrichten werden zwischen einem Absender (engl. *Original Sender*) und einem Empfänger (engl. *Ultimate Receiver*) ausgetauscht. Auf dem Weg der Übermittlung zwischen Absender und Empfänger können SOAP-Nachrichten durch beliebig viele SOAP-Prozessoren, sogenannte *Intermediaries*, verarbeitet und manipuliert werden. Der eigentliche Übermittlungsvorgang kann, je nach Anforderungen von Dienstnutzer und Dienstanbieter, unter Verwendung unterschiedlicher (Netzwerk-) Transportprotokolle erfolgen. Realisiert wird dies durch den Mechanismus der sogenannten *SOAP-Bindings* [GHM+07a, WCL+05]. Ein SOAP-Binding für ein bestimmtes Transportprotokoll definiert zu diesem Zweck eine Serialisierung einer SOAP-Nachricht in einer Form, in der sie von einem Sender über das jeweilige Transportprotokoll (z. B. HTTP [FGM+99], JMS [sun08], SMTP [Kle08]) übermittelt und von einem Empfänger (was Intermediaries einschließt) wieder zu ihrer ursprünglichen Form rekonstruiert und verarbeitet werden kann. Weiterhin legt eine Binding-Beschreibung fest, wie die Kommunikationsprimitive des Transportprotokolls (d. h. dessen Operationen, z. B. *POST* bei HTTP, oder *GET* und *PUT* bei JMS) verwendet werden, um die serialisierte SOAP-Nachricht zwischen den Knoten auf dem Nachrichtenpfad vom Absender zum Empfänger zu übermitteln.

Da der Aufruf komplexer Geschäftsfunktionen in der Regel umfangreichere Interaktionsmuster als eine unidirektionale Nachrichtenübermittlung des Senders an den Empfänger bedingt, erlaubt SOAP in Verbindung mit *WS-Addressing* [GHR06] unter anderem eine eindeutige Identifikation von Nachrichten, sowie die Beschreibung des Protokollschritts eines MEP welcher durch die jeweilige Nachricht repräsentiert wird.

In Abschnitt 6 wird ein Web-Service-Binding für die Kommunikation von Dienstnutzer und Dienstanbieter über *Tuplespaces* [Gel85] beschrieben, das alle oben aufgeführten Aspekte einer Binding-Beschreibung abdeckt und sich auf die vorgestellten Standards SOAP und WS-Addressing stützt.

2.2 Orchestrierung von Web-Services (WS-BPEL)

Die *Web Service Business Process Execution Language (WS-BPEL)* [Org07] ist ein Standard für die Beschreibung von Prozessen, deren funktionale Bestandteile Web-Services sind. Basierend auf den Ideen des sogenannten *Two-Level-Programming* [Ley03] (vgl. Abschnitt 1.1) werden einzelne Web-Services durch Orchestrierungslogik zu einer Anwendung verbunden, welche selbst wiederum potentiellen Dienstnutzern in Form eines Web-Service angeboten wird.

Die BPEL-Spezifikation definiert dabei sowohl Prozesse, die durch ein entsprechendes WfMS automatisiert ausgeführt werden können – sogenannte *ausführbare Prozesse (Executable Processes)* – als auch nur unvollständig spezifizierte, in dieser Form nicht ausführbare *abstrakte Prozesse* (engl. *Abstract Processes*), welche beispielsweise zur Dokumentation von Arbeitsabläufen oder als Vorlage für eine *ausführbare Vervollständigung* (engl. *Executable Completion*) des Prozesses dienen können. Da der Fokus dieser Arbeit die automatisierte dezentrale Ausführung von Prozessen ist, werden abstrakte Prozesse in der folgenden Vorstellung von BPEL nicht weiter betrachtet.

Das Mittel der Beschreibung der Orchestrierungslogik eines BPEL-Prozesses sind die sogenannten *Aktivitäten*. Aktivitäten lassen sich wie folgt klassifizieren. Sogenannte *Strukturierte Aktivitäten* (engl. *Structured Activities*) erlauben die Beschreibung des Prozesskontrollflusses; Beispiele für strukturierte Aktivitäten sind: SEQUENCE für sequentielle Ausführung von Aktivitäten, FLOW für parallele

Ausführung von Aktivitäten und graphbasierte Beschreibung von Kontrollflussabhängigkeiten, IF für blockstrukturierte bedingte Aktivitätsausführung und verschiedene Aktivitäten für die iterative Aktivitätsausführung. BPEL vereint damit sowohl die nötigen Mittel zur graph- als auch zur blockstrukturierten Modellierung von Prozessen [KMWL08, KMWL09]. Sogenannte *einfache Aktivitäten* (engl. *Basic Activities*) erlauben die Interaktion des WfMS mit WfMS-externen Web-Services (z. B. INVOKE zum Web-Service-Aufruf oder RECEIVE und PICK zum Empfang eingehender Nachrichten) oder realisieren WfMS-interne Verarbeitungsschritte.

Neben Aktivitäten erlaubt BPEL die Verwendung instanzspezifischer typisierter Variablen, deren Werte durch die Aktivitäten des Prozesses gelesen und manipuliert werden können. Es gilt, dass mehrere Instanzen eines BPEL-Prozesses auf demselben WfMS während ihrer Ausführung isoliert voneinander ausgeführt werden, d. h. sich diese weder in der Ausführung ihrer Aktivitäten noch im Wert ihrer Instanzvariablen gegenseitig beeinflussen.

BPEL ist eine kontrollflussgetriebene Prozessbeschreibungssprache; d. h. die Aktivitäten eines BPEL-Prozesses sind dann zur Ausführung durch das WfMS bereit, wenn die durch die Kontrollkonstrukte [Seb07] des Prozessmodells definierten Kontrollflussabhängigkeiten es erlauben. Die Evaluierung des Kontrollflusses einer in Ausführung befindlichen Prozessinstanz erfolgt in traditionellen WfMS zur Ausführung von BPEL durch einen *logisch zentralen Navigator* (vgl. Abschnitt 1.2).

Unterschiedliche Arten der Realisierung der Navigation in WfMS zur Ausführung von BPEL mit einer zentralen Architektur werden in Abschnitt 2.3.2 vorgestellt. Abschnitt 2.4 stellt darüber hinaus die Architektur und den Navigationsvorgang verschiedener Systeme zur verteilten Prozessausführung vor.

Im weiteren Verlauf der Arbeit findet BPEL an verschiedenen Stellen Verwendung. Dies sind im Wesentlichen das in Kapitel 4 aufgezeigte Partitionierungsverfahren für BPEL-Prozesse und die Realisierung der in den Abschnitten 7.2 und 7.2 vorgestellten PS-Klienten.

2.3 Workflow Management Systeme mit zentraler Navigation

Gegenstand dieses Abschnitts ist die Vorstellung eines WfMS-Referenzmodells (Abschnitt 2.3.1) sowie die Präsentation der Architektur und einiger Implementierungsaspekte unterschiedlicher quelloffener WfMS zur Ausführung von BPEL-Prozessen (Abschnitt 2.3.2).

2.3.1 WfMC-Referenzmodell

Das *WfMC Reference Model* [wfm95] der *Workflow Management Coalition*[1] beschreibt ein Referenzmodell für WfMS, bestehend aus der Vorstellung ihrer wesentlichen Eigenschaften und Funktionen, einer Terminologie zu deren Beschreibung und einer Übersicht typischer Komponenten und Schnittstellen eines WfMS.

Generell lassen sich die von einem WfMS bereitgestellten Funktionen den Bereichen *Build-Time* und *Run-Time* zuordnen: Unter dem Begriff Build-Time werden all jene Funktionen eines WfMS zusammengefasst, die mit dem Vorgang der Prozessmodellierung befasst sind. Die Run-Time eines WfMS vereint all jene Funktionen, die für die Ausführung von Prozessen und die damit verbundene Interaktion des WfMS mit Menschen und/oder WfMS-externen Diensten befasst sind. Die Überführung eines Prozesses von der Build-Time in die Run-Time wird als *Deployment* [LR99] bezeichnet[2].

Abbildung 2.2 stellt die Komponenten und Schnittstellen des WfMC-Referenzmodells graphisch dar. Dieses definiert fünf Schnittstellen, über die die einzelnen *Workflow Engine(s)* des sogenannten *Workflow Enactment Service* mit den unterschiedlichen Komponenten des WfMS interagieren. Im Einzelnen sind dies: *Process Definition Tools (Interface 1)*, *Workflow Client Applications (Interface 2)*, *Invoked Applications (Interface 3)*, *Other Workflow Enactment Services (Interface 4)* und *Administration & Monitoring Tools (Interface 5)*. Die Gesamtheit der Schnittstellen bezeichnet man als das *Workflow Application Programming Interface (WAPI)*.

[1] http://www.wfmc.org/

[2] Im weiteren Verlauf der Arbeit wird für einen Prozess, der den Deploymentvorgang durchlaufen hat, der Begriff *installiert* als Übersetzung des englischen Begriffs *deployed* verwendet.

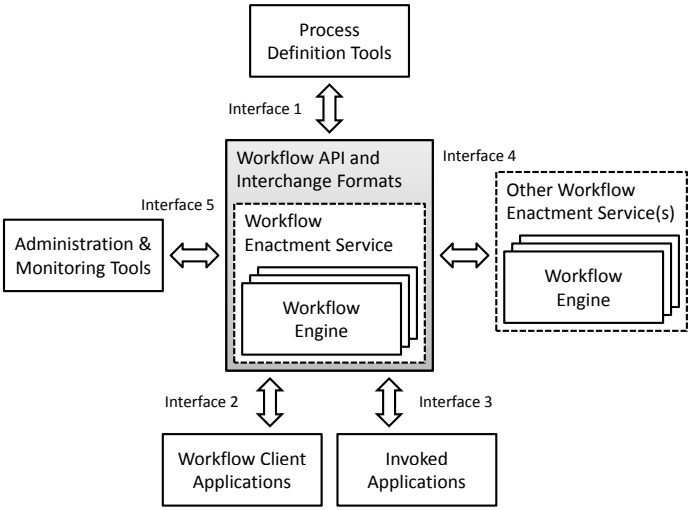


Abbildung 2.2: WfMC-Referenzmodell

Der *Workflow Enactment Service (WfES)* umfasst eine oder mehrere sogenannte *Workflow Engines*, welche Prozessinstanzen erzeugen, verwalten und ausführen, d. h. navigieren. Im Rahmen der Ausführung von Prozessinstanzen durchgeführte Interaktionen mit WfMS-externen Ressourcen erfolgen über die Schnittstellen 2 und 3, abhängig davon, ob die Interaktion mit einem Klienten (z. B. einer Applikation zur Verwaltung sogenannter *Worklists*) oder einem WfMS-externen Dienst durchgeführt werden soll. Schnittstelle 1 beinhaltet die zur Übergabe von Prozessmodellen aus der Build-Time in die Run-Time eines WfMS notwendigen Funktionen. Schnittstelle 4 beschreibt Funktionen, die für eine kooperative Ausführung eines Prozesses durch mehrere unterschiedliche WfES, aus potentiell unterschiedlichen Domänen, erforderlich sind.

Im Hinblick auf die Ausführung von Prozessen unter Verwendung mehrerer WfES beschreibt das WfMC-Referenzmodell vier unterschiedliche Szenarien:

Szenario 1: Concrete Discrete (Chained) In diesem Szenario kooperativer Workflow-Ausführung wird der Prozess in einzelne Abschnitte eingeteilt

und jeder Abschnitt einem WfES zugeordnet. Die Ausführung der Abschnitte erfolgt dabei sequentiell verkettet; d. h. nach Ausführung des durch den ersten WfES ausgeführten Abschnitts wird die Kontrolle an das, für die Ausführung des zweiten Abschnitts zuständige, WfES übergeben. Analog wird mit den nachfolgenden Abschnitten verfahren. Eine Rückgabe der Kontrolle an den WfES eines vorangehenden Abschnitts erfolgt in diesem Szenario nicht.

Szenario 2: Hierarchical (Nested Subprocesses) Szenario 2 beschreibt eine Ausführung eines Prozesses, in welcher eine, durch einen bestimmten WfES ausgeführte, Aktivität durch einen Prozess realisiert wird, welcher durch einen anderen WfES ausgeführt wird. Die Ausführung der entsprechenden Aktivität führt also zur Übergabe der Kontrolle an den WfES, welcher den durch die Aktivität repräsentierten Prozessteil (auch *Sub-Prozess* genannt) ausführt. Nach der Ausführung des Sub-Prozesses wird die Kontrolle an den aufrufenden WfES zurückgegeben.

Szenario 3: Connected Indiscrete (Peer-to-Peer) Im Gegensatz zu Szenarien 1 und 2, in welchen jeweils gesamte Prozessteile durch unterschiedliche WfES ausgeführt werden, beschreibt Szenario 3 eine Ausführung eines Prozesses durch mehrere WfES, wobei theoretisch jede Aktivität des Prozesses durch einen unterschiedlichen WfES ausgeführt werden kann.

Szenario 4: Parallel Synchronized Szenario 4 beschreibt die Synchronisation der Ausführung unterschiedlicher Prozesse durch unterschiedliche WfES an bestimmten Punkten der Workflows. Dabei erfolgt die eigentliche Ausführung der Workflows auf beiden WfES weitgehend unabhängig. Erreicht ein WfES allerdings den Synchronisationspunkt, so blockiert es die weitere Ausführung des Workflows so lange, bis auch die anderen an der Ausführung des Workflows teilnehmenden WfES ihre entsprechenden Synchronisationspunkte erreicht haben.

In der Terminologie des WfMC-Referenzmodells kann die in Kapitel 5 vorgestellte Infrastruktur zur dezentralen Ausführung von BPEL-Prozessen als eine

Realisierung von Szenario 3, *Connected Indiscrete*, klassifiziert werden.

2.3.2 Implementierungsbeispiele zentraler WfMS

Anknüpfend an die Vorstellung einiger Elemente der Architektur klassischer WfMS anhand des WfMC-Referenzmodells werden in der Folge drei Beispiele quelloffener WfMS zur Ausführung von BPEL vorgestellt. Einen Schwerpunkt der Darstellung bildet, neben der Vorstellung der Architektur der Systeme, die Erläuterung des Navigationsvorgangs während der Instanzausführung im jeweiligen System.

2.3.2.1 Apache ODE

Apache ODE[1] (in der Folge kurz *ODE*) ist ein quelloffenes WfMS der *Apache Software Foundation*. ODE unterstützt WS-BPEL 2.0 [Org07] als Prozessbeschreibungssprache. Abbildung 2.3 zeigt die Architektur von ODE. Die internen Komponenten lassen sich entsprechend ihrer Nähe zum funktionalen Kern des WfMS entweder der sogenannten *BPEL-Runtime* oder dem ODE-Gesamtsystem zuordnen. Die Komponenten und deren Funktionen sind im Einzelnen:

Process Deployment Die *Process-Deployment*-Komponente stellt die Schnittstelle für das Einbringen sogenannter *Deployment-Units* in das WfMS dar. Eine Deployment-Unit beinhaltet das Prozessmodell, zugehörige WSDL-Dateien, die die durch den Prozess implementierte Web-Service-Schnittstelle beschreiben, und einen Deployment-Deskriptor. Dieser beinhaltet Informationen über die Interaktionspartner des Prozesses, wobei für jedes PARTNERLINK-Element spezifiziert ist, ob der Prozess auf dem jeweiligen PARTNERLINK einen Dienst anbietet (`provide`) oder entlang diesem einen Dienst nutzt (`invoke`). Weiterhin besteht die Möglichkeit, im Deployment-Deskriptor das Persistierungsverhalten des WfMS für die Instanzen des jeweiligen Prozessmodells zu beeinflussen, um beispielsweise zu Gunsten höherer Performance zur Ausführungszeit (unter

[1] `http://ode.apache.org`

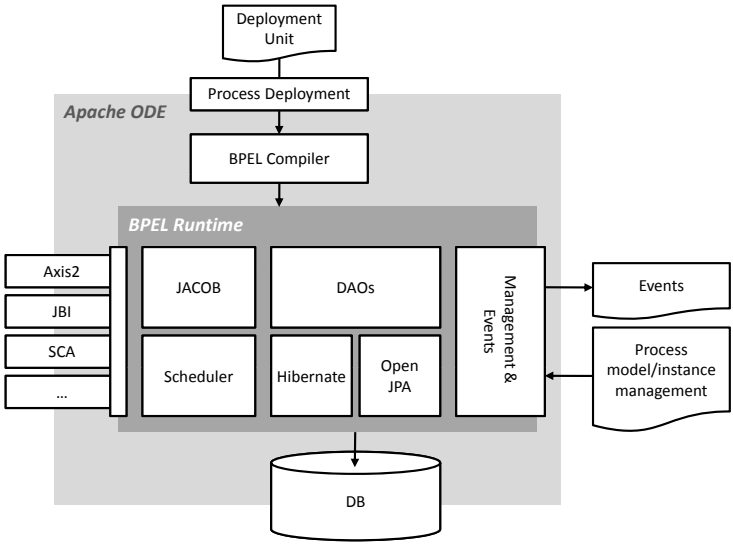


Abbildung 2.3: Architektur von *Apache ODE* (basierend auf [Les08]).

Verlust der Wiederherstellbarkeit nach einem Systemabsturz) auf eine Persistierung des Zustands von Prozessinstanzen zu verzichten.

BPEL Compiler Während des Prozess-Deployment-Vorgangs überführt der sogenannte *BPEL-Compiler* die installierten Prozessmodelle in ein internes Objektmodell (das sogenannte `OModel`) und ergänzt dieses durch Informationen aus den WSDL-Dokumenten und dem Deployment-Deskriptor. Die Objekte der `OModel`-Hierarchie bilden die Basis für die Verarbeitung der Prozessbeschreibung zur Laufzeit von Prozessinstanzen durch die sogenannte *BPEL-Runtime*.

BPEL Runtime Die *BPEL-Runtime* bietet (i) Implementierungen der einzelnen BPEL-Aktivitäten und realisiert (ii) die persistente Haltung von Prozessinstanzdaten. Darüber hinaus bietet sie Schnittstellen für externes Management von Prozessen und Prozessinstanzen und für die Integration unterschiedlicher Mechanismen zur Interaktion mit WfMS-externen

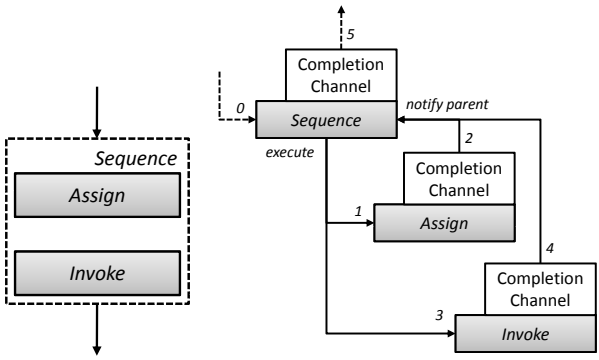
Diensten, wie beispielsweise *Apache Axis 2*[1] oder *Java Business Integration (JBI)*[2].

Data Access Objects Um eine verlässliche Ausführung von Prozessinstanzen zu garantieren, werden sämtliche Informationen, die im Laufe der Verarbeitung einer Prozessinstanz anfallen, in einem persistenten Speicher abgelegt. Realisiert wird dies unter Verwendung einer Abstraktionsschicht aus *Data Access Objects* [Fow03], welche auf unterschiedliche sogenannte *Persistenz-Provider*, wie beispielsweise *Hibernate*[3] oder *OpenJPA*[4], aufgesetzt werden kann. Die persistent gehaltenen Daten beinhalten, die aktuellen Werte der Instanzdaten des Prozesses (d. h. z. B. dessen Variablen oder Partner-Links) und seinen Kontrollflusszustand. Weiterhin werden eingehende Nachrichten vor ihrer Verarbeitung durch das WfMS persistent zwischengespeichert.

Management Für die Integration externer Werkzeuge bietet ODE eine Reihe von Schnittstellen, die beispielsweise das externe Abfragen aktuell installierter Prozessmodelle sowie den Zustand von deren Instanzen ermöglichen. Diese Schnittstellen stehen sowohl zur programmatischen Nutzung in Form eines API als auch für einen menschlichen Nutzer durch einen Web-Browser nutzbares Web-Interface zur Verfügung.

Basis der Navigation in Apache ODE bildet das auf den Konzepten des π-Kalküls (siehe auch Abschnitt 6.3.1) aufbauenden *Java Concurrent Objects (JACOB)* Framework[5]. Dieses realisiert zwei für die Prozessausführung notwendige Kernfunktionalitäten: zum einen ist dies eine persistente Haltung des aktuellen Ausführungszustands einer Prozessinstanz (welche in ODE durch eine Menge von Objekten repräsentiert wird, die durch Kanäle (engl. *Channels*) miteinander verbunden sind), zum anderen die Ausführung nebenläufiger Aktionen.

[1] http://ws.apache.org/axis2
[2] http://jcp.org/aboutJava/communityprocess/final/jsr208/index.html
[3] http://www.hibernate.org
[4] http://openjpa.apache.org
[5] http://ode.apache.org/jacob.html



(a) Sequenz zweier Aktivitäten.

(b) Navigation der Sequenz unter Verwendung von JACOB.

Abbildung 2.4: Navigation unter Verwendung von JACOB.

Kernidee von JACOB ist, dass Abfolgen von Operationen durch die Aktionen *Continue* und *ListenOn* in einzelne Einheiten zerlegt werden können. Dabei persistiert *Continue* den aktuellen Ausführungszustand des Systems und gibt die Kontrolle über die Ausführung anschließend an eine Folgeoperation weiter. *ListenOn* persistiert den Systemzustand gleichermaßen, wartet anschließend allerdings auf das Eintreten eines externen Ereignisses. Ein derartiges Aufsplitten von Operationsfolgen erreicht zum einen die geforderte Persistenz des Ausführungszustands, zum anderen erlaubt sie eine quasi-parallele Verarbeitung nebenläufiger Operationen.

Abbildung 2.4 zeigt ein Beispiel, das als Grundlage für die Erläuterung des Navigationsvorgangs auf Basis des oben beschriebenen Modells dienen soll. Abbildung 2.4(a) zeigt einen Ausschnitt eines BPEL-Prozesses, in welchem die Aktivitäten ASSIGN und INVOKE innerhalb einer SEQUENCE-Aktivität angeordnet sind; es existiert also eine Kontrollflussabhängigkeit zwischen der Ausführung der SEQUENCE-Aktivität und der ASSIGN-Aktivität sowie der ASSIGN- und der INVOKE-Aktivität. Abbildung 2.4(b) zeigt den Ablauf des in Abbildung 2.4(a) dargestellten Prozessausschnitts zur Ausführungszeit. Die Ausführung der SE-

QUENCE-Aktivität hat den Aufruf der *Continue*-Operation und damit sowohl die Persistierung des Ausführungszustands der Prozessinstanz als auch die Weitergabe der Ausführungskontrolle an die ASSIGN-Aktivität zur Folge. Als Teil der Weitergabe der Kontrolle übergibt SEQUENCE einen sogenannten *Completion Channel* an ASSIGN, auf welchem SEQUENCE eine Benachrichtigung über die Fertigstellung der Ausführung von ASSIGN erwartet, um die Ausführung nachfolgender Aktivitäten veranlassen zu können. Damit ist die Bedingung für die Ausführung der ASSIGN-Aktivität erfüllt. Im Fall, dass zu diesem Zeitpunkt die Ausführungsbedingungen weiterer Aktivitäten erfüllt sind, kann jede dieser Aktivitäten als nächste zur Ausführung kommen. Die Fertigstellung der Ausführung signalisiert ASSIGN über den *Completion Channel* an die SEQUENCE-Aktivität, welche daraufhin die Ausführung der INVOKE-Aktivität veranlasst. Eine INVOKE-Aktivität kann eine prinzipiell lang andauernde Interaktion mit einem WfMS-externen Kommunikationspartner nach sich ziehen, was zur Folge hat, dass – nach Ausführung des Dienstaufrufs – die INVOKE-Aktivität die Kontrolle der Prozessausführung an eine andere ausführungsbereite Aktivität abgibt und mit der *Listen On*-Operation auf das Auftreten eines sogenannten *externen Ereignisses* – in diesem Fall das Eintreffen der entsprechenden Antwortnachricht zur gesendeten Anfragenachricht – wartet. Nach Eintreten des Ereignisses setzt ODE die Ausführung der INVOKE-Aktivität fort, welche (analog ASSIGN) die Fertigstellung ihrer Ausführung über ihren *Completion Channel* an SEQUENCE zurückmeldet, woraufhin die Ausführung des dargestellten Prozessfragments abgeschlossen ist.

2.3.2.2 JBoss jBPM

JBoss jBPM[1] ist ein WfMS, welches, im Gegensatz zu den meisten anderen WfMS, nicht auf die Ausführung von Prozessen eines bestimmten Prozessmetamodells festgelegt ist, sondern die Ausführung von Prozessen unterschiedlicher Prozessmetamodelle erlaubt, indem für die einzelnen Prozessmetamodelle eine Abbildung auf ein generisches Metamodell (Abbildung 2.5) definiert wird. Die Ausführungsumgebung für dieses generische Prozessmetamodell ist die

[1] http://www.jboss.org/jbossjbpm

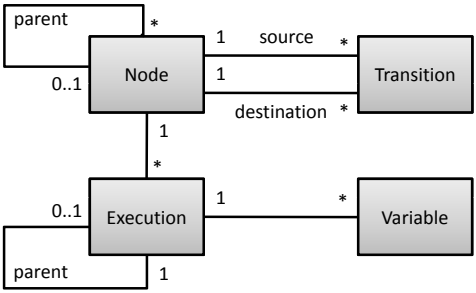


Abbildung 2.5: Vereinfachte Sicht auf das PVM-Metamodell (basierend auf [WWWa]).

sogenannte *Process Virtual Machine (PVM)* [WWWa].

Eine Aktivität wird im Metamodell der PVM durch einen *Knoten* (engl. *Node*) repräsentiert. Ein Knoten kann auf zwei Arten mit anderen Knoten in Beziehung gesetzt werden. Zum einen werden durch sogenannte *Transitionen* (engl. *Transitions*) Kontrollflussabhängigkeiten zwischen einem Knoten und seinen Vorgänger- und Nachfolger-Knoten realisiert; zum anderen kann durch die *Parent*-Relation eine Vater-Kind-Beziehung zwischen Knoten aufgebaut werden.

Eine *Execution* repräsentiert eine Instanz eines Prozessmodells zur Ausführungszeit. Bei der Instantiierung eines Prozesses wird ein zur Start-Aktivität korrespondierendes *Execution*-Objekt erzeugt. Eine *Execution* implementiert dabei die Methode `proceed`, welche die Kontrolle über die Prozessausführung an die Nachfolger-Aktivitäten der jeweiligen Aktivität (durch einen synchronen Funktionsaufruf) weitergibt. Um die im Verlauf der Ausführung einer Prozessinstanz anfallenden Daten zwischenzuspeichern, unterstützt PVM Instanzvariablen, die *Executions* zugeordnet sind. *Aktionen* (engl. *Actions*) realisieren beliebige Anwendungslogik, welche an bestimmten Ankerpunkten einer *Execution* ausgeführt werden kann. Mögliche Ankerpunkte sind beispielsweise der Eingang einer Transition in einen Knoten, der Ausgang eines Knotens in eine Transition oder die Transitionen des Prozesses selbst. Gestattet der vom

Prozess spezifizierte Kontrollfluss die parallele Verarbeitung von Knoten, so wird jeder parallele Ausführungsstrang des Prozesses durch eine eigene *Execution* realisiert. Die Beziehung zwischen dem Knoten, der einen sogenannten *Fork* [RHAM06] realisiert, und den, von diesem gestarteten, Ausführungssträngen des Prozesses wird durch die *Parent*-Relation zwischen den beteiligten *Execution*-Objekten ausgedrückt. Der *Fork*-Knoten selbst bleibt über die Ausführung der von ihm gestarteten parallelen Ausführungsstränge hinweg inaktiv. Zum Zeitpunkt der Evaluierung des dem *Fork* zugehörigen *Join* wird – nach erfolgter Ausführung sämtlicher eingeschlossener Knoten – der *Fork*-Knoten wieder aktiviert und die Ausführung von dessen *Execution* fortgesetzt. Für den Fall, dass die Ausführung der Anwendungslogik von Knoten oder Transitionen rechenintensiv oder zeitaufwändig sind und dementsprechend eine synchrone Weitergabe der Kontrolle über die Prozessausführung nicht praktikabel ist, unterstützt PVM sogenannte *Asynchronous Continuations*. Diese basieren auf dem Prinzip der transaktionsgeschützten Einstellung eines Arbeitsauftrags in eine *Message Queue* [BHL95] und deren asynchrone Verarbeitung durch einen sogenannten *Job Executor*.

PVM bietet darüber hinaus eine Reihe zusätzlicher Funktionen für die Ausführung von Prozessen. Dies sind beispielsweise: die prozessinstanzspezifische Änderung von Prozessmodellen durch sogenannte *Process Updates*, die Protokollierung des Ablaufs von Prozessinstanzen in Form einer *History*, die zeitgestützte Ausführung von Aktionen durch einen *Timer*-Dienst und die persistente Vorhaltung des Ausführungszustands von Prozessinstanzen unter Verwendung von *Object Relational Mapping (ORM)* Technologien.

2.3.2.3 ActiveEndpoints ActiveVOS

Das Produkt *ActiveVOS*[1] der Firma *ActiveEndpoints* vereint Werkzeuge zur Prozessmodellierung, -simulation und -verwaltung von BPEL-Prozessen mit einer Laufzeitumgebung für deren Ausführung. Neben dem Standard WS-BPEL 2.0 [Org07] implementiert *ActiveVOS* die Spezifikationen *WS-BPEL Extensions for People* [AAD+07b] und *WS-Human Task* [AAD+07a]. Des Weiteren bietet *Acti-*

[1] http://www.activevos.com/

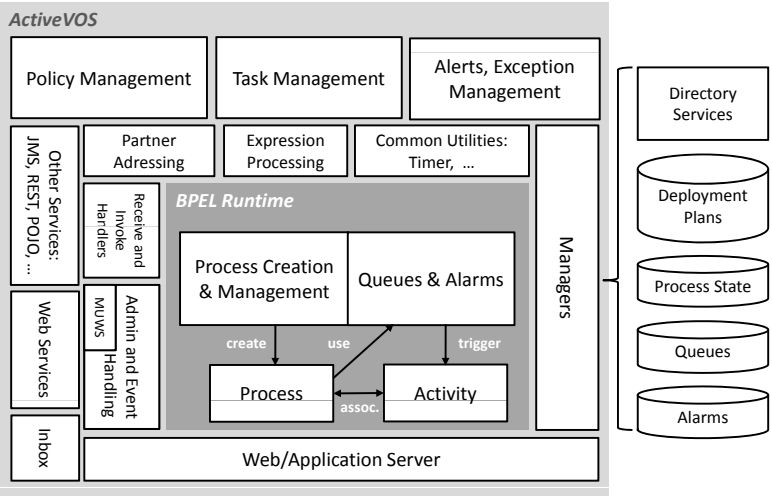


Abbildung 2.6: *ActiveVOS*-Architektur [WWWb].

veVOS durch ein entsprechendes Rahmenwerk die Möglichkeit zur Erweiterung des WfMS um die Unterstützung zusätzlicher *Expression Languages*; im Lieferumfang unterstützt werden XPath 1.0 [CD+99], XPath 2.0 [BBC+07] und JavaScript [Int99].

Abbildung 2.6 zeigt die Architektur von *ActiveVOS*. Basis des Systems bildet ein Application-Server, welcher zum einen die Plattform für die BPEL-Ausführungsumgebung des WfMS sowie die externen Web-Service-Schnittstellen der Administrations-, Taskmanagement- und Monitoring-Komponenten bietet.

Den funktionalen Kern von *ActiveVOS* bildet *ActiveBPEL*, eine Ausführungsumgebung für BPEL-Prozesse, welche die für Prozess-Deployment und -navigation notwendigen Kernfunktionen bereitstellt. Datenzugriff erfolgt in *ActiveBPEL* durch eine Abstraktionsschicht aus sogenannten *Managern*, was den transparenten Austausch der Technologien zur Persistierung von Prozessinstanzdaten erlaubt. An den Kern des WfMS angeschlossen sind Komponenten für beispielsweise die Verarbeitung von sogenannten *Expressions*, Adressierung der Interaktionspartner eines Prozesses und die Realisierung der Interaktion selbst.

Weiter entfernt vom Kern des WfMS sind die, für die Verarbeitung von Prozessen, die die *WS-BPEL Extensions for People* nutzen, notwendigen Funktionen, wie beispielsweise das *Taskmanagement* oder eine *Inbox*, angesiedelt. Um ggf. aufgetretene Fehler manuell durch einen Administrator zu korrigieren, erlaubt *ActiveVOS* den Eingriff in laufende Prozessinstanzen zur Ausführungszeit. Die hierfür notwendige Funktionalität wird durch die *Exception Management* Komponente bereitgestellt.

Das Deployment von Prozessen in *ActiveBPEL* bzw. *ActiveVOS* erfolgt durch einen Deployment-Container, welcher neben dem eigentlichen Prozess auch dessen Schnittstellenbeschreibung in Form eines WSDL-Dokuments, angeschlossene Schema-Beschreibungen sowie die WSDL-Beschreibungen der vom Prozess orchestrierten Dienste und einen Deployment-Deskriptor enthalten kann. Analog ODE (vgl. Abschnitt 2.3.2.1) beinhaltet dieser Deployment-spezifische Konfigurationsparameter, wie beispielsweise das sogenannte *Partner-Binding* oder Informationen über das Persistierungsverhalten der Prozessinstanzen.

Während des Prozess-Deployment-Schritts werden die vorhandenen Deployment-Container extrahiert und jedes in diesen enthaltene BPEL-Prozessmodell wird in eine XML-DOM-Repräsentation überführt. Der XML-Baum des Prozesses wird, dem *Visitor Pattern* [GHJV95] folgend, traversiert, wobei für jede Aktivität des Prozesses ein sogenanntes *Definition-Object* erzeugt wird. Für sämtliche instanzerzeugenden Nachrichten des Prozessmodells wird ein entsprechender *Receive-Handler* in der externen Kommunikationsschnittstelle des WfMS registriert.

Sendet ein Klient eine instanzerzeugende Nachricht an das WfMS, so wird sie vom entsprechenden *Receive-Handler* empfangen und die diesem zugeordnete Prozessinstanz instantiiert. Während der Prozessinstantiierung wird für jedes (während des Prozess-Deployment-Vorgangs) erzeugte *Definition-Object* ein entsprechendes Instanz-spezifisches *Implementation-Object* erzeugt. Der Vorgang der Prozessnavigation wird in *ActiveVOS* (bzw. *ActiveBPEL*) durch eine instanzspezifische Navigationswarteschlange – die sogenannte *Execution-Queue* – abgewickelt. Das Einstellen von Aktivitäten in die Navigationswarteschlange erfolgt durch deren Eltern-Aktivität sobald die Ausführungsbedingung der jeweilgen Aktivität erfüllt ist (bei einer SEQUENCE gilt dies beispielsweise, wenn die sämt-

liche Vorgänger-Aktivitäten der Aktivität erfolgreich ausgeführt wurden). Für die exemplarische Beschreibung der Navigation in *ActiveBPEL* sei angenommen, dass ein Prozess vorliegt, dessen primäre Aktivität eine SEQUENCE ist. Nach erfolgreicher Instantiierung des Prozesses wird das Implementation-Object der primären Aktivität des Prozesses in die Navigationswarteschlange eingestellt. Der Navigator des WfMS konsumiert dieses und führt dessen `execute` Methode aus. Sie implementiert die Anwendungslogik der jeweiligen BPEL Aktivität. Im Fall der SEQUENCE Aktivität umfasst diese das Einstellen der ersten Kind-Aktivität der SEQUENCE und der SEQUENCE-Aktivität selbst in die Navigationswarteschlange. Nach erfolgreichem Konsumieren der ersten Kind-Aktivität der SEQUENCE-Aktivität durch den Navigator wird diese in analoger Weise verarbeitet. Nach Abschluss deren Ausführung signalisiert die Kind-Aktivität ihren erfolgreichen Abschluss an ihre Vater-Aktivität, d. h. die SEQUENCE-Aktivität. Wird die SEQUENCE-Aktivität durch den Navigator erneut ausgeführt, so prüft dieser, ob noch weitere Kind-Aktivitäten zu verarbeiten sind und stellt sie ggf. in die Navigationswarteschlange zur weiteren Verarbeitung ein.

2.4 Verteilte Ausführung von Prozessen

Anknüpfend an die Erläuterung der Architekturen einiger zentraler WfMS zur Ausführung von BPEL-Prozessen im vorangehenden Abschnitt, werden nachfolgend einige Ansätze zu deren verteilter Ausführung vorgestellt.

Abhängig vom jeweiligen Anwendungsfall des Systems motivieren unterschiedliche Beweggründe die verteilte Prozessausführung. Die Motivationen reichen dabei von einer Verbesserung des Laufzeitverhaltens der Prozessausführung, über optimale Nutzung zur Verfügung stehender Ressourcen und der flexiblen Anpassung von Prozessen an hochgradig dynamische Umgebungen, bis hin zum sogenannten *Refactoring* von Prozessen aus Gründen des *Process Outsourcing*, d. h. der Ausgliederung einzelner Teile der Prozesse an externe Dienstleister.

2.4.1 OSIRIS

OSIRIS [SWSS03, SWSS04, Sch05] ist ein System zur verteilten Ausführung von Prozessen unter Verwendung einer sogenannten *Hyper-Datenbank (HDB)*.

Eine HDB ist ein verteiltes *Datenbank-Management-System (DBMS)*, welches durch eine HDB-Softwareschicht auf jedem Knoten des verteilten Systems realisiert wird und Klienten die Möglichkeit bietet, mit entfernten Diensten unter Einhaltung von ACID-Eigenschaften und Garantien hinsichtlich Korrektheit der übertragenen Daten zu interagieren. Die Dienste werden dabei durch die HDB virtualisiert, d. h. ein Klient spezifiziert lediglich den Typ des Diensts, mit dem eine Interaktion erfolgen soll; das Finden des entsprechenden Diensts und das sogenannte *Routing* der Anfrage zum Dienst erfolgt für den Klienten transparent durch die HDB.

Das Auffinden angebotener Dienste wird durch ein globales, verteilt vorgehaltenes Dienstverzeichnis auf Basis einer *Publish-Subscribe-Infrastruktur* realisiert. Dienstanbieter melden sich am Dienstverzeichnis an, um Anfragen für die von ihnen angebotenen Dienste zu erhalten. Ein Klient kann einen Dienst nutzen, indem er seine Anfrage über das, dem Typ des aufzurufenden Diensts entsprechende, *Topic* veröffentlicht, woraufhin ein beim Dienstverzeichnis angemeldeter Dienst durch die HDB aufgerufen wird. Um zu verhindern, dass das Dienst-Verzeichnis selbst eine zentrale Komponente des Gesamtsystems wird, kommen Mechanismen zum Einsatz, die eine Replikation von Dienstinformationen auf die verschiedenen HDB-Knoten des Systems mit unterschiedlichen nicht-funktionalen Eigenschaften (z. B. hinsichtlich verlangter Aktualität durch sogenannte *Freshness Predicates* oder Vollständigkeit der replizierten Daten mittels sogenannter *Partial Replication*) erlauben. Diese Publish-Subscribe-Infrastruktur zum Dienstaufruf bildet auch die Grundlage der verteilten Ausführung von Prozessen in OSIRIS.

Die elementaren Bausteine des OSIRIS-Prozessmodells sind sogenannte *Activities*, welche einen angebotenen Dienst repräsentieren. Dienste sind in OSIRIS gekennzeichnet durch eine statisch definierte Schnittstelle und folgen stets einer *Request-Response*-Aufrufsemantik mit ACID-Eigenschaften. Weiterhin können Aktivitäten durch die Eigenschaften *wiederholbar* (engl. *repeatable*) oder

kompensierbar (engl. *compensatable*) gekennzeichnet sein. Die Eigenschaft der Wiederholbarkeit besagt, dass im Fall eines, während der Ausführung auftretenden, Fehlers eine Aktivität so oft wiederholt ausgeführt werden kann, bis der Aufruf erfolgreich abgeschlossen ist. Für eine kompensierbare Aktivität gilt, dass die von ihr erzeugten Effekte nach ihrer erfolgreichen Ausführung zurückgerollt werden können. Sogenannte *Pivot*-Aktivitäten beschreiben nicht-kompensierbare Aktivitäten, deren erfolgreicher Abschluss implizit einen *Commit* aller, im Prozessmodell vorangehenden, Aktivitäten zur Folge hat (so dass eine weitere Kompensation der Effekte der Aktivitäten nicht mehr möglich ist). Ein OSIRIS-Prozessmodell ist ein Graph, bestehend aus dessen Aktivitäten (repräsentiert als Knoten des Graphen) und einer partiellen Ordnung auf den Aktivitäten (repräsentiert durch die Kanten des Graphen). Unterschiedliche Kantentypen beschreiben die einer Aktivität nachfolgenden Aktivitäten in den Fällen (i) erfolgreiche Aktivitätsausführung, (ii) Auftreten eines Fehlers während der Aktivitätsausführung und (iii) Aktivität wurde kompensiert. Die Ausführung einer Prozessinstanz basiert auf der generellen Vorgehensweise, die Instanzdaten eines Prozesses über die, von der HDB angebotene Publish-Subscribe-Infrastruktur, an die, in der Prozessbeschreibung nachfolgenden Dienste, zu kommunizieren. Auf diese Weise *migriert* die gesamte Prozessinstanz als sogenanntes *Whiteboard* durch das verteilte System. Im Fall paralleler Ausführungsstränge im Prozessmodell wird das Whiteboard entsprechend den, auf dem jeweiligen Prozessstrang ausgeführten Diensten (und den Daten die diese benötigen), aufgespalten und anteilig an die Nachfolger-Dienste übermittelt. Am Ende des parallelen Prozessabschnitts werden die einzelnen Prozessstränge durch einen sogenannten *Join*-Knoten vereinigt; Teil der Vereinigung der Prozessstränge ist auch die Aggregation der einzelnen Teile des Whiteboards.

Die Architektur von OSIRIS umfasst die im Folgenden aufgeführten Komponenten. Der *Process-Manager* nimmt eingehende Dienstaufrufe von der *Communication*-Komponente entgegen und führt den eigentlichen Dienstaufruf unter Verwendung der *Service-Manager*-Komponente aus. Beinhaltet die Ausführung der Dienstfunktionalität konkurrierende Zugriffe auf Ressourcen, die von mehreren Prozessinstanzen gemeinsam genutzt werden, so stellt die *Concurrency-*

Control-Komponente einen synchronisierten Zugriff sicher. Nach Fertigstellung des Dienstaufrufs bestimmt der *Process-Manager* den oder die nachfolgend auszuführenden Dienst(e) und übergibt die Prozessinstanzdaten an die *Communication*-Komponente für die Übermittlung an den oder die nachfolgenden Knoten. Stehen mehrere Implementierungen eines Dienst-Typs zur Verfügung, wird mithilfe der *Load-Balancing*-Komponente eine dieser Implementierungen entsprechend ihrer Auslastung ausgewählt. Die Bestimmung der Ziel-Knoten für die Migration der Prozessinstanz erfolgt dabei durch die *Publish-Subscribe-Routing*-Komponente, welche die hierfür nötigen Information mithilfe der *Replication-Manager*-Komponente aus dem globalen Dienst-Verzeichnis repliziert.

Obwohl das Ziel von OSIRIS – die verteilte Ausführung von Prozessen – dem Ziel der vorliegenden Arbeit entspricht, unterscheidet sich der von OSIRIS verfolgte Ansatz der Virtualisierung von Diensten durch ein Peer-to-Peer-Netzwerk grundsätzlich von dem dieser Arbeit zugrunde liegenden Ansatz. Ziel der PS-Infrastruktur ist die Ausführung von Produktionsprozessen (vgl. Abschnitt 5.1). Die Auswahl der Partner, die an der Ausführung eines Prozesses teilnehmen, ist in Produktionsprozessen durch unterschiedliche – funktionale, nicht-funktionale und organisatorische – Einflussgrößen geprägt (vgl. Abschnitt 4.2), die im Gegensatz zu OSIRIS eine Definition der Partitionierung eines Prozesses zu seiner Deployment-Zeit motivieren. Während in der PS-Infrastruktur jedem Instanzdatum (dies beinhaltet sowohl den Kontroll- als auch den Datenfluss betreffende Daten) über seine gesamte Ausführungszeit eine feste Partition zugewiesen ist (vgl. Abschnitt 4.4), gilt dies in OSIRIS nicht. Vielmehr werden in OSIRIS Instanzdaten zwischen den einzelnen an der Ausführung einer Prozessinstanz teilnehmenden Knoten in Form des Whiteboard migriert. In Geschäftsprozessszenarien ist eine derartige Migration der Instanzdaten eines Prozesses – beispielsweise aus Gründen der Datenhoheit – nicht immer möglich. Verbunden mit der unterschiedlichen Vorgehensweise zum Instanzdatenzugriff entfällt in der PS-Infrastruktur auch die Notwendigkeit für das in OSIRIS notwendige Aufspalten und Zusammenfassen des Whiteboard zur Realisierung paralleler Ausführungspfade. Ein weiterer wesentlicher Unterschied zwischen OSIRIS und dem in dieser Arbeit verfolgten Ansatz ist das

verwendete Prozessmodell. Während sich OSIRIS auf ein eigenes Prozessmodell stützt, verfolgt der in dieser Arbeit vorgestellte Ansatz das Ziel der dezentralen Ausführung von regulären WS-BPEL-Prozessen.

2.4.2 Mentor

Middleware for Enterprise-wide Workflow Management (Mentor) [WWWD96, MWW+98] ist ein System zur Ausführung umfangreicher, unternehmensweiter Prozesse.

Das in *Mentor* verwendete Prozessmodell basiert auf *State-Charts*, welche wiederum eine erweiterte Form von Zustandsdiagrammen sind, die deren hierarchische Schachtelung erlauben und die Möglichkeit bieten, dass mehrere Zustände einer Schachtelungsebene gleichzeitig aktiv sein können (man spricht hierbei auch von *Orthogonalität*) [Har87]. Bestandteile des Prozessmodells sind sogenannte *Activities*, welche die aktiven, funktionalen Bestandteile des Prozesses repräsentieren. Der Datenfluss zwischen Aktivitäten wird durch eine sogenannte *Activity-Chart* beschrieben; Datenflusskanten sind gerichtet und mit einem Bezeichner annotiert. Die Kontrollflussabhängigkeiten zwischen den Aktivitäten der *Activity-Chart* werden durch eine *State-Chart* repräsentiert. Sie hat einen definierten Startzustand und beschreibt die Zustandsübergänge des Prozesses durch *Event-Condition-Action (ECA)* Regeln, welche – ursprünglich aus dem Bereich der Datenbanken stammend [WCD95, Pat99] – definieren, dass eine bestimmte Aktivität als Folge des Auftretens eines Ereignisses unter bestimmten Bedingungen zur Ausführung kommen soll. Eine Aktion einer ECA-Regel kann wiederum eine Aktivität ausführen, ein Ereignis auslösen oder die Parameter einer Bedingung setzen. Durch ein Partitionierungsverfahren werden die einzelnen Aktivitäten eines Prozessmodells auf die an ihrer Ausführung teilnehmenden WfMS verteilt (der Partitionierungsvorgang selbst ist in Abschnitt 2.5 beschrieben). Verlangt eine Aktion die Ausführung einer Aktivität, so wird diese auf dem lokalen WfMS ausgeführt. Spezifiziert eine Aktion hingegen einen Zustandsübergang, so wird zunächst die Ziel-Partition und deren WfMS bestimmt und danach der Zustand der Prozessinstanz an das entsprechende WfMS übergeben.

Die Architektur von Mentor folgt dem *Client-Server*-Stil [Fow03], wobei die Ausführung des Prozesses durch die Server, die Ausführung der aufgerufenen Anwendungen durch die Klienten des Systems erfolgt. Der Architektur der Server stützt sich zur Realisierung der Funktionen der Komponenten weitestgehend auf Standard-Middleware-Komponenten. Zentrale Komponente eines sogenannten *Workflow-Servers* bildet die *Workflow-Engine*-Komponente, welche die dem jeweiligen WfMS zugeordneten Aktivitäten ausführt und ggf. die Interaktion mit der zur Realisierung einer Aktivität verwendeten Anwendung auslöst. Die Interaktion mit der Anwendung geschieht dabei – aus Gründen der Unterstützung heterogener Anwendungsszenarien – unter Verwendung eines *Object Request Broker (ORB)*, dessen Funktionen als Teil der *Common Object Request Broker Architecture (CORBA)* [OMG04] spezifiziert sind und im Fall von Mentor sowohl für das Marshalling und Unmarshalling der Aufrufparameter als auch die Übermittlung des Aufrufs zwischen WfMS und Anwendungsklient verwendet werden. Ein *TP-Monitor* [GR92, Ber97] realisiert sowohl transaktionalen Zugriff auf die von den WfMS verwendeten Ressourcen als auch die Umsetzung fehlertoleranten Zugriffs auf entfernte WfMS. Durch die *Log-Manager*-Komponente wird der Zustand laufender Prozessinstanzen persistent protokolliert, wodurch nach einem Systemabsturz die Wiederherstellung eines konsistenten Instanzzustands ermöglicht wird. Die *History-Manager*-Komponente erfasst und dokumentiert fortlaufend den Ausführungszustand eines Prozesses. Da Mentor Prozesse adressiert, an deren Ausführung Menschen beteiligt sind, realisiert die *Worklist-Manager*-Komponente das Finden eines Akteurs, welcher die von der jeweiligen Aktivität geforderte Rolle implementiert. Verlangt die Ausführung eines Prozesses die Interaktion mit einem entfernten WfMS, so geschieht dies unter Verwendung der *Communication-Manager*-Komponente, die den aktuellen Zustand der Prozessinstanz in eine Synchronisationsnachricht überführt und diese unter Verwendung einer *Nachrichten-orientierten Middleware (engl. Message-oriented Middleware, MOM)* transaktional zum jeweiligen WfMS übermittelt. Dieses nimmt die Synchronisationsnachricht entgegen, aktualisiert seinen lokalen Zustand anhand der in der Synchronisationsnachricht enthaltenen Zustandsinformationen und führt die jeweilige Aktivität aus.

Sowohl das Prozessmetamodell von *Mentor* als auch die der Prozessausführung zugrunde liegende Vorgehensweise unterscheidet sich substantiell von BPEL und der in Kapitel 5 vorgestellten PS-Infrastruktur. Beispielsweise verzichtet das *Mentor*-Prozessmodell auf Mechanismen zur Fehlerbehandlung, Kompensation und Terminierung, die eine Kenntnis des Ausführungszustands anderer Ausführungsteilnehmer einer Prozessinstanz verlangen. Ähnlich OSIRIS gilt auch in *Mentor*, dass der Zustand einer Prozessinstanz zwischen den einzelnen Ausführungsteilnehmern migriert wird.

2.4.3 Process Splitting

In [KL06, Kha08] wird ein Verfahren zur Fragmentierung von BPEL-Prozessen und eine Laufzeitumgebung zu deren verteilter Ausführung vorgestellt. Das der Arbeit zugrunde liegende Szenario ist das sogenannte *Process-Outsourcing*, in welchem ein aus zentraler Sicht modellierter Prozess verteilt ausgeführt werden soll, wobei das Verhalten der Gesamtheit der Fragmente funktional der Ausführung des ursprünglichen BPEL-Prozesses entsprechen soll.

Ein wesentliches Ziel des Ansatzes ist dabei die möglichst weitgehende Wiederverwendung existierender Middleware zur Ausführung der Prozessfragmente durch die einzelnen Teilnehmer der Prozessausführung. Das Verfahren stützt sich daher (ähnlich [NCS04, CCMN05], vgl. Abschnitt 2.4.4) auf BPEL selbst als Mittel zur Beschreibung der einzelnen Fragmente eines Prozesses und auf erweiterte BPEL-WfMS zu deren Ausführung. Die Interaktionen zwischen den einzelnen WfMS, die notwendig sind, um ein zur zentralen Ausführung semantisch äquivalentes Laufzeitverhalten zu erreichen, werden in diesem Ansatz selbst wiederum auf BPEL-Prozessfragmente abgebildet. So erfolgt beispielsweise die Weitergabe der Kontrolle über die Ausführung einer Prozessinstanz durch eine INVOKE-Aktivität auf der Seite desjenigen Ausführungsteilnehmers, der die Kontrolle abgibt, und durch eine RECEIVE-Aktivität auf Seite des Ausführungsteilnehmers, der die Kontrolle über die Prozessausführung erhält. Bedingt der Start der Ausführung eines Teilnehmers eine Abgabe der Kontrolle mehrerer anderer Teilnehmer, so wird dies durch sogenannte *Receiving-Flows* auf Seite des empfangenden Teilnehmers abgebildet.

Da die verteilte Ausführung eines Prozesses allerdings neben der korrekten Realisierung des Prozesskontrollflusses auch die Kommunikation der von den jeweiligen Ausführungsteilnehmern benötigten Instanzdaten erfordert, werden diese als Teil der Weitergabe der Kontrolle über die Prozessausführung zwischen den einzelnen Ausführungsteilnehmern übermittelt. Da BPEL selbst eine rein kontrollflussgetriebene Sprache ist und dementsprechend den Datenfluss nicht explizit als Teil der Prozessmodelle spezifiziert, stützt sich der Ansatz zur Bestimmung der Datenabhängigkeiten zwischen den einzelnen Fragmenten eines Prozesses auf *BPEL-D* [Kha07]. BPEL-D ist eine Variante von BPEL, welche die explizite Beschreibung von Datenabhängigkeiten zwischen Aktivitäten erlaubt. Ein Verfahren zur automatischen Bestimmung dieser Datenabhängigkeiten anhand eines BPEL-Prozesses wird in [KKL08] vorgestellt.

Einige Sprachkonstrukte von BPEL und deren Eigenschaften, wie z. B. Schleifen oder die Mechanismen zur Behandlung von Fehlern bzw. deren Kompensation, lassen sich jedoch nicht rein durch die Dekomposition eines BPEL-Prozesses in BPEL-Prozessfragmente erreichen, da die Ausführung dieser Aktivitäten Kenntnis des globalen Zustands der Prozessinstanz erfordert. Die Bereitstellung dieses Zustands erfolgt im vorgestellten Ansatz durch den sogenannten *Split-Activity-Controller* (vergleichbar der *Monitor*-Komponente in [CCMN05]).

Die Architektur des Systems umfasst im Wesentlichen drei Komponenten: (i) eine zur Prozess-Deployment-Zeit eingesetzte Komponente zur Datenflussanalyse, (ii) ein ebenfalls zur Deployment-Zeit operierendes Werkzeug zur Erzeugung der erweiterten Prozessfragmente anhand der manuellen Nutzervorgaben und (iii) die Laufzeitinfrastruktur, bestehend aus einem WfMS zur Ausführung von BPEL, welches zur Unterstützung des Koordinationsprotokolls zwischen den verteilten WfMS um eine WfMS-externe *Split-Activity-Controller* Komponente auf Basis von *WS-Coordination* [Org09] erweitert wurde. Die Interaktion zwischen WfMS und der *Controller*-Komponente erfolgt bidirektional durch die Kommunikation von WfMS-Events über eine MOM. Für die Anbindung der Controller-Komponente an das WfMS ist in der Regel eine Erweiterung des WfMS notwendig, die zum einen die Zustandsübergänge der Aktivitäten von Prozessinstanzen an den Controller kommuniziert, zum anderen beispielsweise das Anhalten laufender Prozessinstanzen durch den

Controller ermöglicht.

Der Ansatz ähnelt in seinen Zielen der PS-Infrastruktur; beide Arbeiten haben eine verteilte Ausführung von BPEL-Prozessen zum Ziel, unterscheiden sich allerdings in ihrer Vorgehensweise. In der vorgestellten Arbeit wird die verteilte Koordination der Ausführung von Prozessen selbst wiederum auf BPEL-Prozessfragmente abgebildet, die – mit dem Ziel einer möglichst weitgehenden Wiederverwendung existierender Technologien – in einem erweiterten BPEL-WfMS ausgeführt werden. Die aus dieser Vorgehensweise resultierenden Einschränkungen sind die Notwendigkeit (i) eines zentralen Koordinators für die Ausführung bestimmter BPEL-Aktivitäten sowie (ii) die Analyse bzw. Spezifikation des Datenaustauschs zwischen den einzelnen Ausführungsteilnehmern. Darüber hinaus erfolgt die Vorgang der Bestimmung der Prozessfragmente im vorgestellten Ansatz auf Grundlage manueller Nutzervorgaben; in der PS-Infrastruktur ist zusätzlich zur manuellen Vorgabe auch die automatische Bestimmung einer Prozesspartitionierung möglich (vgl. Kapitel 4).

2.4.4 Distributed BPEL4j

In [CCMN04, CCMN05] wird ein verteiltes WfMS zur Ausführung von BPEL-Prozessen auf Basis von *BPWS4j*[1] vorgestellt.

Die dem Ansatz zugrunde liegende Vorgehensweise ist die automatisierte Dekomposition eines BPEL-Prozesses in eine Menge von Prozessfragmente, die selbst wieder vollständige BPEL-Prozesse sind. Die Synchronisation der Prozessfragmente zur Ausführungszeit erfolgt ähnlich dem in Abschnitt 2.4.3 beschriebenen Verfahren durch zusätzliche INVOKE- und RECEIVE-Aktivitäten, die während der Dekomposition des Prozesses in die einzelnen Prozessfragmente eingebaut werden.

Die Navigation der einzelnen Prozessinstanzen erfolgt – bedingt durch die Realisierung der Synchronisation der an der Ausführung teilnehmenden Partner durch reguläre BPEL-Aktivitäten – weitgehend analog zur zentralen Ausführung eines BPEL-Prozesses. Zur Realisierung von Fault-, Compensation- und Termination-Handling aggregiert eine zentrale *Status-Monitor*-Komponente

[1] http://www.alphaworks.ibm.com/tech/bpws4j

den Zustand sämtlicher aktuell in Ausführung befindlicher Fragmente eines Prozesses. Bedient wird der globale *Status-Monitor* durch sogenannte *Local-Monitoring-Agents* der einzelnen WfMS, die den Zustand ihrer lokalen Prozessausführung (sowohl periodisch als auch bedingt durch beispielsweise das Auftreten eines Fehlers) an den *Status-Monitor* weiterleiten. Dieser kann daraufhin globale Aktionen, wie beispielsweise die Terminierung sämtlicher in Ausführung befindlicher Fragmente eines Prozesses, veranlassen.

Der Ansatz ist in seiner Vorgehensweise ähnlich dem in Abschnitt 2.4.3 vorgestellten Verfahren; die dort genannten Unterschiede gelten hier analog.

2.4.5 Pervasive Workflow Execution using BPEL

In [MM05] wird eine verteilte Ausführungsumgebung von Prozessen in *Pervasive-Computing*-Szenarien vorgestellt, in welchen eine Prozessausführung, gesteuert durch ein zentrales WfMS, nicht sinnvoll möglich ist. Grundgedanke der verteilten Ausführung eines Prozesses in diesem Ansatz ist die Definition des Prozesses mit den Mitteln eines relativ einfachen Prozessmetamodells und die Transformation des Prozessmodells in eine Menge einzelner BPEL-Prozessfragmente, die sowohl die Interaktion mit den orchestrierten Diensten als auch die Koordination der einzelnen Teilnehmer der Prozessausführung steuern; in dieser Hinsicht ähnelt das Verfahren den in Abschnitt 2.4.3 und 2.4.4 vorgestellten Ansätzen.

Das System orientiert sich an zwei Kernanforderungen, diese sind: (i) *Unterstützung von verteilter Kontrolle*, wonach die Koordination der Ausführung des Prozesses nicht durch einen Teilnehmer seiner Ausführung bestimmt wird, sondern durch Kollaboration sämtlicher Ausführungsteilnehmer erreicht wird. (ii) *Dynamische Zuweisung von Verarbeitungsschritten*, wonach der Teilnehmer der Ausführung eines Prozesses, der einen bestimmten Verarbeitungsschritt ausführt, erst zur Ausführungszeit des Prozesses bestimmt wird.

Das verwendete Prozessmetamodell repräsentiert Prozesse als gerichtete Graphen, bestehend aus den folgenden Aktivitäten: (i) Die *Task*-Aktivität repräsentiert einen durch einen Teilnehmer der Prozess-Ausführung durchzuführenden Verarbeitungsschritt. (ii) Die *Routing*-Aktivität beschreibt die Weitergabe der

Kontrolle über die Prozessausführung an einen anderen Ausführungsteilnehmer. (iii) Sogenannte *Process-Control-Activities* erlauben die Definition von *sequentieller*, *paralleler* oder *konditionaler* Ordnung von *Task*- und *Routing*-Aktivitäten.

Jeder Teilnehmer der Prozessausführung betreibt eine Instanz des verteilten WfMS. Die von den einzelnen Teilnehmern ausgeführten Verarbeitungsschritte sind (i) die Verarbeitung "eingehender" *Routing*-Aktivitäten, (ii) die Ausführung der für den jeweiligen Teilnehmer spezifizierten *Task*-Aktivitäten und (iii) die Verarbeitung "ausgehender" *Routing*-Aktivitäten. Die Verarbeitung der einzelnen Schritte erfolgt durch jeweils zwei BPEL-Prozesse bei jedem Partner. Ein sogenannter *Public-Process* implementiert die Logik der *Routing*-Aktivitäten und bietet damit die Schnittstelle, die der jeweilige Ausführungsteilnehmer anderen Ausführungsteilnehmern anbietet. Die Ausführung der einzelnen *Tasks*, unter Verwendung von durch den jeweiligen Teilnehmer angebotenen Diensten, erfolgt durch den sogenannten *Private-Process*. Um den hohen Anforderungen eines *Pervasive-Computing*-Szenarios hinsichtlich der Dynamik bekannter potentieller Ausführungsteilnehmer gerecht zu werden, stützt sich der Ansatz auf Mechanismen zum dynamischen Finden möglicher Implementierungen eines *Tasks*, d. h. desjenigen Teilnehmers der Prozessausführung, an den die Kontrolle über die Prozessausführung übergeben werden soll, unter Verwendung von *OWL-S* [MBH+04] zur Instanzlaufzeit.

Aufgrund des Szenarios, das der vorgestellten Arbeit zugrunde liegt, unterscheiden sich sowohl die Methode zur Bestimmung der Teilnehmer der Ausführung eines Prozesses, die einen bestimmten *Task* bzw. eine Aktivität realisieren als auch das zur Ausführung verwendete Verfahren deutlich zu den in dieser Dissertation vorgestellten. Während in [MM05] eine dynamische Bestimmung der Ausführungsteilnehmer anhand semantischer Dienstbeschreibungen zur Laufzeit erfolgt, erfolgt in der PS-Infrastruktur die Partitionierung eines Prozesses auf seine Ausführungsumgebung – aufgrund der Anforderungen des zugrunde liegenden Einsatzszenarios (vgl. Abschnitt 4.1) – zur Deploymentzeit. Ähnlich den in Abschnitten 2.4.3 und 2.4.4 vorgestellten Verfahren stützt sich der Ansatz auf BPEL als Sprache zur Beschreibung einzelner Prozesse, deren Verbund die verteilte Prozessausführung realisiert. Die Navigation der einzelnen Teilprozesse erfolgt auch hier unter Verwendung eines "regulären"

zentralen BPEL-WfMS.

2.4.6 DECS: Decentralized Coordination of Web-Services

In [WPSW05] wird das System *DECS* zur verteilten Prozessausführung vorgestellt. Grundlage für die Beschreibung von Prozessen in *DECS* ist ein proprietäres Prozessmetamodell, dessen Eigenschaften, im Gegensatz zu beispielsweise BPEL, derart gewählt wurden, dass eine verteilte Ausführung begünstigt wird. Ein wesentliches Design-Kriterium des Prozessmetamodells ist die Minimierung der Notwendigkeit globalen Zustands, auf welchen die Teilnehmer der Ausführung einer Prozessinstanz gleichzeitigen Zugriff benötigen, um die Übergabe der Kontrolle über die Prozessausführung zwischen den Teilnehmern soweit wie möglich zu vereinfachen. Weiterhin stützt sich *DECS* zur Beschreibung von Abhängigkeiten zwischen Aktivitäten weitgehend auf den Datenfluss eines Prozesses; hierdurch bedingt entfällt die beispielsweise in [Kha08] notwendige Bestimmung der Datenabhängigkeiten in einer kontrollflussgetriebenen Sprache durch eine Analyse der Prozessmodelle.

Das Prozessmetamodell umfasst drei unterschiedliche *Tasks*: (i) *Request-Response-Tasks*, welche den synchronen Aufruf eines Web-Service erlauben; Ein- und Ausgaben eines Tasks dieses Typs sind durch explizite Datenabhängigkeiten an andere Teile des Workflows gebunden. (ii) *Send-Tasks* erlauben die asynchrone Interaktion mit einem Web-Service durch die Übermittlung einer Anfragenachricht. (iii) *Receive-Tasks* sind das Gegenstück zu *Send-Tasks* und erlauben den Empfang und die Verarbeitung eingehender Daten. Verbunden werden die einzelnen Tasks eines Prozesses durch sogenannte Abhängigkeiten (engl. *Dependencies*), wobei zwei Arten von Abhängigkeiten unterschieden werden: (i) Sogenannte zeitliche (Kontroll-) Abhängigkeiten erlauben die Spezifikation einer Vorgänger-Nachfolger-Relation von Aktivitäten, (ii) Datenabhängigkeiten erlauben es zu spezifizieren, dass ein Task für seine Ausführung ein Resultat eines anderen Tasks benötigt. Das *DECS*-Prozessmetamodell sieht weiterhin Möglichkeiten zur Flexibilisierung der Prozessmodelle vor. So ist es beispielsweise möglich, für einen Task mehrere sogenannte *Input-* bzw. *Output Sources* zu spezifizieren, d. h. die für die Ausführung eines Tasks notwendigen

Daten im Fall der Nicht-Verfügbarkeit einer Datenquelle aus einer anderen zu beziehen bzw. in diese zu schreiben.

Die Komponenten der Architektur des *DECS*-WfMS können drei unterschiedlichen Bereichen zugeordnet werden: (i) *Messaging*, (ii) *Data-Processing* und (iii) *Persistence*. Die einzelnen Komponenten des Systems sind als J2EE-Anwendungen realisiert, die untereinander durch den Austausch von Nachrichten über JMS-Queues kommunizieren. Die *Messaging*-Komponenten des Systems sind für die externe Kommunikation des WfMS (d. h. sowohl die Kommunikation zwischen WfMS und Web-Services als auch die Kommunikation zwischen WfMS) verantwortlich; die Realisierung der Kommunikation erfolgt unter Verwendung der Web-Service-Standards SOAP, WSDL und WS-Addressing. Kernpunkt der *Data-Processing*-Komponenten bildet der sogenannte *Coordinator* (welcher die Rolle des Navigators in einem klassischen WfMS implementiert). Dieser verarbeitet die beim WfMS eingehenden Nachrichten und prüft dabei, ob durch die in der eingehenden Nachricht enthaltenen Daten die Datenabhängigkeiten eines Tasks erfüllt werden können. Ist dies der Fall, so löst der Navigator eine Interaktion mit dem jeweiligen Web-Service aus. Nach dem erfolgten Dienstaufruf bestimmt der Navigator anhand der im Prozessmodell spezifizierten Daten- und Kontrollkanten, an welche WfMS die Kontrolle über die Ausführung des Prozesses weitergegeben werden soll und löst diese aus. Zugriff, sowohl auf installierte Prozessmodelle als auch auf in Ausführung befindliche Prozessinstanzen, erfolgt durch die *Persistence*-Komponenten *Process-Definition-Repository* bzw. *Process-Instance-Repository*.

Wesentliches Unterscheidungsmerkmal zwischen *DECS* und der PS-Infrastruktur ist das verwendete Prozessmetamodell, das in *DECS* explizit mit dem Ziel der Vermeidung globalen Zustands unter Verzicht auf BPEL-Konstrukte wie z. B. SCOPE-Aktivitäten oder Fault- und Compensation-Handler definiert wurde und die hieraus resultierende einfachere Prozessausführung.

2.4.7 Zusammenfassung

In den vorangehenden Abschnitten wurde eine Übersicht über einige existierende Ansätze zur verteilten Prozessausführung gegeben.

Die vorgestellten Ansätze unterscheiden sich teilweise deutlich hinsichtlich ihrer Vorgehensweise zur Prozessausführung. Während beispielsweise *OSIRIS* und *Mentor* eigene Prozessmetamodelle und entsprechende Ausführungsumgebungen definieren, stützen sich [Kha08], [CCMN04] und [MM05] auf BPEL als Prozessmetamodell und auf (erweiterte) bestehende WfMS-Implementierungen.

Ein generelles, in allen Arbeiten erkanntes Problem, ist die Notwendigkeit sogenannten *globalen Zustands*. Das sind Informationen, die zwischen allen (oder zumindest vielen) Teilnehmern der Ausführung eines Prozesses ausgetauscht werden müssen. Arbeiten wie [WPSW05] oder [MM05] adressieren dieses Problem durch eine Vereinfachung ihrer Prozessmodelle und eine damit verbundene Eliminierung globalen Zustands; in [Kha08] und [CCMN04] wird der globale Zustand in einem zentralen *Coordinator* bzw. einem *Status-Monitor* vorgehalten.

Ein spezielles Problem der verteilten Ausführung kontrollflussgetriebener Sprachen ist die Weitergabe der Instanzdaten eines Prozesses bzw. die Bestimmung, welche Instanzdaten zwischen welchen Ausführungsteilnehmern kommuniziert werden müssen. Dies wird in den Arbeiten entweder durch die Verwendung eines datenflussgetriebenen Prozessmetamodells (z. B. [WPSW05]), oder durch eine Datenflussanalyse kontrollflussgetriebener Prozesse (z. B. [Kha08]) erreicht.

2.5 Prozesspartitionierung

Ebenso wie sich die generelle Vorgehensweise der Prozessausführung und die Architekturen der jeweiligen WfMS abhängig von den Anforderungen des jeweiligen Einsatzszenarios unterscheiden, sind auch die Verfahren zur Zerlegung eines Prozesses in einzelne Prozessteile – sogenannte *Partitionen* – charakterisiert durch jeweils unterschiedliche Freiheitsgrade bzw. Einschränkungen. Um das in dieser Dissertation entwickelte Partitionierungsverfahren für BPEL-Prozesse in Kontext zu existierenden Arbeiten zu setzen, werden in der Folge einige relevante Arbeiten aus diesem und angrenzenden Forschungsbereichen vorgestellt. Dabei werden teilweise auch die Partitionierungsverfahren der-

jenigen verteilten WfMS vorgestellt, deren Architektur im vorigen Abschnitt erläutert wurde.

2.5.1 Mentor

Die Prozesspartitionierung in *Mentor* [WW97] umfasst zwei Phasen, in denen zunächst die *Activity-Chart* und danach die *State-Chart* partitioniert werden (vgl. Abschnitt 2.4.2). Basis der Partitionierung der *Activity-Chart* bildet die Annahme, dass jeder Aktivität eines Prozesses eine Person oder eine Rolle zugeordnet ist, die die jeweilige Aktivität implementiert. Sämtliche Aktivitäten der *Activity-Chart*, die derselben Person oder Rolle zugeordnet sind, werden derselben Partition zugeordnet. Bei der Partitionierung der *Activity-Chart* in Mentor handelt es sich also um einen rein *regelbasierten Ansatz*.

Die Partitionierung der *State-Chart* ist ebenfalls ein mehrstufiges Verfahren. In einem ersten Schritt wird jeder State einer Activity der *Activity-Chart* zugewiesen. In einem zweiten Schritt erfolgt das Überführen jedes Zustands der *State-Chart* in einen orthogonalen Zustand, bestehend aus Eingangs- und Ausgangszuständen und entsprechenden Bedingungen an den Zustandsübergängen, so dass die Semantik der ursprünglichen Aktivitäten erhalten bleibt. In einem letzten, dritten Schritt erfolgt die Dekomposition der erzeugten orthogonalisierten *State-Chart* in Partitionen entsprechend der implementierenden Rollen bzw. Personen der diesen zugeordneten Aktivitäten. Die so erzeugten Partitionen werden dann auf den an der Ausführung des Prozesses teilnehmenden WfMS installiert; die Prozesspartitionierung findet also hier vor dem Prozess-Deployment statt.

2.5.2 Adept$^{\text{Distribution}}$

Ähnlich *Mentor* ist Adept$^{\text{Distribution}}$ [BD00] ein WfMS, das die Ausführung von Prozessen mit einer großen Anzahl von Aktivitäten und Klienten zum Ziel hat. Das System strebt dabei eine möglichst gleichmäßige Auslastung seiner WfMS-Server (sowie auch der von dem WfMS verwendete Netzwerkkommunikationsinfrastruktur, wie z. B. Gateways und Router) an.

Die Definition eines Prozesses in $Adept^{Distribution}$ erfolgt durch sogenannte *Workflow-Vorlagen*, welche die Ausführungsreihenfolge der Aktivitäten des Prozesses bzw. der Programme spezifizieren, die durch die jeweilige Aktivität repräsentiert werden. Die Workflow-Vorlagen sind blockorientiert und hierarchisch. Zur Beschreibung der Abhängigkeiten zwischen Aktivitäten – sogenannten *Tasks*– kommen in Adept$^{\text{Distribution}}$ die Kontrollkonstrukte SEQUENZ (entspricht einer SEQUENCE-Aktivität in BPEL), PARALLELE VERZWEIGUNG (entspricht einem FLOW ohne Synchronisationskanten), BEDINGTE VERZWEIGUNG (entspricht einer IF-Aktivität in BPEL), SCHLEIFE (entspricht REPEAT-Aktivität) und eine Synchronisationskante (entspricht einem LINK in einem FLOW) zum Einsatz.

Während ihrer Ausführung durchläuft eine Aktivität mehrere Zustände. Ist eine Aktivität noch nicht zur Verarbeitung durch das WfMS bereit, so befindet sie sich im Zustand NOT_ACTIVATED. Haben alle eingehenden Synchronisationskanten einer Aktivität einen positiven Status TRUE_SIGNALED, so wechselt die Aktivität in den Zustand ACTIVATED. Abhängig davon, ob eine Aktivität automatisch, d. h. durch das WfMS selbst, oder durch einen (menschlichen) Benutzer des Systems ausgeführt wird, wechselt diese entweder direkt in den Zustand RUNNING oder muss, signalisiert durch den Zustand SELECTED, zunächst durch einen Benutzer zur Bearbeitung ausgewählt werden. Der Lebenszyklus einer Aktivität endet, je nach Status der Aktivitätsausführung, entweder positiv im Zustand COMPLETED oder negativ im Zustand FAILED. Kommt eine Aktivität (beispielsweise aufgrund bedingter Aktivitätsausführung) nicht zur Ausführung, so wechselt ihr Status direkt in den Endzustand SKIPPED.

Kriterium für die Verteilung der Tasks, die im Rahmen der Ausführung eines Prozesses zu verarbeiten sind, ist eine Maximierung der Lokalität der Tasks zu den ausführenden Bearbeitern. Der hinsichtlich dieses Kriteriums optimale Ausführungsort eines Tasks ergibt sich damit als derjenige WfMS-Server, der derselben Netzwerk-Domäne wie der ausführende Bearbeiter zugeordnet ist. Die Wahl dieses Optimierungskriteriums in Adept$^{\text{Distribution}}$ begründet sich durch das Wegfallen der Notwendigkeit entfernter Kommunikation zwischen Bearbeiter einer Aufgabe und dem jeweiligen WfMS-Server, was in der Folge auch zu einer Entlastung der Netzwerkinfrastrukturkomponenten zwischen

Bearbeiter und WfMS-Server führt. Ist eine direkte Verarbeitung einer Aufgabe auf einem der Domäne des Nutzers zugeordneten WfMS-Server (beispielsweise aus Gründen der Auslastung des lokalen WfMS-Servers) nicht möglich, so ist das Ziel der Partitionierung die Erhaltung einer möglichst starken Lokalität (beispielsweise zur Vermeidung "teurer" WAN-Kommunikation aus Gründen geringerer Bandbreite, geringerer Zuverlässigkeit und höherer Latenz im Vergleich zu einer Kommunikation im selben LAN).

Die Partitionierung selbst erfolgt in Adept$^{\text{Distribution}}$ auf Basis eines Kostenmodells. Dieses stützt sich unter anderem auf die Ein- und Ausgabeparameter der Aktivitätsimplementierungen, die Kosten der Aktualisierung von Arbeitslisten sowie der Migration von Workflow-Instanzen und der Kommunikation von Aktivitätsimplementierungen mit externen Datenquellen.

Der eigentliche Partitionierungsvorgang bewertet (auf Basis eines Greedy-Ansatzes) mögliche Partitionierungen eines Prozesses auf die einzelnen Server auf Grundlage des Kostenmodells. Ziel des verwendeten Verfahrens ist die Bestimmung einer "optimalen" Partitionierung für jede einzelne Aktivität eines Prozesses in einem ersten Schritt und daraufhin die Zusammenfassung mehrerer Ein-Aktivitäten-Partitionen. Dabei gilt, dass eine Partition "ein zusammenhängender Teilgraph des Kontrollflussgraphen [eines Prozesses]" [Bau01] ist. Dieses Verfahren wird in Adept$^{\text{Distribution}}$ als sogenannte *statische Serverzuordnung* bezeichnet. Für Szenarien, in denen eine Vorabbestimmung der Partitionierung eines Prozesses aufgrund dynamischer Einflussgrößen während der Instanzausführung nicht möglich ist, werden während der Prozesspartitionierung Regeln bestimmt, die, zur Instanzlaufzeit evaluiert, bestimmen, auf welche WfMS-Server eine Instanz unter bestimmten Bedingungen migriert werden soll.

2.5.3 Distributed BPEL4j

In dem in [NCS04] vorgestellten Verfahren wird die automatische Partitionierung eines BPEL-Prozesses durch ein Verfahren realisiert, welches aus dem Forschungsbereich der Optimierung von Programmen für parallele Ausführung stammt. Das Verfahren unterscheidet dabei zwischen fixen (engl. *fixed*) und

portablen (engl. *portable*) Knoten, für welche jeweils mit unterschiedlichen Verfahren eine Ausführungspartition bestimmt wird. Fixe Aktivitäten sind: (i) RECEIVE- und REPLY-Paare, die auf dem WfMS-Server installiert werden, der von einem Nutzer kontaktiert werden muss, um die Erzeugung einer neuen Prozessinstanz auszulösen. (ii) INVOKE-Aktivitäten, die immer beim Anbieter des jeweiligen Diensts ausgeführt werden. Der Fall mehrerer verfügbarer, funktional äquivalenter, Implementierungen eines bestimmten Diensts wird in diesem Ansatz nicht betrachtet.

Ziel der Platzierung portabler Aktivitäten ist die Minimierung der Kommunikation zwischen den an der Prozessausführung teilnehmenden Partnern. Das für die Partitionierung von portablen Knoten verwendete Verfahren stützt sich auf eine Analyse des Prozesses, welche das Prozessmodell traversiert und anhand des Kontrollflusses des Prozesses ein sogenannter *Threaded Control Flow Graph (TCFG)* [Kri98] erstellt. Der TCFG wird um die, durch eine Datenflussanalyse gewonnenen Informationen ergänzt und in einen *Program Dependence Graph (PDG)* [FOW87] überführt. Der PDG dient als Grundlage für den eigentlichen Vorgang der Partitionierung der portablen Knoten; durch Umsortierung und Zusammenfassung von Knoten im PDG des Prozesses ist hier das Ziel, die Anzahl der Datenkanten zwischen zwei Knoten des PDG zu minimieren.

2.5.4 Optimale Stratifizierung von Transaktionen

Stratifizierte Transaktionen [Ley97, LR99] sind ein Mittel zur Aufspaltung einer (umfangreichen) globalen Transaktion in einzelne sogenannte Strata unter Beibehaltung wesentlicher transaktionaler Eigenschaften der globalen Transaktion und gleichzeitiger Verbesserung des Laufzeitverhaltens. In Abschnitt 5.5.1 wird das Modell stratifizierter Transaktionen näher erläutert.

In komplexen Szenarien ist die Bestimmung einer Stratifikation einer Gesamttransaktion mit optimalem Laufzeitverhalten aufwändig. [Dan08, DKL09] stellt unterschiedliche Verfahren zur automatischen, optimalen Stratifikation von Transaktionen vor. Ziel der Verteilung ist die Minimierung der Kommunikation zwischen Strata und die damit verbundene Verbesserung des Laufzeitverhaltens der durch die Teil-Transaktionen realisierten globalen Transaktion. Das Problem

der Transaktionsstratifizierung wird in der vorgestellten Arbeit auf ein kombinatorisches Optimierungsproblem zurückgeführt. Als Grundlage der Lösung des Optimierungsproblems wird eine Kostenfunktion definiert, die auf Basis unterschiedlicher Parameter – wie beispielsweise erwarteter Ausführungsdauer einzelner Transaktionen, erwartete Wahrscheinlichkeitswerte für *Abort* und *Commit* von Transaktionen und den Kosten für eine eventuelle Transaktions-Recovery – die Bewertung einer möglichen Lösung des kombinatorischen Optimierungsproblems erlaubt. Unter Verwendung dieser Kostenfunktion werden in der Arbeit unterschiedliche Verfahren zur Lösung des Optimierungsproblems hinsichtlich ihres Laufzeitverhaltens und der Kosten der erzeugten Stratifizierung evaluiert. Die in der Arbeit vorgestellten Verfahren sind *Evolutionäre Algorithmen*, *Hill-Climbing* [RN02] und *Simulated Annealing* [KGV83] .

2.5.5 Continuation Passing

Im Gegensatz zu den bisher vorgestellten Verfahren zur Prozesspartitionierung, in denen die (Erst-) Partitionierung vor dem Prozess-Deployment-Zeitpunkt bestimmt wird, erfolgt die Bestimmung der Partitionierung in [YY07] zur Ausführungszeit des Prozesses; eine Vorabpartitionierung findet hier nicht statt. Das Verfahren ist auf offene, hochgradig dynamische Ad-Hoc-Umgebungen ausgelegt, die sich durch eine hohe Änderungsfrequenz der zur Verfügung stehenden Ressourcen auszeichnen. Damit verbunden ist ein Erlangen globalen Wissens über die aktuell verfügbaren Ressourcen nicht möglich. In diesem Verfahren beginnt die Ausführung des Prozesses auf einem Knoten des Systems. Nach Ausführung eines Navigationsschritts des Prozesses evaluiert der Knoten eine Migration der Prozessinstanz auf einen ihm bekannten “benachbarten” Knoten. Wurde ein Kandidat für die Migration identifiziert, so erfolgt die Migration zu diesem durch Weiterreichen des kompletten Ausführungszustands der Instanz in Form einer sogenannten *Continuation*. Diese bezeichnet ein Objekt, welches den aktuellen Ausführungszustand eines Programms beschreibt und beinhaltet sowohl Informationen über die aktuelle Position der Ausführung des Programms als auch den aktuellen Zustand der vom Programm verwendeten Daten, z. B. Variablen. Dieser Ausführungszustand kann persistiert (ggf.

weitergegeben) und zu einem späteren Zeitpunkt wieder reaktiviert, d. h. die Ausführung des Programms fortgesetzt werden.

2.5.6 Zusammenfassung

In den vorangehenden Abschnitten wurden unterschiedliche Ansätze und Verfahrensweisen zur Partitionierung von Prozessen unterschiedlicher Prozessmetamodelle vorgestellt. Die vorgestellten Ansätze unterscheiden sich wesentlich in *Zeitpunkt*, *Granularität* und dem zur Partitionierung verwendeten *Verfahren*.

Der überwiegende Teil der vorgestellten Verfahren verfolgt eine *A-Priori*-Strategie zur Partitionierung; d. h. die Partitionierung eines Prozesses wird bereits vor dem Deployment des Prozesses auf seine Laufzeitumgebung bestimmt. Ausnahmen bilden Adept$^{\text{Distribution}}$ [Bau01, BD00] und der in [YY07] vorgestellte *Continuation-Passing*-Ansatz, bei denen eine Adaption der Partitionierung dynamisch zur Laufzeit erfolgen kann oder der "nächste" Ausführungsteilnehmer gar erst zur Laufzeit bestimmt wird.

Auch die, durch die vorgestellten Verfahren erreichte, Partitionierungsgranularität umfasst ein breites Spektrum; die Granularität einer Partitionierung bezeichnet dabei die Beschaffenheit der Teile eines Prozesses, die während der Partitionierung auf die einzelnen Ausführungsteilnehmer verteilt werden können. Während die beispielsweise in [Kha08] und [CCMN04] erzeugten Partitionierungen vollständige BPEL-Prozesse sind, werden in OSIRIS und Adept$^{\text{Distribution}}$ einzelne Aktivitäten des Prozesses über die einzelnen Partner verteilt.

Darüber hinaus unterscheiden sich die verschiedenen Prozesspartitionierungsverfahren in ihrer Wahl der zur Bestimmung der Partitionen verwendeten Methode. Die vorgestellten Verfahren lassen sich dabei hinsichtlich des verwendeten Partitionierungsansatzes klassifizieren in einerseits *a-priori*, *dynamische* und *rein-dynamische* sowie andererseits in *kombinatorische*, *Programmanalyse-gestützte* und *manuelle* bzw. *regelbasierte* Verfahren. Die Verfahren der Klasse der kombinatorischen Partitionierungsverfahren reduzieren das Prozesspartitionierungsproblem auf das Problem der Graphpartitionierung, welches zur Klasse der kombinatorischen Optimierungsprobleme gehört. Sie definieren

eine Funktion zur Bestimmung eines Kostenwerts einer Lösung des Partitionierungsproblems und verwenden etablierte Strategien (z. B. *Hill-Climbing*) zur Lösung des Optimierungsproblems. Kombinatorische Verfahren verfolgen im Allgemeinen eine *a-priori*-Strategie zur Partitionierung, in welcher zumindest eine Erstpartitionierung vor der Laufzeit eines Prozesses erstellt wird (z. B. [DKL09]). Einige Verfahren, z. B. [Bau01], erlauben zur Laufzeit eine *dynamische* Anpassung der Partitionierung. Demgegenüber kommen in *Programmanalyse-gestützten* Partitionierungsverfahren (z. B. das in [NCS04] vorgestellte *a-priori*-Verfahren) aus dem Compilerbau bekannte Verfahren zur Analyse von Daten- und Kontrollfluss zum Einsatz. In *rein-dynamischen* Verfahren (z. B. [YY07]), welche zumeist in hochgradig dynamischen Systemen wie Netzwerken von mobilen Endgeräten zum Einsatz kommen, erfolgt keine Vorabpartitionierung, sondern zur Laufzeit einzelner Prozessinstanzen wird, abhängig von aktuell verfügbaren "Nachbarn", eine Migration der kompletten Prozessinstanz vollzogen. Ein Problem dieser Verfahren ist eine schlechte *Prognostizierbarkeit* des Laufzeitverhaltens einzelner Prozessinstanzen, da dieses stark von unterschiedlichen Prozess-externen Faktoren (wie z. B. die aktuelle Verfügbarkeit und Auslastung "benachbarter" WfMS) beeinflusst wird. In *manuellen* Partitionierungsverfahren (wie z. B. [Kha08]) erfolgt die Partitionierung des Prozesses manuell durch einen Benutzer, welcher die Partitionen der zu partitionierenden Objekte statisch vorgibt. *Regelbasierte* Partitionierungsverfahren (z. B. [MWW+98]) sind eine Erweiterung manueller Verfahren, welche die Partitionen einzelner Objekte anhand vorgegebener Regeln (z. B. Zugehörigkeit zu bestimmten Organisationseinheiten) festlegen.

2.6 Tuplespaces

Die konzeptuelle Grundlage für die *Tuplespace*-Technologie bildet das sogenannte *Linda*-Modell [Gel85], welches vor dem Hintergrund der Vereinfachung komplizierter und fehleranfälliger Verfahren zur Inter-Prozess-Kommunikation und -Synchronisation (beispielsweise durch Shared-Memory und Semaphore [Tan01]) entwickelt wurde. Das Kernkonzept der Technologie ist der sogenannte *Tuplespace*, ein von den Teilnehmern einer Interaktion gemeinsam genutzter,

ungeordneter Raum, über den Informationen ausgetauscht werden können. Linda definiert drei grundlegende Operationen auf einem Tuplespace.

Mittels der *out*-Operation können Klienten eines Tuplespace Daten in Form sogenannter Tupel im Tuplespace ablegen. Die über einen Tuplespace ausgetauschten Daten folgen immer der Struktur einer geordneten Liste typisierter Felder, wobei konzeptuell keine Einschränkung hinsichtlich deren Typisierung bestehen. Durch die *in*- und *rd*-Operationen können geschriebene Daten von Klienten aus dem Tuplespace konsumiert werden. Dabei wird mit der *in*-Operation sowohl eine destruktive (also ein "Entnehmen") als auch mit *rd* eine nichtdestruktive Lese-Operation angeboten. Operationen zur Aktualisierung von Tupeln im Tuplespace sind nicht Teil von Gelernters Linda-Schnittstelle. Eine Aktualisierung eines Tupels wird durch die Kombination aus destruktivem Lesen, Aktualisierung auf Seite des Klienten und anschließendem Zurückschreiben des Tupels realisiert.

Das Konsumieren von Daten erfolgt in Tuplespace-basierter Kommunikation durch ein sogenanntes *Template-Matching*-Verfahren. Hierbei beschreibt der potentielle Konsument eines Tupels dessen Struktur durch eine *Schablone* (engl. *Template*), welche die Struktur des zu empfangenden Tupels widerspiegelt. Diese Schablone kann dabei den Typ eines oder mehrerer Felder eines Tupels, konkrete Werte für einzelne Felder sowie einen *Wildcard*-Operator spezifizieren. Generell ähnelt das Template-Matching-Verfahren in Tuplespaces dem aus den Datenbanken bekannten *Query-by-Example* [Zlo77].

Das *Linda*-Kommunikationsparadigma ist seit seiner Entwicklung Gegenstand verschiedener wissenschaftlicher Arbeiten, die es beispielsweise hinsichtlich Mächtigkeit [BGZ97], dem Grad der Entkopplung von Kommunikationspartnern [AADH05, Ley06], Unterschieden zu anderen Kommunikationstechnologien [FKLT07] und auf einen möglichen Einsatz in Szenarien der *Enterprise Application Integration (EAI)* [MWSL07] hin untersuchen.

Darüber hinaus existieren verschiedene Arbeiten, die die Erweiterung des klassischen Linda-Modells und deren Implementierung zum Gegenstand haben. Einige Beispiele für Linda-Erweiterungen sind:

Batch-Operationen Die Operationen `getAll` und `getAny` in *Jada* [CR97,

CR96] erlauben das Konsumieren aller Tupel eines Tuplespace, die entweder zu einem (`getAny`) oder allen (`getAll`) Templates eines Klienten konform sind, innerhalb eines Operationsaufrufs. Ein weiteres Beispiel für diese Klasse von Erweiterungen ist die von der *JavaSpace05*-Schnittstelle [sun03] spezifizierte `take`-Operation auf Java-*Collection*-Objekten.

Transaktionales Verhalten Beispiele für die Unterstützung von Transaktionen sind einerseits Operationen, die die atomare Verarbeitung von Tupelmengen innerhalb eines Operationsaufrufs erlauben, wie z. B. die `writeAll`-Operation in *TSpaces* [WMLF98, LMW99]. Andere Ansätze bieten die Möglichkeit zum Verbund mehrerer einzelner Operationsaufrufe (auch auf potentiell mehreren Tuplespaces) innerhalb einer Transaktion, wie z. B. *MARS* [CLZ00] oder *JavaSpaces* und *Gigaspaces*[1], welche sich zur Realisierung transaktionalen Verhaltens auf *Jini*-Transaktionen [WJT00] stützen oder *CORSO* [Küh94], welches das Flex-Transaktionsmodell implementiert.

Erweiterung des Template-Matching-Mechanismus Neben den oben bereits genannten Beispielen zum Matching auf Template-Mengen erweitert *TSpaces* den Template-Matching-Mechanismus um die Möglichkeit zur Evaluierung von XQL- bzw. XPath-Ausdrücken. Ähnliche Möglichkeiten zum Zugriff auf XML-Daten in Tupeln bietet *XMLSpaces* [TG01]. Zur besseren Integration in bestehende Anwendungen erlaubt *Gigaspaces* den Zugriff auf den Tuplespace durch verschiedene etablierte Schnittstellen, wie z. B. *JMS* [sun08], *JDBC* [EHF06].

Neue Koordinationsprimitive Beispiele für erweiterte Koordinationsprimitive sind u.a. *Bonita* [RW97] mit den Operationen `dispatch`, `arrived` und `obtain` oder die `rhonda`-Operation zur Synchronisation von Interaktionspartnern in *TSpaces*. Zur Realisierung beliebiger Koordinationsprimitive definiert XVSM [KRJ05, KMS08] die Konzepte *Container* und *Coordinator*. Ein Coordinator legt hier – in Verbindung mit einem

[1] `http://www.gigaspaces.org/`

sogenannten *Selector* – die Ordnung fest, in der ein Zugriff auf die Tupel eines Containers erfolgt.

“Global View”-Operationen Um eine erschöpfende Untersuchung sämtlicher Tupel eines Tuplespace zu erlauben, definieren sowohl *Jada* als auch *JavaSpaces* nicht-blockierende Operationen zum Konsumieren von Tupeln bzw. die *JavaSpace05*-Schnittstelle die Operation `contents`. Weitere Beispiele für Operationen, die eine globale Sicht auf einen Tuplespace erlauben, sind die Operationen `scan` und `count` in *TSpaces*.

2.7 Zusammenfassung

In diesem Kapitel wurden zwei unterschiedliche Themenbereiche adressiert: Zum einen wurden die im weiteren Verlauf der Dissertation verwendeten Technologien und Standards einführend vorgestellt. Diese umfassen verschiedene Standards aus dem Bereich der Web-Services sowie eine Erläuterung der Architektur existierender WfMS und der in diesen jeweils verwendeten Vorgehensweise zur Prozessnavigation.

Zum anderen wurde ein Überblick über relevante wissenschaftliche Arbeiten aus den Forschungsbereichen der verteilten Ausführung und Partitionierung von Prozessen gegeben. Deren Vorstellung bildet einerseits eine wesentliche Grundlage des in Kapitel 4 vorgestellten Verfahrens zur Partitionierung von BPEL-Prozessen sowie der in Kapitel 5 vorgestellten Architektur der PS-Infrastruktur und diente andererseits der Abgrenzung zu den Vorgehensweisen und Ergebnissen dieser Dissertation.

Abschließend wurden die Kernkonzepte der Tuplespaces anhand des Linda-Modells einführend dargestellt und existierende Erweiterungen erläutert. Diese Konzepte haben einen wesentlichen Einfluss auf die in Kapitel 5 präsentierte PS-Infrastruktur.

KAPITEL 3

DEZENTRALE PROZESSAUSFÜHRUNG

Die dezentrale Ausführung von BPEL-Prozessen unter Verwendung der in dieser Dissertation vorgestellten PS-Infrastruktur folgt einem zweistufigen Verfahren. Den ersten Schritt bildet die Dekomposition eines Prozesses in einzelne, konzeptuell voneinander unabhängige, funktionale Bausteine. Diese werden in einem zweiten Schritt auf die zur Verfügung stehende PS-Infrastruktur verteilt und koordinieren sich zur Laufzeit von Prozessinstanzen derart, dass ein zur operationalen Semantik des BPEL-Prozesses äquivalentes Verhalten entsteht. Eine derartige Verteilung sämtlicher funktionaler Bausteine eines Prozesses wird in der Folge als *Konfiguration* bezeichnet. Ein wesentliches Ziel der Dekomposition ist es dabei, ein möglichst umfangreiches Spektrum unterschiedlicher Prozesskonfigurationen zu unterstützen.

Gegenstand dieses Kapitels ist die Erläuterung des EWFN-Prozessmetamodells, welches die oben genannte Prozessdekomposition realisiert. Dessen Vorstellung beschränkt sich dabei auf lediglich diejenigen Aspekte, die für ein Verständnis der in den nachfolgenden Kapiteln vorgestellten Konzepte unmittelbar

erforderlich sind. Eine detaillierte Beschreibung des EWFN-Metamodells und dessen theoretischer Grundlagen ist Teil von [WML08b, MWL08a, MWL08b] und [Mar10].

Das Kapitel ist wie folgt strukturiert: in Abschnitt 3.1 wird die Vorgehensweise der dezentralen Navigation eines Prozesses auf Basis der PS-Infrastruktur anhand eines Beispiels erläutert. Hieran anknüpfend werden in Abschnitt 3.2 die wesentlichen Konzepte des EWFN-Metamodells vorgestellt. In Abschnitt 3.4 wird das, durch die Dekomposition eines BPEL-Prozesses in ein EWFN-Modell, erreichbare Verteilungsspektrum beschrieben. Die Realisierung der verteilten Prozessausführung stellt besondere Anforderungen an die Verarbeitungsschritte, die vor, während und nach der Ausführung des Prozesses durchlaufen werden müssen. Diese Verarbeitungsschritte werden in Abschnitt 3.5 erläutert und in einen Prozesslebenszyklus eingeordnet.

3.1 Dezentrale Prozessnavigation

Abbildung 3.1 zeigt eine (vereinfachte) graphische Darstellung des *Loan Approval*-Prozesses aus [Org07]. Schwarz gefärbte Kreise repräsentieren *Basic Activities*, weiß gefärbte Rechtecke *Structured Activities*, weiß gefärbte gestrichelte Kreise Variablen. Die vom Prozess verwendeten Dienste sind als grau gefärbte Rechtecke dargestellt. Die grau gefärbten Rechtecke P_1, P_2, P_3 gruppieren die abgebildeten Elemente und repräsentieren die an der Ausführung des Prozesses teilnehmenden Partner, wobei P_1 das BPEL-WfMS und P_2 und P_3 die vom Prozess verwendeten Dienste betreiben.

Durchgezogene Pfeile beschreiben Synchronisationskanten (LINK) in einem FLOW-Konstrukt. Gestrichelte Pfeile zwischen Aktivitäten und Variablen beschreiben Variablenzugriff; gepunktete Pfeile zwischen Aktivitäten und Diensten repräsentieren eine Interaktion zwischen einem WfMS und dem jeweiligen Dienst. Zwischen lesendem und schreibendem Variablenzugriff wird in der Abbildung nicht unterschieden; ebenso werden weder Anzahl noch Abfolge der während der Interaktion mit einem Web-Service ausgetauschten Nachrichten dargestellt.

Als Grundlage für die Vorstellung der Vorgehensweise zur dezentralen Aus-

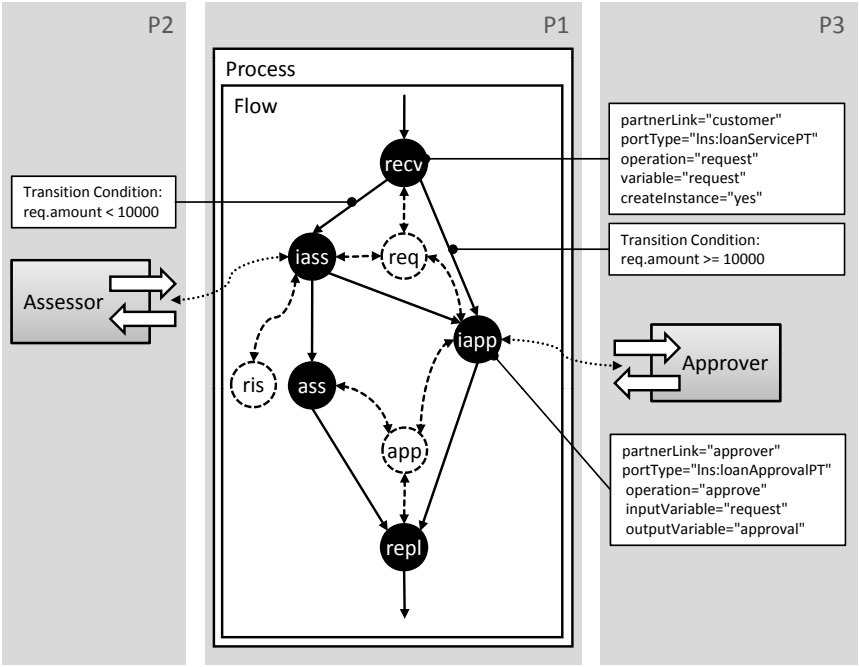


Abbildung 3.1: Prozessbeispiel: *Loan Approval*

führung des Prozesses und eines entsprechenden Prozessmetamodells wird zunächst erläutert, wie eine Ausführung des dargestellten Prozesses in einem regulären BPEL-WfMS mit einem logisch zentralen Navigator abläuft.

Die Ausführung einer Prozessinstanz beginnt mit ihrer Erzeugung als Folge des Eingangs einer Nachricht beim WfMS, welche die durch die RECEIVE-Aktivität *receive* (kurz *recv*) spezifizierte Kombination aus PARTNERLINK, PORTTYPE und OPERATION erfüllt. Die Nutzdaten der eingehenden Nachricht werden vom WfMS während der Ausführung der Aktivität *recv* in der Variablen *request* (kurz *req*) abgelegt und stehen nach Beendigung der Ausführung von *recv* anderen Aktivitäten zur Verarbeitung zur Verfügung. Ebenso erfolgt bei Beendigung der Ausführung von *recv* eine Bestimmung des Status der ausgehen-

den Synchronisationskanten von *recv* durch eine Evaluierung ihrer jeweiligen TransitionCondition. Im Fall des dargestellten Beispiels realisieren diese Bedingungen einen gegenseitigen Ausschluss, d. h. es kommt – abhängig vom Wert der Variable *req* – entweder die Invoke-Aktivität *invoke assessor* (kurz *iass*) oder die Invoke-Aktivität *invoke approver* (kurz *iapp*) zur Ausführung, die jeweils eine Interaktion des WfMS mit externen Web-Service-Implementierungen beschreiben. Evaluiert eine TransitionCondition zu einem Booleschen `true`, so erhält der Link einen positiven Status, andernfalls ist der Link-Status `false`. Der Status der eingehenden Kanten einer Aktivität bestimmt – in Verbindung mit der sogenannten JoinCondition der Aktivität – ob die jeweilige Aktivität zur Ausführung kommt. Die JoinCondition einer Aktivität ist ein Boolescher Ausdruck auf Grundlage des Status ihrer eingehenden Synchronisationskanten[1]. Diese wird evaluiert, nachdem der Status sämtlicher eingehender Synchronisationskanten bekannt ist. Führt die Evaluierung der JoinCondition zu einem Booleschen `true`, so kommt die jeweilige Aktivität zur Ausführung. Im Fall einer Evaluierung zu `false` bestimmt der Wert des Attributs SuppressJoinFailure der jeweiligen Aktivität das Verhalten des WfMS und löst gegebenenfalls den sogenannten *Dead-Path-Elimination*-Mechanismus aus, welcher im Fall der Nicht-Ausführung einer Aktivität diese Information entlang aller ausgehender Synchronisationskanten einer Aktivität an deren Folgeaktivitäten durch Setzen eines entsprechenden Link-Status propagiert. Kam in Abbildung 3.1, aufgrund eines Werts von `req = 5000`, beispielsweise *iass* zur Ausführung, so kann – bedingt durch den Link zwischen *iass* und *iapp* und eine entsprechende JoinCondition bei *iapp* – auch Aktivität *iapp*, trotz des negativen Status des Link zwischen *recv* und *iapp*, noch zur Ausführung kommen.

Angeschlossen an die Ausführung der Aktivitäten *iapp* und *ass* nach derselben Vorgehensweise schließt die Reply-Aktivität *repl* die Ausführung der Prozessinstanz, durch Senden des in Variable *approval* (kurz *app*) vorgehaltenen Werts als Antwort-Nachricht an den aufrufenden Klienten, ab.

Abbildung 3.2 zeigt eine Darstellung desselben Prozessmodells wie Abbildung 3.1, allerdings eine andere *Konfiguration*, d. h. Verteilung seiner Elemente

[1] Definiert eine Ziel-Aktivität von Synchronisationskanten keine explizite JoinCondition, so gilt für diese implizit ein Boolesches `OR`.

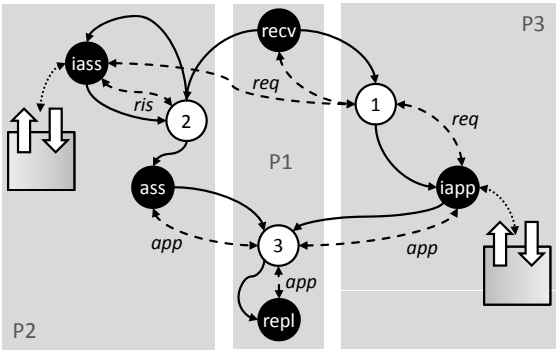


Abbildung 3.2: Konfiguration für dezentrale Prozessausführung

auf die zur Verfügung stehende Ausführungsumgebung. In dieser Konfiguration verteilt sich die Ausführung der einzelnen Aktivitäten des Prozessmodells sowie die in diesem definierten Instanzdaten auf die dargestellten Partner P_1, P_2, P_3. Die durchgezogenen Kreise $1, 2, 3$ in der Abbildung repräsentieren Puffer für die Übertragung von Nachrichten, über die die Implementierungen der einzelnen Aktivitäten des Prozesses, sogenannte *Process-Space-Klienten (PS-Klienten)*, asynchron und über die Grenzen mehrerer Systeme hinweg miteinander interagieren können; in der PS-Infrastruktur werden diese Nachrichtenpuffer als *Process-Spaces (PS)* bezeichnet. Für die Interaktion von PS-Klienten über PS gelten die Eigenschaften der Entkopplung in Zeit, Raum und Referenz [WML09b, AADH05, Ley06].

Im Vergleich zu Abbildung 3.1 gilt in der hier dargestellten Konfiguration, dass die Ausführung der Aktivitäten *iass* und *iapp* von P_1 zu P_2 bzw. P_3 verschoben wurde. Als Folge wird nach Ausführung von Aktivität *recv* – analog der zentralen Ausführung – die Kontrolle über die Prozessausführung, abhängig vom Resultat der Evaluierung der TransitionCondition der ausgehenden Synchronisationskanten von *recv* entweder über PS 1 an Aktivität (bzw. PS-Klient) *iapp* oder über PS 2 an Aktivität *iass* übergeben. Dieser Weitergabe der Prozesskontrollflussinformationen geschieht durch einen *Token-Passing*-Mechanismus, indem *recv* ein entsprechendes Datum in den jeweiligen PS schreibt. Analog

wird für das im Rahmen der Ausführung von *recv* empfangene Datum *req* verfahren. Dieses wird unabhängig vom ausgeführten "linken" oder "rechten" Prozesspfad in PS 1 abgelegt. Die Aktivitäten *iapp* und *iass* überwachen jeweils die PS 1 bzw. 2 auf das Vorhandensein der von *recv* erzeugten Kontrollflussinformation in Form einer Nachricht, die die (erfolgreiche) Beendigung der Ausführung von *recv* signalisiert[1]. Sind die von einer Aktivität jeweils erwarteten Kontrollflussinformationen in den von dieser überwachten PS vorhanden, so ist die *Eingangsbedingung* der Aktivität erfüllt; sie kann mit ihrer Ausführung beginnen. Am Beispiel von *iapp* gilt dies, sobald die Ausführung von *recv* abgeschlossen und die JoinCondition von *iapp* zu einem positiven Wert ausgewertet wurde.

Als Teil der Ausführung einer Aktivität werden sowohl die "eingehenden" Kontrollflussinformationen als auch die von der Aktivität während ihrer Ausführung verwendeten Instanzdaten aus den jeweiligen PS konsumiert. Ein Instanzdatum wird, ebenso wie die Kontrollflussinformationen, in einem PS abgelegt und steht somit den unterschiedlichen PS-Klienten zur Verfügung. Für Aktivität *iapp* sind beispielsweise nicht nur die Kontrollflussinformationen, sondern auch das, für die Ausführung der Aktivität benötigte, Datum *req* in PS 1 abgelegt. Nach dem Konsumieren der Daten führt *iapp*, unter Verwendung dieser Daten, einen internen Verarbeitungsschritt aus; in diesem Fall die Interaktion mit dem dargestellten Dienst. In dieser Prozesskonfiguration wird der so aufgerufene Dienst durch denselben Partner P_3 bereitgestellt, von welchem auch die entsprechende, den Aufruf auslösende, Aktivität *iapp* betrieben wird. Nach der erfolgreichen Interaktion wird (i) die vom Dienst erhaltene Antwort-Nachricht (*app*) in PS 3 geschrieben und damit den anderen Aktivitäten (im Beispiel insbesondere *repl*) zugänglich gemacht sowie (ii) eine Kontrollflussnachricht über den erfolgreichen Abschluss der Ausführung von *iapp* ebenfalls in PS 3 veröffentlicht, welche Teil der Eingangsbedingung von *repl* ist.

Wie aus dem Beispiel ersichtlich ist, kann die Kommunikation von Instanzdaten und Kontrollflussinformationen sowohl über dieselben wie auch unterschiedliche Puffer erfolgen. Die Ausführung der weiteren Aktivitäten des

[1] Aus Gründen der Übersichtlichkeit der Abbildung wird das Verhalten des Workflows im Fall einer fehlerhaften Ausführung einer Aktivität an dieser Stelle nicht weiter diskutiert.

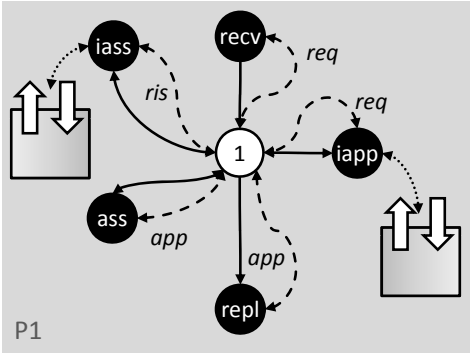


Abbildung 3.3: Konfiguration für zentrale Prozessausführung

Prozesses erfolgt analog dem oben beschriebenen Verfahren.

Ein Kernpunkt des in der Arbeit entwickelten Ansatzes und der zugehörigen Infrastruktur ist die Nicht-Invasivität der Bestimmung einer Prozesskonfiguration. Dies bedeutet, dass die Bestimmung einer Konfiguration eines Prozesses derart vorgenommen werden kann, dass dies keine Anpassung des Prozessmodells zur Folge hat (vgl. Abschnitt 1.2). Zur Verdeutlichung zeigt Abbildung 3.3 eine alternative Konfiguration des in Abbildung 3.1 dargestellten Prozessmodells. In dieser erfolgt die Interaktion der einzelnen PS-Klienten allerdings über denselben zentralen PS 1 und ist dementsprechend vergleichbar der in Abbildung 3.1 dargestellten zentralen Prozessausführung.

Betrachtet man die beiden in Abbildung 3.2 und 3.3 dargestellten Konfigurationen des *Loan-Approval*-Prozesses (Abbildung 3.1), so wird ersichtlich, dass die Ausführung eines derart zerteilten und dezentral navigierten BPEL-Prozesses den Austausch unterschiedlicher Information zwischen seinen PS-Klienten bedingt. Diese Informationen lassen sich wie folgt klassifizieren:

Kontrollfluss Ein Prozessmodell (bzw. dessen *Structured Activities*) beschreibt eine Ordnung auf den Aktivitäten des Prozesses. Das bedeutet beispielsweise, dass die erfolgreiche Ausführung einer Aktivität Voraussetzung für die Ausführung einer oder mehrerer anderer Aktivitäten sein kann (z. B.

bedingt *recv* die Ausführung von *iapp*). In einem logisch zentralen WfMS wird die Einhaltung dieser Ordnung durch dessen Navigator-Komponente sichergestellt, welche den Ausführungszustand sämtlicher Aktivitäten des Prozesses kennt, und anhand diesem die auszuführende(n) Nachfolgeraktivität(en) bestimmt (vgl. Abschnitt 2.3.2).

Da in einem logisch dezentralen WfMS kein derartiger Navigator existiert, ist hier eine explizite Weitergabe von Informationen zwischen den einzelnen Aktivitäten des Prozesses (bzw. ihren implementierenden PS-Klienten) erforderlich. Die explizite Beschreibung der Weitergabe dieser Kontrollflussinformationen ist folglich eine wesentliche Anforderung an eine Prozessmetamodell zur Beschreibung dezentral ausgeführter Prozesse.

Tote Pfade Eine Sonderform der oben genannten Kontrollflussinformationen sind die sogenannten *Toten Pfade*. Führt die Evaluierung der JoinCondition einer Kind-Aktivität eines Flow zu einem negativen Resultat, so kommt diese nicht zur Ausführung. Gilt für die *Join*-Aktivität die Eigenschaft `suppressJoinFailure` (vgl. Abschnitt 11.6.3. *Dead-Path-Elimination* in [Org07]) nicht, so führt dies zum BPEL-Fehler `joinFailure`, welcher im Rahmen der regulären Fehlerbehandlung in BPEL verarbeitet wird. Für den Fall, dass für die Aktivität die `suppressJoinFailure`-Eigenschaft gilt und die Evaluierung der JoinCondition der Aktivität zu einem negativen Wert führt, unterbleibt die Auslösung und Verarbeitung des Fehlers. Kommt eine Aktivität aufgrund eines negativen Status einer JoinCondition nicht zur Ausführung, so muss dies aufgrund der synchronisierenden *Join*-Semantik in BPEL[1] an nachfolgende Aktivitäten kommuniziert werden. Wäre dies nicht der Fall, so würde die Ausführung einer Join-Aktivität unter Umständen endlos blockieren und damit zu einer Verklemmung des Prozesses führen.

In Verbindung mit der oben vorgestellten Evaluierung des “positiven” Prozesskontrollflusses verlangt die Prozessausführung also Wissen über

[1] Eine JoinCondition wird in BPEL erst dann evaluiert, wenn die Status der eingehenden Synchronisationskanten der Aktivität bestimmt sind.

sowohl (i) die Ausführung als auch (ii) die Nicht-Ausführung im Prozessmodell vorangehender Aktivitäten.

Instanzdaten Neben Informationen, die den Kontrollfluss betreffen, erfordert die Ausführung eines Prozesses Kenntnis über die Werte von dessen Instanzdaten für die folgenden Zwecke: (i) Sie beeinflussen die Navigation eines Prozesses durch beispielsweise die Verwendung in der TRANSITIONCONDITION einer Synchronisationskante, die Bedingung eines IF oder des Zählers eines WHILE. (ii) Sie dienen als Ein- oder Ausgabedaten für Aktivitäten wie beispielsweise ASSIGN, INVOKE oder RECEIVE. (iii) Sie beinhalten im Prozessmodell nicht sichtbare (d. h. nicht explizit deklarierte) Informationen, wie z. B. Informationen über in Ausführung befindliche "offene" Interaktionen mit WfMS-externen Kommunikationspartnern als Teil der Ausführung einer RECEIVE-Aktivität.

Da der Wert von Instanzdaten Einfluss auf die Ausführung einer Prozessinstanz hat, ist ein "gemeinsamer" Zugriff der einzelnen PS-Klienten auf die von ihnen benötigten Instanzdaten erforderlich. Dies bedingt insbesondere eine explizite Kenntnis eines PS-Klienten darüber, welche Instanzdaten er konsumieren bzw. manipulieren muss und deren Ort in der PS-Infrastruktur während der Ausführung einer Instanz einer bestimmten Prozesskonfiguration. Folglich müssen auch Informationen über Instanzdatenzugriffe durch die Elemente eines Prozessmetamodells für dezentral ausgeführte Prozesse beschreiben werden können.

Zustand von Aktivitäten Aktivitäten in BPEL sind zustandsbehaftet; [KKS+06] identifiziert die Aktivitätszustände *Inactive*, *Ready*, *Executing*, *Waiting*, *Faulted*, *Terminated*, *Dead Path*, *Complete* für alle BPEL-Aktivitäten. Für SCOPE-Aktivitäten gelten weiterhin die Zustände *EventHandling*, *FaultHandling*, *FaultHandlingNoHandler*, *CompensationExecuting*, *Compensated* und *CompensatedWithFault*. Der Zustand einer Aktivität kann Auswirkungen auf die Ausführung anderer Aktivitäten des Prozesses haben. So gilt beispielsweise, dass die im Kontext einer SCOPE-Aktivität definierten EVENTHANDLER erst nach Aktivierung des SCOPE aktiviert werden.

Analog den oben genannten Instanzdaten gilt auch hinsichtlich des Zustands von Aktivitäten, dass bestimmte Aktivitäten über Zustandsänderungen anderer Aktivitäten in Kenntnis gesetzt werden müssen, die Kommunikation dieser Informationen bei der Nicht-Existenz eines logisch zentralen Navigators also explizit durch das ausgeführte Prozessmodell beschrieben werden muss.

3.2 Beschreibung dezentraler Navigation: Executable Workflow Nets

Entsprechend den oben formulierten Anforderungen ist der Kerngedanke des Prozessmetamodells der *Executable Workflow Nets (EWFN)* [WML08b, MWL08a, MWL08b, Mar10] die explizite Beschreibung jedweder Kommunikation, die zwischen den verschiedenen PS-Klienten eines Prozesses notwendig ist, um die operationale Semantik von BPEL (auch bei dezentraler Ausführung) zu bewahren. Diese Kommunikation umfasst die vier in Abschnitt 3.1 vorgestellten Klassen von Informationen: positiver und negativer Kontrollfluss, durchgeführte Instanzdatenzugriffe und die Repräsentation des internen Zustands in Ausführung befindlicher Aktivitäten.

Konzeptuell orientiert sich der EWFN-Formalismus stark an *Gefärbten Petri-Netzen* (engl. *Colored Petri Nets, CPN*) [Jen96] und unterscheidet, diesen entsprechend, die Elemente *Transitionen*, *Marken*, *Stellen* und *Kanten*.

Entsprechend ihrer Funktion in den sogenannten *Place-Transition Nets* [Pet80, Rei85] und den CPN repräsentieren *Transitionen* (engl. *Transitions*) im EWFN-Formalismus die aktiven Elemente eines Prozesses, d. h. dessen Aktivitäten bzw. auf Ebene der zur Ausführung verwendeten PS-Infrastruktur die einzelnen PS-Klienten. Eine Transition ist dann *schaltbereit*, wenn ihre *Eingangsbedingung* erfüllt ist. Die Beendigung ihres Schaltvorgangs signalisiert sie durch Erzeugung sogenannter *Ausgangseffekte*. Es gilt, dass eine Transition, je nach vorliegender Eingangsbedingung, unterschiedliche Ausgangseffekte erzeugen kann.

Eine *Marke* (engl. *Token*) repräsentiert ein zwischen Transitionen ausgetauschtes Datum. Im EWFN-Formalismus gilt, dass Marken, im Gegensatz zu

beispielsweise *Place-Transition-Nets*, eine interne Struktur besitzen: sie folgen dem Aufbau einer endlichen Liste typisierter Felder.

Eine *Stelle* (engl. *Place*) repräsentiert einen passiven Puffer, welcher als Zwischenspeicher für Marken während ihrer Kommunikation zwischen Transitionen dient, d. h. einen PS.

Eine *Kante* (engl. *Arc*) beschreibt die Kommunikation einer Marke zwischen entweder einer Stelle und einer Transition oder einer Transition und einer Stelle. Ein Markenaustausch erfolgt niemals zwischen Transitionen direkt, sondern immer über eine Stelle. Ähnlich gefärbten Petri-Netzen gilt, dass eine Kante spezifizieren kann, welche Marken entlang dieser produziert bzw. konsumiert werden. Weiterhin sind Kanten typisiert; jeder Kantentyp realisiert dabei eine bestimmte Semantik hinsichtlich der Verarbeitung einer Marke (z. B. destruktives oder nicht-destruktives Konsumieren von einer oder mehreren Marken).

Zur Verdeutlichung dieser Konzepte und des Ablaufs der Ausführung einer Instanz eines EWFN-Modells zeigt Abbildung 3.4 schematisch die Koordination der Ausführung der Transitionen Aktivität 1 bis Aktivität 4 über zwei unterschiedliche Stellen 1 und 2. Aufgrund der Ähnlichkeit des EWFN-Formalismus zu CPN orientiert sich auch die hier verwendete graphische Notation an typischen Notationen zur Beschreibung von Petri-Netzen. Rechtecke symbolisieren Transitionen, große Kreise Stellen, Pfeile Kanten und kleine schwarz und weiß gefärbte Kreise unterschiedlich typisierte Marken. Zur Verdeutlichung der Darstellung mehrerer Kommunikationsvorgänge über dieselbe Stelle verbindet eine Kante in dieser Darstellung nicht eine Transition mit einer Stelle (wie in der sonst für CPN gebräuchlichen graphischen Notation üblich), sondern stattdessen eine Transition mit einer Marke.

Entlang den unterschiedlichen Kantentypen des EWFN (*write*, *take*, *read*, *update*; eine Beschreibung der Semantik der einzelnen Operationen ist Gegenstand von Abschnitt 5.3.1) werden unterschiedlich typisierte Marken kommuniziert. Sogenannte *Kontrollflussmarken* ($CF1, CF2, CF3, CF4$) repräsentieren die kontrollflussbezogenen Informationen eines Prozesses. Sogenannte *Datenmarken* ($D1, D2$) beschreiben die Instanzdaten und die Informationen über den Ausführungszustand einzelner Aktivitäten, die während der Ausführung

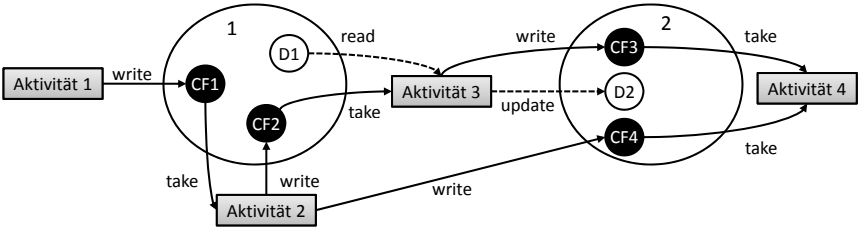


Abbildung 3.4: Dezentrale Koordination von Transitionen durch den Austausch von Marken über Stellen.

einer Prozessinstanz zwischen den einzelnen Aktivitäten ausgetauscht werden müssen.

Die Eingangsbedingung von Aktivität 2 ist das Vorhandensein von $CF1$ in Stelle 1, ihr Ausgangseffekt die Erzeugung der Marke $CF2$ in Stelle 1 und $CF4$ in Stelle 2. Die von Aktivität 2 erzeugte Marke $CF2$ dient wiederum als Eingangsbedingung für Aktivität 3. Zusätzlich konsumiert Aktivität 3 Instanzdaten in Form der Datenmarken $D1$ und aktualisiert weiterhin den Wert des Datums $D2$ in Stelle 2. Die von Aktivität 3 erzeugte Marke $CF3$ und die von Aktivität 2 erzeugte Marke $CF4$ erfüllen gemeinsam die Eingangsbedingung von Aktivität 4; diese realisiert also einen *Join* unterschiedlicher Ausführungspfade.

Analog des in Abbildung 3.4 dargestellten Kommunikation von (positiven) Kontrollflussinformationen und der Instanzdatenzugriffe erfolgt auch die Kommunikation des negativen Kontrollflusses durch Marken speziellen Typs (sogenannte *Dead Tokens* [MWL08b], ähnlich [LSW97, AH05]).

3.3 Transformation zwischen BPEL-Prozessen und deren EWFN-Repräsentation

Abbildung 3.5 verdeutlicht den Zusammenhang zwischen dem BPEL- und dem EWFN-Prozessmetamodell sowie deren jeweiligen Ausführungsumgebungen. Das EWFN-Metamodell bildet mit seinen Mitteln zur expliziten Beschreibung

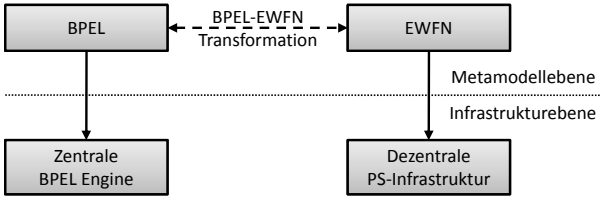


Abbildung 3.5: Beziehung zwischen BPEL-Prozessen, deren EWFN-Repräsentation und den jeweiligen Ausführungsumgebungen

der Koordination der einzelnen PS-Klienten und der Kommunikation von Instanzdaten die Grundlage für die Prozessausführung unter Verwendung der in Kapitel 5 vorgestellten PS-Infrastruktur.

Die sogenannte *BPEL-EWFN Transformation* überführt ein BPEL-Prozessmodell in ein EWFN-Modell, das sich bei seiner Ausführung semantisch äquivalent zum BPEL-Prozess verhält. Die Abbildung der einzelnen Aktivitäten des BPEL-Prozesses erfolgt in Form komponierbarer *EWFN-Patterns*. Abbildung 3.6 zeigt ein Beispiel eines EWFN-Patterns einer ASSIGN- und einer REPLY-Aktivität als Kind-Aktivität einer SEQUENCE-Aktivität. In der Darstellung bezeichnen schwarze Rechtecke Transitionen, schwarze Kreise Stellen und die verschiedenen Pfeile unterschiedliche Kantentypen. Die Annotationen der Kanten bezeichnen den Typ der Marken, welche entlang der jeweiligen Kante kommuniziert werden.

Zur Repräsentation des Prozesskontrollflusses verfügt jedes dieser EWFN-Patterns über eine einheitliche Schnittstelle, die die Komposition einzelner EWFN-Pattern zu komplexeren erlaubt. In Abbildung 3.6 ist z. B. ersichtlich, dass sich die ENDED-Stelle der ASSIGN-Aktivität mit der START-Stelle der REPLY-Aktivität überlagert, um die von der umgebenden SEQUENCE-Aktivität verlangte sequentielle Ausführung der Kind-Aktivitäten zu erreichen. Die detaillierte Beschreibung der Realisierung sämtlicher BPEL-Aktivitäten mit den Mitteln des EWFN-Metamodells ist Teil von [Mar10].

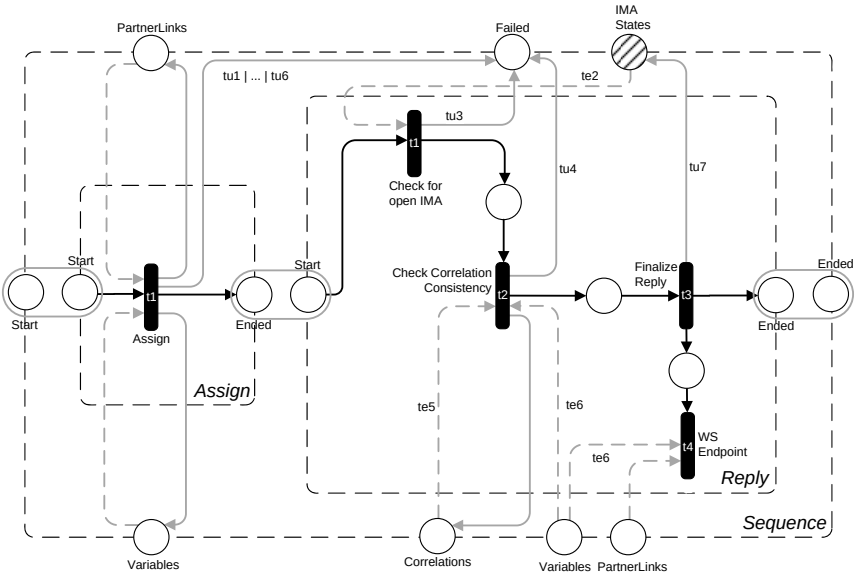


Abbildung 3.6: Darstellung eines Ausschnitts eines BPEL-Prozesses in seiner EWFN-Repräsentation

3.4 Unterstütztes Verteilungsspektrum

Das, in Abschnitt 3.2 vorgestellte, EWFN-Prozessmetamodell ermöglicht, in Verbindung mit der Bestimmung des Ortes der PS-Klienten und der Instanzdaten in der Ausführungsumgebung des Prozesses sowie einer entsprechenden Laufzeitinfrastruktur, die Nutzung eines breiten Spektrums möglicher Prozesskonfigurationen, ohne dass dies Auswirkungen auf das jeweilige BPEL-Prozessmodell hat [WML09b].

Die, in Abbildung 3.7 dargestellte, zentrale Prozesskonfiguration markiert das eine Extrem des realisierbaren Verteilungsspektrums. In dieser Konfiguration werden sämtliche Aktivitäten des Prozesses von demselben WfMS (dar-

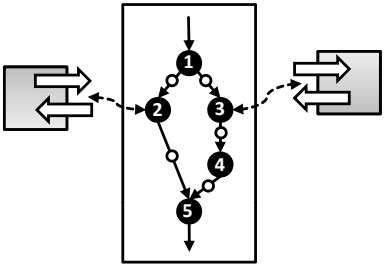


Abbildung 3.7: Zentrale Prozesskonfiguration

gestellt durch das Rechteck) ausgeführt[1]. Obwohl in dieser Konfiguration sämtliche Interaktionen zwischen den Aktivitäten des Prozesses auf einem physikalischen Rechner ablaufen, ist zu bemerken, dass auch in diesem Fall die Koordination und Kommunikation der einzelnen PS-Klienten durch "Navigation" des EWFN-Modells des Prozesses erfolgt.

Abbildung 3.8 zeigt das andere Ende des möglichen Verteilungsspektrums. In diesem Konfigurationsbeispiel wurden minimale Partitionen mit jeweils genau einer Aktivität gebildet und jede dieser Partitionen auf einem WfMS installiert. Weiterhin zeigt das Szenario zwei unterschiedliche Möglichkeiten des Deployment der PS im Verhältnis zu den PS-Klienten auf: So wird beispielsweise die Kommunikation zwischen Aktivität (bzw. PS-Klient) 1 und 3 über einen PS abgewickelt, welcher zwischen Aktivität 1 und 3 auf einem separaten physikalischen Rechner bereitgestellt wird. Die Interaktion von 1 und 2 hingegen findet über einen PS statt, der auf demselben physikalischen Rechner wie Aktivität 2 betrieben wird; d. h. von Aktivität 1 ausgelöste Interaktionen sind entfernte Operationsaufrufe, Operationen von 2 werden lokal ausgeführt.

Eine maximal verteilte Prozessausführung, in der in Abbildung 3.8 darge-

[1] Analog Abbildung 3.2 werden PS-Klienten als schwarz gefärbte Kreise $1, \ldots, 5$ und die zur Kommunikation zwischen diesen verwendeten PS als kleinere, weiß gefärbte Kreise dargestellt. Aus Gründen einer übersichtlicheren Darstellung zeigt Abbildung 3.7 genau einen PS für die Interaktion zwischen jeweils zwei Aktivitäten. Tatsächlich gilt, dass (i) sämtliche PS auf demselben physikalischen Rechner betrieben werden können (vgl. Abschnitt 5.2) und (ii) dass die Kommunikation sämtlicher Marken auch über einen einzelnen PS erfolgen kann (vgl. Abbildung 3.3).

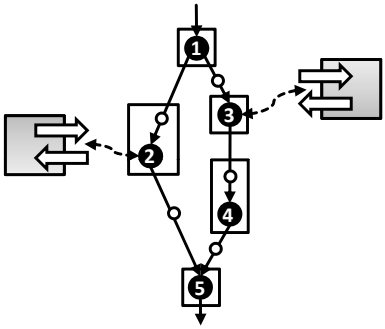
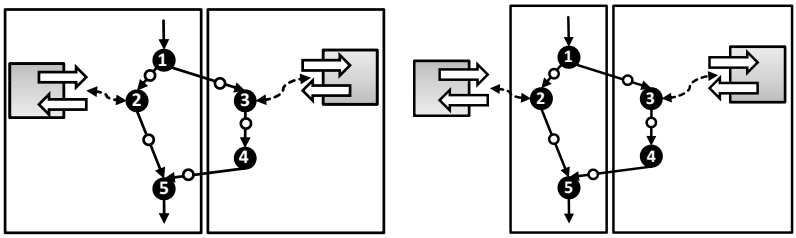


Abbildung 3.8: Maximal dezentrale Prozesskonfiguration

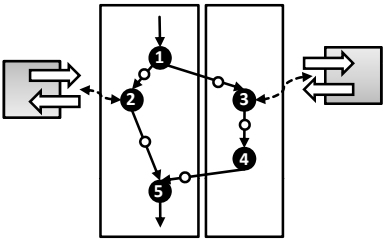
stellten Weise, resultiert offensichtlich in einer großen Menge an entfernter Kommunikation zwischen den beteiligten PS-Klienten. Entsprechend dem Beispiel dezentraler Ausführung in Abbildung 3.2 ist es demnach oftmals sinnvoller, mehrere PS-Klienten sowie die von diesen verwendeten Instanzdaten zu einer sogenannten *Partition* zusammenzufassen und gemeinsam einer Einheit, beispielsweise einem physikalischen Rechner zuzuordnen.

Abbildung 3.9 zeigt ein Beispiel einer derartigen Partitionierung und verdeutlicht drei unterschiedliche Verfahren zum Deployment der vom Prozess verwendeten Dienste. Im dargestellten Beispiel werden die Aktivitäten $1, 2, 5$ durch einen Ausführungsteilnehmer, Aktivitäten 3 und 4 durch einen anderen ausgeführt. Diese Konfiguration des Prozesses kombiniert also *Interaktion innerhalb einer Partition* mit *Interaktion über die Grenzen einer Partition*, d. h. Interaktionen, die über die Grenzen eines physikalischen Rechners hinausreichen.

Abbildung 3.9(a) veranschaulicht anhand eines Beispiels die Zuordnung der Orchestrierungslogik eines Diensts zu demjenigen Ausführungsteilnehmer, der auch den jeweiligen Dienst betreibt. In diesem Szenario sind lediglich die Interaktion zwischen den Aktivitäten 1 und 3 und zwischen 4 und 5 entfernte Interaktionen. Die Interaktionen innerhalb der beiden abgebildeten Partitionen erfolgen lokal, was auch die Interaktionen von 2 und 3 mit den jeweiligen



(a) Lokale Dienste (b) Hybrid



(c) Entfernte Dienste

Abbildung 3.9: Mögliches Dienst-Deployment bei dezentraler Prozessausführung

Diensten einschließt.

In Szenarien, in denen keine unmittelbare Zuordnung der Orchestrierungslogik eines Diensts zur jeweiligen Dienstimplementierung erfolgen soll, d. h. deren Aufruf beispielsweise virtualisiert durch einen *Enterprise Service Bus (ESB)* [Cha04] erfolgt, kann die Ausführung der Orchestrierungslogik auch teilweise 3.9(b) oder vollständig 3.9(c) unabhängig von diesen erfolgen.

Die Bestimmung einer "sinnvollen" Partitionierung eines Prozesses im Hinblick auf ein bestimmtes Optimierungskriterium ist von einer Reihe unterschiedlicher Faktoren abhängig. In Kapitel 4 werden sowohl diese Faktoren als auch ein entsprechendes Partitionierungsverfahren, welches diese Faktoren berücksichtigt, vorgestellt.

3.5 Ein Verfahren zur Verarbeitung dezentral ausgeführter BPEL-Prozesse

Wesentliches Unterscheidungsmerkmal des entwickelten Ansatzes zur dezentralen Prozessnavigation zur gegenwärtigen Vorgehensweise der logisch zentralen Ausführung der Orchestrierungslogik eines Prozesses ist die Möglichkeit zur Aufteilung des Navigationsvorgangs eines BPEL-Prozesses selbst – während seiner *Partitionierung* – auf seine Ausführungsteilnehmer und die damit verbundene Verlagerung der Ausführung der Orchestrierungslogik "nahe" zu den jeweils orchestrierten Diensten (vgl. Abbildung 3.9(a)). Eine derartige Prozessausführung stellt besondere Anforderungen an die angewandte Vorgehensweise zur Verarbeitung von Prozessen vor, während und nach ihrer Ausführung; diese wird nachfolgend – in Form der sogenannten *PS-Methode* – erläutert. Die Beschreibung gibt dabei Verweise auf unterschiedliche nachfolgende Kapitel dieser Dissertation und setzt diese so zueinander in Bezug.

3.5.1 Erweiterter Prozesslebenszyklus

Die typischen Phasen des Lebenszyklus eines ausführbaren Prozesses sind *Modellierung*, *Deployment*, *Ausführung* und *Evolution* [wfm95, AH04, JB96].

Ziel der *Modellierungsphase* ist die Erstellung eines Prozessmodells, welches einen Geschäftsprozess der realen Welt in einer Form widerspiegelt, die, durch ein WfMS unterstützt, (automatisiert) ausgeführt werden kann. Bei der Realisierung komplexer Geschäftsprozesses gliedert sich die Modellierungsphase oft in die Teilschritte *fachliche Prozessmodellierung* und *technische Prozessmodellierung*. Während der fachlichen Prozessmodellierung wird der Prozess, oft unter Verwendung selbst nicht ausführbarer Prozessbeschreibungssprachen auf hoher Abstraktionsebene [KMWL09, KMWL08, RM06], von Domänenexperten beschrieben. In diesem Teilschritt der Modellierungsphase werden technische Aspekte, wie eine Beschreibung der Nachrichtenformate der verwendeten Dienste, oftmals weitgehend außer Acht gelassen. Die Ergänzung dieser Informationen in den Prozessmodellen und die damit verbundene Aufbereitung der Prozesse im Hinblick auf deren Ausführung ist Teil des technischen Pro-

zessmodellierungsschritts. In diesem ergänzen bzw. überführen IT-Spezialisten die nicht-ausführbare Prozessbeschreibung in ein Prozessmodell, das die, für eine automatisierte Ausführung notwendigen Information beinhaltet. Sowohl zur Unterstützung des fachlichen als auch des technischen Prozessmodellierers existieren entsprechende, zumeist graphische, Werkzeuge zur Prozessmodellierung.

Während der *Deployment-Phase* wird ein Prozessmodell aus der Build-Time- eines WfMS in dessen Run-Time-Umgebung überführt (vgl. Abschnitt 2.3.1). Einige für die Prozessausführung relevante Parameter sind nicht Teil des Prozessmodells und werden erst während der Deployment-Phase in Form eines sogenannten *Deployment-Deskriptors* erfasst und der Prozess durch diese parametrisiert. Begründet ist diese Vorgehensweise durch das Ziel, größtmögliche Wiederverwendbarkeit von Prozessen zu gewährleisten. Beispiele für die Parametrisierung von Prozessen während der Deployment-Phase sind die Anpassung eines Prozesses an seine jeweilige Ausführungsumgebung (bzw. deren vorliegende Dienstlandschaft) in Form des sogenannten *Bindens* WfMS-externer Interaktionspartner des Prozesses; ein weiteres Beispiel ist die Konfiguration von Laufzeiteigenschaften (z. B. das Persistierungsverhalten während der Ausführung) des zur Ausführung verwendeten WfMS (vgl. Abschnitt 2.3.2). Unterstützt wird ein Benutzer während des Deployment-Vorgangs typischerweise durch Werkzeuge des WfMS; je nach Grad der Integration der verwendeten Werkzeuge sind diese zum Teil bereits Teil der Prozessmodellierungswerkzeuge.

Nach erfolgreichem Deployment eines Prozesses auf ein WfMS steht dieser potentiellen Nutzern zur Ausführung während der *Ausführungsphase* zur Verfügung. Eine wesentliche Anforderung der automatisierten Ausführung von Geschäftsprozessen – insbesondere bei Prozessen mit einem hohen Geschäftswert – ist die Gewährleistung der Nachvollziehbarkeit der Ausführung der Prozessinstanzen. Zu diesem Zweck wird durch das WfMS während der Instanzausführung ein Ausführungsprotokoll erstellt [wfm98].

Die während der Ausführung erfassten Protokolldaten bilden eine Einflussgröße der sogenannten *Evolution* des Prozesses, seiner fortlaufenden Weiterentwicklung sowie seiner Anpassung an Änderungen in seiner Ausführungsumgebung.

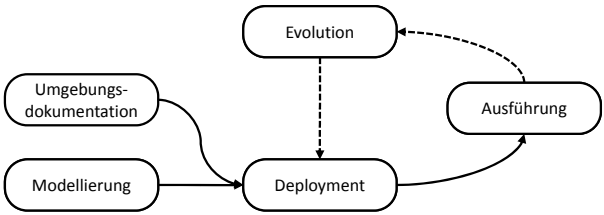


Abbildung 3.10: Übersicht über den Lebenszyklus eines auf der PS-Infrastruktur ausgeführten Prozesses.

Die Vorgehensweise zur Entwicklung dezentral ausgeführter Prozesse nach der PS-Methode orientiert sich soweit möglich an den oben beschriebenen Phasen eines typischen Prozesslebenszyklus. Allerdings wird der Prozesslebenszyklus um die Phase der *Umgebungsdokumentation* ergänzt und existierende Phasen um zusätzliche Verarbeitungsschritte erweitert, die für die Unterstützung einer dezentralen Prozessausführung notwendig sind.

Abbildung 3.10 zeigt eine Übersicht des erweiterten Lebenszyklus eines auf der PS-Infrastruktur ausgeführten Prozesses. In der Abbildung bezeichnen durchgezogene Pfeile Voraussetzungen, gestrichelte optionale Abhängigkeiten.

In der zusätzlichen Phase *Umgebungsdokumentation* erfolgt die Erfassung der Ausführungsumgebung (d. h. die Kombination aus Dienstlandschaft und PS-Infrastruktur), die für die Prozessausführung zur Verfügung steht. Da diese eine Einflussgröße der im Rahmen der Deployment-Phase ausgeführten Prozesspartitionierung darstellen, ist eine Ersterfassung dieser Daten vor der Partitionierung eines Prozesses (d. h. insbesondere vor der Deployment-Phase) erforderlich. Dabei gilt, dass die Dokumentation der Ausführungsumgebung nicht spezifisch für die Partitionierung einzelner Prozessmodelle erfolgt, sondern bereits (für die Partitionierung anderer Prozesse) erfasste Informationen soweit möglich wiederverwendet oder aktualisiert werden. Da sich die zur Verfügung stehende Ausführungsumgebung im Verlauf der Zeit ändern kann, erfolgt auch während der Ausführung von Prozessen eine fortlaufende Erfassung neuer sowie eine Änderung bestehender Daten, welche in zukünftig

verarbeiteten Prozessen berücksichtigt werden kann.

Ein wesentlicher Aspekt der Phasen der PS-Methode ist, wie nachfolgend vorgestellt, die möglichst weitgehende Wiederverwendung existierender Standards und Technologien. Die hieraus resultierenden Vorteile sind die Möglichkeit zur Wiederverwendung bestehender Prozesse und Dienste sowie die Verwendung existierender und etablierter Werkzeuge.

3.5.2 Phasen des Prozesslebenszyklus und beteiligte Rollen

Basierend auf dem in Abbildung 3.10 vorgestellten erweiterten Prozesslebenszyklus gibt Abbildung 3.11 eine Detailübersicht über die im Rahmen des vorgeschlagenen Verfahrens zur Verarbeitung dezentral ausgeführter BPEL-Prozesse notwendigen Schritte. Entsprechend Abbildung 3.10 sind optionale Verarbeitungsschritte der Methode gestrichelt dargestellt, obligatorische in durchgezogenen Linien.

Abgedeckt werden die einzelnen Phasen der PS-Methode durch die Rollen *Prozessmodellierer*, *Dienstanbieter*, *WfMS-Anbieter*, *Partitionierer* und *Deployer*.

3.5.2.1 Modellierung

Grundlage für die dezentrale Ausführung eines BPEL-Prozesses ist eine, zur WS-BPEL-2.0-Spezifikation [Org07] kompatible, ausführbare Prozessbeschreibung. Die Erstellung einer derartigen Prozessbeschreibung und ihre mögliche Aufbereitung für eine dezentrale Ausführung sind Teil der Phase *Modellierung (M)*, welche durch die Rolle *Prozessmodellierer (engl. Process Modeler)* abgedeckt werden. Die Schritte der Modellierungsphase adressieren dabei lediglich die technische Modellierung des Prozesses (vgl. Abschnitt 3.5.1). Erfolgt die Modellierung des Prozesses vorab zunächst auf höherer Abstraktionsebene, beispielsweise unter Verwendung der *Business Process Modeling Notation (BPMN)* [OMG08] oder *Ereignisgesteuerter Prozessketten (EPK)* [KNS92], so muss dieser zunächst in einen entsprechenden ausführbaren BPEL-Prozess überführt werden. Dies kann entweder vollständig manuell oder teilautomatisiert, unter Anwendung existierender Verfahren, wie beispielsweise [ODHA06, RM06, SKI08, WDGW08] für

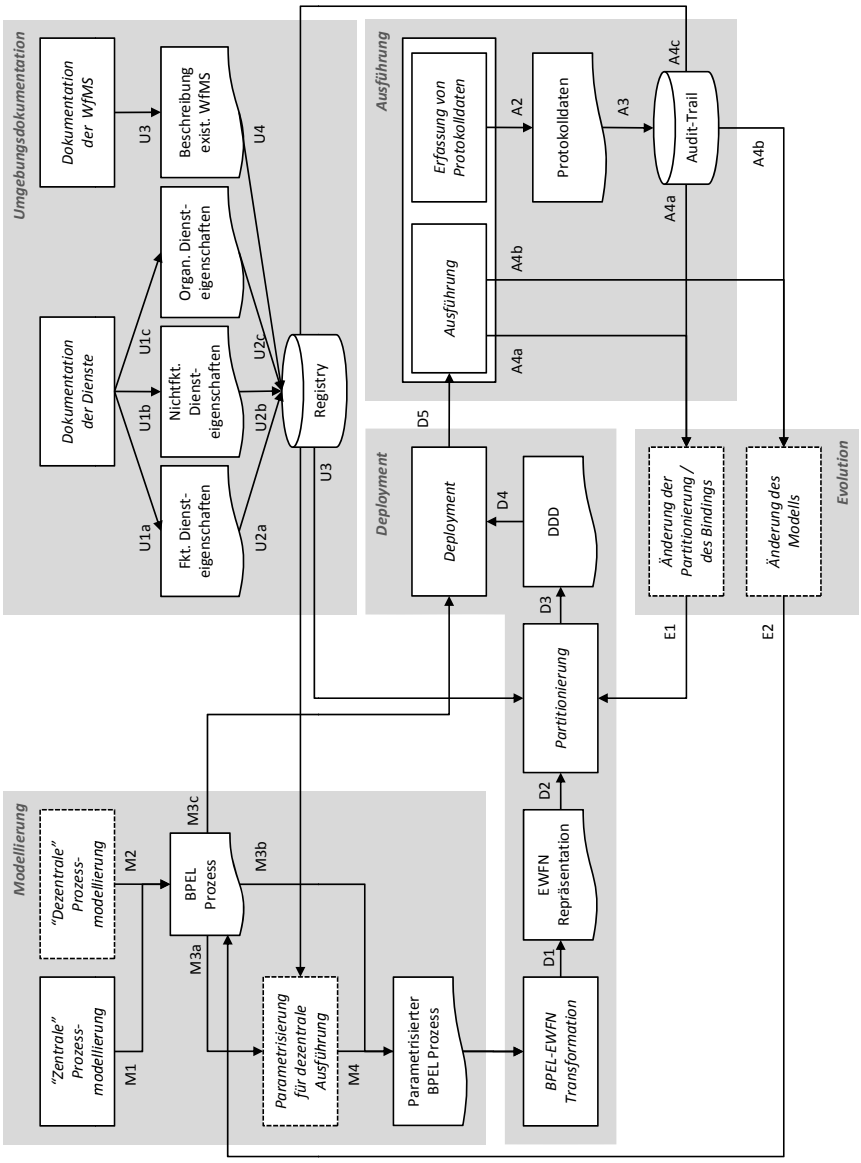


Abbildung 3.11: Detailübersicht der Verarbeitungsschritte eines dezentral ausgeführten Prozesses.

BPMN oder [ZM05, KUL06] für EPK, erfolgen. Im weiteren Verlauf der Arbeit werden diese Fälle allerdings nicht weiter betrachtet.

Die Modellierung des BPEL-Prozesses kann dabei auf unterschiedliche Arten vorgenommen werden. Ein Ziel dieser Arbeit ist es, dass jedes "regulär" (d. h. in Erwartung einer zentralen Ausführung) beschriebene BPEL-Prozessmodell (M1) unter Verwendung der PS-Infrastruktur ausgeführt werden kann, um eine Wiederverwendung existierender Prozessmodelle zu erlauben. Einige Eigenschaften von Prozessmodellen haben bei dezentraler Ausführung einen negativen Einfluss auf das Prozesslaufzeitverhalten. Abschnitt 5.9 zeigt verschiedene Beispiele für diese Eigenschaften auf und gibt Empfehlungen, wie deren Einfluss durch geeignete Prozessmodellierung vermindert werden kann. Ist also bereits zum Modellierungszeitpunkt eines Prozesses bekannt, dass dieser auf der PS-Infrastruktur verteilt ausgeführt werden soll, so kann seine Modellierung diese Empfehlungen berücksichtigen. Im Diagramm wird eine derartige Prozessmodellierung als "dezentrale Prozessmodellierung" (M2) bezeichnet. Das BPEL-Prozessmodell dient in seiner XML-Repräsentation als Eingabe für die Prozess-Deployment-Phase; dabei kann der Prozess entweder direkt weiterverarbeitet (M3b) oder zunächst hinsichtlich seiner Partitionierung geeignet parametrisiert werden (M3a, M4).

Realisiert werden die Schritte der Prozessmodellierungsphase durch die *Modellierer*-Rolle. Da die Eingabe des Systems ein standardkonformer BPEL-2.0-Prozess ist, können für die Realisierung der Arbeitsschritte dieser Phase existierende Werkzeuge zur Entwicklung von BPEL-Prozessen verwendet werden. Beispiele hierfür sind der *Eclipse BPEL Designer*[1] oder der *Oracle JDeveloper*[2].

Nach seiner Modellierung kann ein Prozess im Hinblick auf seine dezentrale Ausführung mittels der PS-Infrastruktur aufbereitet werden. Diese Aufbereitung erfolgt durch Parametrisierung des BPEL-Prozesses in seiner XML-Repräsentation (eine direkte Parametrisierung des EWFN-Modells eines Prozesses ist nicht erforderlich). Die Prozessparametrisierung kann hinsichtlich unterschiedlicher Kriterien erfolgen. Als Teil der Beschreibung des entwickelten Partitionierungsverfahrens werden diese im Detail vorgestellt (Abschnitt 4.2).

[1] http://www.eclipse.org/bpel/
[2] http://www.oracle.com/technology/products/jdev/index.html

Dabei gilt, um der Anforderung der Nicht-Invasivität des Ansatzes nachzukommen, dass die Parametrisierung des Prozesses prozessmodellextern erfolgt. Die Beschreibung, in welcher Form diese Parametrisierung durchgeführt wird, ist Teil der Erläuterungen in Abschnitt 4.3.

3.5.2.2 Umgebungsdokumentation

Voraussetzung für die Partitionierung eines Prozesses ist die Kenntnis seiner *Ausführungsumgebung*; diese erfolgt als Teil der Phase *Umgebungsdokumentation (U)*.

Eine derartige Dokumentation umfasst die Erfassung von Informationen über die verwendete PS-Infrastruktur, d. h. die Gesamtheit der zur Verfügung stehenden PS und (soweit durch den Betreiber der jeweiligen Domäne gewünscht, vgl. Abschnitt 5.8) auch den von diesen jeweils angebotenen PS-Klienten. In Abschnitt 5.2.2 wird erläutert, in welcher Form diese Informationen erfasst (U3) und in einer Service-Registry abgelegt werden können (U4).

Neben der Erfassung von Informationen über die PS-Infrastruktur erfordert das Prozesspartitionierungsverfahren auch eine Beschreibung sowohl funktionaler (U1a) als auch nicht-funktionaler (U1b) und organisatorischer (U1c) Merkmale der Dienste in der zur Verfügung stehenden Dienstlandschaft. Diese werden in Form von Dienst-Metadaten erfasst (U1a, U1b, U1c) und in einem maschinenlesbaren Format in einer Service-Registry aggregiert und damit dem Partitionierungswerkzeug zugänglich gemacht (U2a, U2b, U2c). Die detaillierte Beschreibung, welche Informationen jeweils erfasst werden, sowie die Formate, in denen diese in der Registry abgelegt werden, ist Teil der Erläuterungen in Abschnitt 4.2.2.1.

Realisiert werden die Schritte der Phase Umgebungsdokumentation durch die Rollen *WfMS-Anbieter* und *Dienstanbieter*, welche, neben der Dokumentation der von ihnen angebotenen Dienste, auch für deren Betrieb verantwortlich sind. Da sich sowohl die Beschreibung von Diensten als auch der PS-Infrastruktur an existierenden Standards und Formaten orientiert, ist auch hierfür eine Verwendung existierender Werkzeuge möglich.

3.5.2.3 Deployment

Der erste Schritt der Installation eines Prozesses auf seine Ausführungsumgebung in der Phase *Deployment (D)* ist die Erzeugung seiner EWFN-Repräsentation aus der BPEL-Prozessbeschreibung (D1). Wie in Abschnitt 3.2 erläutert, beschreibt diese sowohl die Kontrollflussabhängigkeiten zwischen den PS-Klienten als auch deren Instanzdatenzugriffe explizit und kann nativ auf der PS-Infrastruktur ausgeführt werden. Die Erzeugung der EWFN-Repräsentation stellt sicher, dass der Bezug zwischen einer BPEL-Aktivität und den Elementen ihrer EWFN-Repräsentation erhalten bleibt. Wurde also in Schritt M3a und M4 während der Modellierungsphase eine Parametrisierung des Prozesses vorgenommen, so ist eine Interpretation dieser Informationen auch im EWFN-Modell des Prozesses möglich.

Die EWFN-Repräsentation eines Prozesses dient, gemeinsam mit den Informationen über seine Ausführungsumgebung (U3), als Eingabe für die Partitionierung (D2). Im Rahmen des Partitionierungsvorgangs wird jedem Element des EWFN-Modells eine Partition in dessen Ausführungsumgebung zugewiesen. Erfasst wird die Partitionierung des Prozesses (in Verbindung mit weiteren Deployment-Informationen) im sogenannten *Distributed Deployment Descriptor (DDD)*, welcher als Resultat des Partitionierungsvorgangs erzeugt wird (D3). Die Beschreibung des zur Partitionierung verwendeten Verfahrens ist Gegenstand von Abschnitt 4.4. Der DDD wiederum dient (D4), zusammen mit dem ursprünglichen BPEL-Prozess (M3c), als Eingabe für den Deployment-Schritt. Die Übergabe des BPEL-Prozesses ist notwendig, da dessen EWFN-Repräsentation zwar die Kommunikation von Kontrollflussinformationen und Instanzdaten explizit beschreibt, jedoch auf eine erschöpfende Beschreibung der Attribute der einzelnen BPEL-Aktivitäten (z. B. dem PARTNERLINK oder der OPERATION einer INVOKE-Aktivität) verzichtet und hierfür auf die jeweilige Aktivität im BPEL-Prozessmodell verweist. Im DDD erfolgt die Konfiguration der Ausführungsteilnehmer des Prozesses für die durch diese jeweils auszuführenden Verarbeitungsschritte. Sowohl der Ablauf des Deployment-Vorgangs als auch das Format des DDD werden in Abschnitt 5.8 vorgestellt. Realisiert werden die Verarbeitungsschritte der Deployment-Phase durch die Rollen *Pro-*

zesspartitionierer (engl. Process Partitioner) und *Deployer*. Zur Durchführung des Partitionierungsvorgangs steht dem Partitionierer ein entsprechendes Partitionierungswerkzeug zur Verfügung, welches das entwickelte und in Kapitel 4 vorgestellte Partitionierungsverfahren umsetzt. Die Vorstellung einer prototypischen Umsetzung des Partitionierungswerkzeugs ist Gegenstand von Abschnitt 7.3.

Nach Abschluss des, durch die *Deployer*-Rolle ausgeführten, Deployment-Vorgangs (D5) ist der Prozess bereit zu seiner Ausführung.

3.5.2.4 Ausführung und Evolution

Während der Phase *Ausführung (A)* erfolgt die Verarbeitung des Prozessmodells durch konfigurierte PS-Klienten entsprechend den Vorgaben der EWFN-Repräsentation des Prozesses. Aus Gründen einer erforderlichen Nachvollziehbarkeit des Ausführungsablaufs jeder Prozessinstanz werden während deren Ausführung durch die einzelnen Ausführungsteilnehmer Protokolldaten erfasst. Die dezentrale Natur der PS-Infrastruktur und die Eigenschaften eines verteilten Systems (beispielsweise keine zuverlässig synchronen Uhren) resultieren in bestimmten Anforderungen an sowohl den Ort an welchem die Protokolldaten erfasst werden, als auch an die während der Protokollierung erfassten Informationen. In Abschnitt 5.7 wird der Vorgang der Protokolldatenerfassung (A2) detailliert erläutert. Neben der Gewährleistung der Nachvollziehbarkeit der Prozessausführung können die erfassten Protokolldaten sowohl als Grundlage für eine Präzisierung der Parametrisierung der Prozesspartitionierung (A4c) als auch der Prozessevolution (A4a, A4b) Verwendung finden.

Unterschiedliche Gründe können eine Anpassung eines existierenden Prozesses bzw. einer Prozesskonfiguration erforderlich machen [HSB98, AJ00, Nar00, Kar06]. Beispiele hierfür sind: (i) Änderungen des Geschäftsprozesses, welcher der ausführbaren technischen Repräsentation des Prozesses zugrunde liegt, (ii) Änderungen in der Schnittstelle der vom Prozess orchestrierten Dienste oder (iii) Änderungen in der Dienstlandschaft des Prozesses, wie beispielsweise die Änderung des Endpunkts, unter welchem mit einem bestimmten Dienst interagiert werden kann. Eine zusätzliche Dimension möglicher Änderung eines

Prozesses im Fall einer dezentralen Ausführung seiner Orchestrierungslogik ist die Anpassung seiner Partitionierung. Die optionalen Verarbeitungsschritte der Phase *Evolution (E)* tragen diesen Gründen Rechnung.

Erfordern die notwendigen Änderungen eine Anpassung des Prozessmodells selbst – man spricht hierbei von den sogenannten *Changes in Goals* – so wird nach der Anpassung des Prozessmodells (A4b) ggf. eine erneute Parametrisierung des Prozessmodells sowie ein erneutes Durchlaufen sämtlicher Verarbeitungsschritte der Deployment-Phase notwendig (E2). Haben die notwendigen Änderungen lediglich Auswirkungen auf die vom Prozess orchestrierten Dienst-Implementierungen oder die Partitionierung des Prozesses – sogenannten *Changes in Business Relationships and Environment* – so genügt oft die wiederholte Abarbeitung der Partitionierung (E1) und der nachfolgenden Schritte der Deployment-Phase.

3.6 Zusammenfassung

In diesem Kapitel wurde eine Einführung in die grundlegenden Konzepte der *Executable Workflow Nets* gegeben sowie die dem Ansatz zugrunde liegende PS-Methode vorgestellt.

Als Grundlage hierfür wurden zunächst anhand der Ausführung eines Beispielprozesses all jene Informationen aufgezeigt, die innerhalb eines WfMS während der Ausführung einer Prozessinstanz benötigt werden. Hieran anknüpfend folgte die Vorstellung des EWFN-Metamodells mit seinen Elementen *Transitionen*, *Stellen*, *Marken* und *Kanten*. Durch dieses wird die Kommunikation der für die Ausführung von Prozessinstanzen notwendigen Informationen zwischen den einzelnen Ausführungsteilnehmern explizit beschrieben. Aufbauend auf diesem wurde die generelle Vorgehensweise der Realisierung der einzelnen BPEL-Aktivitäten mit den Mitteln des EWFN-Metamodells in Form der EWFN-Patterns erläutert sowie das durch das EWFN-Metamodell und die in Kapitel 5 vorgestellte PS-Infrastruktur erreichbare Verteilungsspektrum der Prozesspartitionierung aufgezeigt.

Abschließend wurde ein erweiterter Prozesslebenszyklus präsentiert, der den speziellen Anforderungen einer dezentralen Prozessausführung Rechnung

trägt und dessen einzelne Phasen, implementierende Rollen sowie jeweils verwendete Werkzeuge vorgestellt. Ein wesentlicher Aspekt hierbei ist die Einhaltung etablierter Standards und die Wiederverwendung bestehender Technologien, um weitestgehend auf existierende Werkzeuge zur Realisierung der einzelnen Verarbeitungsschritte zurückgreifen zu können.

Für den weiteren Verlauf der Arbeit können folgende relevante Kernpunkte der verteilten Prozessausführung auf Basis des EWFN-Prozessmetamodells und der PS-Infrastruktur festgehalten werden:

Ein EWFN-Modell ist ein "Ausführungsplan" für die Navigation eines Prozesses

Ein EWFN-Modell beschreibt einen Prozess durch die explizite Repräsentation sämtlicher lesender und schreibender Datenzugriffe seiner funktionalen Einheiten; dies umfasst die Weitergabe der Kontrolle über die Prozessausführung sowie die Zugriffe auf die Prozessinstanzdaten. Ein EWFN-Modell kann nativ auf der PS-Infrastruktur ausgeführt werden. Dabei gilt, dass die Stellen und Transitionen eines EWFN die atomaren Einheiten seiner Verteilung auf seine Ausführungsumgebung sind.

Abbildung von BPEL

Der Realisierung der operationalen Semantik der einzelnen BPEL-Aktivitäten erfolgt durch komponierbare *EWFN-Patterns* [Mar10].

Kommunikation erfolgt niemals direkt zwischen Aktivitätsimplementierungen

Sämtliche Kommunikation zwischen Aktivitäten erfolgt niemals direkt, sondern über PS, die die Eigenschaften räumlicher, zeitlicher und referentieller Entkopplung aufweisen. Dies gilt sowohl für die Koordination der Aktivitätsausführung als auch für von diesen gemeinsam genutzten Instanzdaten.

Breites unterstütztes Verteilungsspektrum ohne die Notwendigkeit zur Änderung des BPEL-Prozessmodells

Auf Basis des entwickelten Ansatzes und der zugehörigen Laufzeitinfra-

struktur kann ein breites Spektrum möglicher Prozesskonfigurationen – von zentralem bis hin zu maximal dezentralem Deployment – allein mittels Prozessmodell-externer Parametrisierung realisiert werden, ohne dass dies Änderungen am Prozessmodell bedingt.

KAPITEL 4

EIN VERFAHREN FÜR DIE AUTOMATISCHE PROZESSPARTITIONIERUNG

In Abschnitt 3.2 wurde das EWFN-Prozessmetamodell für die Beschreibung dezentral ausgeführter Prozesse durch Dekomposition in einzelne funktionale Einheiten, welche sich durch den Austausch von Informationen entlang explizit beschriebener virtueller Kanäle koordinieren, vorgestellt. Die Dekomposition ermöglicht die Unterstützung eines breiten Spektrums möglicher Prozesskonfigurationen, welche – entsprechend der PS-Methode (vgl. Abschnitt 3.5) – durch einen nicht-invasiven Abbildungsvorgang derart bestimmt werden können, dass dies keine Änderung des BPEL-Prozessmodells bedingt, welches als Eingabe für die BPEL-EWFN-Transformation diente (vgl. Abschnitt 3.4). Da jedoch die Ausschöpfung des maximal möglichen Verteilungsgrads im Allgemeinen (aufgrund des erforderlichen Kommunikationsaufwands) nicht sinnvoll ist, wird in diesem Kapitel ein Partitionierungsverfahren vorgestellt, das die automatische Bestimmung möglicher Prozesskonfigurationen sowie deren Bewertung

hinsichtlich eines Kostenkriteriums erlaubt.

Grundlage für die Beschreibung des entwickelten Partitionierungsverfahrens bildet die Präsentation einer Zusammenstellung von Anforderungen und Eigenschaften des betrachteten Einsatzszenarios (Abschnitt 4.1) sowie die Vorstellung der sogenannten *Partitionierungsobjekte* eines BPEL-Prozessmodells, d. h. all jener Objekte eines Prozesses und seiner Laufzeitumgebung, für die während des Partitionierungsvorgangs eine sogenannte *Partition* (ein logischer Identifikator eines Orts in der Ausführungsumgebung des Prozesses) bestimmt werden muss (Abschnitt 4.2).

Eine Reihe unterschiedlicher Einflussgrößen beeinflussen den Partitionierungsvorgang, wie beispielsweise die durch das Prozessmodell vorgegebene Ausführungsordnung der Aktivitäten und die durch den Prozess verwendeten Dienste. In Abschnitt 4.3 werden diese unterschiedlichen Einflussgrößen des Partitionierungsverfahren in Form sogenannter *Partitionierungsparameter* vorgestellt und es wird erläutert, wir die Parametrisierung unter Verwendung existierender Standards und Technologien technisch umgesetzt werden kann.

In den Abschnitten 4.4 bis 4.7 wird das entwickelte mehrphasige Partitionierungsverfahren zunächst in Form einer Gesamtübersicht aufgezeigt und anschließend die einzelnen Phasen im Detail ausgeführt.

Abschnitt 4.8 fasst die erarbeiteten Resultate zusammen und schließt damit das Kapitel ab.

4.1 Einsatzszenario

Prozesse können anhand der Dimensionen *Geschäftswert* und *Wiederholungshäufigkeit* in vier Kategorien eingeordnet werden [LR99]. Der Geschäftswert eines Prozesses spiegelt dabei seine Bedeutung für die Erfüllung des Kerngeschäfts eines Unternehmens wider, seine Wiederholungshäufigkeit die erwartete Anzahl der Instanzen eines Prozessmodells. Abbildung 4.1 verdeutlicht die Klassifikation graphisch.

Für das der Arbeit zugrunde liegende Einsatzszenario gelten die folgenden Eigenschaften und die daraus resultierenden Anforderungen an das Partitionierungsverfahren:

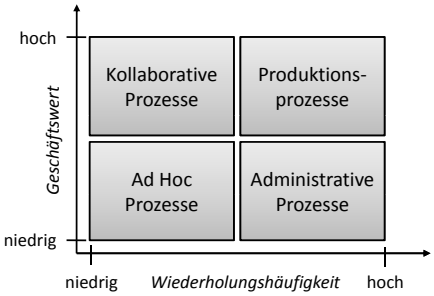


Abbildung 4.1: Klassifikation von Prozessen (aus [LR99]) anhand der Größen *Geschäftswert* und *Wiederholungshäufigkeit*.

Eigenschaft 4.1 (Produktionsprozesse)**.** *In der Terminologie der oben genannten Klassifikation ist das Ziel dieser Arbeit die verteilte Ausführung von* Produktionsprozessen *(engl.* Production Workflows*). Im Gegensatz zu anderen Arten von Prozessen zeichnen sich diese im Allgemeinen durch einen relativ hohen Geschäftswert und eine relativ große Anzahl an Instanzen desselben Modells aus. Weiterhin gilt für Produktionsprozesse, dass Änderungen am Prozessmodell, im Verhältnis zur Anzahl der Instanzen des Modells, selten sind.*

Anforderung 4.2 (Laufzeitbeeinflussung und Prognostizierbarkeit)**.** *Produktionsprozesse realisieren typischerweise komplexe Anwendungen und umfassen daher potentiell eine große Anzahl von Aktivitäten. Der hohe Geschäftswert der Produktionsprozesse motiviert den Wunsch nach effizienten und prognostizierbaren Ausführungsverhalten*[1]*. Für den verwendeten Ansatz zur Prozesspartitionierung bedeutet dies, dass ein* rein-dynamisches *Partitionierungsverfahren – aufgrund der fehlenden Prognostizierbarkeit der Ausführung – für das vorliegende Einsatzszenario nicht geeignet ist, sondern eine* a-priori-*Strategie zur Partitionierung verfolgt wird.*

[1] Entsprechend den Erläuterungen in Abschnitt 2.5.6 bezeichnet der Begriff *prognostizierbar* hier eine Ausführung eines Prozesses, dessen Laufzeitverhalten nur möglichst wenig durch Prozess-externe Faktoren (wie beispielsweise stark schwankende Verfügbarkeit der vom Prozess verwendeten Dienste oder unterschiedliche Lastverhältnisse der WfMS bzw. der Dienste) beeinflusst wird.

Anforderung 4.3 (Erhaltung der operationalen Semantik von BPEL)**.** *Durch das Partitionierungsverfahren sind eventuelle Einschränkungen zu berücksichtigen, die aus der Erhaltung der operationalen Semantik von BPEL resultieren. Die Partitionierung eines Prozesses darf demnach nicht derart vorgenommen werden, dass sich das Ausführungsverhalten eines partitionierten Prozesses von dem einer zentralen Ausführung unterscheidet.*

Hinsichtlich der Dienstumgebung der zu partitionierenden Prozesse gelten die folgenden Eigenschaften und Anforderungen:

Eigenschaft 4.4 (Dienstverfügbarkeit)**.** *In der Dienstumgebung eines Prozesses ist mindestens eine Realisierung (d. h. eine Implementierung) für jeden, vom Prozess verwendeten, Diensttyp verfügbar. Stehen mehrere funktional äquivalente Implementierungen desselben Diensttyps zur Verfügung, so können sich diese hinsichtlich ihrer nicht-funktionalen Eigenschaften (vgl. Definition 4.25) unterscheiden. Auch hinsichtlich nicht-funktionaler Eigenschaften gilt, dass zumindest eine Dienstimplementierung in der Dienstumgebung eines Prozesses die von diesem verlangten Anforderungen erfüllt.*

Anforderung 4.5 (Dienstbeschreibungen)**.** *Um ein Auffinden von Diensten zu erlauben, die zu den Anforderungen einer bestimmten Aktivität kompatibel sind, beschreiben Anbieter ihre Dienste hinsichtlich deren funktionaler sowie nicht-funktionaler Eigenschaften und machen diese Dienstbeschreibungen den Prozesspartitionierern zugänglich (vgl. Abschnitt 3.5.2.2). Die Dienstbeschreibungen müssen dabei in einem Format vorliegen, das eine maschinelle Auswertung durch das Partitionierungswerkzeug während des Partitionierungsvorgangs erlaubt.*

Eigenschaft 4.6 (Dienstdynamik)**.** *Vergleichbar den betrachteten Produktionsprozessmodellen gilt auch für die Schnittstellen der Dienste in der Dienstumgebung der Prozesse ein relativ geringer Grad an Dynamik. So ist davon auszugehen, dass sich die Schnittstelle eines angebotenen Diensts in der Regel über die Zeit der Ausführung von Instanzen eines bestimmten Prozesses hinweg nicht ändert. Sollte der Fall einer Schnittstellenänderung eintreten, so wird das jeweilige Prozessmodell durch den Modellierer angepasst, neu partitioniert und erneut auf der*

PS-Infrastruktur installiert (vgl. Erläuterung von Prozessevolution in Abschnitt 3.5.1).

Änderungen der Adresse einer verwendeten Dienstimplementierung bei konstanter Schnittstelle können hingegen – durch Anwendung der Vorgehensweise des sogenannten Dynamic Binding *(beispielsweise durch Zuweisung einer* EndpointReference *auf einen* PartnerLink *oder die Abwicklung der Interaktion durch einen ESB) und der Verwendung entfernter Dienstaufrufe (vgl. Abbildung 3.9(c) in Abschnitt 3.4) – kompensiert werden, ohne dass dies Änderungen am Prozessmodell oder dessen Partitionierung erfordert.*

4.2 Partitionierungsobjekte und deren Einfluss auf den Partitionierungsvorgang

Ziel des Partitionierungsvorgangs ist es, die sogenannten *Partitionierungsobjekte* eines Prozesses auf die zur Verfügung stehende *Ausführungsumgebung* zu verteilen. Dies geschieht, indem jedem Partitionierungsobjekt eine sogenannte *Partition* zugewiesen wird. Dabei wird zugunsten einer einfachen Beschreibung des Partitionierungsverfahrens eine Partition in diesem Kapitel lediglich als logischer Bezeichner verstanden, der einen Ort in der Ausführungsumgebung eines Prozesses identifiziert; die Abbildung dieses Bezeichners auf die Elemente der PS-Infrastruktur wird in Abschnitt 5.2 erläutert. Eine Partition beschreibt somit einen logischen Verbund eines oder mehrerer Partitionierungsobjekte, welche zur Ausführungszeit einer Instanz des Prozessmodells gemeinsam an einem bestimmten "Ort" verarbeitet werden. Die Ausführungsumgebung eines Prozesses umfasst dabei auf Infrastrukturebene entsprechend der Beschreibung in Abschnitt 3.1: (i) die Menge der PS-Klienten, die für die Realisierung der Funktionen der einzelnen BPEL-Aktivitäten des Prozesses verwendet werden, (ii) die Menge der PS, die als Nachrichtenpuffer für die Kommunikation zwischen PS-Klienten verwendet werden und (iii) die Menge der dokumentierten Dienstimplementierungen (vgl. Abschnitt 3.1 und Abschnitt 3.3). Für die Interaktionen zwischen PS-Klienten und PS gilt die Annahme, dass diese innerhalb einer Partition (aufgrund ihrer geographischen und administrati-

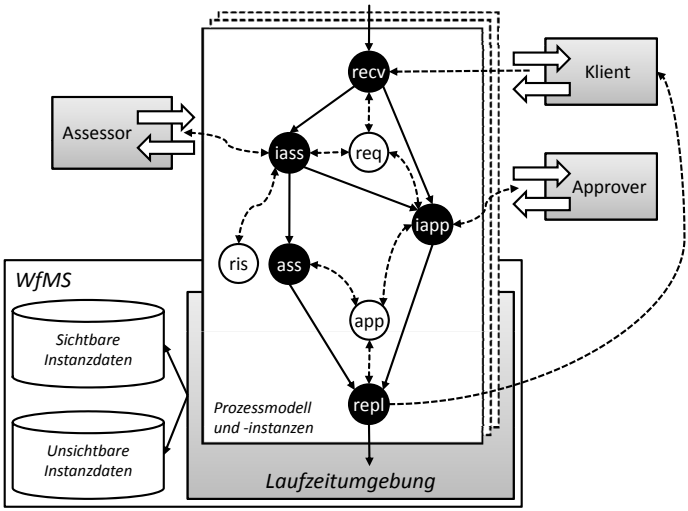


Abbildung 4.2: Partitionierungsobjekte eines BPEL-Prozesses.

ven Nähe) relativ effizient realisierbar und demnach kostengünstiger sind als partitionsübergreifende Interaktionen.

Zur Verdeutlichung der betrachteten Partitionierungsobjekte zeigt Abbildung 4.2 die schematische Darstellung eines WfMS mit einem installierten Prozessmodell und mehreren in Ausführung befindlichen Instanzen des Modells. Die Darstellung des Prozesses verwendet die in Abschnitt 3.1 vorgestellte Notation.

Während der Ausführung einer Prozessinstanz interagiert das WfMS mit einem *Klienten* und zwei WfMS-externen Diensten *Assessor* und *Approver*. Anhand der Abbildung können die folgenden Klassen von Partitionierungsobjekten erkannt werden: *Klienten*, *Dienste*, *Aktivitäten* und "sichtbare" und "unsichtbare" Instanzdaten. Die Begriffe sichtbar und unsichtbar sind hierbei abhängig davon gewählt, ob ein entsprechendes Instanzdatum im Prozessmodell explizit deklariert und verwendet wird, in diesem also "sichtbar" ist. *Sichtbare* Instanzdaten in BPEL sind *Variablen*, *Correlation Sets* und *PartnerLinks*. *Unsichtbare* Instanzdaten hingegen sind Instanzdaten, die auf Ebene des Prozessmodells

nicht explizit sichtbar sind, sondern während der Laufzeit einer Prozessinstanz durch das WfMS erzeugt und durch dieses verwaltet werden. Beispiele für unsichtbare Instanzdaten sind (i) Informationen über laufende *Message Exchanges*, (ii) der interne Zustand einer in Ausführung befindlichen SCOPE-Aktivität oder (iii) Daten für die Realisierung gegenseitigen Ausschlusses der Ausführung von Aktivitäten in beispielsweise einem *Isolated Scope*.

Die einzelnen Klassen werden nachfolgend mit ihrem Einfluss auf das Partitionierungsverfahren vorgestellt.

4.2.1 Klienten

Klienten (engl. *Clients*) bezeichnen Interaktionspartner des Prozesses bzw. des WfMS, die eine Interaktion zwischen sich und dem WfMS initiieren. Dies kann entweder mit dem Ziel geschehen, einen Prozess zu instantiieren oder mit einer bereits in Ausführung befindlichen Prozessinstanz zu interagieren. Auf Prozessebene kann die Interaktion zwischen einem Klienten und dem WfMS auf drei unterschiedliche Arten repräsentiert werden: (i) eine RECEIVE-Aktivität oder eine ONMESSAGE-Aktivität in entweder (ii) einem EVENTHANDLER oder (iii) einer PICK-Aktivität. Die Attribute dieser Aktivitäten beschreiben die Kombination aus PARTNERLINK, PORTTYPE und OPERATION, die durch die jeweilige Aktivität bedient wird sowie die VARIABLE, in welcher die Nutzdaten der empfangenen Nachricht abgelegt werden.

Die Übermittlung einer Nachricht an das WfMS durch einen Klienten bedingt, dass dem jeweiligen Klient die Adresse (bzw. der *Endpunkt*) bekannt ist, auf welcher das WfMS eingehende Nachrichten zur Verarbeitung empfängt. Im Fall existierender (zentraler) WfMS zur Ausführung von BPEL, z. B. den in Abschnitt 2.3.2 vorgestellten BPEL-WfMS, erfolgt dies in der Regel im Deployment-Deskriptor des Prozesses mittels statischer Vorgabe durch einen Benutzer. Diese Vorgehensweise wird in der PS-Infrastruktur übernommen, woraus Anforderung 4.7 an das entwickelte Partitionierungsverfahren folgt.

Anforderung 4.7 (Manuelle Partitionsvorgabe für Aktivitäten, die eingehende Nachrichten verarbeiten)**.** *Um Klienten einen definierten Endpunkt für das Senden von Nachrichten an "empfangende" Interaktionsaktivitäten eines Prozesses zu*

bieten, muss das Partitionierungsverfahren die Möglichkeit vorsehen, Aktivitäten einer bestimmten, von einem Benutzer vorgegebenen, Partition zuzuordnen.

BPEL unterstützt zwei Modi für die Interaktion zwischen einem Klienten und einer Prozessinstanz; man bezeichnet diese durch die Begriffe *synchron* und *asynchron*. Im synchronen Interaktionsmodus gilt, dass eine durch das WfMS empfangene Anfragenachricht in Form einer REPLY-Aktivität "beantwortet" wird. Im Fall eines asynchronen Interaktionsmodus hingegen geschieht dies durch eine INVOKE-Aktivität; Empfangen der Anfragenachricht und Senden der Antwortnachricht sind hier also zwei voneinander unabhängige Aktionen.

BPEL erlaubt die Verwendung unterschiedlicher – sowohl synchroner als auch asynchroner – Transportprotokolle zur Realisierung der Interaktion zwischen WfMS und Klienten bzw. Diensten. Synchrone Transportprotokolle zeichnen sich im Gegensatz zu asynchronen Transportprotokollen dadurch aus, dass sie für die Übermittlung der Antwortnachricht zwischen WfMS und Klient dieselbe Netzwerkverbindung verwenden können, die auch für die Übermittlung der Anfragenachricht verwendet wurde. Erfolgt die Realisierung einer synchronen Interaktion unter Verwendung eines synchronen Kommunikationsprotokolls (wie beispielsweise HTTP [FGM+99] in einem synchronen Kommunikationsmodus[1]), so kann dies Auswirkungen auf die Partitionierung zusammengehöriger RECEIVE/REPLY-Paare haben.

Abbildung 4.3 zeigt ein entsprechendes Beispiel, anhand welchem das entstehende Problem verdeutlicht werden kann. Die Abbildung zeigt eine Interaktion zwischen Klient und WfMS, in welcher RECEIVE- und REPLY-Aktivität auf zwei unterschiedliche Partitionen und damit zwei unterschiedliche Ausführungsteilnehmer auf unterschiedlichen physikalischen Rechnern verteilt sind.

Durch die Verwendung eines synchronen Transportprotokolls müsste für die Übersendung der von P_2, als Teil der Ausführung der REPLY-Aktivität, erzeugten Antwortnachricht derselbe Kanal (d. h. die zwischen Klient und P_1 bestehende Netzwerkverbindung) verwendet werden, der auch für den Empfang der An-

[1] Durch das Setzen des WS-Addressing REPLYTO-Header-Felds in der Anfragenachricht kann beispielsweise HTTP auch in einem asynchronen Kommunikationsmodus verwendet werden. In diesem Fall baut das WfMS zum Versenden der Antwortnachricht eine eigene Verbindung zum Klienten auf.

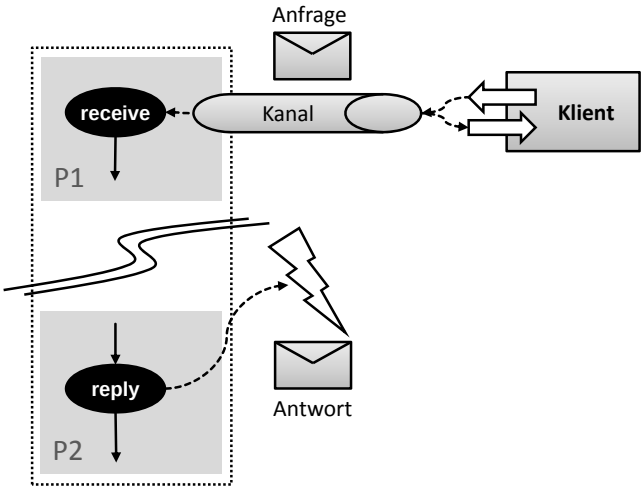


Abbildung 4.3: Beispiel einer nicht erlaubten Partitionierung bei synchroner Interaktion zwischen Klient und WfMS.

fragenachricht durch die RECEIVE-Aktivität bei P_1 verwendet wurde. Da diese Aktivitäten unterschiedlichen Partitionen zugeordnet sind (und damit insbesondere gilt, dass sie auf unterschiedlichen Rechnern bei unterschiedlichen Ausführungsteilnehmern ausgeführt werden), ist eine Wiederverwendung der vom Klienten aufgebauten Netzwerkverbindung zum Übersenden der Antwortnachricht im dargestellten Fall nicht möglich. Die abgebildete Partitionierung ist demnach nicht zulässig und ist durch das entwickelte Partitionierungsverfahren auszuschließen.

Anforderung 4.8 (RECEIVE/REPLY-Paare)**.** *Um entsprechend Anforderung 4.3 – unabhängig vom jeweils für eine Interaktion verwendeten Kommunikationsprotokoll – eine Einhaltung der operationalen Semantik von BPEL zu gewährleisten, sind zusammengehörige* RECEIVE/REPLY*-Paare durch das Partitionierungsverfahren stets derselben Partition zuzuordnen.*

4.2.2 Dienste

Dienste sind neben Klienten weitere Interaktionspartner eines Prozesses; sie repräsentieren die vom WfMS während der Ausführung eines Prozesses verwendeten Geschäftsfunktionen in Form von Web-Services. Die Interaktion mit Diensten erfolgt in BPEL durch die INVOKE-Aktivität. Die Attribute der INVOKE-Aktivität spezifizieren neben dem Ziel des Dienstaufrufs auch die Variablen, die zum einen den Wert beinhalten, der als Eingabe für den Dienstaufruf verwendet wird, zum anderen den vom Dienst erhaltenen Rückgabewert aufnehmen.

Eigenschaft 4.9 (INVOKE-Aktivitäten "nahe" zu den verwendeten Diensten)**.** *Kernidee der in Abschnitt 3.1 erläuterten Vorgehensweise zur dezentralen Ausführung von Prozessen ist die Verteilung der Orchestrierungslogik eines Prozesses zu denjenigen Ausführungsteilnehmern, die auch eine oder mehrere während der Ausführung des Prozesses verwendete Dienstimplementierungen bereitstellen. Demzufolge gilt für die Partitionierung von* INVOKE*-Aktivitäten das generelle Vorgehen, die Aktivität einer Partition zuzuordnen, in welcher auch eine Dienstimplementierung bereitstellt wird, die die Anforderungen der jeweiligen Aktivität erfüllt*[1].

Die Dienste, mit welchen das WfMS – ausgelöst durch eine INVOKE-Aktivität – interagiert, sind durch funktionale sowie nicht-funktionale Eigenschaften charakterisiert, welche den Partitionierungsvorgang beeinflussen.

Funktionale Eigenschaften eines Diensts beschreiben seine Schnittstelle und damit die von diesem angebotenen Operationen; im Umfeld der Web-Service-Technologien werden sie durch die *Web Service Description Language (WSDL)* beschrieben; die wesentlichen Konzepte von WSDL wurden in Abschnitt 2.1.1 einführend dargestellt. Eine INVOKE-Aktivität (Abbildung 4.4) spezifiziert durch ihre Attribute PORTTYPE und OPERATION im Prozessmodell die funktionalen Anforderungen an die (abstrakte) Schnittstelle des Diensts, welche durch die jeweilige Aktivität aufgerufen wird. Durch das konkrete Deployment des Prozesses wird bestimmt, welche Dienstimplementierung zur Laufzeit des Prozesses

[1] Entsprechend den Erläuterungen in Abschnitt 3.4 und der Möglichkeit zum Aufruf entfernter Dienste, gilt allerdings dass auch andere Partitionierungen von INVOKE-Aktivitäten möglich sind.

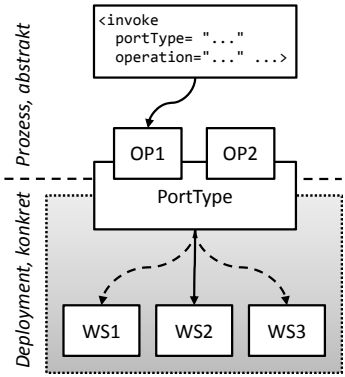


Abbildung 4.4: Zweistufiges *Binden* von Diensten bzw. Dienstimplementierungen in BPEL.

für die Interaktion verwendet werden soll. Bietet die Dienstumgebung eines Prozesses mehrere Implementierungen derselben Dienstschnittstelle, so wird im Rahmen der Prozesspartitionierung auf Grundlage nicht-funktionaler Diensteigenschaften bestimmt, welche dieser Dienstimplementierungen während der Ausführung des Prozesses verwendet wird.

Nicht-funktionale Diensteigenschaften beschreiben, mit welcher Dienstgüte (z. B. hinsichtlich Verfügbarkeit, Verlässlichkeit, Sicherheit) ein Dienst von einem Dienstanbieter zur Verfügung gestellt wird. Dabei ist es möglich, dass ein Dienst mit denselben funktionalen Eigenschaften von unterschiedlichen Dienstanbietern mit unterschiedlichen nicht-funktionalen Eigenschaften angeboten wird. In der Familie der Web-Service-Standards wird für die Beschreibung nicht-funktionaler Diensteigenschaften der Standard *WS-Policy* [OHV+] verwendet. Eine kurze Einführung von WS-Policy ist Teil von Abschnitt 2.1.2.

Eine besondere Form nicht-funktionaler Diensteigenschaften sind Eigenschaften, die im weiteren Verlauf der Arbeit als *organisatorische Eigenschaften* bezeichnet werden. Organisatorische Diensteigenschaften machen eine Aussage über den Betreiber eines Diensts. Aus Gründen existierender Verträge zwischen Geschäftspartnern kann es beispielsweise notwendig sein, dass ein

bestimmter Dienst immer bei einem bestimmten Dienstanbieter in Anspruch genommen werden muss, selbst wenn andere Dienstanbieter weitere funktional und nicht-funktional kompatible Implementierungen des Diensts anbieten.

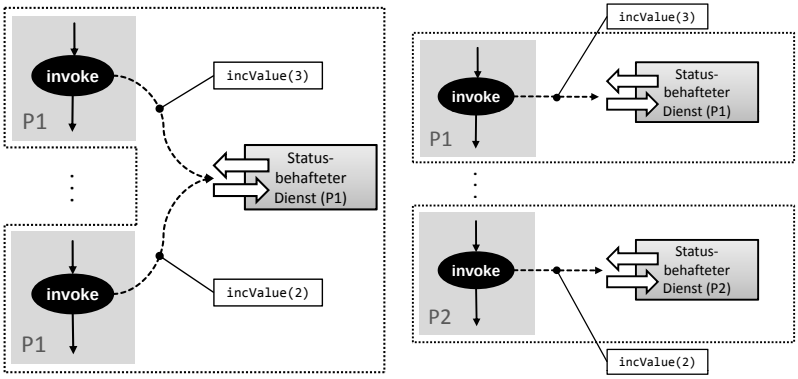
Aufgrund der oben genannten Eigenschaften gelten für die Partitionierung von INVOKE-Aktivitäten, zusätzlich zu Eigenschaft 4.9, die folgenden Kriterien:

Anforderung 4.10 (Berücksichtigung sowohl funktionaler, als auch nicht–funktionaler Diensteigenschaften)**.** *Die Bestimmung der möglichen Dienstimplementierungen, die als Ziel einer* INVOKE*-Aktivität in Frage kommen, erfolgt auf Grundlage der Kompatibilität der funktionalen und nicht-funktionalen Anforderungen der jeweiligen Aktivität bzw. der Eigenschaften der dokumentierten Dienste. Die Berücksichtigung organisatorischer Diensteigenschaften erfolgt in Form* statischer Partitionsvorgaben*, die auch zur Partitionierung von* RECEIVE*-Aktivitäten verwendet werden (Anforderung 4.7).*

Anforderung 4.11 (Minimierung des Partitionierungsgrads eines Prozesses)**.** *Existieren in der Dienstumgebung eines Prozesses mehrere Dienstimplementierungen, die sowohl die erforderlichen funktionalen als auch nicht-funktionalen Anforderungen erfüllen, so soll – aus Gründen der Minimierung potentiell teurer partitionsübergreifender Kommunikation – die Partitionierung derart erfolgen, dass die Anzahl unterschiedlicher Partitionen minimiert wird.*

Ein PARTNERLINK beschreibt in BPEL einen Kanal zwischen dem WfMS und einem Interaktionspartner. Mehrere Interaktionen über denselben PARTNERLINK müssen also im Allgemeinen in einer wiederholten Kommunikation mit demselben Interaktionspartner resultieren.

Die Notwendigkeit für dieses Verhalten lässt sich am Beispiel sogenannter *zustandsbehafteter Dienste* erläutern. Im Gegensatz zu zustandslosen Diensten halten zustandsbehaftete Dienste einen, durch eine Interaktion erlangten, internen Zustand auch nach dem Ende der Interaktion. Nachfolgende Interaktionen mit dem Dienst erfolgen ausgehend vom davor erlangten Zustand und nicht vom ursprünglichen Startzustand des jeweiligen Diensts. Abbildung 4.5 zeigt einen Prozess mit zwei INVOKE-Aktivitäten, die eine Interaktion über denselben PARTNERLINK ausführen. In der Dienstumgebung des Prozesses



(a) Eine Dienstimplementierung. (b) Mehrere Dienstimplementierungen.

Abbildung 4.5: Interaktionen mit zustandsbehafteten Diensten.

existieren zwei Dienstimplementierungen (bei P_1 und P_2), die sowohl die funktionalen als auch die nicht-funktionalen Anforderungen erfüllen, die mit der Ausführung der INVOKE-Aktivität verbunden sind. Im Beispiel realisieren die INVOKE-Aktivitäten Interaktionen mit einem zustandsbehafteten Zähler. Beim Start des Diensts erhält der Zähler den Wert 0, bei nachfolgenden Aufrufen der Funktion `incValue()` wird der interne Zähler des Diensts um den Wert erhöht, der als Parameter des Dienstaufrufs angegeben wurde.

Auf Grundlage der bisher erläuterten Kriterien des Partitionierungsvorgangs ist prinzipiell eine Verteilung der beiden Dienstaufrufe auf P_1 und P_2 möglich (und im Fall eines zustandslosen Diensts auch korrekt). Im Fall eines zustandsbehafteten Diensts führt diese, in Abbildung 4.5(b) dargestellte, Partitionierung zu einem unterschiedlichen Zählerstand nach Ausführung der zweiten INVOKE-Aktivität (3 für den von P_1 und 2 für den von P_2 betriebenen Dienst) als bei der in Abbildung 4.5(a) dargestellten Partitionierung, in der für beide INVOKE-Aktivitäten dieselbe Dienstimplementierung verwendet wird (5).

Anforderung 4.12 (Verarbeitung zustandsbehafteter Dienste)**.** *Um (entsprechend Anforderung 4.3) eine korrekte Orchestrierung zustandsbehafteter Dienste*

zu gewährleisten, gilt das Standardverhalten, dass mehrere INVOKE*-Aktivitäten, die denselben* PARTNERLINK *nutzen, derselben Partition zugeordnet werden. Für den Fall, dass die jeweilige* INVOKE*-Aktivität allerdings eine Interaktion mit einem zustandslosen Dienst repräsentiert, soll ein selektiver Verzicht auf die oben genannte Eigenschaft zugunsten einer flexibleren Partitionierung möglich sein. Auch hierbei gilt, vergleichbar Eigenschaft 4.6, dass die, durch die Sprachmittel von BPEL geschaffene, Möglichkeit der dynamischen Änderung einer verwendeten Dienstimplementierung zur Laufzeit einer Prozessinstanz (durch Zuweisung einer* ENDPOINTREFERENCE *auf einen* PARTNERLINK*) erhalten bleibt.*

4.2.2.1 Veröffentlichung von Dienstbeschreibungen

Um ein Auffinden angebotener Dienste zu erlauben, veröffentlichen Dienstanbieter (entsprechend Eigenschaft 4.5) Beschreibungen ihrer Dienste in einem, den Partitionierern zugänglichen Dienstverzeichnis, einer sogenannten *Service Registry*. Das Dienstverzeichnis dient als Datenquelle für das Prozesspartitionierungswerkzeug und wird während der Prozesspartitionierung durch dieses abgefragt. Konzeptuell ist das entwickelte Verfahren zur Prozesspartitionierung nicht an die Verwendung einer bestimmten Service-Registry-Schnittstelle und -Technologie gebunden. Nachfolgend wird exemplarisch eine Umsetzung unter Verwendung einer *UDDI-Web-Service-Registry* vorgestellt. Der Standard *UDDI* [oas04] wurde aus Gründen seiner Reife und Verbreitung als Beispiel für die nachfolgende Diskussion gewählt. Eine Einführung von UDDI und seinen Konzepten ist Teil von Abschnitt 2.1.3.

Basis für die Beschreibung der Diensteigenschaften bilden die UDDI-Datenmodellelemente BUSINESSSERVICE und BUSINESSENTITY, welche (i) durch die BUSINESSSERVICE-Referenzen einer BUSINESSENTITY die einzelnen Dienste einem Dienstanbieter zuordnen. (ii) Ein Dienstanbieter kann durch die CONTACT-Referenz näher beschrieben werden. (iii) Sowohl BUSINESSENTITY als auch BUSINESSSERVICE können durch die TMODEL-Referenzen im CATEGORYBAG-Attribut in beliebiger Form klassifiziert werden.

Um eine maschinelle Registrierung und Abfrage funktionaler Diensteigenschaften in einer UDDI-Registry zu ermöglichen, definieren [oasa] und [oasb]

Abbildungen der durch ein WSDL-Dokument beschriebenen Informationen auf die UDDI-Datenstruktur. Das in [oasa] beschriebene Verfahren beschränkt sich auf eine Verknüpfung eines Diensts bzw. seiner UDDI-Repräsentation BusinessService mit seinem WSDL-Dokument, überführt die durch das WSDL-Dokument beschriebenen Informationen allerdings selbst nicht in die UDDI-Registry. Dies hat zur Folge, dass Informationen, die Teil der WSDL-Beschreibung eines Diensts sind (z. B. die von diesem implementierten PortType-Elemente) nicht für das Auffinden des Diensts in der Registry verwendet werden können, was eine Voraussetzung für die Lösung des vorliegenden Problems ist.

Im Unterschied hierzu beschreibt das in [oasb] vorgestellte Verfahren eine Abbildung der Informationen eines WSDL-Dokuments bis zur Ebene einzelner PortType Elemente, was zur Lösung des vorliegenden Problems des Findens einer funktional kompatiblen Dienstimplementierung ausreichend ist; Abbildung 4.6 verdeutlicht dies graphisch.

Ein Dienst (bzw. sein WSDL-Element Service) wird auf das UDDI-Element BusinessService abgebildet. Die Kategorisierung des BusinessService als WSDL-Service erfolgt mittels einer entsprechenden KeyedReference im CategoryBag des BusinessService. Die zur Kategorisierung verwendeten *tModel-Key*-UUIDs sind in [oasb] im Einzelnen aufgeführt.

Die BindingTemplate-Elemente des BusinessService repräsentieren einzelne Implementierungen des Diensts und definieren durch das Element AccessPoint die Adresse, über die mit der beschriebenen Dienstimplementierung kommuniziert werden kann. Weiterhin referenzieren sie im Rahmen der tModelInstanceDetails die tModel-Elemente *Binding* und *PortType*.

Das tModel PortType repräsentiert ein durch Namensraum und lokalen Bezeichner benannten WSDL-PortType. Die Kategorisierung des tModel erfolgt, entsprechend der Vorgehensweise zur Kategorisierung eines WSDL-Service durch eine entsprechende KeyedReference.

Analog repräsentiert das tModel Binding ein WSDL-Binding und beinhaltet weitere Informationen hinsichtlich des unterstützten Nachrichtenformats und Transportprotokolls der jeweiligen Dienstimplementierung.

Ein Verfahren, wie die Beschreibung der nicht-funktionalen Eigenschaften eines Diensts mit der Beschreibung seiner funktionalen Eigenschaften in Verbin-

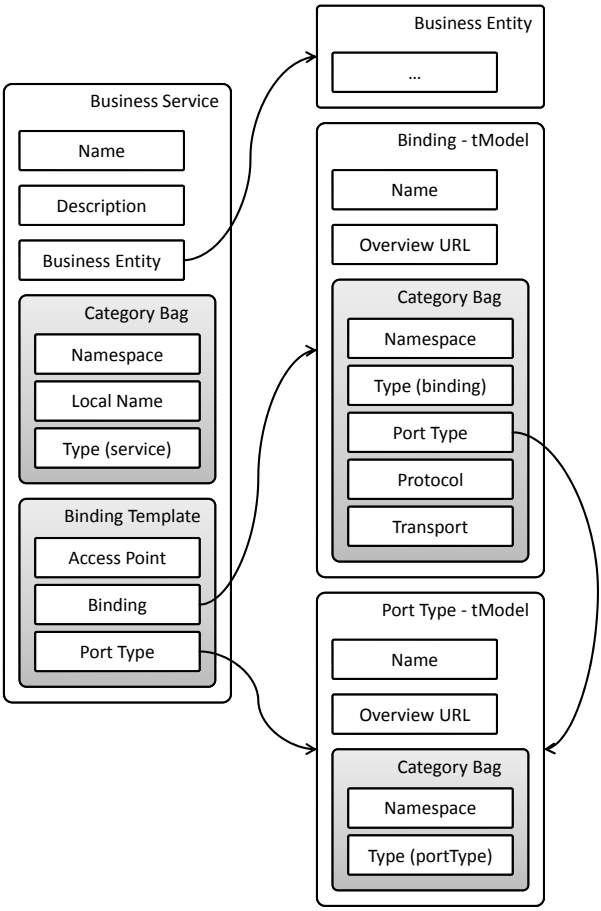


Abbildung 4.6: Abbildung von WSDL-1.1-Konstrukten auf Konzepte des *UDDI Information Models* (basierend auf [oasb]).

dung gebracht werden kann, wird in der *WS-Policy-Attachment*-Spezifikation [YBV+07] vorgestellt. Für die Annotation von Dienstbeschreibungen, welche in einer UDDI-Registry abgelegt wurden, gilt: erlaubt das zu annotierende Element des UDDI Datenmodells eine Kategorisierung durch das CategoryBag-Element,

so kann die URI des mit dem jeweiligen Element zu verknüpfenden Policy-Dokuments im KEYVALUE einer entsprechenden KEYEDREFERENCE referenziert werden. Eine Ausnahme bildet die Annotation des BINDINGTEMPLATE-Elements, da dieses keine Kategorisierung erlaubt. Hier erfolgt die Annotation durch eine geeignete Definition der TMODELINSTANCEDETAILS des BINDINGTEMPLATE.

4.2.3 Instanzdaten

Alle *Structured Activities* sowie alle *Basic Activities* außer EMPTY und EXIT konsumieren oder produzieren während der Ausführung ihrer Funktionalität potentiell *sichtbare* und/oder *unsichtbare* Instanzdaten.

Anforderung 4.13 (Berücksichtigung von Instanzdatenzugriffen)**.** *Instanzdatenzugriffe (insbesondere Variablenzugriffe) können die Übermittlung potentiell umfangreicher Datenmengen zwischen PS-Klienten unterschiedlicher Partitionen zur Folge haben. Um einen negativen Einfluss der Instanzdatenzugriffe auf das Laufzeitverhalten eines Prozesses möglichst weitgehend zu minimieren, ist eine Berücksichtigung der Kosten für lesende und schreibende Zugriffe auf (sowohl sichtbare als auch unsichtbare) Instanzdaten durch das Partitionierungsverfahren erforderlich.*

Die Berücksichtigung der Instanzdatenzugriffskosten erfolgt auf Grundlage von Informationen über die erwarteten (durchschnittlichen) Kosten der einzelnen Instanzdatenzugriffe zur Ausführungszeit. Eine Gewinnung dieser Informationen, so dass diese zum Partitionierungszeitpunkt (der der eigentlichen Prozessausführung gemäß Eigenschaft 4.2 vorgelagert ist) verfügbar sind, kann auf unterschiedliche Arten erfolgen; zwei Ansätze werden nachfolgend erläutert. Die, in Abschnitt 7.3 vorgestellte, prototypische Umsetzung des Partitionierungsverfahrens fokussiert sich dabei auf das zweiter Verfahren.

Der Verlauf der Ausführung von Produktionsprozessinstanzen wird, motiviert durch die Notwendigkeit der Nachvollziehbarkeit des Ausführungsablaufs, in Form von Protokolldaten durch das WfMS dokumentiert. Die Protokolldaten schließen neben der Erfassung der ausgeführten PS-Operationen auch die während dieser kommunizierten Daten ein (vgl. hierzu die Erläuterung der

Protokolldatenerfassung in Abschnitt 5.7). Durch eine Analyse der angefallenen Ausführungsprotokolle kann die Größe der Instanzvariablen während lesenden oder schreibenden Zugriffs gemittelt über eine Menge von Prozessinstanzen bestimmt werden. In Abschnitt 5.7.3 wird anhand der Ausführungswahrscheinlichkeit von Aktivitäten erläutert, wie eine Auswertung der erfassten Protokolldaten erfolgen kann; eine Bestimmung der durchschnittlichen Größe eines Instanzdatenzugriffs kann auf demselben Weg erfolgen.

Soll eine Erstpartitionierung eines Prozesses durchgeführt werden oder liegen keine Protokolldaten abgelaufener Prozessinstanzen vor, so können die entsprechenden Informationen über die Kosten eines Instanzdatenzugriffs durch einen Benutzer vor der Partitionierung abgeschätzt und der Prozess mit diesen geeignet annotiert werden; der Vorgang der Annotation wird in Abschnitt 4.3 erläutert.

4.2.4 Aktivitäten

Die Aktivitäten eines Prozesses repräsentieren die Verarbeitungsschritte, die durch das WfMS während der Ausführung von dessen Instanzen durchgeführt werden. Die Ausführungsreihenfolge der einzelnen Aktivitäten wird in BPEL durch sogenannte Kontrollflussabhängigkeiten definiert. Sind die Kontrollflussabhängigkeiten einer Aktivität erfüllt, so ist sie zur Ausführung durch das WfMS bereit.

Die Kontrollflussabhängigkeiten einer Aktivität und der bisherige Verlauf einer Prozessinstanz bestimmen somit, ob eine Aktivität im weiteren Verlauf einer Prozessinstanzausführung zur Ausführung kommt oder nicht. Laut der BPEL-Spezifikation ist eine Aktivität dann ausführungsbereit, wenn die folgenden Bedingungen gelten:

(i) Die die Aktivität umgebende *Structured Activity* wird ausgeführt.

(ii) Die strukturellen Eigenschaften der umgebenden *Structured Activity* erlauben die Ausführung der Aktivität. Beispielsweise muss in einer IF-Aktivität die Bedingung des jeweiligen Zweigs des IF zu einem Booleschen `true` evaluieren; Aktivitäten in einem Zweig, dessen Bedingung zu einem Booleschen `false` evaluieren kommen nicht zur Ausführung.

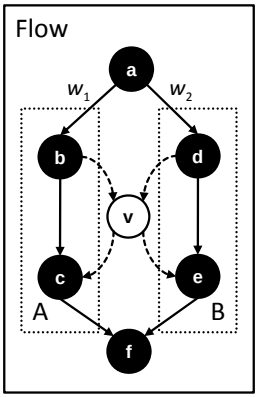


Abbildung 4.7: Einfluss der Ausführungswahrscheinlichkeit einer Aktivität auf den Partitionierungsvorgang.

(iii) Die JoinCondition der Aktivität, ein Boolescher Ausdruck über den Status eingehenden Synchronisationskanten, muss zum Wert `true` evaluieren. Der Status einer Synchronisationskante wird bei Fertigstellung der Ausführung ihrer Quell-Aktivität durch das Resultat der Evaluierung ihres TransitionCondition-Attributs bestimmt.

Anforderung 4.14 (Ausführungswahrscheinlichkeit einer Aktivität)**.** *Die Ausführung der Anwendungslogik einer bestimmten Aktivität, z. B. die Manipulation eines Variablenwerts durch* Assign *oder die Interaktion mit einem WFMS-externen Dienst durch* Invoke*, ist in der Regel mit einem Instanzdatenzugriff verbunden. Dieser ist, entsprechend der Erläuterungen in Abschnitt 4.2.3, eine Einflussgröße des Partitionierungsvorgangs. Da dieser Instanzdatenzugriff allerdings nur dann durchgeführt wird, wenn die jeweilige Aktivität auch zur Ausführung kommt, hat die Ausführungswahrscheinlichkeit einer Aktivität direkten Einfluss auf den Partitionierungsvorgang.*

Abbildung 4.7 veranschaulicht diesen Einfluss anhand eines Beispiels graphisch. Die Aktivitäten der Zweige b, c und d, e des Prozesses greifen jeweils schreibend bzw. lesend auf die Variable v zu. Für das Beispiel wird angenom-

men, dass die Voraussetzungen für die Ausführung der abgebildeten FLOW-Aktivität selbst erfüllt sind und dass sämtliche Zugriffe auf die Variable v eine Übermittlung derselben Datenmenge erfordern. Die beiden ausgehenden Synchronisationskanten von a sind mit unterschiedlichen TRANSITIONCONDITION-Attributen belegt; für die restlichen Synchronisationskanten des dargestellten Prozessausschnitts ist keine explizite TRANSITIONCONDITION festgelegt. Weiterhin gilt, dass für die einzelnen abgebildeten Aktivitäten keine explizite JOINCONDITION definiert ist und dementsprechend die Standard-JOINCONDITION, d. h. die Disjunktion der Status sämtlicher eingehender Synchronisationskanten, gilt.

Die Wahrscheinlichkeit, dass die TRANSITIONCONDITION einer von a ausgehenden Synchronisationskante – gemittelt über eine Anzahl von Prozessinstanzen – zu einem Booleschen `true` evaluiert ist mit w_1 bzw. w_2 angegeben. Aufgrund der definierten JOINCONDITION der Ziel-Aktivitäten dieser Synchronisationskanten entspricht dies auch der Wahrscheinlichkeit, dass b bzw. d zur Ausführung kommen. Gilt nun $w_2 > w_1$, so erfolgt die Ausführung von Aktivitäten d, e und die damit verbundenen Variablenzugriffe mit höherer Wahrscheinlichkeit als die Variablenzugriffe von b, c. Gelten keine anderen Einflussgrößen hinsichtlich der Partitionierung von v oder heben sich diese auf, so ist es im dargestellten Szenario sinnvoller, Variable v der Partition B als der Partition A zuzuordnen, da dies – wiederum über mehrere Instanzausführungen gemittelt – zu weniger partitionsübergreifender Kommunikation führt[1].

Da BPEL (i) die Verwendung beliebiger Sprachen zur Definition einer TRANSITIONCONDITION erlaubt und das Resultat deren Evaluierung (ii) abhängig vom aktuellen Wert der in der TRANSITIONCONDITION verwendeten Variablen zum Evaluierungszeitpunkt ist, ist eine automatische Bestimmung der Wahrscheinlichkeit eines positiven LINK-Status anhand des Prozessmodells im Allgemeinen nicht möglich. Für die Bestimmung dieser Wahrscheinlichkeiten können somit prinzipiell dieselben Verfahrensweisen zum Einsatz kommen, die auch für die

[1] Die hier gewählte Erläuterung ist lediglich exemplarischer Natur und stellt eine starke Vereinfachung der angewendeten Verfahrensweise dar. Tatsächlich wird die Ausführungswahrscheinlichkeit einer Aktivität, abhängig von der Ausführungswahrscheinlichkeit aller ihrer Eltern- bzw. Vorgängeraktivitäten durch rekursiven Aufstieg bestimmt (vgl. Abschnitt 4.7.2).

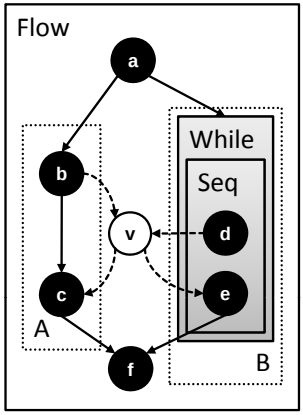


Abbildung 4.8: Einfluss der Ausführungsanzahl einer Aktivität auf den Partitionierungsvorgang.

Bestimmung der zu erwarteten Kosten eines Variablenzugriffs Verwendung finden (d. h. Auswertung von Protokolldaten bereits ausgeführter Instanzen oder manuelle Vorgabe durch einen Benutzer).

Anforderung 4.15 (Wiederholungshäufigkeit einer Aktivität)**.** *Eine weitere zu berücksichtigende Einflussgröße des Partitionierungsvorgangs ist die Anzahl der Ausführungen von Aktivitäten, ausgelöst beispielsweise durch die Aktivitäten* WHILE *bzw.* FOREACH *oder durch die mehrfache (asynchrone) Ausführung eines* EVENTHANDLER*-Konstrukts. Hinsichtlich der Verfügbarkeit von Informationen zur erwarteten Ausführungsanzahl einer Iterationsaktivität gelten dieselben Aussagen wie in Anforderung 4.13.*

Abbildung 4.8 veranschaulicht den Einfluss der erwarteten Anzahl von Ausführungen einer Gruppe von Aktivitäten auf den Partitionierungsvorgang. Das abgebildete Szenario folgt einem ähnlichen Aufbau wie das in Abbildung 4.7 dargestellte; allerdings sind in diesem Fall die von *a* ausgehenden Synchronisationskanten nicht mit TRANSITIONCONDITION-Attributen belegt. Hier gilt also, dass beide nachfolgenden Ausführungspfade des Prozesses nach erfolgreicher

Beendigung von a zur Ausführung kommen. Weiterhin gilt, dass die Aktivitäten d und e innerhalb einer SEQUENCE in einer WHILE-Aktivität mehrfach ausgeführt werden, was insbesondere die wiederholte Ausführung ihrer Variablenzugriffe auf v zur Folge hat. Als Resultat dieser mehrfachen Ausführung von d, e im Vergleich zu b, c ist es im dargestellten Szenario sinnvoller (d. h. hinsichtlich der Kosten der Instanzdatenzugriffe günstiger), die Variable v Partition B als Partition A zuzuordnen, da durch diese Partitionierung die Kosten für die partitionsübergreifende Kommunikation (wiederum gemittelt über eine Menge von Prozessinstanzen) reduziert werden.

Das Auftreten eines Fehlers während der Ausführung einer Aktivität löst den Mechanismus der Fehlerbehandlung und -kompensation aus; dies resultiert insbesondere in der Unterbrechung der "regulären" Prozessausführung. Abhängig von der Struktur des Prozesses und dessen FAULT- und COMPENSATIONHANDLER-Elementen kann diese Unterbrechung der Prozessausführung zur Folge haben, dass bisher noch nicht ausgeführte Aktivitäten im weiteren Verlauf der Prozessinstanz nicht mehr zur Ausführung kommen. Bedingt durch die Möglichkeit zum asynchronen Auslösen von Fehlern während beispielsweise der Ausführung eines EVENTHANDLER, dem Auslösen eines Fehlers durch die THROW-Aktivität in einem nebenläufigen Ausführungspfad des Prozesses oder die Struktur der vorhandenen FAULT- und COMPENSATIONHANDLER, ist eine vollständige Berücksichtigung der Wahrscheinlichkeit des Auftretens eines Fehlers in der Partitionierung eines Prozesses im Allgemeinen (d. h. ohne eine Simulation des zeitlichen Ausführungsverlaufs von Prozessinstanzen) nicht ohne weiteres möglich. Entsprechendes gilt für die Beendigung eines Prozesses durch die EXIT-Aktivität oder der expliziten Terminierung von SCOPE-Aktivitäten durch die Aktivitäten TERMINATE bzw. TERMINATESCOPE.

Eigenschaft 4.16 (Erfolgreiche Ausführung einer Aktivität)**.** *Die Wahrscheinlichkeit, dass ein Fehler während der Ausführung einer Aktivität auftritt stellt einen Einfluss auf die Ausführungswahrscheinlichkeit einer Aktivität dar. Aufgrund oben genannter Eigenschaften von BPEL berücksichtigt das Partitionierungsverfahren die Effekte eines während der Ausführung einer Aktivität auftretenden Fehlers nur für diejenigen Aktivitäten, die in einer Kontrollabhängigkeit zur fehlerhaft*

ausgeführten Aktivität stehen. Explizit ausgelöste Fehler oder die explizite Terminierung einzelner SCOPE*-Aktivitäten werden durch das Partitionierungsverfahren nicht berücksichtigt.*

Eigenschaft 4.17 (Explizite Parametrisierung von HANDLER-Aktivitäten)**.** *Die Kind-Aktivitäten von* FAULTHANDLER*-,* COMPENSATIONHANDLER*-,* EVENTHANDLER*- und* TERMINATIONHANDLER*-Aktivitäten werden nur dann ausgeführt, wenn auch die entsprechende* HANDLER*-Aktivität zur Ausführung kommt. Die Bestimmung der durchschnittlichen Ausführungswahrscheinlichkeit (und Ausführungsanzahl) einer* HANDLER*-Aktivität erfolgt entweder durch manuelle Vorgabe des Benutzers bzw. die Analyse angefallener Protokolldaten erfolgen.*

4.3 Parametrisierung des Partitionierungsvorgangs

Die Adressierung der in den vorangehenden Abschnitten formulierten Anforderungen hinsichtlich der identifizierten Partitionierungsobjekte und deren Einfluss auf das Partitionierungsverfahren erfordert – für den Fall, dass noch keine Ausführungsprotokolle abgelaufener Prozessinstanzen vorliegen – geeignete Mechanismen zur Parametrisierung eines Prozesses vor seiner Partitionierung. In den folgenden Abschnitten wird erläutert, wie die identifizierten Einflussgrößen in Form sogenannter *Partitionierungsparameter* repräsentiert und in Verbindung zu den Elementen eines BPEL-Prozesses gebracht werden können.

4.3.1 Parameter

Tabelle 4.1 zeigt eine Übersicht der identifizierten Partitionierungsparameter eines Prozesses sowie deren Typen.

Zur Adressierung von Anforderung 4.13 erlauben die Parameter *VAC* und *VACL* eine Annotation von Variablenzugriffen mit Kostenwerten. Der Kostenwert eines Instanzdatenzugriffs bestimmt sich abhängig von der Größe des mit diesem übermittelten Datenvolumens. Für den Fall, dass die Ausführung einer Aktivität mit dem Zugriff auf mehr als eine Variable verbunden ist, erfolgt die Annotation mittels des Parameters *VACL* in Form mehrerer Variablenname-Kostenwert-Paare, andernfalls durch den Parameter *VAC*. Auf eine Annotation

Parameter	Typ	Erklärung
VariableAccessCost (VAC)	Integer	Kosten eines Variablenzugriffs
VariableAccessCostList ($VACL$)	List: Variable, VAC	Liste von Variablenzugriffskosten
OneServiceOnly (OSO)	Boolean	Beschränkung der Partitionierung von INVOKE-Aktivitäten eines bestimmten PARTNERLINK
Probability (P_{Exec}, P_{Succ}, P_{Link})	Float	Erwartete Wahrscheinlichkeitswerte für Aktivitätsausführung, Beendigung der Aktivitätsausführung ohne Fehler und positiven LINK-Status
ProbabilityList (PL_{Exec})	List: Float	Liste von Ausführungswahrscheinlichkeiten
NumberOfIterations (NOI)	List: Integer	Erwartete Ausführungsanzahl
NonFunctionalProperties (NFP)	WS-Policy	Nicht-funktionale Anforderungen einer INVOKE-Aktivität
FixedAssignment (FIX)	String	Manuelle Partitionsvorgabe

Tabelle 4.1: Partitionierungsparameter zur Abbildung der in Abschnitt 4.2 identifizierten Einflussgrößen der Partitionierung.

der Zugriffskosten auf Instanzdaten anderen Typs (wie z.B. Correlation-Sets oder Message-Exchanges) wird aufgrund ihrer im Allgemeinen geringen Größe verzichtet.

Für die Erfüllung von Anforderung 4.12 zur Flexibilisierung der Partitionierung im Hinblick auf zustandslose Dienste erlaubt der Parameter OSO die Festlegung, ob für die Realisierung der PARTNERROLE eines bestimmten PARTNERLINK immer dieselbe Dienstimplementierung verwendet werden muss. Um im Fall der Nicht-Vorgabe einer Parametrisierung eine korrekte Partitionierung (d. h. eine Partitionierung, die einer zentralen Ausführung des Prozesses

entspricht) zu erhalten, gilt für den Parameter OSO der Standardwert eines Booleschen `true`.

Die Parameter P_{Exec}, PL_{Exec}, P_{Succ}, P_{Link} erlauben eine Annotation unterschiedlicher BPEL-Elemente mit Wahrscheinlichkeitswerten hinsichtlich der Wahrscheinlichkeit der Ausführung von Aktivitäten (P_{Exec}, PL_{Exec}), dem Abschluss der Aktivitätsausführung ohne Fehler (P_{Succ}) und eines positiven LINK-Status (P_{Link}) zur Erfüllung von Anforderungen 4.14 und 4.16. Da die Bedeutung der einzelnen Parameter abhängig vom jeweils parametrisierten Objekt ist, ist deren Erläuterung Teil der Beschreibung der parametrisierbaren Objekte in Abschnitt 4.3.2.

Die Gewichtung von Aktivitäten, deren Kind-Aktivitäten potentiell mehrfach ausgeführt werden (Anforderung 4.15), wird im Partitionierungsverfahren durch den Parameter NOI realisiert. Dabei gilt, dass der Wert von NOI die tatsächlich erwartete Ausführungsanzahl einer Iterationsaktivität bzw. einer Handler-Aktivität angibt. Gilt beispielsweise, dass innerhalb einer Schleife mit 100 spezifizierten Ausführungen bereits typischerweise nach 50 Ausführungen ein Fehler auftritt, so ist die korrekte Parametrisierung der Schleife der Wert $NOI = 50$.

Eine gesonderte Parametrisierung der INVOKE-Aktivitäten hinsichtlich ihrer funktionalen Anforderungen ist nicht erforderlich, da die hierfür notwendigen Informationen bereits als Teil des Prozessmodells (bzw. des Deployment-Deskriptors) definiert sind. Die Beschreibung der mit einer INVOKE-Aktivität verbundenen nicht-funktionalen Anforderungen an den jeweils zu verwendenden Dienst (Anforderung 4.10) erfolgt durch den Parameter NFP welcher die Anforderungen an die zu verwendende Dienstimplementierung in Form eines WS-Policy-Dokuments beinhaltet.

Die Realisierung von Anforderung 4.7 nach manueller Partitionsvorgabe durch den Partitionierer erfolgt durch eine Partitionierungsparametrisierung vom Typ FIX.

4.3.2 Parametrisierbare Objekte

Ein wichtiger Aspekt der dezentralen Prozessausführung unter Verwendung der PS-Infrastruktur ist, dass ein Benutzer des Systems keine Kenntnis der EWFN-Repräsentation verarbeiteter Prozesse benötigt; diese dient lediglich als Ausführungsvorschrift für deren dezentrale Ausführung. Hierdurch bedingt, erfolgt die Parametrisierung der Elemente eines Prozesses auf Ebene seiner BPEL-Repräsentation durch die oben genannten Parameter wie folgt:

Variablen

- *VAC*: Da sich der Wert und damit die Größe eines Instanzdatums während der Ausführung einer Prozessinstanz ändern kann, erfolgt die Festlegung der Kosten eines Variablenzugriffs nicht als Teil der Variablendeklaration sondern durch Parametrisierung der Aktivität, die den Variablenzugriff durchführt.

Partner Links

- *OSO*: Gilt für einen PartnerLink die Eigenschaft $OSO = \texttt{false}$, so ist eine Partitionierung zulässig, die mehrere Invoke-Aktivitäten, die eine Interaktion über desselben PartnerLink führen, auf unterschiedliche Partitionen verteilt. Die Parametrisierung erfolgt durch Annotation der Deklaration des PartnerLink im Prozessmodell.

From/To in Copy in Assign

- *VACL*: In den Attributen From und To eines Copy-Elements einer Assign-Aktivität können beliebig komplexe *Expressions* in unterschiedlichen *Expression Languages* verwendet werden, welche insbesondere die Verwendung mehrerer unterschiedlicher Instanzdaten erlauben. Dementsprechend erfolgt die Annotation durch potentiell mehrere Variable-Zugriffskosten-Wertpaare mittels des *VACL*-Parameters.

INVOKE

- *VACL*: Zur Beschreibung der Kosten für den Zugriff auf die INPUT- und OUTPUT-Variable der Aktivitäten werden die Kosten des Variablenzugriffs für die Variablennamen `inputVariable` bzw. `outputVariable` definiert.
- *NFP*: Erfolgt keine explizite Annotation einer INVOKE-Aktivität hinsichtlich ihrer nicht-funktionalen Eigenschaften, so kann jede zu einer INVOKE-Aktivität funktional kompatible Dienstimplementierung (vgl. Definition 4.25) als Ziel des Dienstaufrufs verwendet werden. Ist eine entsprechende Annotation vorhanden, so ist eine Dienstimplementierung zu verwenden, die die verlangten nicht-funktionalen Eigenschaften erfüllt.

RECEIVE, ONMESSAGE (in PICK und EVENTHANDLER)

- *VAC*: Analog der INVOKE-Aktivität erfolgt eine Bestimmung der Kosten für die Ablage der empfangenen Daten. Da während der Ausführung der Aktivität lediglich ein Variablenzugriff durchgeführt wird, erfolgt die Annotation unter Verwendung des *VAC*-Parameters.

IF

- *VACL*: Analog der Annotation der *Expressions* einer ASSIGN-Aktivität gilt auch für das CONDITION-Attribut einer IF-Aktivität die Möglichkeit der Verwendung mehrerer Variablen.
- PL_{Exec}: Mittels des Parameters PL_{Exec} ist eine Parametrisierung der Ausführungswahrscheinlichkeiten der einzelnen Zweige einer IF-Aktivität möglich. Aufgrund des gegenseitigen Ausschlusses der einzelnen Ausführungspfade gilt, dass die Summe der annotierten Wahrscheinlichkeitswerte einen Wert ≤ 1 ergeben muss. Die Abfolge der Wahrscheinlichkeiten in PL_{Exec} entspricht der Reihenfolge der Zweige der IF-Aktivität in der Prozessbeschreibung.

Synchronisationskanten

- $VACL$: Für jede Variable, die in einer TransitionCondition eines Link-Elements verwendet wird, kann ein entsprechendes Paar von Variablenname und zugeordnetem Kostenwert definiert werden.
- P_{Link}: Der Parameter P_{Link} beschreibt die Wahrscheinlichkeit, dass die TransitionCondition während der Ausführung einer Prozessinstanz zu einem positiven Link-Status evaluiert wird. Da ausgehende Synchronisationskanten einer Aktivität nicht notwendigerweise einen gegenseitigen Ausschluss realisieren müssen, gilt – im Gegensatz zu den Zweigen einer If-Aktivität – hier die Einschränkung der Summe der Wahrscheinlichkeitswerte auf einen Wert ≤ 1 nicht.

ForEach

- NOI: Die Attribute StartCounterValue und FinalCounterValue einer ForEach-Aktivität definieren die Anzahl der Iterationen der Scope-Aktivität im Rumpf des ForEach. Da die Werte von StartCounterValue und StopCounterValue während der Ausführung einer Prozessinstanz dynamisch bestimmt werden können, ist eine automatische Bestimmung ihrer Werte durch das Partitionierungsverfahren im Allgemeinen nicht möglich. Der Parameter NOI erlaubt dementsprechend die Annotation der Aktivität mit der erwarteten durchschnittlichen Iterationsanzahl.

While/RepeatUntil

- NOI: Die Parametrisierung der While- und RepeatUntil-Aktivitäten durch das NOI-Attribut erfolgt analog der Parametrisierung der ForEach-Aktivität.
- $VACL$: Die Definition der Kosten für den Zugriff auf die, im Condition-Element einer While bzw. Repeat-Aktivität verwendeten Variablen, erfolgt unter Verwendung des $VACL$-Parameters.

EventHandler

- NOI: Da die EventHandler eines Prozesses potentiell mehrfach zur Ausführung kommen können, erlaubt der Parameter NOI auch hier die Annotation der erwarteten Anzahl der Ausführungen.

Alle Aktivitäten

- P_{Exec}: Zur selektiven Parametrisierung einzelner Aktivitäten eines Prozesses sowie zur Parametrisierung von Fault-, Compensation- und TerminationHandler-Aktivitäten, kann jede Aktivität durch den Parameter P_{Exec} statisch mit einer Ausführungswahrscheinlichkeit annotiert werden.
- P_{Succ}: Der Parameter P_{Succ} erlaubt die Angabe eines Wahrscheinlichkeitswerts für die Ausführung einer Aktivität, ohne dass diese während ihrer Verarbeitung einen Fehler an ihre umgebende Scope-Aktivität propagiert und damit zur Unterbrechung der regulären Prozessausführung führt. Dies bedeutet insbesondere, dass, falls die annotierte Aktivität selbst ein (impliziter oder expliziter) Scope ist, P_{Succ} auch die Fälle einschließt, in denen zwar ein Fehler von einer Kind-Aktivität des Scope ausgelöst wird, dieser allerdings durch den FaultHandler der parametrisierten Scope-Aktivität erfolgreich verarbeitet wird und damit nicht in einer Weitergabe eines Fehlers an deren umgebenden Scope resultiert. Ist kein Wert für P_{Succ} explizit angegeben, so gilt implizit der Wert 1. Das bedeutet, dass für diesen Fall angenommen wird, dass eine nicht gesondert parametrisierte Aktivität stets erfolgreich ausgeführt wird.

Alle parametrisierbaren Objekte

- FIX: Durch Parametrisierung eines Objekts mit einer Annotation vom Typ FIX kann dieses statisch einer Partition zugeordnet werden.

4.3.3 Realisierung mit WS-Policy und WS-PolicyAttachment

Abbildung 4.9 verdeutlicht graphisch, wie eine Realisierung der Parametrisierung eines BPEL-Prozesses unter Verwendung der Web-Service-Standards

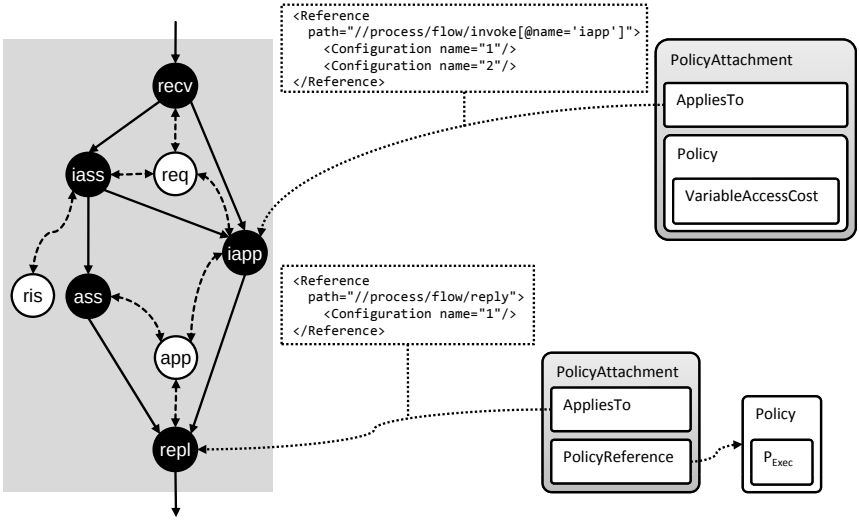


Abbildung 4.9: Annotation eines BPEL-Prozesses mittels WS-Policy-Dokumenten.

WS-Policy [OHV+] und *WS-PolicyAttachment* [YBV+07] möglich ist. Für die Verwendung des Standards *WS-Policy* spricht, neben der Möglichkeit zur Prozessexternen Annotation durch *WS-PolicyAttachment*, auch die, mit der Konformität zu einem etablierten Standard verbundene, Möglichkeit zur Wiederverwendung existierender Werkzeuge.

Die Parametrisierung der Objekte des Prozesses verläuft wie folgt: eine Menge von annotierten *Subjekten* (z. B. eine Aktivität in einer bestimmten Prozesskonfiguration) wird, durch eine Menge sogenannter *PolicyAttachment-Dokumente* als Bindeglied, in Verbindung zu sogenannten *Policy-Dokumenten* gesetzt, welche die zu annotierende Information beinhalten. Die Liste der Subjekte sind die in Abschnitt 4.3.2 aufgeführten Elemente eines BPEL-Prozesses. Die Policy-Dokumente sind XML-Repräsentationen der in Abschnitt 4.3.1 genannten Parameter.

In Abbildung 4.9 gilt beispielsweise, dass durch das, in der Abbildung oben

```
<xs:schema
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified">
 <xs:element name="VariableAccessCost">
  <xs:complexType>
   <xs:attribute name="Cost"
                 use="required"
                 type="xs:decimal"/>
   <xs:attribute name="VariableName"
                 use="required"
                 type="xs:NCName"/>
  </xs:complexType>
 </xs:element>
 <xs:element name="VariableAccessCostList">
  <xs:complexType>
   <xs:sequence>
    <xs:element minOccurs="1"
                maxOccurs="unbounded"
                ref="VariableAccessCost"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

```
<VariableAccessCostList>
   <VariableAccessCost
      VariableName="req"
      Cost="100" />
   <VariableAccessCost
      VariableName="ris"
      Cost="75" />
</VariableAccessCostList>
```

Abbildung 4.10: Schema und Beispiel eines *VACL* Parameters.

dargestellte, XML-Element `Reference` im *AppliesTo* der *VariableAccessCost*-Policy die INVOKE-Aktivität *iapp* durch einen XPath-Ausdruck über deren `name`-Attribut identifiziert. Um die Parametrisierung auf bestimmte Konfigurationen des Prozesses (vgl. Abschnitt 3.1) einzuschränken, werden die Identifikatoren der Prozesskonfigurationen im Attribut `name` der `Configuration`-Elemente der `Reference` erfasst. Für eine gleichzeitige Parametrisierung mehrerer Konfigurationen, ist die Angabe mehrerer `Configuration`-Elemente möglich.

Zusätzlich zu dieser Art der Parametrisierung wird durch die *WS-PolicyAttachment*-Spezifikation auch die, in der Abbildung unten dargestellte, Vorgehensweise der Annotation unterstützt, welche die Policy vom Policy-Attachment-Dokument entkoppelt und so eine getrennte Pflege der Dokumente vereinfacht. Hierbei erfolgt die Identifikation des zu parametrisierenden Prozessobjekts analog dem obigen Beispiel, allerdings ist die eigentliche Policy hier nicht Teil des *PolicyAttachment*-Dokuments, sondern liegt von diesem getrennt vor und wird lediglich aus dem *PolicyAttachment*-Dokument referenziert.

Abbildung 4.10 zeigt das XML-Schema des *VACL*-Parameters sowie ein

```
<ps:Reference
   xmlns:ps="http://www.iaas.uni−stuttgart.de/ns/2009/process−space"
   xmlns:example="http://example.org"
   process="example:LoanApproval"
   path="//process/flow/invoke[@name='iapp']">
   <ps:Configuration name="bf53f446−d7b5−1078−7b4d−833908470970"/>
   <ps:Configuration name="bf53f436−061f−50e0−c828−608996fb980c"/>
</ps:Reference>
```

Abbildung 4.11: Annotation eines Prozesselements in zwei unterschiedlichen Konfigurationen.

entsprechendes Beispieldokument. Das Element `VariableAccessCostList` dient als Container für die Elemente `VariableAccessCost` mit den Attributen `VariableName` und `Cost`. Der komplexe Typ `VariableAccessCostType` beschreibt ein Element der Parameterliste vom Typ *VACL*. Die Policy-Dokumente der anderen Partitionierungsparameter folgen einem ähnlichen Aufbau.

Die Verknüpfung einer Annotation mit dem zu annotierenden Prozesselement erfolgt, der WS-Policy-Attachment-Spezifikation [YBV+07] entsprechend, durch das Kind-Element `AppliesTo` des `PolicyAttachment`-Elements. Der Wert der Annotation kann entweder eingebettet in das *PolicyAttachment*-Dokument oder, mittels einer sogenannten *PolicyReference* verbunden, in Form eines externen Dokuments vorliegen.

Die Referenzierung des annotierten Prozesselements erfordert (i) die Identifikation des Prozessmodells selbst, (ii) die Prozesskonfiguration(en), für die jeweiligen Parametrisierungen gelten sollen sowie (iii) die Identifikation des zu annotierenden Elements innerhalb des Prozessmodells. Die Repräsentation dieser Referenz erfolgt durch das, in Abbildung 4.11 dargestellte, XML-Element `Reference`. Für die Identifikation des annotierten Prozessmodells wird dessen qualifizierter Name (QName) im Attribut `process` verwendet; die Bestimmung des parametrisierten Objekts im Prozess durch einen XPath-Ausdruck [CD+99] im Attribut `path`. Da eine Parametrisierung spezifisch für bestimmte Prozesskonfiguration (d. h. der Abbildung der Partitionierungsobjekte eines Prozesses auf seine Ausführungsumgebung, vgl. Abschnitt 3.1) ist und unterschiedliche

Parametrisierungen für unterschiedliche Konfigurationen existieren können, werden die Identifikatoren der parametrisierten Prozesskonfigurationen im Attribut `name` des XML-Elements `Configuration` erfasst.

4.4 Übersicht des Partitionierungsverfahrens

In den folgenden Abschnitten wird ein Verfahren vorgestellt, welches auf Grundlage der identifizierten Partitionierungsparameter (Abschnitt 4.2) eine automatische Partitionierung eines BPEL-Prozesses bzw. seiner EWFN-Repräsentation durchführt [WML09a]. Die entwickelte Vorgehensweise ist ein hybrides Verfahren, das Ideen verschiedener Verfahren vereint und diese im Hinblick auf das in dieser Arbeit vorliegende Einsatzszenario anpasst und erweitert. Ähnlich [NCS04] ist das entwickelte Partitionierungsverfahren mehrphasig. Das Partitionierungsresultat früher Partitionierungsphasen wird in späteren Phasen durch Partitionierung bisher noch nicht partitionierter Objekte verfeinert. Das entwickelte Verfahren verfolgt allerdings – im Vergleich zu [NCS04] – einen anderen Ansatz zur Partitionierung von INVOKE-, RECEIVE- und REPLY-Aktivitäten, um den speziellen Eigenschaften einer SOA-Anwendungsumgebung Rechnung zu tragen. Zu diesem Zweck wird die Klasse der sogenannten *Portable Nodes* unterteilt in die Klasse der *Heavy* und *Light Nodes*, deren Partitionierung in unterschiedlichen Phasen gesondert adressiert wird. Ähnlich der in [Bau01] und [DKL09] vorgestellten Ansätze erfolgt die Realisierung einer Phase des Partitionierungsverfahrens durch Reduktion des Prozesspartitionierungsproblems auf das *Graphpartitionierungsproblem*; Grundlage hierfür bildet allerdings im vorliegenden Fall nicht das ursprüngliche BPEL-Prozessmodell selbst, sondern eine, um eine Kantengewichtung erweiterte, Form der EWFN-Repräsentation des Prozesses. Die Bestimmung der Kantengewichte des EWFN-Graphen stützt sich – wie in Abschnitt 4.2.4 erläutert – unter anderem auf die erwartete Ausführungsanzahl und -wahrscheinlichkeit der Aktivitäten des Prozesses. Für den Fall, dass zum Partitionierungszeitpunkt noch keine Informationen über den Ausführungsverlauf früherer Instanzen desselben Prozessmodells vorliegen, erlaubt das entwickelte Partitionierungsverfahren – ähnlich beispielsweise [MJGN08] oder [MKL+09, Mon08] – eine approximative Bestimmung der

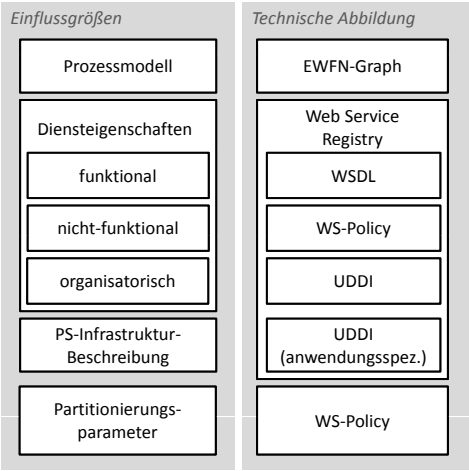


Abbildung 4.12: Zusammenfassung: Einflussgrößen für die Partitionierung von BPEL-Prozessen und deren technische Abbildung.

Ausführungswahrscheinlichkeit der Prozessaktivitäten auf Grundlage der vom Partitionierer definierten Partitionierungsparameter.

Abbildung 4.12 fasst die unterschiedlichen Eingaben des Partitionierungsverfahrens, sowohl auf konzeptueller als auch technischer Ebene, graphisch zusammen und Abbildung 4.13 zeigt – in Form einer Gesamtübersicht – wie diese durch das Partitionierungsverfahren verarbeitet werden.

Eingaben des Partitionierungsverfahrens bilden (i) das Prozessmodell, (ii) die Parametrisierung des Partitionierungsverfahrens, die (iii) funktionalen sowie (iv) nicht-funktionalen Eigenschaften der Dienste in der Dienstumgebung des Prozesses und (v) die Beschreibung der PS-Infrastruktur, auf welcher der partitionierte Prozess ausgeführt werden soll. Die unterschiedlichen Eingaben werden dem Partitionierungswerkzeug in unterschiedlicher Weise zugänglich gemacht: Die Beschreibungen der Diensteigenschaften sowie der PS-Infrastruktur

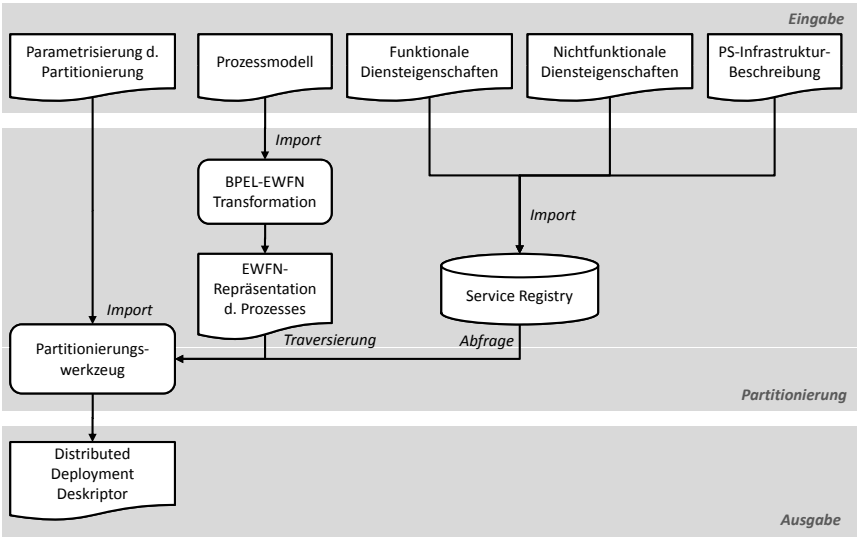


Abbildung 4.13: Gesamtübersicht über das entwickelte Verfahren zur Partitionierung von BPEL-Prozessen.

werden durch die Dienstanbieter in eine *Service Registry* publiziert[1]. Diese wird während des Partitionierungsverfahrens durch das Partitionierungswerkzeug abgefragt. Das Prozessmodell in seiner BPEL-XML-Repräsentation wird durch einen BPEL-EWFN-Transformator [Mar10] in eine entsprechende Repräsentation im EWFN-Metamodell überführt. Die EWFN-Repräsentation des Prozessmodells dient als Grundlage des weiteren Partitionierungsvorgangs. Die Prozessparametrisierung in Form von WS-Policy- und zugehörigen WS-Policy-Attachment-Dokumenten wird direkt an das Partitionierungswerkzeug übergeben. Ausgabe des Partitionierungsverfahrens ist die bestimmte Prozesskonfiguration in Form des sogenannten *Verteilten Deployment-Deskriptors* (engl. *Distributed Deployment Descriptor (DDD)*). Dieser beschreibt unter anderem,

[1]Die Publikation der Beschreibung der Dienste wurde in Abschnitt 4.2.2.1 erläutert. Da sich die Erklärung des Partitionierungsvorgangs, wie in Abschnitt 4.2 erwähnt, lediglich auf den logischen Partitionsbegriff stützt, kann die Erläuterung der Publikation der Beschreibung der PS-Infrastruktur in Abschnitt 5.2.2 erfolgen.

welcher Partition welches partitionierbare Objekt des Prozesses zugeordnet wird. Die formale Beschreibung des DDD ist Teil von Abschnitt 4.4.1, ein entsprechendes XML-Format wird in Abschnitt 5.8.1 vorgestellt.

4.4.1 Definitionen

Die zur Beschreibung des Partitionierungsverfahren verwendete Notation orientiert sich an der in [KML08] definierten abstrakten Syntax für WS-BPEL 2.0 und verwendet aus dieser im Besonderen:

- die Mengen $\mathcal{A}$ (sowie ihre Teilmengen, z. B. $\mathcal{A}_{Invoke}$, $\mathcal{A}_{If}$), $\mathcal{E}$, $\mathcal{L}$, $\mathcal{V}$, $CORSET$, PL, MEX, $FROM$, TO;
- die Menge $\mathcal{WS}$ der Web-Services sowie die Menge PT der PortType-Elemente;
- die Menge $\mathsf{HR} \subseteq \mathcal{A} \times \mathcal{B} \times \mathcal{A}$, die die hierarchische Schachtelung der Aktivitäten eines Prozesses repräsentiert; dabei beschreibt $\mathcal{B}$ die Menge der sogenannten *Labels* mit $\mathcal{B} = \mathcal{E} \cup \bot$ den *Events* $\mathcal{E}$;
- die Mengen $\mathsf{SR} \subseteq \mathcal{A} \times \mathcal{C} \times \mathcal{L}$, $\mathsf{TR} \subseteq \mathcal{A} \times \mathcal{L}$ und $\mathsf{LR} \subseteq \mathcal{A} \times \mathcal{L} \times \mathcal{C} \times \mathcal{A}$ sowie die Funktion $\mathcal{L}_{in} : \mathcal{A} \to 2^{\mathcal{L}}$ zur Beschreibung der Link-Elemente einer Flow-Aktivität;
- die Funktionen $<^{s}_{seq}$ und $<^{i}_{if}$ zur Beschreibung der Ordnung der Kind-Aktivitäten einer Sequence-Aktivität $s \in \mathcal{A}_{Sequence}$ bzw. der Ordnung der Aktivitäten in der einzelnen Zweigen einer If-Aktivität $i \in \mathcal{A}_{If}$;
- die Funktion $\mathsf{type}_{\mathbb{L}} : \mathbb{L} \to$ types, die den Typ eines Elements zurückgibt;
- die Funktion $\mathsf{children} : \mathcal{A} \to 2^{\mathcal{A}}$ zur Bestimmung der direkten Kind-Aktivitäten einer Aktivität;
- sowie die Funktionen $\mathsf{partnerLink}_{\mathcal{CO}} : \mathcal{CO}_{message} \to PL$ und $\mathsf{portType}_{\mathcal{CO}} : \mathcal{CO}_{message} \to PT$, die den PartnerLink bzw. den PortType einer Interaktionsaktivität zurückgibt.

Weiterhin wird die Notation $\pi_i(t)$ für die Projektion des Tupels t auf dessen i-te Komponente verwendet; die Notation $\pi_{i,j}(t)$ projiziert das Tupel t auf ein Tupel mit den Komponenten i und j von t. Zusätzlich notwendige Definitionen

(z. B. zur formalen Repräsentation der in Abschnitt 4.3.1 identifizierten Partitionierungsparameter und der Partitionierungsobjekte) sowie Hilfsfunktionen zur Vereinfachung der Darstellung des Partitionierungsverfahrens werden in der Folge vorgestellt:

Definition 4.18 (Partitionierungsobjekt)**.** *Die Menge der Partitionierungsobjekte O_{part} beinhaltet sämtliche Elemente eines BPEL-Prozesses, denen im Rahmen der Partitionierung eine Partition zugewiesen wird. Dies sind (i) die Aktivitäten $\mathcal{A}$ des Prozesses, (ii) dessen Events $\mathcal{E}$[1], (iii) die Links $\mathcal{L}$ sowie dessen Instanzdaten, bestehend aus (iv) der Menge der Variablen $\mathcal{V}$, (v) der Correlation-Sets $CORSET$, (vi) der Partner-Links PL und (vii) der Message-Exchanges MEX. Es gilt demnach: $O_{part} = \mathcal{A} \cup \mathcal{E} \cup \mathcal{L} \cup \mathcal{V} \cup CORSET \cup PL \cup MEX$.*

Definition 4.19 (Parametrisierbares Objekt)**.** *Die Menge der parametrisierbaren Objekte O_{param} beschreibt all jene Elemente eines BPEL-Prozesses, die vor dem Partitionierungsvorgang parametrisiert werden können und beinhaltet (i) die Menge der partitionierbaren Objekte O_{part}, da jedem partitionierbaren Objekt statisch eine Partition zugewiesen werden kann (Anforderung 4.7) und weiterhin (entsprechend den Erläuterungen in Abschnitt 4.3.2) die Menge (ii) $FROM \cup TO$ zur Annotation von* Assign*-Aktivitäten. Es gilt demnach: $O_{param} = O_{part} \cup FROM \cup TO$.*

Definition 4.20 (Partitionierungsparameter)**.** *Die Menge $\mathcal{PAR}$ umfasst die zur Parametrisierung des Partitionierungsvorgangs verwendeten Parameter, welche in Abschnitt 4.3.1 vorgestellt wurden. Die Parameter sind jeweils, abhängig von ihrem Typ aus einer Teilmenge von $\mathcal{PAR}$.*

Für die Menge PAR der Parametertypen gilt:

$$PAR = \{\texttt{VAC}, \texttt{VACL}, \texttt{P}_{EXEC}, \texttt{P}_{SUCC}, \texttt{P}_{LINK}, \texttt{PL}_{EXEC}, \texttt{OSO}, \texttt{NOI}, \texttt{NFP}, \texttt{FIX}\}$$

und damit $\mathcal{PAR} = \mathcal{PAR}_{VAC} \cup \mathcal{PAR}_{VACL} \cup \mathcal{PAR}_{P_{Exec}} \cup \mathcal{PAR}_{P_{Succ}} \cup \mathcal{PAR}_{P_{Link}} \cup \mathcal{PAR}_{PL} \cup \mathcal{PAR}_{OSO} \cup \mathcal{PAR}_{NOI} \cup \mathcal{PAR}_{NFP} \cup \mathcal{PAR}_{FIX}$.

[1] Entsprechend Definition 3.5.1 und 3.5.2 [KML08] gilt, dass die Handler-Elemente von Ereignissen nicht expliziter Teil der Formalisierung sind, sondern diese implizit durch das jeweilige Ereignis aus $\mathcal{E}$ beschrieben werden.

Die Funktion $\mathsf{type}_{\mathcal{PAR}} : \mathcal{PAR} \rightarrow PAR$ *ordnet (entsprechend Definition 2.1.13 in [KML08]) einem Partitionierungsparameter dessen Typ zu, d. h. es gilt beispielsweise* $\forall p \in \mathcal{PAR}_{FIX} : \mathsf{type}_{\mathcal{PAR}}(p) = \mathtt{FIX}$.

Definition 4.21 (Von einem Web-Service implementierte PortTypes)**.** *Die Funktion* $\mathsf{portType}_{\mathcal{WS}} : \mathcal{WS} \rightarrow 2^{PT}$ *ordnet einem Dienst die Menge der von diesem implementierten WSDL-*PortTypes *zu.*

Definition 4.22 (Parametrisierung)**.** *Die Funktion* $\mathsf{param} : O_{param} \rightarrow 2^{\mathcal{PAR}} \cup \bot$ *ordnet parametrisierbaren Objekten eine Menge von Partitionierungsparametern zu. Liegen für ein Objekt* $o \in O_{param}$ *keine Partitionierungsparameter vor, so gilt* $\mathsf{param}(o) = \bot$. *Zur Vereinfachung der Darstellung existiert für jeden Parametertyp in PAR eine Funktion, die den direkten Zugriff auf den Parameter des jeweiligen Typs erlaubt.*

So gilt beispielsweise $\mathsf{param}_{FIX} : O_{param} \rightarrow \mathcal{PAR}_{FIX} \cup \bot$ *für jedes* $o \in O_{param}$ *mit:*

$$\mathsf{param}_{FIX}(o) \mapsto \begin{cases} p & \textit{wenn } \exists\ p \in \mathsf{param}(o) :\ \mathsf{type}_{\mathcal{PAR}}(p) = \mathtt{FIX} \\ \bot & \textit{sonst} \end{cases}$$

Definition 4.23 (Bestimmung nicht-funktionaler Kompatibilität von Dienstnutzer und Dienstanbieter)**.** *Die Funktion* $\mathsf{compatible} : \mathcal{PAR}_{NFP} \times \mathcal{PAR}_{NFP} \rightarrow \mathbb{B}$ *beschreibt den Schnitt zweier WS-Policy-Dokumente zur Repräsentation nicht-funktionaler Eigenschaften und Anforderungen. Sind die Policy-Dokumente zueinander kompatibel, so ist der Wert von* compatible *ein Boolesches* `true`*, andernfalls* `false`*. Die von* compatible *durchgeführte Überprüfung deckt dabei sowohl die sogenannte domänenunabhängige als auch die domänenspezifische Kompatibilität der Policy-Dokumente ab. Die Beschreibung der Semantik von* compatible *wurde einführend in Abschnitt 2.1.2 erläutert; im Detail wir sie in Abschnitt 4.5 in [OHV+] vorgestellt.*

Definition 4.24 (Partition)**.** $\mathcal{P}$ *bezeichnet die Menge der Partitionen. Die Funktion* $\mathsf{partition} : O_{part} \cup \mathcal{PAR}_{FIX} \cup \mathcal{WS} \rightarrow \mathcal{P} \cup \bot$ *ordnet partitionierbaren Objekten, statischen Partitionsvorgaben und Web-Services eine Partition zu. Ist einem Objekt*

$o \in O_{part}$ *keine Partition zugeordnet, so gilt* $\mathsf{partition}(o) = \bot$. *Entsprechend der Erläuterung in Abschnitt 4.2 wird eine Partition für die Beschreibung des entwickelten Partitionierungsverfahrens als logischer Bezeichner verstanden, welcher die Zusammenfassung mehrerer Partitionierungsobjekte zum Zweck deren Kolokation während der Prozessausführung erlaubt. Als Teil der Beschreibung der Übersicht der PS-Infrastruktur in Abschnitt 5.2 wird dieser Partitionsbegriff verfeinert und es wird erläutert, wie dieser auf die Elemente der PS-Infrastruktur abgebildet wird.*

Definition 4.25 (Funktionale Kompatibilität von Dienstnutzer und -anbieter)**.** *Ein Dienst* $ws \in \mathcal{WS}$ *ist zu den funktionalen Anforderungen einer* INVOKE*-Aktivität* $a \in \mathcal{A}_{Invoke}$ funktional kompatibel – *dargestellt durch* $a \preceq_f ws$ *– genau dann, wenn ws den von a erwarteten Kontrakt erfüllt, d. h. der Web-Service zumindest den von a verwendeten* PORTTYPE *und dessen* OPERATION *unterstützt (d.h. diese unter demselben Namen und mit denselben Nachrichtentypen implementiert).*

Formal gilt: $a \preceq_f ws :\Leftrightarrow \exists\, pt \in \mathsf{portType}_{\mathcal{WS}}(ws) :\ \mathsf{portType}_{CO}(a) = pt$

Definition 4.26 (Nicht-funktionale Kompatibilität von Dienstnutzer und -anbieter)**.** *Ein Dienst* $ws \in \mathcal{WS}$ *ist zu den nicht-funktionalen Anforderungen einer* INVOKE*-Aktivität* $a \in \mathcal{A}_{Invoke}$ nicht-funktional kompatibel – *dargestellt durch* $a \preceq_{nf} ws$ *– genau dann, wenn ws die von a geforderten nicht-funktionalen Anforderungen erfüllt.*

Formal gilt: $a \preceq_{nf} ws :\Leftrightarrow \mathsf{compatible}(\mathsf{param}_{NFP}(a), \mathsf{param}_{NFP}(ws)) = \mathtt{true}$

Definition 4.27 (Eltern-Aktivität einer Aktivität)**.** *Die Funktion* $\mathsf{parent} : \mathcal{A} \rightarrow \mathcal{A} \cup \mathcal{B}$ *erlaubt den Zugriff auf die Eltern-Aktivität einer Aktivität. Es gilt:*

$$\mathsf{parent}(a) \mapsto \begin{cases} a' & \textit{wenn } \exists\, (a', e, a) \in \mathsf{HR} :\ e = \bot \\ e & \textit{wenn } \exists\, (a', e, a) \in \mathsf{HR} :\ e \neq \bot \\ \bot & \textit{sonst} \end{cases}$$

Die Funktion $\mathsf{parents} : \mathcal{A} \rightarrow 2^{\mathcal{A}}$ *bildet eine Aktivität auf die Menge der ihr*

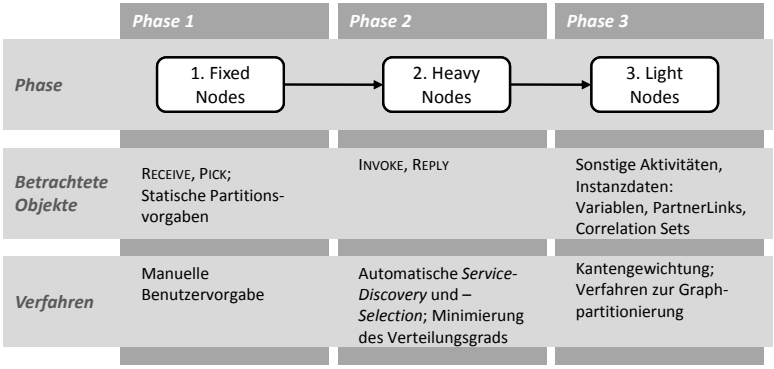


Abbildung 4.14: Phasen des entwickelten Verfahrens zur Partitionierung von BPEL-Prozessen.

hierarchisch übergeordneten Aktivitäten ab. Es gilt:

$$\mathsf{parents}(a) = \{a' \in \mathcal{A} | \exists\, k : (a_1, a_2), \ldots, (a_{k-1}, a_k), (a_i, a_{i+1}) \in \pi_{1,3}(\mathsf{HR}), \\ 1 \leq i \leq k-1, a_1 = a', a_k = a\}$$

Definition 4.28 (Quelle und Ziel eines Links)**.** *Die Funktionen* $\mathsf{source}_{\mathcal{L}} : \mathcal{L} \rightarrow \mathcal{A}$ *und* $\mathsf{target}_{\mathcal{L}} : \mathcal{L} \rightarrow \mathcal{A}$ *erlauben den Zugriff auf die Quelle bzw. das Ziel einer Synchronisationskante. Es gilt:*

$$\forall\, l \in \mathcal{L} : \mathsf{source}_{\mathcal{L}}(l) = \pi_1((s, c, l) \in \mathsf{SR})$$
$$\forall\, l \in \mathcal{L} : \mathsf{target}_{\mathcal{L}}(l) = \pi_1((t, l) \in \mathsf{TR})$$

4.4.2 Phasen des Partitionierungsverfahrens

Das entwickelte Partitionierungsverfahren umfasst zur Verarbeitung der Partitionierungsobjekte eines Prozesses, unter Adressierung sämtlicher identifizierter Einflussgrößen, drei unterschiedliche Phasen. Abbildung 4.14 veranschaulicht diese graphisch.

Phase 1 *Fixed Nodes* adressiert statische Partitionsvorgaben für beliebige Par-

titionierungsobjekte sowie Aktivitäten, die einen Nachrichtenempfang durch das WfMS beschreiben. Phase 2 *Heavy Nodes* beschreibt die, auf Mechanismen zur *Service Discovery* und *Service Selection* sowie der Minimierung des Partitionierungsgrads basierende, Bestimmung der Partition von INVOKE-Aktivitäten sowie die regelbasierte Partitionierung von REPLY-Aktivitäten. Phase 3 *Light Nodes* beinhaltet schließlich die Bestimmung der Partitionen bisher noch nicht partitionierter Objekte des EWFN-Modells des Prozesses auf Grundlage des Optimierungskriteriums der Minimierung von partitionsübergreifender Kommunikation.

Dabei gilt, dass in den einzelnen Phasen der Partitionierung jeweils nur die Partitionierungsobjekte betrachtet werden, die in vorherigen Phasen des Partitionierungsverfahrens noch keiner Partition zugeordnet wurden, d. h. $\forall o \in O_{part} : \mathsf{partition}(o) = \bot$.

Für das Partitionierungsverfahren gilt, dass die Partitionierung – entsprechend den Erläuterungen in Abschnitt 3.5 – zur Deployment-Zeit eines Prozesses stattfindet. Dies ist aufgrund der Eigenschaft der geringen Dynamik, sowohl der Dienstumgebung (Eigenschaft 4.6) als auch der Prozesse selbst (Eigenschaft 4.1), im der Arbeit zugrunde liegenden Szenario möglich.

Weiterhin gilt, dass eine Repartitionierung eines in Ausführung befindlichen Prozesses durch das Partitionierungsverfahren nicht unterstützt wird. Entsprechend den Erläuterungen in Abschnitt 3.5.2 ist in diesem Fall eine erneute Partitionierung und ein erneutes Deployment des Prozesses erforderlich. Dementsprechend fließen auch dynamische Größen, wie beispielsweise die aktuelle Auslastung der von einem Prozess verwendeten Dienste, nicht als Einflussgröße in das Partitionierungsverfahren ein.

4.5 Phase 1: Fixed Nodes

Der Partitionierer kann als Teil von Phase 1 mittels einer statischen Partitionsvorgabe für jedes Partitionierungsobjekt eines Prozesses manuell bestimmen, welcher Partition dieses zugeordnet werden soll. Wurde die Partition eines Objekts auf diese Weise bestimmt, so wird dieses als *Fixed Node* bezeichnet.

Entsprechend der in Abschnitt 4.3 erläuterten Vorgehensweise erfolgt die

```
<wsp:PolicyAttachment
  xmlns:wsp="http://www.w3.org/ns/ws−policy"
  xmlns:ps="http://www.iaas.uni−stuttgart.de/ns/2009/process−space"
  xmlns:wsu="http://docs.oasis−open.org/wss/2004/01/oasis−200401−wss−wssecurity−utility−1.0.xsd">
  <wsp:AppliesTo>
    <ps:Reference
      xmlns:example="http://example.org"
      process="example:LoanApproval"
      path="//process/variables/variable[@name='request']">
      <ps:Configuration name="bf53f446−d7b5−1078−7b4d−833908470970"/>
    </ps:Reference>
  </wsp:AppliesTo>
  <wsp:Policy wsu:Id="request−partition">
    <ps:FixedNode>
      <ps:Partition>ps://organisation−a.org</ps:Partition>
    </ps:FixedNode>
  </wsp:Policy>
</wsp:PolicyAttachment>
```

Abbildung 4.15: Beispiel der Zuweisung einer statischen Partitionsvorgabe zu einem *Fixed Node* mittels *WS-PolicyAttachment*.

Vorgabe der Partition eines *Fixed Node* mittels Annotation des jeweiligen Partitionierungsobjekts durch ein Prozess-externes WS-Policy-Dokument vom Typ `FIX` unter Verwendung von *WS-PolicyAttachment*.

Abbildung 4.15 zeigt anhand des Beispiels einer Variablen die statische Vorgabe einer Partition. In diesem Fall wird die Variable `request` derjenigen Partition zugeordnet, die durch den logischen Bezeichner `ps://organisation-a.org` identifiziert wird. Eine Erläuterung der Bedeutung der URL und die Bestimmung der Menge der verfügbaren PS ist Teil der Abschnitte 5.2.2 und 5.8. Die statische Partitionsvorgabe für andere Typen von Partitionierungsobjekten erfolgt analog.

Algorithmus 4.1 zeigt das zur Partitionierung von *Fixed Nodes* verwendete Verfahren in einer Pseudocode-Darstellung. Eingabe des dargestellten Algorithmus bilden die Menge der statischen Partitionsvorgaben $\mathcal{PAR}_{FIX}$ sowie die Mengen der partitionierbaren Objekte $\mathcal{O}_{part}$. Als Annahme für den Algorithmus gilt, dass für jedes partitionierbare Objekt o_{part} maximal eine statische Partitionsvorgabe existiert. Während einer Iteration über die Menge der statischen Partitionsvorgaben (C1), wird für jede Partitionsvorgabe $f \in \mathcal{PAR}_{FIX}$ dasjenige Objekt o_{part} bestimmt, auf das sich die jeweilige Annotation bezieht und die

Algorithmus 4.1 Partitionierung von FIXEDNODES.

Require: $\forall o_{part} \in O_{part} : |\mathsf{param}_{FIX}(o_{part})| \leq 1 \wedge \mathsf{partition}(o_{part}) = \bot$

```
procedure PARTITIONFIXEDNODES
    for all f ∈ PAR_FIX do                                   ▷ C1
        if ∃ o_part ∈ O_part : param_FIX(o_part) = f then
            partition(o_part) ← partition(f)                 ▷ C2
        end if
    end for
end procedure
```

durch f definierte statische Partitionsvorgabe als Partitionszuordnung für o_{part} übernommen (C2).

Gemäß Anforderung 4.7 muss gelten, dass für sämtliche Aktivitäten, die beim WfMS eingehende Nachrichten empfangen und verarbeiten (d. h. die Aktivitäten aus $\mathcal{A}_{receive} \cup \mathcal{E}_{message}$), nach Abschluss von Phase 1 des Partitionierungsverfahrens eine Partition bestimmt wurde.

4.6 Phase 2: Heavy Nodes

Phase 2 des Partitionierungsverfahrens adressiert die Partitionierung der Interaktionsaktivitäten INVOKE und REPLY, den sogenannten *Heavy Nodes*. Gemäß Anforderung 4.10 berücksichtigt das Partitionierungsverfahren für INVOKE-Aktivitäten sowohl funktionale als auch nicht-funktionale Diensteigenschaften; Grundlage hierfür bilden die (entsprechend Eigenschaft 4.5) erfassten Diensteigenschaften.

Abbildung 4.16 zeigt die zur Partitionierung von *Heavy Nodes* notwendigen Verarbeitungsschritte. Diese sind: (i) Die Bestimmung der Menge der dokumentierten Dienste, die eine zu den funktionalen Anforderungen der jeweiligen INVOKE-Aktivität kompatible Dienstschnittstelle bieten sowie deren nicht-funktionale Anforderungen erfüllen (Phase 2.1). Diese Bestimmung berücksichtigt dabei auch die Einschränkungen, die aus einer mehrfachen Interaktion über denselben PARTNERLINK resultieren (Anforderung 4.12). Die

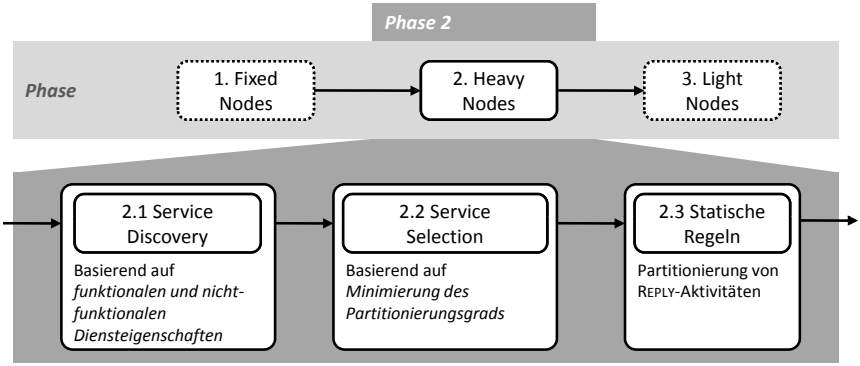


Abbildung 4.16: Teilschritte der Partitionierung von *Heavy Nodes*.

Wahl einer der kompatiblen Dienstimplementierungen erfolgt – aufgrund Anforderung 4.11 – auf Grundlage des Kriteriums der Minimierung des Partitionierungsgrads des Prozesses (Phase 2.2). Die Bestimmung der Partitionen von Reply-Aktivitäten (Phase 2.3) schließt die Phase 2 des Partitionierungsverfahrens ab, indem eine Reply-Aktivität immer derselben Partition zugeordnet wird, der auch die entsprechende Receive-Aktivität zugeordnet wurde (Anforderung 4.8).

Algorithmus 4.2 zeigt eine Hilfsfunktion der *Heavy Node*-Partitionierung auf Grundlage der Menge $\mathcal{A}_{Invoke}$ der Invoke-Aktivitäten des Prozesses und den OSO-Parametern $\mathcal{PAR}_{OSO}$. Während der Ausführung von determinePartnerLinkSets erfolgt eine Iteration über die Menge $\mathcal{A}_{Invoke}$. Dabei wird überprüft, ob die jeweilige Interaktion entlang eines PartnerLink erfolgt, für welchen die Eigenschaft *OneServiceOnly* (vgl. Abschnitt 4.3.1) spezifiziert ist (C1). Gilt für die OSO-Eigenschaft der Wert `true`, so wird in $\mathcal{PL}$ ein Eintrag in Form eines Tupels erzeugt, der zum jeweiligen PartnerLink sämtliche Invoke-Aktivitäten erfasst, die entlang diesem eine Interaktion ausführen. Das dritte Feld des Tupels dient zur Erfassung möglicher Dienstimplementierungen aus $\mathcal{WS}$, die die Anforderungen des jeweiligen PartnerLink bzw. der entsprechenden Invoke-Aktivitäten erfüllen; da dies in einem nachfolgenden Schritt durch die

Algorithmus 4.2 Bestimmung von Dienstaufrufen entlang desselben PARTNERLINK.

procedure DETERMINEPARTNERLINKSETS
  for all $a \in \mathcal{A}_{Invoke}$ **do**
    if $\exists\, p_{OSO} \in \mathsf{param}_{OSO}(\mathsf{partnerLink}_{CO}(a)) :\ p_{OSO} = \texttt{true}$ **then** ▷ C1
      if $\neg\, \exists\, pl \in \mathcal{PL} :\ \pi_1(pl) = \mathsf{partnerLink}_{CO}(a)$ **then**
        $\mathcal{PL} \leftarrow \mathcal{PL} \cup (\mathsf{partnerLink}_{CO}(a), \{a\}, \{\})$
      else
        $\pi_2(pl) \leftarrow \pi_2(pl) \cup a$
      end if
    else
      $\mathcal{PL} \leftarrow \mathcal{PL} \cup (\mathsf{partnerLink}_{CO}(a), \{a\}, \{\})$
    end if
  end for
end procedure

Funktion DETERMINECANDIDATESETS (Algorithmus 4.3) erfolgt, wird das entsprechende Feld des Tupels hier lediglich mit der leeren Menge belegt. Gilt für die *OSO*-Eigenschaft der Wert `false`, so wird für jede INVOKE-Aktivität über diesen PARTNERLINK ein eigener Eintrag in $\mathcal{PL}$ erzeugt, was in nachfolgenden Verarbeitungsschritten die in Anforderung 4.12 beschriebene Flexibilisierung des Partitionierung von INVOKE-Aktivitäten zur Folge hat.

Die Bestimmung des Werts des letzten Felds der Elemente von $\mathcal{PL}$ wird durch die in Algorithmus 4.3 vorgestellte Funktion DETERMINECANDIDATESETS realisiert. Diese bestimmt den Wert von $\pi_3(pl)$ für alle $pl \in \mathcal{PL}$ und erfasst in diesem funktional sowie nicht-funktional kompatible Dienstimplementierungen. Das Verfahren prüft hierzu für jedes Element pl der Menge $\mathcal{PL}$ (welche als Teil der Ausführung von Algorithmus 4.2 erstellt wurde) zunächst, ob für eine Aktivität $a \in \pi_2(pl)$, d.h. der Menge der INVOKE-Aktivitäten, die über den jeweiligen PARTNERLINK interagieren, eine statische Partitionsvorgabe existiert. Ist dies der Fall, so werden sämtliche Aktivitäten aus $\pi_2(pl)$ aufgrund Anforderung 4.12 der statisch vorgegebenen Partition zugeordnet (C1). Aufgrund der statischen Partitionsvorgabe kann das Suchen einer kompatiblen Dienstimplementierung hier entfallen. Existieren unterschiedliche statische Partitionsvorgaben für

Algorithmus 4.3 Bestimmung kompatibler Dienstimplementierungen.

```
procedure DETERMINECANDIDATESETS
    for all pl ∈ 𝒫ℒ do
        if ∃ a ∈ π2(pl) : paramFIX(a) ≠ ⊥ then                          ▷ C1
            if ∃ a′ ∈ π2(pl) : partition(paramFIX(a′)) ≠ paramFIX(a) then
                ERROR("conflicting partition assignments")
            else
                π3(pl) ← π3(pl) ∪ partition(paramFIX(a))
            end if
        else                                                             ▷ C2
            if ∃ 𝒲𝒮cand ⊆ 𝒲𝒮, ∀ a ∈ π2(pl), ∀ wscand ∈ 𝒲𝒮cand :
                (a ⪯f wscand) ∧ (a ⪯nf wscand) then
                for all wscand ∈ 𝒲𝒮cand do
                    π3(pl) ← π3(pl) ∪ partition(wscand)
                end for
            else
                ERROR("no suitable service")                             ▷ C3
            end if
        end if
    end for
end procedure
```

einzelne INVOKE-Aktivitäten aus $\pi_2(pl)$, so bricht das Verfahren mit einem Fehler ab. Der Fall dass für eine bestimmte Aktivität aus $\pi_2(pl)$ unterschiedliche statische Partitionsvorgaben existieren wird durch die Eigenschaft $\forall o_{part} \in \mathcal{O}_{part} : |\mathsf{param}_{FIX}(o_{part})| \leq 1$ in Algorithmus 4.1 ausgeschlossen.

Existiert keine statische Partitionsvorgabe, so wird für jede INVOKE-Aktivität in $\pi_2(pl)$ die Menge $\mathcal{WS}_{cand}$ der funktional und nicht-funktional kompatiblen Dienste bestimmt und deren Partitionen in $\pi_3(pl)$ abgelegt (C2).

Falls weder eine statische Partitionsvorgabe existiert noch eine kompatible Dienstimplementierung gefunden werden kann, bricht das Verfahren mit einem Fehler ab (C3); ist Eigenschaft 4.4 erfüllt, so kann dieser Fall allerdings nicht eintreten.

Neben der Partition der gefundenen kompatiblen Dienstimplementierungen werden an dieser Stelle weitere Informationen über die jeweilige gefunde-

ne Dienstimplementierung erfasst. Diese werden während des Deployment-Schritts (vgl. Abschnitt 5.8) zur Konfiguration der PS-Klienten verwendet, die die jeweiligen INVOKE-Aktivitäten implementieren. Da diese Informationen keinen Einfluss auf den weiteren Verlauf des Partitionierungsvorgangs haben, wurde in der Formalisierung des Partitionierungsverfahrens aus Gründen der Übersichtlichkeit auf deren explizite Darstellung verzichtet.

Algorithmus 4.4 zeigt das zur Partitionierung von *Heavy Nodes* verwendete Verfahren. Der Algorithmus realisiert die in Abbildung 4.16 dargestellten Phasen nicht streng sequentiell, sondern adressiert diese teilweise überlappend. Zunächst wird die Menge $\mathcal{PL}$ durch Aufruf der Funktion DETERMINEPARTNERLINKSETS erzeugt und deren Elemente durch Aufruf der Funktion DETERMINECANDIDATESETS um mögliche Dienstkandidaten ergänzt.

Die Erfüllung der Anforderung nach Minimierung des Partitionierungsgrads (Anforderung 4.11) erfolgt durch die Anwendung des *Greedy Algorithmus* zur Lösung des sogenannten *Set-Cover*-Problems [CLRS09]. Zur Anwendung des Algorithmus beinhaltet die Menge $\mathcal{U}$ die zu partitionierenden INVOKE-Aktivitäten (C1) und die Menge $\mathcal{F}$ Tupel bestehend aus einer Partition und der Menge der INVOKE-Aktivitäten, für die in der jeweiligen Partition eine kompatible Dienstimplementierung gefunden wurden (C2). Nun wird fortlaufend jeweils die Partition bestimmt, die die Anforderungen der meisten INVOKE-Aktivitäten erfüllt (C3). Die entsprechenden Aktivitäten werden daraufhin $\mathcal{U}$ entnommen (C4) und die Aktivitäten der Menge $\pi_2(f)$ der jeweiligen Partition zugeordnet (C5).

Im Anschluss hieran erfolgt die Partitionierung von REPLY-Aktivitäten (C6). Da eine REPLY-Aktivität immer die Existenz einer Aktivität aus $\mathcal{A}_{receive} \cup \mathcal{E}_{message}$ bedingt (und dieser gemäß Anforderung 4.7 bereits in Phase 1 einer entsprechenden Partition zugeordnet wurde), erfolgt die Zuordnung der Aktivität zur Partition der jeweiligen Aktivität aus $\mathcal{A}_{receive} \cup \mathcal{E}_{message}$ gemäß Anforderung 4.8.

Es ist zu bemerken, dass die Anwendung von Algorithmus 4.4 lediglich eine Erstpartitionierung der Interaktionsaktivitäten des Prozesses auf Grundlage des Kriteriums einer möglichst großen Nähe zwischen Interaktionsaktivität und verwendeter Dienstimplementierung zum Partitionierungszeitpunkt vornimmt (vgl. Anforderung 4.9). Zur Laufzeit einer Prozessinstanz kann allerdings –

Algorithmus 4.4 Partitionierung von HEAVYNODES.

procedure PARTITIONHEAVYNODES
DETERMINEPARTNERLINKSETS()
DETERMINECANDIDATESETS()

$\mathcal{U} \leftarrow \bigcup_{pl \in \mathcal{PL}} \pi_2(pl)$ ▷ C1
$\mathcal{F} \leftarrow \{\}$
for all $pl \in \mathcal{PL}$ **do** ▷ C2
for all $p \in \pi_3(pl)$ **do**
if $\exists f \in \mathcal{F} : \pi_1(f) = p$ **then**
$\pi_2(f) \leftarrow \pi_2(f) \cup \pi_2(pl)$
else
$\mathcal{F} \leftarrow \mathcal{F} \cup (p, \pi_2(pl))$
end if
end for
end for

while $\mathcal{U} \neq \emptyset$ **do**
wähle $f \in \mathcal{F}$ mit maximalem $|\pi_2(f) \cap \mathcal{U}|$ ▷ C3
$\mathcal{U} \leftarrow \mathcal{U} \backslash \pi_2(f)$ ▷ C4
for all $a \in \pi_2(f)$ **do**
$\mathsf{partition}(a) \leftarrow \pi_1(f)$ ▷ C5
end for
end while

for all $a \in \mathcal{A}_{Reply} :\ \mathsf{partition}(a) = \perp$ **do**
if $\exists\ a' \in (\mathcal{A}_{receive} \cup \mathcal{E}_{message}) :$
$\mathsf{partnerLink}_{CO}(a') = \mathsf{partnerLink}_{CO}(a)$ **then**
$\mathsf{partition}(a) \leftarrow \mathsf{partition}(a')$ ▷ C6
end if
end for
end procedure

beispielsweise bedingt durch die Möglichkeit der dynamischen Zuweisung einer ENDPOINTREFERENCE auf einen PARTNERLINK – auch eine Interaktion mit einer anderen Dienstimplementierung (potentiell auch aus einer anderen Partition)

notwendig werden. In diesem Fall erfolgt die Interaktion mit der jeweiligen Dienstimplementierung als *entfernte* Interaktion (vgl. Abschnitt 3.4).

4.7 Phase 3: Light Nodes

Abbildung 4.17 verdeutlicht die Teilschritte der Phase 3 des entwickelten Partitionierungsverfahrens, der sogenannten *Light-Node*-Partitionierung, graphisch.

Basis für die *Light-Node*-Partitionierung ist die Bestimmung der Interaktionen zwischen PS-Klienten und PS, die zur Ausführung der einzelnen Aktivitäten des zu partitionierenden Prozesses notwendig sind (Phase 3.1). Der Beschreibung in Abschnitt 3.2 entsprechend beinhalten diese sowohl den Zugriff auf Informationen, die den Prozesskontrollfluss betreffen, als auch auf sichtbare und unsichtbare Instanzdaten; jeder dieser Datenzugriffe wird in der Folge als sogenannte *Datenabhängigkeit* bezeichnet.

Auf Grundlage dieser Datenabhängigkeiten wird ein *parametrisierter Partitionierungsgraph* des Prozesses erstellt, in welchem die durchgeführten Datenzugriffe und die in Abschnitt 4.3.1 erläuterten Partitionierungsparameter (z. B. kommuniziertes Datenvolumen von Instanzdatenzugriffen, Wahrscheinlichkeit der Aktivitätsausführung und bereits bestimmte Partitionen einzelner Partitionierungsobjekte) miteinander in Verbindung gebracht werden (Phase

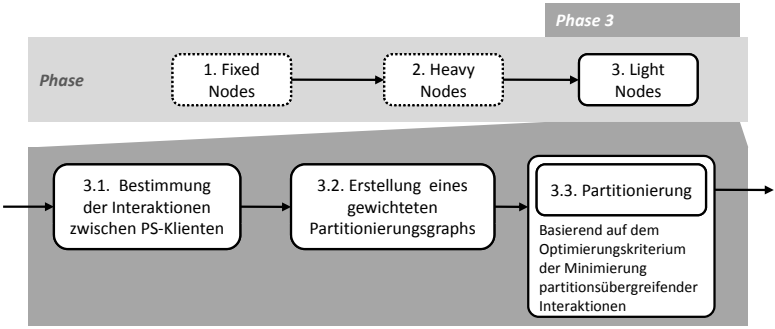


Abbildung 4.17: Partitionierung von *Light Nodes*.

3.2).

Die Knoten des Partitionierungsgraphs, denen nach Abschluss der Phasen 1 und 2 noch keine Partition zugewiesen ist, sind sogenannte *Light Nodes* und werden während Phase 3 des Partitionierungsverfahrens bearbeitet. Der eigentliche *Light-Node*-Partitionierungsvorgang erfolgt durch Reduktion auf ein kombinatorisches Optimierungsproblem als Teil von Phase 3.3, welches unter Verwendung des *Simulated Annealing*-Verfahrens [KGV83] gelöst wird. Die Anwendung dieses Lösungsverfahrens verlangt die Möglichkeit zur Bewertung der möglichen Lösungen des vorliegenden Optimierungsproblems; realisiert wird dies in Form einer Funktion, die jede Lösung des Problems auf einen Kostenwert abbildet. Im Rahmen dieser Dissertation wird exemplarisch eine Kostenfunktion definiert, welche das Ziel der Minimierung partitionsübergreifender Interaktionskosten zum Ziel hat. Durch Definition einer alternativen Kostenfunktion kann das hier vorgestellte Partitionierungsverfahren für *Light Nodes* auf andere Optimierungskriterien hin angepasst und erweitert werden (siehe Abschnitt 4.7.3).

4.7.1 Phase 3.1: Bestimmung der Interaktionen von PS-Klienten

In Abschnitt 3.2 wurde das EWFN-Metamodell zur Beschreibung der Koordination und Kommunikation potentiell verteilter Interaktionspartner auf Grundlage des Petri-Netz-Formalismus eingeführt. In [Mar10] ist ein EWFN zur Beschreibung von BPEL-Prozessen definiert durch:

$$\text{EWFN}_{\text{BPEL}} = \langle \Sigma, \Phi, P, T, S, F, X, A, B, I_0, L_{write} \rangle$$

Die für die weitere Erläuterung des Partitionierungsverfahrens relevanten Teile der Definition von $\text{EWFN}_{\text{BPEL}}$ werden an dieser Stelle aus Gründen eines besseren Verständnis kurz zusammengefasst. Entsprechend Petri-Netzen besteht ein EWFN aus einer Menge von Transitionen T (in der Ausführungsumgebung werden diese durch PS-Klienten repräsentiert) und einer Menge von Stellen P (bzw. PS in der Ausführungsumgebung). Die Elemente von T und P können jeweils durch gerichtete Kanten der Menge S verbunden

werden. Definiert durch die Funktion $F : S \rightarrow (T \times P) \cup (P \times T \times R)$ mit $R = \{\texttt{read}, \texttt{take}, \texttt{readall}, \texttt{takeall}, \texttt{update}, \texttt{sync}\}$ verbindet eine Kante entweder eine Transition und eine Stelle oder eine Stelle und eine Transition. Im Fall einer Kante zwischen einer Stelle und einer Transition ist diese zusätzlich mit einem Operationstyp aus R annotiert. Durch die Verbindung aus (i) der Funktion F, (ii) der Funktion $A : (P \times T \times R) \rightarrow X$, welche eingehenden Kanten einer Transition jeweils ein Linda-Template zuweist, (iii) der Funktion $L_{read} : (X \times 2^{\Sigma^{MS}}) \rightarrow \Sigma$, welche eine Linda-Leseoperation auf einem Multiset von Tupeln repräsentiert, und (iv) der Funktion $L_{write} : (T \times P) \rightarrow \Sigma$, welche den Vorgang des Schreibens eines Tupels durch eine Transition beschreibt, sind die für die Ausführung eines PS-Klienten erforderlichen Interaktionen vollständig definiert.

Für die Realisierung von Phase 3.1 sind also, über die Bestimmung der EWFN-Repräsentation des zu partitionierenden Prozesses hinaus, keine zusätzlichen Verarbeitungsschritte notwendig. Diese kann also direkt als Eingabe für Phase 3.2 des Partitionierungsverfahrens verwendet werden.

4.7.2 Phase 3.2: Erstellen des Partitionierungsgraphs

Die Berücksichtigung der in Abschnitt 4.3.1 vorgestellten Partitionierungsparameter in der Bewertung einer Partitionierung eines Prozesses erfolgt in Form gewichteter EWFN-Kanten.

Dabei gilt, dass das Gewicht einer Kante beeinflusst wird durch: (i) die Ausführungswahrscheinlichkeit der BPEL-Aktivität, zu deren EWFN-Repräsentation die jeweilige Kante gehört, (ii) die Anzahl, wie oft diese Aktivität ausgeführt wird, und (iii) die mit den, durch eine Kante repräsentierten, Datenzugriff verbundenen (Kommunikations-) Kosten. Aufgrund der Eigenschaften von BPEL, wonach die Ausführung eines Prozesses potentiell zu jedem Zeitpunkt (und an jeder Stelle) durch beispielsweise ein asynchron auftretendes Ereignis in einem EventHandler oder einen Fehler (und dessen Behandlung durch Fault- und Compensation-Handler) unterbrochen werden kann, ist eine exakte Bestimmung der Ausführungswahrscheinlichkeit der Aktivitäten sowie deren Ausführungsshäufigkeit im Allgemeinen nicht möglich (vgl. Eigenschaft 4.16).

Hieraus folgt, dass auch die bestimmten Kantengewichte den tatsächlichen Verlauf der Ausführung des EWFN-Modells lediglich approximativ widerspiegeln. Hinsichtlich einer Prozesspartitionierung die auf Grundlage dieser approximativen Kantengewichte erzeugt wurde, gilt allerdings dennoch die nachfolgend definierte Korrektheitseigenschaft.

Eigenschaft 4.29 (Korrektheit einer Partitionierung)**.** *Es gilt, dass sämtliche Partitionszuordnungen, die in einer Ausführungssemantik resultieren, die nicht der zentralen Ausführung des Prozesses entspricht (vgl. Anforderung 4.3), durch die Phasen 1 und 2 des Partitionierungsverfahrens adressiert werden.*

Aufgrund dieser Eigenschaft gilt, dass jede mögliche Partitionszuordnung der Light Nodes *des Prozesses in Phase 3 des Partitionierungsverfahrens zu einer korrekten Ausführungssemantik führt, solange sichergestellt ist, dass die Prozesspartitionierung vollständig erfolgt, d. h. nach Durchlaufen von Phase 3 jeder Stelle bzw. jeder Transition eines EWFN-Modells genau eine Partition zugeordnet ist. Dies wird durch die nachfolgend vorgestellten Algorithmen sichergestellt (vgl. Definition 4.33).*

Im Besonderen bedeutet dies, dass auch eine, hinsichtlich des gewählten Optimierungskriteriums der Light Node*-Partitionierung, nicht optimale Partitionierung lediglich zu einem nicht optimalen Laufzeitverhalten führen kann, niemals jedoch zu einer, im Vergleich zum ursprünglichen BPEL-Prozess, veränderten und damit nicht korrekten Ausführungssemantik.*

Aufgrund der oben genannten Eigenschaft gilt, dass im Besonderen auch die lediglich approximative Bestimmung der Ausführungswahrscheinlichkeit niemals zu einer nicht korrekten Partitionierung eines Prozesses führen kann.

Die Notwendigkeit, den Kanten eines EWFN-Modells ein Kantengewicht zuordnen zu können, erfordert eine Erweiterung des in [Mar10] definierten Metamodells $\text{EWFN}_{\text{BPEL}}$ zum sogenannten *Partitionierungsgraph* $\text{EWFN}_{\text{Part}}$.

Definition 4.30 (EWFN-Partitionierungsgraph)**.** *Ein Partitionierungsgraph, welcher als Grundlage für die* Light-Node*-Partitionierung verwendet wird, ist definiert als:*

$$EWFN_{Part} = \langle \Sigma_{BPEL}, \Phi, P, T, S, \tilde{F}, X, A, B, I_0, L_{write} \rangle$$

Wobei die Funktion $\tilde{F}$ die Funktion F von $EWFN_{BPEL}$ beschreibt, welche um ein Kantengewicht erweitert wurde. Sie ist definiert als:

$$\tilde{F} : S \to (T \times P \times \mathbb{R}) \cup (P \times T \times R \times \mathbb{R})$$

Die anderen Mengen in der Definition von $EWFN_{Part}$ entsprechen ihrer Definition in $EWFN_{BPEL}$.

Die Vorgehensweise zur Überführung eines EWFN-Modells vom Typ $\text{EWFN}_{\text{BPEL}}$ in einen Modell vom Typ $\text{EWFN}_{\text{Part}}$ wird im weiteren Verlauf dieses Abschnitts erläutert. Für die Beschreibung der während dieser Überführung notwendigen Verarbeitungsschritte werden, zusätzlich zu den in [Mar10] und [KML08] aufgeführten Funktionen, die folgenden weiteren Hilfsfunktionen verwendet.

Zur Vereinfachung der Darstellung der einzelnen Algorithmen erlaubt die Funktion $\text{weight}_S : S \to \mathbb{R}$ direkten Zugriff auf das Gewicht einer Kante eines EWFN, unabhängig davon, ob diese eine Stelle mit einer Transition oder eine Transition mit einer Stelle verbindet. Sie ist für alle $s \in S$ definiert als:

$$\text{weight}_S(s) \mapsto \begin{cases} \pi_3(\tilde{F}(s)) & \text{wenn } \pi_1(\tilde{F}(s)) \in T \wedge \pi_2(\tilde{F}(s)) \in P \\ \pi_4(\tilde{F}(s)) & \text{wenn } \pi_1(\tilde{F}(s)) \in P \wedge \pi_2(\tilde{F}(s)) \in T \end{cases}$$

Weiterhin erlaubt die Funktion $\text{isDataAccess}_S : S \to \mathbb{B}$ die Unterscheidung, ob eine Kante eines EWFN einen Zugriff auf ein Instanzdatum beschreibt. Die Funktion ist auf Grundlage der Struktur der entlang einer Kante verarbeiteten Tupel aus Σ_{BPEL} für alle $s \in S$ definiert durch:

$$\mathsf{isDataAccess}_S(s) \mapsto \begin{cases} \texttt{true} & \text{wenn } (\pi_{1,2,3}(\tilde{F}(s)) \in (P \times T \times R) \wedge \\ & \quad \pi_1(A(\pi_{1,2,3}(\tilde{F}(s)))) = \texttt{DATA}) \vee \\ & (\pi_{1,2}(\tilde{F}(s)) \in (T \times P) \wedge \\ & \quad \pi_1(L_{write}(\pi_{1,2}(\tilde{F}(s)))) = \texttt{DATA}) \\ \texttt{false} & \text{sonst} \end{cases}$$

Die Herstellung des Bezugs zwischen einem Element eines EWFN-Modells und dem entsprechenden Element des BPEL-Prozesses, der als Eingabe für die BPEL-EWFN-Transformation diente, erfolgt in der Beschreibung der Algorithmen durch die Funktionen mapsToBPELElement bzw. mapsToParam.

Definition 4.31 (Bezug zwischen einem Element des EWFN-Modells und dessen zugehörigen BPEL-Element)**.** *Die Funktion* mapsToBPELElement : $P \cup S \cup T \rightarrow O_{part}$ *ordnet einem Element eines EWFN (d. h. einer Stelle, einer Transition oder einer Kante) genau das Element des BPEL-Modells des Prozesses zu, zu dessen EWFN-Pattern das jeweilige EWFN-Element gehört. Realisiert wird* mapsToBPELElement *dadurch, dass mit jedem EWFN-Element, bereits zum Zeitpunkt seiner Erzeugung während der BPEL-EWFN-Transformation [Mar10], ein Identifikator desjenigen BPEL-Modell-Elements assoziiert wird, das als Eingabe für den Transformationsvorgang diente.*

Definition 4.32 (Partitionierungsparameter eines EWFN-Elements)**.** *Aufbauend auf* mapsToBPELElement *erlaubt die Funktion* mapsToParam : $P \cup S \cup T \rightarrow 2^{\mathcal{PAR}} \cup \perp$ *den Zugriff auf die Partitionierungsparameter des Partitionierungsobjekts, zu deren EWFN-Pattern das jeweilige EWFN-Element gehört. Analog der Funktion* param *existieren jeweils einzelne Funktionen für den Zugriff auf Parametrisierung eines bestimmten Typs, z. B.* $\mathsf{mapsToParam}_{VAC}$*. Weiterhin realisiert die Funktion* mapsToParam *auch eine Abbildung von Parametern in Listen-Form (*$\mathcal{PAR}_{VACL}$*,* $\mathcal{PAR}_{EPL}$*), auf die jeweiligen Typen ihrer Elemente (d. h. VAC, EP). So gilt beispielsweise, dass die Funktion* $\mathsf{mapsToParam}_{VAC}$ *bei Anwendung auf eine EWFN-Kante, die einen Zugriff auf die* `inputVariable` *einer* INVOKE*-Aktivität* $a \in \mathcal{A}_{Invoke}$ *repräsentiert, den jeweils entsprechenden Wert aus* $\mathsf{param}_{VACL}(a)$ *zurückliefert.*

Die Realisierung von mapsToParam *erfolgt analog* mapsToBPELElement *auf Grundlage der, während der BPEL-EWFN-Transformation eingefügten, BPEL-Element-Identifikatoren.*

Zur Erfassung der Resultate der approximativen Berechnung der Ausführungswahrscheinlichkeit und der Ausführungsanzahl von Aktivitäten in den folgenden Algorithmen ordnen die Funktionen noi : $O_{part} \rightarrow \mathbb{N} \cup \perp$ und ep : $O_{part} \rightarrow \mathbb{R} \cup \perp$ einem Partitionierungsobjekt seine Ausführungsanzahl bzw. Ausführungswahrscheinlichkeit zu.

Algorithmus 4.5 zeigt die Realisierung der Funktion GENERATEPARTITIONINGGRAPH, die zu einem gegebenen EWFN-Modell und dessen Partitionierungsparametern einen entsprechenden EWFN-Partitionierungsgraph erzeugt.

Die approximative Bestimmung der Ausführungsanzahl einer Aktivität erfolgt durch die Betrachtung der Parameter $\mathcal{PA}_{NOI}$ der umgebenden Iterationsaktivitäten WHILE, REPEATUNTIL und FOREACH sowie umgebender EVENTHANDLER (C1), welche jeweils multiplikativ in die Bestimmung der Ausführungshäufigkeit der betrachteten Aktivität eingehen. Die Bestimmung der Ausführungswahrscheinlichkeit der einzelnen Aktivitäten kann auf unterschiedliche Weise erfolgen. Hat der Partitionierer die erwartete Ausführungswahrscheinlichkeit einer Aktivität direkt durch einen Parametrisierung aus P_{Exec} festgelegt, so wird deren Wert für die Ausführungswahrscheinlichkeit der betrachteten Aktivität übernommen (C2). Existiert keine derartige Parametrisierung, so wird die Ausführungswahrscheinlichkeit durch die Funktion EP (Algorithmus 4.6) berechnet (C3); diese wird im Anschluss an die Erläuterung von GENERATEPARTITIONINGGRAPH vorgestellt. Auf Basis der bestimmten Werte für Ausführungsanzahl und Ausführungswahrscheinlichkeit wird für jede Kante des EWFN ein Kantengewicht bestimmt (C4). Diese Bestimmung ist dabei abhängig von der Art der Interaktion, die durch eine Kante beschrieben wird. Handelt es sich bei einer Kante um einen Zugriff auf ein Instanzdatum, so gehen für diesen (i) die, mittels des *VAC*-Parameters definierten Kosten für den Instanzdatenzugriff, (ii) die Wahrscheinlichkeit für die Ausführung der jeweiligen Aktivität und (iii) die Anzahl der Ausführungen der jeweiligen Aktivität multiplikativ ein (C5). Handelt es sich bei einer Kante hingegen um eine Kommunikation von Informationen, die

Algorithmus 4.5 Erzeugung des Partitionierungsgraphs zu einem gegebenen EWFN.

Require: $\forall a \in \mathcal{A} : \mathsf{noi}(a) \leftarrow 1$

function GENERATEPARTITIONINGGRAPH
  for all $a_{cur} \in \mathcal{A}$ **do**
    for all $a_{parent} \in \mathsf{parents}(a_{cur})$ **do**
      if $a_{parent} \in \mathcal{A}_{While} \cup \mathcal{A}_{RepeatUntil} \cup \mathcal{A}_{ForEach} \cup \mathcal{T}_{normal}$ **then**
        $\mathsf{noi}(a_{cur}) \leftarrow \mathsf{noi}(a_{cur}) \cdot \mathsf{param}_{NOI}(a_{parent})$ ▷ C1
      end if
    end for
    if $\mathsf{param}_{P_{Exec}}(a_{cur}) \neq \perp$ **then** ▷ C2
      $\mathsf{ep}(a_{cur}) \leftarrow \mathsf{param}_{P_{Exec}}(a_{cur})$
    else ▷ C3
      $\mathsf{ep}(a_{cur}) \leftarrow \mathrm{EP}(a_{cur})$
    end if
  end for

  for all $s \in S$ **do** ▷ C4
    $a \leftarrow \mathsf{mapsToBPELElement}(s)$
    if $\mathsf{isDataAccess}(s)$ **then**
      $c \leftarrow \mathsf{mapsToParam}_{VAC}(s) \cdot \mathsf{ep}(a) \cdot \mathsf{noi}(a)$ ▷ C5
    else
      $c \leftarrow 1 \cdot \mathsf{noi}(a)$ ▷ C6
    end if
    $\tilde{F}(s) \leftarrow F(s) \circ c$ ▷ C7
  end for
return $\tilde{F}$
end function

den Prozesskontrollfluss betreffen, so werden diese konstant mit dem Faktor 1 gewichtet und mit der Zahl der Ausführungen einer Aktivität multipliziert (C6). Die Berücksichtigung der Ausführungswahrscheinlichkeit der Aktivität entfällt hier, da auch im Fall einer fehlerhaften Aktivitätsausführung ein sogenanntes *Dead*-Token entlang der Kontrollkante kommuniziert werden muss [Mar10].

Das auf diese Weise bestimmte Gewicht c einer Kante s wird durch den

Konkatenationsoperator $\circ$ als letzter Wert an das Tupel $F(s)$ angehängt (C7). Das Resultat dieser Konkatenation ist die entsprechende gewichtete Kante $\tilde{F}(s)$.

Algorithmus 4.6 Bestimmung der Ausführungswahrscheinlichkeit einer BPEL-Aktivität.

```
function EP(a ∈ 𝒜)
    if parent(a) ∈ 𝒜_Flow ∪ 𝒜_Scope then
        return EPINFLOW(a)
    else if parent(a) ∈ 𝒜_Sequence then
        return EPINSEQUENCE(a)
    else if parent(a) ∈ 𝒜_If then
        return EPINIF(a)
    else if parent(a) ∈ 𝒜_While ∪ 𝒜_RepeatUntil ∪ 𝒜_ForEach then
        return EPINLOOP(a)
    else if parent(a) ∈ ℰ then
        return EPINHANDLER(a)
    else if parent(a) = ⊥ then
        return 1
    end if
end function
```

Algorithmus 4.6 zeigt die Funktion EP zur approximativen Bestimmung der Ausführungswahrscheinlichkeit einer BPEL-Aktivität. Da die Berechnung der Ausführungswahrscheinlichkeit einer Aktivität unter anderem abhängig vom Typ ihrer Eltern-Aktivität ist, wurde die Funktion EP in die Funktionen EPINFLOW, EPINSEQUENCE, EPINIF, EPINLOOP und EPINHANDLER aufgeteilt. Die nachfolgend dargestellte Beschreibung der Berechnung der Ausführungswahrscheinlichkeit durch diese Funktionen umfasst jeweils eine graphische Darstellung der betrachteten Kontrollabhängigkeiten und das jeweilige Berechnungsverfahren in einer Pseudocode-Darstellung. Für das vorgestellte Verfahren gilt dabei, zusätzlich zu den Annahmen in Abschnitt 4.2.4, dass ein negatives Ergebnis der Evaluierung einer JOINCONDITION nicht zur Auslösung eines Fehlers führt (es gilt also SUPPRESSJOINFAILURE = `true`[1]) sowie, dass ein eventueller gegenseitiger Ausschluss der TRANSITIONCONDITION mehrerer ausgehender Syn-

[1] Prozesse die diese Eigenschaft nicht berücksichtigen, werden durch das in Abschnitt 7.3 vorgestellte Partitionierungswerkzeug zurückgewiesen.

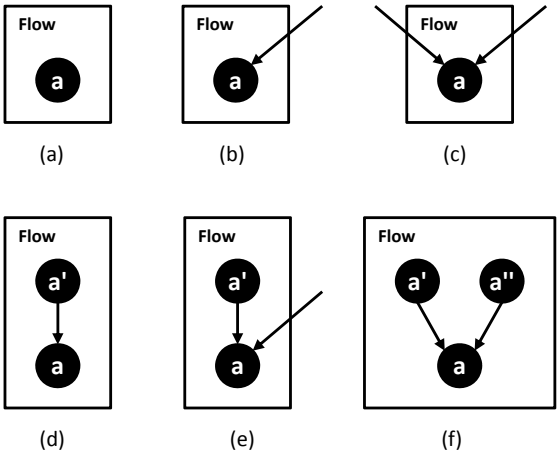


Abbildung 4.18: Betrachtete Szenarien für die Berechnung der Ausführungswahrscheinlichkeit der Kind-Aktivitäten einer FLOW-Aktivität.

chronisationskanten einer Aktivität in einem FLOW durch das Verfahren nicht gesondert betrachtet wird.

4.7.2.1 FLOW-Aktivität

Für die Berechnung der Ausführungswahrscheinlichkeit der Kind-Aktivitäten einer FLOW-Aktivität werden die in Abbildung 4.18 dargestellten Szenarien betrachtet[1]. Szenarien (a)-(c) zeigen Fälle, in welchen die betrachtete Aktivität a die erste Aktivität in einem FLOW ist und Kontrollflussabhängigkeiten zu Aktivitäten außerhalb der FLOW-Aktivität bestehen (diese also Kind-Aktivitäten einer umgebenden FLOW-Aktivität sind). Die Fälle (d)-(f) zeigen Szenarien mit sowohl FLOW-internen als auch FLOW-externen Kontrollflussabhängigkeiten.

Einfluss auf die Bestimmung der Ausführungswahrscheinlichkeit P_{Exec} der Kind-Aktivität a einer FLOW-Aktivität haben (i) die Wahrscheinlichkeit, dass die

[1] Sowohl für diese als auch die im Folgenden vorgestellten Szenarien gilt, dass Aktivität a die jeweils betrachtete Aktivität ist und Aktivität a' (bzw. a'') zur Konstruktion unterschiedlicher Kontrollflussabhängigkeiten verwendet wird.

FLOW-Aktivität (parent(a)) zur Ausführung kommt und (ii) die JOINCONDITION von a zu einem Booleschen `true` evaluiert wird.

Da es sich bei der JOINCONDITION einer Aktivität um einen Booleschen Ausdruck auf den Status der eingehenden Synchronisationskanten handelt, kann die Bestimmung der Wahrscheinlichkeit für eine positive Evaluierung der JOINCONDITION einer Aktivität wie folgt realisiert werden: (i) Die JOINCONDITION wird zunächst unter Verwendung der *De Morgan'schen Gesetze* [Goo07] in ihre *Disjunktive Normalform (DNF)* überführt. Im einfachsten Fall kann dies durch eine wiederholte Anwendung der Regel $P \wedge (Q \vee R) \rightarrow (P \wedge Q) \vee (P \wedge R)$ realisiert werden; da verschiedene Umsetzungen dieser Transformation existieren (z. B. in Form der Bibliothek *Orbital*[1] wird auf eine detaillierte Beschreibung an dieser Stelle verzichtet. (ii) Anschließend erfolgt die Bestimmung der Wahrscheinlichkeit einer Evaluierung zu einem Booleschen `true` auf Grundlage der Wahrscheinlichkeiten positiver LINK-Status der eingehenden Synchronisationskanten der Aktivität (vgl. Parameter P_{Link}, Abschnitt 4.3.1) und der Annahme stochastischer Unabhängigkeit [DH04]. In den folgenden Algorithmen werden diese Schritte durch die Funktion join : $\mathcal{A} \rightarrow \mathbb{R}$ realisiert; diese wird nachfolgend anhand eines Beispiels erläutert.

Angenommen, eine Aktivität a hat Kontrollflussabhängigkeiten in Form von Synchronisationskanten l_1, l_2, l_3 zu drei vorangehenden Aktivitäten a_1, a_2, a_3 in einem FLOW. Für die JOINCONDITION von a gelte beispielhaft: $l_1 \wedge (l_2 \vee (\neg l_3))$. Jede der Synchronisationskanten ist mit einer TRANSITIONCONDITION belegt. Als Teil der Parametrisierung hat der Partitionierer die folgenden Werte für einen positiven LINK-Status annotiert: $\mathsf{param}_{P_{Link}}(l_1) = 0.5$, $\mathsf{param}_{P_{Link}}(l_2) = 0.3$, $\mathsf{param}_{P_{Link}}(l_3) = 0.1$.

Unter einmaliger Anwendung der genannten Umformungsregel wird die JOINCONDITION in ihre disjunktive Normalform überführt; diese lautet: $(l_1 \wedge l_2) \vee (l_1 \wedge (\neg l_3))$.

Die Wahrscheinlichkeit (dargestellt durch P) für eine positive Evaluierung dieser DNF der JOINCONDITION von a berechnet sich, auf Grundlage obiger

[1] http://symbolaris.com/orbital/

Annahmen, zu:

$$
\begin{aligned}
\mathsf{join}(a) &= P((l_1 \cap l_2) \cup (l_1 \cap \overline{l_3})) \\
&= P(l_1 \cap l_2) + P(l_1 \cap \overline{l_3}) - (P(l_1 \cap l_2) \cdot P(l_1 \cap \overline{l_3})) \\
&= (P(l_1) \cdot P(l_2)) + (P(l_1) \cdot (1 - P(l_3))) \\
&\quad - ((P(l_1) \cdot P(l_2)) \cdot (P(l_1) \cdot (1 - P(l_3)))) \\
&= (0.5 \cdot 0.3) + (0.5 \cdot 0.9) - ((0.5 \cdot 0.3) \cdot (0.5 \cdot 0.9)) \\
&= 0.15 + 0.45 - (0.15 \cdot 0.45) \\
&= 0.5325
\end{aligned}
$$

Damit die Bestimmung der JoinCondition von a wie oben beschrieben erfolgen kann, müssen weiterhin die folgenden Eigenschaften berücksichtigt werden:

- Die TransitionCondition eines Link wird nur dann evaluiert, wenn die Quell-Aktivität des jeweiligen Link ausgeführt wird. Gilt dies nicht, so wird der Status des Link nicht durch Evaluierung von dessen TransitionCondition sondern – aus Gründen der Realisierung der DPE – zu einem Booleschen `false` bestimmt.
- Propagiert die Quell-Aktivität während ihrer Ausführung einen Fehler an ihre umgebende Scope-Aktivität, so wird der Fehlerbehandlungsmechanismus von BPEL ausgelöst, der eine Terminierung des Scope der Quell-Aktivität zur Folge hat. In dem Fall, dass Quelle und Ziel des Link Kind-Aktivitäten desselben Scope sind, führt diese Terminierung insbesondere zu einer Nichtausführung der Ziel-Aktivität[1]. In dem Fall, dass Quelle und Ziel des Link Kind-Aktivitäten unterschiedlicher Scope-Aktivitäten sind, führt die Terminierung des Scope der Quell-Aktivität dazu, dass der Status der ausgehenden Kanten der Quell-Aktivität zu einem Booleschen `false` bestimmt wird.

[1] Zusätzlich gilt, dass in diesem Fall auch sämtliche andere Aktivitäten im jeweiligen Scope nicht ausgeführt werden, auch dann, wenn diese selbst nicht in einer Kontrollabhängigkeit zur Quell-Aktivität stehen. Gemäß Eigenschaft 4.17 werden diese Fälle durch die hier vorgestellte Berechnung allerdings nicht betrachtet.

Aus den oben genannten Gründen gehen zusätzlich zu $\mathsf{join}(a)$ auch Ausführungswahrscheinlichkeit P_{Exec}, sowie die Wahrscheinlichkeit der erfolgreichen Ausführung P_{Succ} dieser Quell-Aktivität(en) multiplikativ in die Bestimmung der Ausführungswahrscheinlichkeit von a in den nachfolgend vorgestellten Algorithmen mit ein.

Algorithmus 4.7 Approximative Bestimmung der Ausführungswahrscheinlichkeit der Kind-Aktivitäten einer FLOW-Aktivität.

function EPINFLOW($a \in \mathcal{A}$)
  $p \leftarrow \mathsf{parent}(a)$
  if $\mathcal{L}_{in}(a) = \emptyset$ **then** ▷ C1
    if $\mathsf{parent}(p) = \perp$ **then**
      return 1 ▷ C2
    else
      return EP(p) ▷ C3
    end if
  else ▷ C4
    $t \leftarrow \mathsf{join}(a) \cdot \prod_{\forall l \in \mathcal{L}_{in}(a)} \text{EP}(\mathsf{source}_{\mathcal{L}}(l)) \cdot \mathsf{param}_{P_{Succ}}(\mathsf{source}_{\mathcal{L}}(l))$
    if $\neg \exists\, (a', a) \in \pi_{1,4}(\mathsf{LR}) : \mathsf{parent}(a') = \mathsf{parent}(a)$ **then**
      return EP(p) $\cdot\, t$
    else
      return t
    end if
  end if
end function

Das in Algorithmus 4.7 dargestellte Verfahren zur Bestimmung der Ausführungswahrscheinlichkeit der Kind-Aktivitäten einer FLOW-Aktivität gliedert sich in zwei Abschnitte, abhängig von der Anzahl der eingehenden Synchronisationskanten: keine (C1), oder eine bzw. mehrere (C4). Generell erfolgt sowohl in diesem als auch in allen weiteren Algorithmen die Bestimmung der Ausführungswahrscheinlichkeiten der Aktivitäten durch *rekursiven Aufstieg*, d. h. ausgehend von der jeweiligen Aktivität "von innen nach außen". Im Fall des Fehlens eingehender Synchronisationskanten gilt für eine Aktivität entweder die Ausführungswahrscheinlichkeit 1 falls der FLOW selbst keine Eltern-

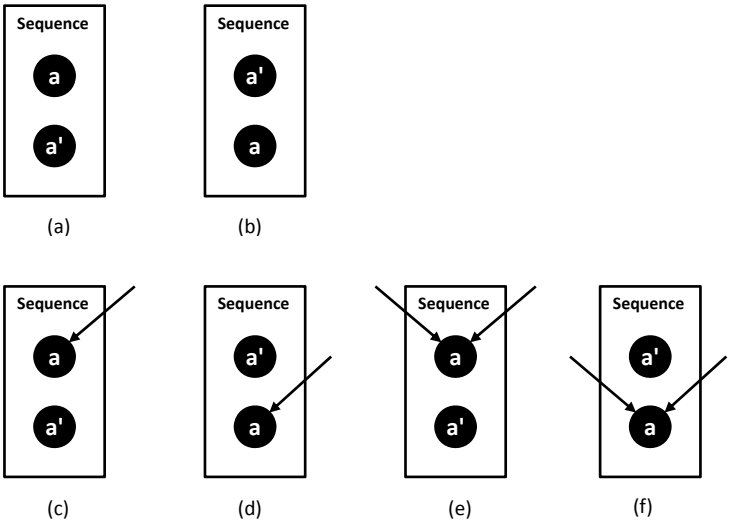


Abbildung 4.19: Betrachtete Szenarien für die Berechnung der Ausführungswahrscheinlichkeit der Kind-Aktivitäten einer SEQUENCE-Aktivität.

Aktivität hat (C2), bzw. die Ausführungswahrscheinlichkeit der FLOW-Aktivität selbst (C3). Im Fall einer oder mehrerer eingehenden Synchronisationskante(n) (C4) berechnet sich die Ausführungswahrscheinlichkeit aus dem Produkt der Wahrscheinlichkeit der Ausführung der Vorgängeraktivitäten und der Funktion join(a) entsprechend dem oben erläuterten Verfahren. Dabei ist eine Berücksichtigung der Ausführungswahrscheinlichkeit von p selbst, aufgrund der verfolgten Vorgehensweise des rekursiven Aufstiegs, nur dann erforderlich, wenn eine Aktivität keine Vorgängeraktivitäten im selben FLOW hat.

4.7.2.2 SEQUENCE-Aktivität

Abbildung 4.19 zeigt die betrachteten Szenarien der Bestimmung der Ausführungswahrscheinlichkeit einer Kind-Aktivität einer SEQUENCE-Aktivität. Szenario (a) zeigt Aktivität a als erste Kind-Aktivität einer SEQUENCE-Aktivität.

In Szenario (b) befindet sich die betrachtete Aktivität nicht am Anfang der SEQUENCE-Aktivität, sondern hat eine Vorgängeraktivität a' innerhalb der SEQUENCE. Szenarien (c)-(f) zeigen Varianten der vorangehenden Szenarien, in denen die Kind-Aktivitäten der SEQUENCE-Aktivität zusätzliche eingehende Synchronisationskanten aufweisen, welche in der Berechnung der Ausführungswahrscheinlichkeit berücksichtigt werden.

Algorithmus 4.8 Approximative Bestimmung der Ausführungswahrscheinlichkeit der Kind-Aktivitäten einer SEQUENCE-Aktivität.

function EPINSEQUENCE($a \in \mathcal{A}$)
  $p \leftarrow \mathsf{parent}(a)$
  if $\mathcal{L}_{in}(a) = \emptyset$ **then** ▷ C1
    if $\neg\exists\,(a', a) \in <^{a}_{seq} \wedge \mathsf{parent}(p) = \bot$ **then**
      return 1 ▷ C2
    else if $\neg\exists\,(a', a) \in <^{a}_{seq} \wedge \mathsf{parent}(p) \neq \bot$ **then**
      return EP(p) ▷ C3
    else
      return EP(a') $\cdot\ \mathsf{param}_{P_{Succ}}(a')$ ▷ C4
    end if
  else ▷ C5
    $t \leftarrow \mathsf{join}(a) \cdot \prod_{\forall l \in \mathcal{L}_{in}(a)} \text{EP}(\mathsf{source}_{\mathcal{L}}(l)) \cdot \mathsf{param}_{P_{Succ}}(\mathsf{source}_{\mathcal{L}}(l))$
    if $\neg\exists\,(a', a) \in <^{a}_{seq}$ **then**
      return EP(p) $\cdot\ t$ ▷ C6
    else
      return EP(a') $\cdot\ \mathsf{param}_{P_{Succ}}(a') \cdot t$ ▷ C7
    end if
  end if
end function

Algorithmus 4.8 zeigt die Bestimmung der Ausführungswahrscheinlichkeiten der Kind-Aktivitäten einer SEQUENCE-Aktivität. In seinem Aufbau folgt das Verfahren im Wesentlichen Algorithmus 4.7 und gliedert sich in zwei Teile bei C1 bzw. C5, abhängig von der Anzahl eingehender Synchronisationskanten. Entsprechend der Bestimmung der Ausführungswahrscheinlichkeiten der Kind-Aktivitäten einer FLOW-Aktivität gilt: ist die betrachtete Aktivität a die

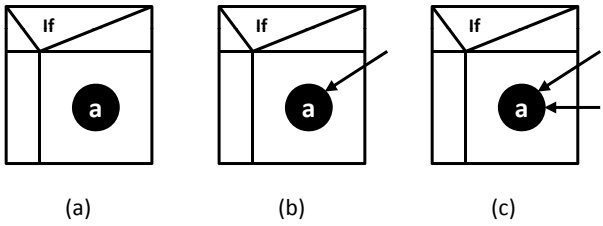


Abbildung 4.20: Betrachtete Szenarien für die Berechnung der Ausführungswahrscheinlichkeit der Kind-Aktivität einer IF-Aktivität.

erste Aktivität der SEQUENCE und hat diese keine weiteren Eltern-Aktivitäten und keine eingehenden Synchronisationskanten, so ist a die erste Aktivität des Prozesses und erhält damit die Ausführungswahrscheinlichkeit 1 (C2). Ist die SEQUENCE hingegen von weiteren strukturierten Aktivitäten umgeben, so ist die Ausführungswahrscheinlichkeit von a die Ausführungswahrscheinlichkeit der SEQUENCE selbst (C3). Hat die betrachtete Aktivität Vorgängeraktivitäten in der SEQUENCE, so hängt die Ausführungswahrscheinlichkeit von a von der Wahrscheinlichkeit einer erfolgreichen Ausführung der Vorgängeraktivität ab (C4), bzw. rekursiv angewendet deren Vorgängeraktivitäten. Durch den rekursiven Ansatz des Algorithmus ist eine gesonderte Betrachtung des Falls mehrerer vorangehender Aktivitäten nicht notwendig.

Im Fall einer oder mehrerer eingehender Synchronisationskanten (C5) gilt, dass zusätzlich zur Ausführungswahrscheinlichkeit der Eltern-Aktivität (C6) bzw. der erfolgreichen Ausführung einer eventuell vorhandenen Vorgängeraktivität (C7) bzw. auch die Wahrscheinlichkeit der erfolgreichen Ausführung der Quellaktivitäten der Synchronisationskanten sowie die Wahrscheinlichkeit zur positiven Evaluierung der JOINCONDITION $\mathsf{join}(a)$ in die Wahrscheinlichkeitsbestimmung multiplikativ eingehen.

4.7.2.3 IF-Aktivität

Abbildung 4.20 zeigt die Szenarien, die für die Berechnung der Ausführungswahrscheinlichkeiten der Kind-Aktivität einer IF-Aktivität betrachtet werden[1]. Die BPEL-Spezifikation erlaubt, dass auch die Kind-Aktivität einer IF-Aktivität das Ziel einer oder mehrerer Synchronisationskanten sein kann (sofern die IF-Aktivität selbst unterhalb einer FLOW-Aktivität liegt); diese Eigenschaft wird in den Szenarien (b) und (c) berücksichtigt. Das XML-Schema von WS-BPEL 2.0 erlaubt als Kind eines Zweigs einer IF-Aktivität lediglich eine Aktivität. Sollen in einem Zweig mehrere Aktivitäten ausgeführt werden, so werden diese wiederum als Kind-Aktivitäten einer strukturierten Aktivität modelliert, welche das direkte Kind der IF-Aktivität ist.

Algorithmus 4.9 Hilfsfunktion Bestimmung des Zweigs der Aktivität a in einer IF-Aktivität i.

function BRANCH($i \in \mathcal{A}_{If}, a \in \mathcal{A}$)
  $e \leftarrow \pi_1(a', b') \in <^i_{if}$ mit $\neg\exists\,(a'', b'') \in <^i_{if} : a'' < a'$
  $c \leftarrow 1$
  while $e \neq a$ **do**
    $e \leftarrow \pi_2(e, b) \in <^i_{if}$
    $c \leftarrow c + 1$
  end while
  return c
end function

Die Funktion BRANCH (Algorithmus 4.9) ist eine Hilfsfunktion, die zu einer gegebenen IF-Aktivität $i \in \mathcal{A}_{If}$ sowie einer Kind-Aktivität $a \in \mathsf{children}(i)$ den Index des Zweigs von i bestimmt, in welchem sich Aktivität a befindet. Die Funktion wird verwendet, um die Darstellung des wahlfreien Zugriffs auf die Aktivität in einem bestimmten Zweig einer IF-Aktivität in Algorithmus 4.10 zu vereinfachen.

Algorithmus 4.10 zeigt das Verfahren zur Bestimmung der Ausführungswahrscheinlichkeit der Kind-Aktivität einer IF-Aktivität. In Szenario (a) gilt für die

[1] Um die Darstellung zu vereinfachen, wurde in der Abbildung auf eine Unterscheidung der einzelnen Zweige der IF-Aktivität verzichtet.

Algorithmus 4.10 Approximative Bestimmung der Ausführungswahrscheinlichkeit der Kind-Aktivität einer IF-Aktivität.

function EP-IN-IF($a \in \mathcal{A}$)
  $p \leftarrow \mathsf{parent}(a)$
  $i_a \leftarrow \text{BRANCH}(p, a)$
  if $\mathcal{L}_{in}(a) = \emptyset$ **then**
    if $\mathsf{parent}(p) = \perp$ **then**
      return $1 \cdot \pi_{i_a}(\mathsf{param}_{EPL}(p))$ ▷ C1
    else
      return $\text{EP}(p) \cdot \pi_{i_a}(\mathsf{param}_{EPL}(p))$ ▷ C2
    end if
  else ▷ C3
    $t \leftarrow \mathsf{join}(a) \cdot \prod_{\forall l \in \mathcal{L}_{in}(a)} \text{EP}(\mathsf{source}_{\mathcal{L}}(l)) \cdot \mathsf{param}_{P_{Succ}}(\mathsf{source}_{\mathcal{L}}(l))$
    return $\text{EP}(p) \cdot \pi_{i_a}(\mathsf{param}_{EPL}(p)) \cdot t$
  end if
end function

Ausführungswahrscheinlichkeit von a, dass diese entweder der Ausführungswahrscheinlichkeit des Zweigs von a entspricht, falls die IF-Aktivität selbst keine Eltern-Aktivität hat (C1). Falls die IF-Aktivität einer Eltern-Aktivität hat, so geht deren Ausführungswahrscheinlichkeit multiplikativ in die Berechnung der Ausführungswahrscheinlichkeit von a ein (C2). Die Berücksichtigung zusätzlicher eingehender Synchronisationskanten wird durch den Fall (C3) abgedeckt und ist vergleichbar der Berechnung der Ausführungswahrscheinlichkeit der Kind-Aktivitäten einer FLOW-Aktivität, berücksichtigt allerdings zusätzlich die Ausführungswahrscheinlichkeit des jeweiligen IF-Zweigs, dem die betrachtete Aktivität untergeordnet ist, entsprechend Fall (C2).

4.7.2.4 Schleifen und Handler

Von den beiden in Abbildung 4.21 aufgeführten Szenarien zur Berechnung der Ausführungswahrscheinlichkeit der Kind-Aktivität der WHILE-Aktivität, wird lediglich Szenario (a) im Algorithmus berücksichtigt. Eine Prozessmodellierung entsprechend Szenario (b) ist durch die Einschränkung *SA00070* der BPEL-

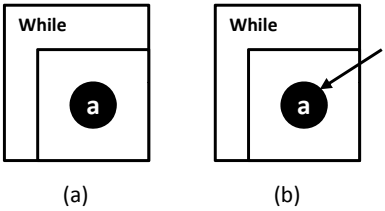


Abbildung 4.21: Betrachtete Szenarien für die Berechnung der Ausführungswahrscheinlichkeit der Kind-Aktivität einer WHILE-Aktivität.

Spezifikation [Org07] explizit untersagt. Für die Aktivitäten REPEATUNTIL und FOREACH gelten dieselben Szenarien, weshalb diese in der Folge nicht gesondert aufgeführt werden. Die Berechnung der Ausführungswahrscheinlichkeit erfolgt daher für sämtliche Iterationsaktivitäten durch die generische Funktion EPINLOOP (Algorithmus 4.11). Eine gesonderte Berücksichtigung der *completion condition* einer FOREACH-Aktivität erfolgt durch den Algorithmus nicht.

Algorithmus 4.11 Approximative Bestimmung der Ausführungswahrscheinlichkeit der Kind-Aktivität von Schleifen.

```
function EPINLOOP(a ∈ 𝒜)
    p ← parent(a)
    if parent(p) = ⊥ then
        return 1
    else
        return EP(p)
    end if
end function
```

Die erwartete Anzahl der Ausführungen der Schleife findet in der Berechnung der Ausführungswahrscheinlichkeit keine Verwendung, da diese im zweiten Teil von Algorithmus 4.5 (d. h. dem Teil nach C4) berücksichtigt wird.

Abschließend zeigt Algorithmus 4.12 das Verfahren zur Bestimmung der Ausführungswahrscheinlichkeit eines Fault-/Termination-/Event-/Compensation-Handlers. Die Berechnung stützt sich auf die Vorgabe der Ausführungswahr-

Algorithmus 4.12 Approximative Bestimmung der Ausführungswahrscheinlichkeit der Kind-Aktivität eines Handlers.

```
function EPINHANDLER(a ∈ 𝒜)
    p ← parent(a)
    if param_{P_Exec}(p) ≠ ⊥ then
        s ← π_1(s', p, a) ∈ HR
        return EP(s) · param_{P_Exec}(p)
    else
        return 1
    end if
end function
```

scheinlichkeit der Handler-Aktivität durch den Partitionierer (durch den Parameter $\mathcal{PAR}_{P_{Exec}}$) sowie die Ausführungswahrscheinlichkeit des SCOPE, dem der jeweilige Handler zugeordnet ist, da die Handler erst bei Ausführung des SCOPE aktiviert werden.

4.7.3 Phase 3.3: Partitionierung des Partitionierungsgraphs

In Phase 3.3 des entwickelten Partitionierungsverfahrens erfolgt die Partitionierung des EWFN-Partitionierungsgraphs $\text{EWFN}_{\text{Part}}$, welcher durch Anwendung der in Abschnitt 4.7.2 vorgestellten Algorithmen bestimmt wurde.

Da der EWFN-Partitionierungsgraph ein bipartiter gerichteter und gewichteter Multigraph und damit insbesondere ein Graph ist, kann seine Partitionierung auf das *Graphpartitionierungsproblem* zurückgeführt werden. Definition 4.33 gibt eine formale Definition des Graphpartitionierungsproblems, mit den für den vorliegenden Fall geltenden Spezialisierungen.

Definition 4.33 (Partitionierung des EWFN-Partitionierungsgraph)**.** *Das Problem der Partitionierung des EWFN-Partitionierungsgraphs beschreibt die Zerlegung von EWFN_{Part} in eine Menge von Partitionen $\mathcal{P}$.*

Ergänzend zur Funktion partition *auf Ebene des BPEL-Prozesses ordnet die Funktion* $\text{partition}_{PT} : P \cup T \rightarrow \mathcal{P}$ *einer Stelle oder einer Transition des EWFN-Partitionierungsgraphs eine Partition zu. Die Ausgabe des Partitionierungsver-*

fahrens, der DDD (vgl. Abschnitt 4.4 und 5.8.1), definiert eine Partition für jede Transition und jede Stelle von EWFN$_{Part}$. Es gilt also: $\forall\, o \in P \cup T\ \exists\, p \in \mathcal{P}$: $\mathsf{partition}_{PT}(o) = p \wedge p \neq \bot$, *wobei* $\bot$ *die nicht zugewiesene Partition bezeichnet.*

Im Gegensatz zum klassischen Graphpartitionierungsproblem [Els97], in welchem typischerweise eine Partitionierung angestrebt wird, in der die Knoten des Graphen auf möglichst gleich große Partitionen zu verteilen sind, ist diese Anforderung für das vorliegende spezialisierte Graphpartitionierungsproblem nicht gegeben. Das bedeutet, es muss nicht unbedingt gelten:

$$\forall p_1, p_2 \in \mathcal{P} \text{ mit } p_1 \neq p_2 : |\{o | o \in P \cup T : \mathsf{partition}_{PT}(o) = p_1\}| \approx |\{o' | o' \in P \cup T : \mathsf{partition}_{PT}(o') = p_2\}|$$

Das Graphpartitionierungsproblem gehört zur Klasse der *Kombinatorischen Optimierungsprobleme*; Definition 4.34 gibt eine formale Definition dieser Klasse von Problemen auf Basis von [PS98].

Definition 4.34. *Ein* Optimierungsproblem $(\Omega, \omega, \preceq)$ *ist definiert durch einen Lösungsraum* Ω*, eine Kostenfunktion* ω *zur Bewertung eines Elements des Lösungsraums sowie einen Operator* $\preceq$*, der eine Ordnung auf den Elementen des Lösungsraums entsprechend der Kostenfunktion definiert. Gilt beispielsweise die Kostenfunktion* $\omega : \Omega \rightarrow \mathbb{R}$*, so wird die Ordnung der Elemente des Lösungsraums beispielsweise durch den Operator* $\leq$ *bestimmt.*

Das (globale) Optimum $O_{global} \in \Omega$ *des Optimierungsproblems bezeichnet diejenigen Elemente des Lösungsraums, die hinsichtlich der Kostenfunktion "besser" als alle anderen Elemente des Lösungsraums* $\Omega \backslash O_{global}$ *sind. Das globale Optimum ist definiert als* $O_{global} = \{o \in \Omega | \forall\ o' \in \Omega : o' \preceq o\}$.

Für den vorliegenden Fall der Partitionierung der Stellen und Transitionen eines EWFN$_{\text{Part}}$ gilt, dass der Lösungsraum des Partitionierungsproblems prinzipiell alle möglichen Zuordnungen von Stellen und Transitionen zu Partitionen umfasst.

Soll beispielsweise eine Partitionierung des Graphen mit den Stellen a, c und

der Transition b auf zwei Partitionen p_1, p_2 erfolgen, so gilt:

$$\begin{aligned}\Omega = \{&\{(a,p_1),(b,p_1),(c,p_1)\},\{(a,p_1),(b,p_1),(c,p_2)\},\\ &\{(a,p_1),(b,p_2),(c,p_1)\},\{(a,p_2),(b,p_1),(c,p_1)\},\\ &\{(a,p_1),(b,p_2),(c,p_2)\},\{(a,p_2),(b,p_1),(c,p_2)\},\\ &\{(a,p_2),(b,p_2),(c,p_1)\},\{(a,p_2),(b,p_2),(c,p_2)\}\}\end{aligned}$$

In diesem Fall gilt also $|\Omega| = 8$. Im Allgemeinen gilt für die Partitionierung eines $\text{EWFN}_{\text{BPEL}}$: $|\Omega| = |\mathcal{P}|^{|P|+|T|}$.

Die Kosten der notwendigen partitionsübergreifenden Interaktionen sind, wie in Abschnitt 4.7 bereits erläutert, das, in dieser Dissertation gewählte, Optimierungskriterium der *Light Node*-Partitionierung. Entsprechend diesem ist ω im vorliegenden Fall definiert als die Summe der Gewichte der partitionsübergreifenden Kanten in einer bestimmten Prozesskonfiguration $o \in \Omega$ eines Prozesses. Formal berechnet sich diese zu:

$$\begin{aligned}\omega(o) = &\sum_{s \in \text{cArcs}} \text{weight}_S(s) \text{ mit}\\ &\text{cArcs} = \{s' | \text{partition}_{PT}(\pi_1(\tilde{F}(s'))) \neq \text{partition}_{PT}(\pi_2(\tilde{F}(s'))\}\end{aligned}$$

Hinsichtlich des Vergleichs zweier Elemente des Lösungsraums durch die Kostenfunktion gilt, dass im vorliegenden Fall ein Element des Lösungsraums des Optimierungsproblems $o \in \Omega$ immer genau dann "besser" als ein anderes Element des Lösungsraums $o' \neq o$ ist, wenn die Summe der Gewichte der partitionsübergreifenden Kanten in Lösung o kleiner ist als der entsprechende Wert für Lösung o'. Formal gilt also: $o' \preceq o \Leftrightarrow \omega(o) \leq \omega(o')$.

Generell unterscheiden Optimierungsprobleme zwei Arten von Optima: das *globale Optimum* (vgl. Definition 4.34) und *lokale Optima*. Im Gegensatz zu einem globalen Optimum beschreibt ein lokales Optimum einen Punkt $o \in \Omega$, der hinsichtlich der Kostenbewertung besser als seine direkten Nachbarn ist, allerdings in weiterer Nachbarschaft weitere, bessere Optima existieren. Die Nachbarschaft eines Elements des Lösungsraums ist dabei definiert durch die Nachbarschaftsfunktion $\varphi : \Omega \to 2^{\Omega}$, welche einem Element des Lösungsraums die

Menge sämtlicher benachbarter Elemente des Lösungsraums zuordnet. Betrachtet man im oben genannten Beispiel das Element $l_1 = \{(a, p_1), (b, p_2), (c, p_1)\}$, so wäre beispielsweise

$$\begin{aligned}\varphi(l_1) = \{&\{(a, p_2), (b, p_2), (c, p_1)\},\\ &\{(a, p_1), (b, p_1), (c, p_1)\},\\ &\{(a, p_1), (b, p_2), (c, p_2)\}\}\end{aligned}$$

Für ein lokales Optimum gilt also: $\Omega_{lokal} = \{o \in \Omega | \forall\, o' \in \varphi(o) : o' \preceq o \wedge \exists\, o'' \in \varphi^\star(o) : o \preceq o''\}$, wobei $\varphi^\star$ die mehrfache Anwendung der Nachbarschaftsfunktion bezeichnet.

Ein möglicher Ansatz zur Lösung kombinatorischer Optimierungsprobleme ist die Bewertung jedes Elements des Lösungsraums hinsichtlich seiner Kosten und die Bestimmung des Elements mit den geringsten Kosten [RN02]. Obwohl garantiert ist, dass die vollständige Bewertung des Lösungsraums zu einer optimalen Lösung des Partitionierungsproblems führt, ist dieses Vorgehen aufgrund des exponentiellen Wachstums der Größe des Lösungsraums mit der Anzahl der zu partitionierenden Objekte – die Komplexität dieser Vorgehensweise hat die Komplexität $O(m^n)$, wobei m die Anzahl der Partitionen und n die Anzahl der partitionierenden Objekte bezeichnet – bei nicht trivialen Prozessen nicht praktikabel anwendbar.

In vielen Anwendungsgebieten der Graphpartitionierung, das vorliegende eingeschlossen, ist eine *optimale* Lösung – d. h. das globale Optimum – allerdings nicht unbedingt notwendig; eine "gute" Lösung (d. h. eine Lösung, die den Kriterien eines Nutzers genügt) ist bereits ausreichend (vgl. Eigenschaft 4.29). Zur Lösung dieser Probleme existieren eine Reihe von heuristischen Verfahren, die zugunsten besseren Laufzeitverhaltens auf eine erschöpfende Traversierung des Lösungsraums und der Bewertung jedes einzelnen Elements verzichten, sondern diesen nur unvollständig entsprechend bestimmten Strategien traversieren.

Beispiele für anwendbare etablierte Algorithmen zur Lösung des vorliegenden Partitionierungsproblems sind die sogenannten *Greedy-Algorithmen*

[Far88], z. B. der *Kernighan-Lin-Algorithmus* [KL70], oder generische Verfahren zur Lösung kombinatorischer Optimierungsprobleme wie *Hill Climbing* [RN02] oder *Simulated Annealing* [KGV83, JS93, JAMS89].

Generell kann jedes der hier genannten Verfahren für die Partitionierung von $\text{EWFN}_{\text{Part}}$ verwendet werden. Aufgrund guten Laufzeitverhaltens und der Eigenschaft des Verfahrens, unter bestimmten Umständen auch eine "schlechtere" als die gegenwärtig gefundene Lösung zu akzeptieren (was dazu führt, dass das Verfahren ein gefundenes lokales Optimum auch wieder zugunsten eines eventuell vorhandenen globalen Optimums verlassen kann), wird in der Folge die Anwendung von *Simulated Annealing* für die Partitionierung von $\text{EWFN}_{\text{Part}}$ erläutert.

Kerngedanke von *Simulated Annealing* (wörtlich übersetzt "Simuliertes Aushärten") ist die Nachahmung des physikalischen Vorgangs des Aushärtens von Metallen durch langsames Abkühlen, in welchem sich Atome aus energetisch ungünstigen in energetisch günstigere Positionen bewegen [KGV83]. Die Bereitschaft des Verfahrens, eine Lösung des Lösungsraums zu akzeptieren, wird dabei von der sogenannten *Temperatur* beeinflusst. Je höher der Temperaturwert, desto eher ist das Verfahren bereit, auch ein Element des Lösungsraums als Lösung zu akzeptieren, das hinsichtlich der Kostenfunktion schlechter ist als die aktuell gefundene Lösung. Durch schrittweise Absenkung der Temperatur wird die Bereitschaft des Verfahrens reduziert, auch schlechtere Lösungen zu akzeptieren, und das Verfahren konvergiert zu einem Punkt im Lösungsraum (welcher allerdings nicht notwendigerweise das globale Optimum sein muss).

Die Abkühlung bei *Simulated Annealing* kann durch unterschiedliche Strategien realisiert werden. Beispiele für etablierte Abkühlungsstrategien sind *Lineares Cooling*, *Logarithmisches Cooling* und *Exponentielles Cooling*; weitere Abkühlungsstrategien sowie deren Evaluierung anhand verschiedener Anwendungsszenarien sind Gegenstand von unter anderem [NA98, AKD99, JAMS89].

Lineares Cooling ($T_k = \alpha \cdot \frac{T_0}{k}$) zeichnet sich durch einen relativ schnellen Abkühlungsvorgang aus. Dies führt zu einer relativ hohen Wahrscheinlichkeit, dass ein lokales Optimum nicht mehr verlassen wird. Demgegenüber erfolgt bei logarithmischem Cooling ($T_k = \alpha \cdot \frac{T_0}{1+\ln k}$) der Abkühlungsvorgang extrem langsam. In unendlicher Zeit führt logarithmisches Cooling zum Finden des glo-

balen Optimums. Exponentielles Cooling ($T_k = \alpha \cdot \frac{T_0}{e^k}$) beschreibt den Mittelweg zwischen den beiden vorgenannten Standardverfahren und weist eine mittlere Laufzeit und gutes Verhalten hinsichtlich der Wahrscheinlichkeit, eine Lösung nahe des globalen Optimum zu erlangen, auf. Es ist daher die bevorzugte Cooling-Strategie für die Partitionierung des EWFN-Partitionierungsgraphs.

Da die Cooling-Strategie beliebig austauschbar ist, wird sie in Algorithmus 4.15 durch die Funktion $\mathsf{reduceTemperature} : \mathbb{R} \times \mathbb{N} \rightarrow \mathbb{R}$ dargestellt, die eine Starttemperatur und einen Iterationszähler auf den Temperaturwert nach einem Abkühlungsschritt abbildet.

Zur Vereinfachung der Darstellung Algorithmus zur *Light-Node*-Partitionierung werden die folgenden Hilfsfunktionen verwendet.

Definition 4.35 (Quelle und Ziel einer Kante in $\text{EWFN}_{\text{Part}}$)**.** *Die Funktion* $\mathsf{source}_S : S \rightarrow P \cup T$ *gibt den Quellknoten einer Kante in* $EWFN_{Part}$ *an, d. h. es gilt* $\mathsf{source}_S(s) = \pi_1(\tilde{F}(s))$*. Analog hierzu beschreibt die Funktion* $\mathsf{target}_S : S \rightarrow P \cup T$ *den Ziel-Knoten einer Kante, d. h. es gilt* $\mathsf{target}_S(s) = \pi_2(\tilde{F}(s))$*.*

Definition 4.36 (Partitionsübergreifende Kanten eines Knotens)**.** *Die Funktion* $\mathsf{crossingArcs}_{PT} : P \cup T \rightarrow 2^S$ *beschreibt die Menge der partitionsübergreifenden (sowohl ein- als auch ausgehenden) Kanten eines Knoten, d. h. es gilt:*

$$\begin{aligned}\forall o \in P \cup T : \mathsf{crossingArcs}_{PT}(o) = \{s \in S | \exists o' \in P \cup T \text{ mit} \\ (\mathsf{partition}_{PT}(\mathsf{source}_S(s)) = \mathsf{partition}_{PT}(o) \wedge \\ \mathsf{partition}_{PT}(\mathsf{target}_S(s)) \neq \mathsf{partition}_{PT}(o)) \vee \\ (\mathsf{partition}_{PT}(\mathsf{source}_S(s)) \neq \mathsf{partition}_{PT}(o) \wedge \\ \mathsf{partition}_{PT}(\mathsf{target}_S(s)) = \mathsf{partition}_{PT}(o))\}\end{aligned}$$

Zusätzlich kommen die in Algorithmus 4.13 vorgestellten Hilfsfunktionen INITIALIZE, INITIALSOLUTION und DETERMINECOST zum Einsatz. Für Aktivitäten, für die in vorangehenden Phasen des Partitionierungsverfahrens bereits eine Partition bestimmt wurde, überträgt die Funktion INITIALIZE diese Partitionierungsinformation auf all jene Knoten von $\text{EWFN}_{\text{Part}}$, die zur EWFN-Repräsentation der jeweiligen BPEL-Aktivität gehören. Die Partition dieser Knoten darf wäh-

Algorithmus 4.13 Partitionierung von *Light Nodes*: Hilfsfunktionen (1)

procedure INITIALIZE
  for all $o \in O_{part}$ **do**
    if $\exists$partition(o) **then**
      for all $o' \in P \cup T$: mapsToBPELElement$(o') = o$ **do**
        partition$_{PT}(o') \leftarrow$ partition(o)
        fixed$_{PT}(o') \leftarrow$ `true` ▷ C1
      end for
    end if
  end for
end procedure

procedure INITIALSOLUTION
  for all $o \in P \cup T$: partition$_{PT}(o) = \perp$ **do**
    partition$_{PT}(o) \leftarrow$ FINDNEIGHBOR(o) ▷ C2
  end for
end procedure

function FINDNEIGHBOR$(o \in P \cup T)$
  $e \leftarrow$ **random element from** $\{s \in S | \text{source}_S(s) = o \vee \text{target}_S(s) = o\}$
  if source$_S(e) = o$ **then**
    if partition$_{PT}$(target$_S(e)$) $\neq \perp$ **then**
      partition$_{PT}(o) \leftarrow$ partition$_{PT}$(target$_S(e)$)
    else partition$_{PT}(o) \leftarrow$ FINDNEIGHBOR(target$_S(e)$)
    end if
  else if target$_S(e) = o$ **then**
    if partition$_{PT}$(source$_S(e)$) $\neq \perp$ **then**
      partition$_{PT}(o) \leftarrow$ partition$_{PT}$(source$_S(e)$)
    else partition$_{PT}(o) \leftarrow$ FINDNEIGHBOR(source$_S(e)$)
    end if
  end if
  return partition$_{PT}(o)$
end function

function DETERMINECOST
  cArcs $= \{s' | \text{partition}_{PT}(\pi_1(\tilde{F}(s'))) \neq \text{partition}_{PT}(\pi_2(\tilde{F}(s')))\}$
  return $\sum_{s \in \text{cArcs}} \text{weight}_S(s)$ ▷ C3
end function

rend Phase 3.3 des Partitionierungsverfahrens nicht mehr verändert werden. Repräsentiert wird diese Information durch die Funktion $\text{fixed}_{PT} : P \cup T \to \mathbb{B}$ (C1). Ausgangssituation für die Optimierung ist ein Element des Lösungsraums Ω des Optimierungsproblems, also eine vollständige Partitionierung des EWFN-Graphen. Die Funktion INITIALSOLUTION bestimmt eine derartige Lösung, indem jeder Knoten der Partition eines benachbarten Knoten zugeordnet wird. INITIALSOLUTION stützt sich dazu auf die Hilfsfunktion FINDNEIGHBOR (C2). Die Funktion DETERMINECOST erlaubt die Bewertung einer gefundenen Lösung des Optimierungsproblems und realisiert damit die Kostenfunktion ω. Die Funktion stützt sich hierbei auf die im Rahmen von Phase 3.2 bestimmten Kantengewichte. Der Kostenwert eines Elements des Lösungsraums erfolgt dabei durch Aufsummierung der Gewichte der partitionsübergreifenden Kanten (C3).

Algorithmus 4.14 Partitionierung von *Light Nodes*: Hilfsfunktionen (2)

```
function findCandidate
    node ← random element from (P ∪ T) :
            fixed_PT(node) = false∧
            crossingArcs_PT(node) ≠ ∅
    arc ← random element from crossingArcs_PT(node)
    return (node, arc)                                          ▷ C1
end function

function repartitionCandidate(node ∈ P ∪ T, arc ∈ S)
    old ← partition_PT(node)                                    ▷ C2
    if node = source_S(arc) then                                ▷ C3
        partition_PT(node) ← partition_PT(target_S(arc))
    else                                                        ▷ C4
        partition_PT(node) ← partition_PT(source_S(arc))
    end if
    return old
end function
```

Simulated Annealing ist ein iteratives Verfahren, d. h. in jedem Schritt des Verfahrens dient die gegenwärtig gefundene Lösung des Problems als Ausgangspunkt. Mögliche Kandidaten für die nächste Iteration sind die Elemente

der Lösungsraums, die sich in der sogenannten *Nachbarschaft* des aktuellen Elements befinden.

Die Funktionen FINDCANDIDATE und REPARTITIONCANDIDATE (Algorithmus 4.14) erlauben die zufällige Wahl eines Elements der Nachbarschaft der aktuell betrachteten Lösung sowie die Ausführung eines Repartitionierungsschritts. Die Funktion FINDCANDIDATE bestimmt einen zufälligen Knoten node des EWFN für den (i) in den vorherigen Schritten des Partitionierungsverfahrens keine Partition bestimmt wurde und der (ii) ein- oder ausgehende partitionsübergreifende Kanten aufweist. Nach Bestimmung eines derartigen Knotens wird eine der partitionsübergreifenden Kanten arc des Knotens ausgewählt, entlang welcher der Knoten in eine andere Partition "verschoben" werden kann (C1). Die Funktion REPARTITIONCANDIDATE nimmt die durch FINDCANDIDATE bestimmte Kombination aus Knoten und Kante als Eingabe. Da bei der Anwendung von Simulated-Annealing eine schlechtere Lösung des Optimierungsproblems nur bei entsprechender Temperatur akzeptiert wird, muss die Möglichkeit vorgesehen werden, eine durchgeführte Repartitionierung eines Knotens rückgängig machen zu können. Zu diesem Zweck wird die Partition des zu verschiebenden Knotens vor der Repartitionierung abgelegt (C2). Ist der zu verschiebende Knoten node die Quelle der Kante arc, entlang welcher die Repartitionierung erfolgen soll (gilt also node $= \mathsf{source}_S(\text{arc})$), so wird der Knoten in die Partition des Ziel-Knotens der Kante verschoben (C3). Im umgekehrten Fall wird der Knoten in die Partition des Quell-Knotens der Kante verschoben (C4).

Algorithmus 4.15 zeigt die eigentliche Partitionierung des EWFN-Abhängigkeitsgraphs durch Anwendung von Simulated-Annealing. Nach der Initialisierung und der Bestimmung der Ausgangslösung erfolgt die wiederholte Anwendung der nachfolgend erläuterten Schritte, bis die sogenannte *Exit-Condition* zu einem positiven Wert evaluiert (C4). Die Exit-Condition gibt an, ob ein zufriedenstellendes Partitionierungsergebnis erreicht wurde und der Partitionierungsvorgang beendet werden kann. Das als Teil von Algorithmus 4.15 nicht weiter ausformulierte Abbruchkriterium ist in diesem Fall eine Konvergenz des Optimierungsvorgangs und ein damit verbundener, über mehrere Iterationen hinweg, relativ konstanter Wert von DETERMINECOST.

Die in der Schleife (C1) durchgeführten Schritte sind im Einzelnen: (i) die

Algorithmus 4.15 Partitionierung von *Light Nodes*: Partitionierung des Partitionierungsgraphs

Require: T_0 : Starttemperatur; $i = 1$: Iterationszähler

```
INITIALIZE()
INITIALSOLUTION()

T ← reduceTemperature(T_0, i)
while T > 0 do                                                    ▷ C1
    cost_current ← DETERMINECOST()
    (node, arc) ← FINDCANDIDATE()
    old ← REPARTITION(node, arc)
    cost_new ← DETERMINECOST()
    cost_delta ← cost_current − cost_new
    if cost_delta < 0 then                                        ▷ C2
        if random[0, 1] < e^(−cost_delta / T) then                ▷ C3
            partition_PT(node) ← old
        end if
    end if
    i ← i + 1
    T ← reduceTemperature(T_0, i)
    if EXITCONDITIONSATISFIED() then                              ▷ C4
        break
    end if
end while
```

Bestimmung des Kostenwerts der aktuell betrachteten Lösung des Optimierungsproblems $\text{cost}_{\text{current}}$, (ii) die Bestimmung eines Kandidaten $(\text{node}, \text{arc})$ für einen Repartitionierungsschritt, (iii) die vorbehaltliche Ausführung des Repartitionierungsschritts, (iv) die Bestimmung des Kostenwerts cost_{new} der neuen Lösung und (v) die Bestimmung der Kostendifferenz $\text{cost}_{\text{delta}}$ von neuer und vorheriger Lösung. Weist die neu bestimmte Lösung einen Kostenwert auf, der schlechter als der Kostenwert der zuvor betrachteten Lösung ist (C2), und gilt weiterhin, dass – bedingt durch den aktuellen Temperaturwert und einen Zufallsfaktor [KGV83] – die schlechtere Lösung nicht akzeptiert wird, so wird

der vorbehaltlich ausgeführte Partitionierungsschritt rückgängig gemacht (C3). Nach Senkung der Temperatur des Systems durch die Funktion REDUCETEMPERATURE wird eine neue Iteration des Schleifenrumpfs (C1) gestartet.

4.8 Zusammenfassung

In diesem Kapitel wurde ein automatisches Verfahren für die Partitionierung von BPEL-Prozessen vorgestellt. Basis für das entwickelte Verfahren bildete eine Analyse des Einsatzszenarios, welches dieser Arbeit zugrunde liegt, und den aus diesem resultierenden Eigenschaften und Anforderungen (Abschnitt 4.1). Hieran anknüpfend folgten Ausführungen zu den sogenannten Partitionierungsobjekten eines Prozesses und die Beschreibung ihres Einflusses auf den Partitionierungsvorgang (Abschnitt 4.2). Für einen manuellen Eingriff des Partitionierers in den Partitionierungsvorgang hat dieser die Möglichkeit, eine Reihe unterschiedlicher Parameter vorzugeben. In Abschnitt 4.3 wurden diese Parameter im Einzelnen vorgestellt und dazu erläutert, wie sowohl der Vorgang der Annotation des Prozessmodells als auch die Beschreibung seiner Dienst- und Ausführungsumgebung unter Verwendung der Web-Service-Standards WSDL, WS-Policy und UDDI erfolgen kann. Die detaillierte Vorstellung des entwickelten mehrphasigen Partitionierungsverfahrens bildete den inhaltlichen Schwerpunkt des Kapitels. Das Verfahren adressiert dabei sowohl die Anforderungen eines Service-orientierten Anwendungsumfelds (z. B. die Verfügbarkeit mehrerer funktional äquivalenter Dienste mit unterschiedlichen nicht-funktionalen Eigenschaften bei unterschiedlichen Dienstanbietern), der Ausführung von Geschäftsprozessen (z. B. manuelle Vorgabe der Partitionierung, beispielsweise aus Gründen der Berücksichtigung bestehender Verträge oder der Datenhoheit) und angestrebter Laufzeiteigenschaften (z. B. die Minimierung der Kosten partitionsübergreifender Kommunikation).

Ein wesentlicher Aspekt des entwickelten Partitionierungsverfahrens ist dabei, eine detaillierte Parametrisierung der Partitionierung zwar zu ermöglichen, diese jedoch nicht zu verlangen. Erreicht wurde dieses Ziel zum einen durch die Möglichkeit zur selektiven Parametrisierung, zum anderen durch die Eigenschaft, dass auch eine nicht-optimale Partitionierung lediglich zu

nicht-optimalem Laufzeitverhalten führen kann, die semantische Äquivalenz zur lokalen Ausführung jedoch in jedem Fall bestehen bleibt.

KAPITEL 5

EINE INFRASTRUKTUR ZUR VERTEILTEN PROZESSAUSFÜHRUNG

In den folgenden Abschnitten wird ein verteiltes WfMS – die sogenannte *Process-Space*-Infrastruktur (kurz *PS-Infrastruktur*) – vorgestellt, welches die dezentrale Ausführung von WS-BPEL-2.0-Prozessen auf Grundlage des in Abschnitt 3.2 vorgestellten EWFN-Modells ermöglicht.

Ausgangspunkt der Erläuterung des Systems und seiner Architektur bildet eine Anforderungsanalyse (Abschnitt 5.1) in welcher – ähnlich der Vorgehensweise zur Erläuterung des Partitionierungsverfahrens in Kapitel 4 – eine Reihe sowohl funktionaler als auch nicht-funktionaler Anforderungen an das zu entwickelnde WfMS formuliert werden. Ausgehend von diesen Anforderungen wird die Architektur der PS-Infrastruktur sowie deren Komponenten und Schnittstellen vorgestellt (Abschnitt 5.2). In den hieran anschließenden Abschnitten werden unterschiedliche Aspekte der Prozessausführung unter Verwendung der PS-Infrastruktur, wie beispielsweise die Gewährleistung transaktionalen Verhal-

tens oder der Aufbau der zwischen den einzelnen PS-Klienten ausgetauschten Daten, erläutert. Die Realisierung nicht unmittelbar mit der Prozessausführung befasster Aspekte, wie Protokollierung der Instanzausführung und das Deployment von Prozessmodellen auf der PS-Infrastruktur, wird in den Abschnitten 5.7 und 5.8 vorgestellt. Abschnitt 5.9 schließt das Kapitel mit einer qualitativen Betrachtung der Eigenschaften der entwickelten PS-Infrastruktur hinsichtlich des erreichbaren Dezentralisierungsgrads der Prozessausführung und der Autonomie der Ausführungsteilnehmer eines Prozesses ab.

5.1 Anforderungsanalyse

Anwendungsfeld dieser Arbeit und damit maßgeblich für die Anforderungen an die entwickelte Infrastruktur ist die Ausführung von *Produktionsprozessen*, welche sich durch einen hohen Geschäftswert und eine relativ hohe Anzahl an Wiederholungen auszeichnen (vgl. Eigenschaft 4.1). Aufgrund der Nähe zum Kerngeschäft eines Unternehmens und die erwartet hohe Zahl von Prozessinstanzen gelten für Produktionsprozesse im Allgemeinen hohe Anforderungen hinsichtlich Dienstgütegarantien, wie Skalierbarkeit, Verfügbarkeit und Robustheit. Die geforderten funktionalen sowie nicht-funktionalen Eigenschaften werden in der Folge im Einzelnen erläutert.

5.1.1 Funktionale Anforderungen

Anforderung 5.1 (WS-BPEL 2.0)**.** *Um eine Ausführung von Prozessen, die unter Verwendung des BPEL-Metamodells beschrieben sind, zu erlauben, muss die entwickelte Infrastruktur die operationale Semantik der BPEL-Aktivitäten implementieren. Diese beinhaltet dabei all jene Verarbeitungsschritte, die ein WfMS während der Ausführung einer bestimmten Aktivität durchführt; die durch die einzelnen BPEL-Aktivitäten zu realisierenden Funktionen sind in [Mar10] in Form der EWFN-Pattern beschrieben. Einige Beispiele hierfür sind: (i) die Möglichkeit zur Interaktion mit WfMS-externen Diensten und Klienten, (ii) Zugriff auf Instanzdaten, (iii) Verarbeitung während der Instanzausführung auftretender Fehler und (iv) Instanz-Terminierung.*

Anforderung 5.2 (Prozess-Deployment)**.** *Vor der Instantiierung eines Prozesses muss dieser während des Deployment-Schritts (vgl. Abschnitt 3.5.2.3) aus der Build-Time-Umgebung in die Run-Time-Umgebung des WfMS überführt werden [wfm95]. Grundlage für das Deployment in der PS-Infrastruktur ist der in Abschnitt 5.8.1 beschriebene und durch das Partitionierungsverfahren (vgl. Abschnitt 4.4) erzeugte DDD.*

Anforderung 5.3 (Erfassung von Protokolldaten)**.** *Das sogenannte* Process (Instance) Monitoring *und das* Process Auditing *(oder auch* Controlling*) bezeichnen den Vorgang der Überwachung der Ausführung von Prozessinstanzen zu jeweils unterschiedlichen Zeitpunkten im Prozesslebenszyklus [MR00, Mue01a]. Sowohl Monitoring als auch Auditing bedienen sich dabei der Protokolldaten, die während der Ausführung von Prozessinstanzen durch das WfMS erfasst werden und deren Ablauf dokumentieren. Hierbei sind besondere Anforderungen zu berücksichtigen, die aus der Verteilung der PS-Infrastruktur resultieren, z. B. nicht zuverlässig synchronisierte Uhren auf unterschiedlichen Systemen (vgl. Abschnitt 5.7).*

5.1.2 Nicht-funktionale Anforderungen

Anforderung 5.4 (Verfügbarkeit)**.** *Eine wesentliche Anforderung an ein WfMS zur Ausführung von Produktionsprozessen ist* Verfügbarkeit *(engl.* Availability*). Dabei bezeichnet man ein System immer dann als* verfügbar*, wenn es in der Lage ist, Anfragen zu empfangen, diese zu verarbeiten und sie ggf. auch durch eine korrekte Antwort abzuschließen [LR99].*

Anforderung 5.5 (Wartung und Aktualisierung von Systemkomponenten)**.** *Ein weiterer Aspekt der Anforderung nach Verfügbarkeit ist eine Minimierung der Ausfallzeiten des Systems während Wartungsintervallen (engl.* Continuous Operation*). Beispielsweise sollte es möglich sein, einen selektiven Austausch einzelner Komponenten des Gesamtsystems vorzunehmen, ohne dass dies zum Abbruch in Ausführung befindlicher Prozessinstanzen führt. Ebenso gilt – besonders in einem potentiell sowohl geographisch als auch administrativ verteilten System wie der PS-Infrastruktur – die Anforderung nach der Möglichkeit zur Aktualisierung der Komponenten des WfMS bei einem Ausführungsteilnehmer eines Prozesses, ohne*

dass dies eine gleichzeitige entsprechende Aktualisierung der WfMS der anderen Ausführungsteilnehmer erfordert.

Anforderung 5.6 (Skalierbarkeit)**.** Skalierbarkeit *(engl.* Scalability*) bezeichnet die "Fähigkeit, steigende Systemlast kompensieren zu können ohne zugesicherte Dienstgüteeigenschaften zurücknehmen zu müssen, solange die Systemressourcen ebenfalls vergrößert werden" [LR99], und ist weitere Anforderung an eine Laufzeitumgebung für Produktionsprozesse.*

Die Verarbeitung von Anfragen durch das System kann nicht erwartete Fehlerzustände zur Folge haben. Das System muss in der Lage sein, die Fehlerzustände zu erkennen und diesen entsprechend zu begegnen. Dies resultiert in zwei Anforderungen an das WfMS.

Anforderung 5.7 (Transaktionales Verhalten)**.** *Auch im Fall eines auftretenden Fehlers muss sichergestellt sein, dass (i) von einem Aufrufer entgegengenommene Anfragen (deren Empfang und damit implizit deren Verarbeitung dem Aufrufer bestätigt wurde) nicht verloren gehen und (ii) das System sich auch bei nebenläufiger Ausführung oder auftretenden Fehlern stets in einem konsistenten Zustand befindet oder in diesem überführen lässt.*

Anforderung 5.8 (Wiederherstellbarkeit nach einem Systemfehler)**.** *Bei erfolgreicher Ausführung eines internen Verarbeitungsschritts des Systems gilt, dass die Resultate dieses Verarbeitungsschritts in einer Form abgelegt werden müssen, die es auch im Fall eines (fatalen) Systemfehlers erlauben, dass beim Wiederanlaufen des Systems (i) der Fehlerzustand des Systems erkannt und (ii) ein entsprechender Wiederherstellungsvorgang (engl.* Recovery*) durchgeführt werden kann.*

Neben den oben genannten gelten oft zusätzliche nicht-funktionale Anforderungen hinsichtlich Sicherheitsaspekten wie Datenintegrität, Authentifizierung und Autorisierung. Diese werden allerdings im Verlauf der Arbeit nicht weiter verfolgt.

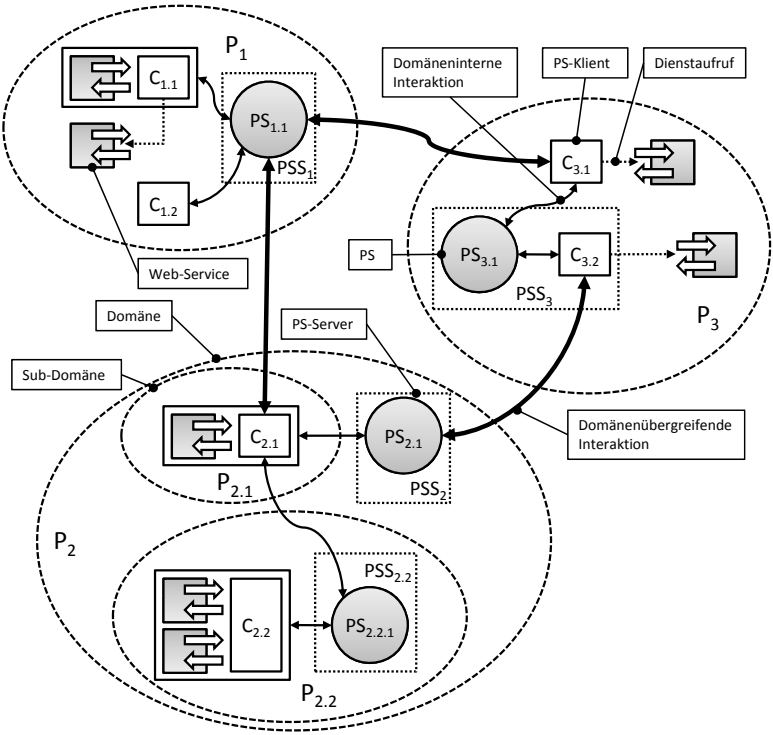


Abbildung 5.1: Architektur des PS-Gesamtsystems.

5.2 Architektur des verteilten WfMS

In diesem Abschnitt wird die Architektur der PS-Infrastruktur präsentiert. Sie besteht aus der Vorstellung einer Architekturübersicht des Gesamtsystems, in welcher erläutert wird, wie die einzelnen Elemente des EWFN-Metamodells durch die PS-Infrastruktur realisiert und zueinander in Bezug gesetzt werden (Abschnitt 5.2.1). Anknüpfend an die Architekturübersicht folgt eine Erläuterung des internen Aufbaus von PS-Servern und PS-Klienten (Abschnitt 5.2.3). Die nachfolgenden Abschnitte des Kapitels konkretisieren die einzelnen Komponenten der PS-Infrastruktur und erläutern deren Funktion im Detail.

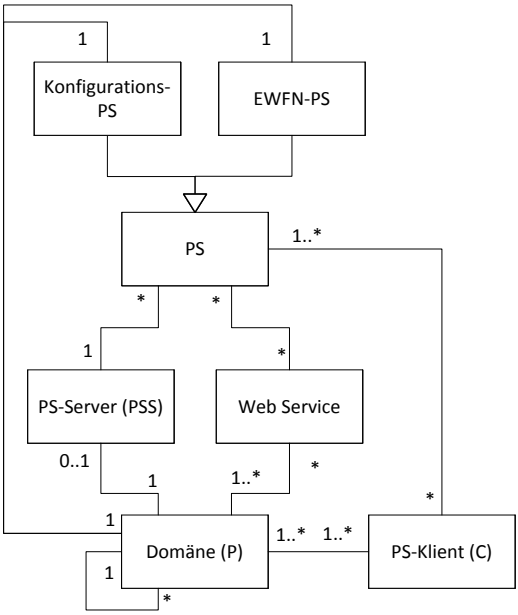


Abbildung 5.2: Beziehungen zwischen den Elementen der PS-Infrastruktur und deren Kardinalitäten.

Abbildung 5.1 zeigt einen schematischen Überblick über die PS-Infrastruktur auf einer hohen Abstraktionsebene. Die Darstellung des Systems umfasst *Process-Space-Server* (*PS-Server*, PSS_1, ..., PSS_3), *Process-Spaces* (*PS*, $PS_{1.1}$, ..., $PS_{3.1}$) und *Process-Space-Klienten* (*PS-Klienten*, C_1, ..., C_3) sowie die von den jeweiligen Ausführungsteilnehmern angebotenen Web-Services.

Abbildung 5.2 veranschaulicht die Beziehungen der einzelnen Elemente der PS-Infrastruktur untereinander. Eine *Domäne* bezeichnet einen logischen Verbund aus PS-Server, PS-Klienten und Web-Services (beispielsweise motiviert durch eine Organisationseinheit in einem Unternehmen). Domänen können mit anderen Domänen assoziiert werden, um einen sogenannten *Domänenverbund* zu bilden. Eine Domäne hat immer genau einen oder keinen *extern sichtbaren* (d. h. für die Interaktion mit PS-Klienten anderer Domänen zur Ausführung

von EWFN-Modellen verwendeten) PS-Server; für andere Anwendungen ist der Betrieb beliebig vieler interner PS-Server möglich. Wie im Fall von P_2 in Abbildung 5.1 dargestellt, kann eine hierarchische Unterteilung einer Domäne in sogenannte *Sub-Domänen* (P_2, $P_{2.1}$ bzw. P_2, $P_{2.2}$) – vorgenommen werden. Weiterhin sind einer Domäne ein oder mehrere PS-Klienten zugeordnet; ebenso gilt, dass ein PS-Klient zu mehr als einer Domäne gehören kann. Dies erlaubt es beispielsweise, denselben PS-Klienten an zwei PS unterschiedlicher aber verbundener Domänen zu betreiben. Verwendung findet dies beispielsweise zur Erfüllung von Anforderung 5.6 nach Skalierbarkeit des Gesamtsystems, wenn die Interaktionen einer Domäne über mehr als einen PS abgewickelt werden sollen. Als Grundlage für die Partitionierung der *Heavy Nodes* (vgl. Abschnitt 4.6) gilt, dass auch jeder für die Ausführung eines Prozesses auf der PS-Infrastruktur verwendete Web-Service mindestens einer Domäne zugeordnet sein muss.

Auf einem PS-Server können beliebig viele PS ($PS_{1.1}, PS_{2.1}, PS_{2.2.1}, PS_{3.1}$) betrieben werden; die Verteilung eines PS über mehrere PS-Server ist nicht vorgesehen. Ist eine Verteilung von Daten über mehrere PS-Server (d. h. mehrere physikalische Rechner) gewünscht, so müssen diese durch geeignete Partitionierung auf einzelne PS verteilt werden, welche wiederum auf unterschiedlichen PS-Servern betrieben werden[1]. Um eine Identifikation der einzelnen PS zu erlauben, ist jeder PS durch einen *Uniform Resource Locator (URL)* [BLMM94] eindeutig benannt. Dies erlaubt, ähnlich [Gel89], den Betrieb mehrerer PS auf einem PS-Server. Durch Verwendung des *Domain Name System (DNS)* [Moc87] kann der *Authority*-Anteil der URL zu der *Internet Protocol (IP)* Adresse [Pos81a] des Rechners aufgelöst werden, auf dem der jeweilige PS betrieben wird.

Es werden zwei Spezialisierungen von PS unterschieden: *Konfigurations-PS*, die zur Konfiguration der PS-Klienten verwendet werden, und *EWFN-PS*, über die die Kommunikation mit PS-Klienten anderer Domänen erfolgt. Jeder Domäne ist genau ein Konfiguration-PS sowie ein EWFN-PS zugeordnet. Dies gilt

[1] Konzeptuell ist allerdings durch die Anwendung von Replikationsmechanismen auch eine Verteilung über mehrere physikalische Rechner möglich. Beispiele für Tuplespace-Umsetzungen, die diese Funktionalität bieten, sind *SwarmLina* [TM04], *CORSO* [Küh94, Küh03] oder *Gigaspaces* (`http://www.gigaspaces.com`).

auch für jene Fälle, in welchen in einer Domäne kein eigener PS-Server betrieben wird (z. B. $P_{2.1}$ und PSS_2 in Abbildung 5.1). Auf Konfigurations-PS erfolgt lediglich domäneninterner Zugriff. Der EWFN-PS einer Domäne ist hingegen die Schnittstelle für domänenübergreifende Kommunikation, dementsprechend gilt, dass die URLs dieser PS von den jeweiligen Ausführungsteilnehmern veröffentlicht werden (vgl. Abschnitt 5.2.2) und die jeweiligen PS-Server einen Zugriff auf diesen von außerhalb der Domäne erlauben müssen.

Das Deployment der PS-Server kann auf unterschiedliche Weise erfolgen, entweder allein auf einem physikalischen Rechner (wie z. B. bei PSS_1), oder in Kombination mit einem oder mehreren PS-Klienten (wie z. B. bei $PSS_3, C_{3.2}$). Vorteil des letzteren Deployments ist eine effiziente lokale Kommunikation, Vorteil des ersteren die Möglichkeit zur flexiblen Anpassung des Systems an geänderte Lastverhältnisse (vgl. Anforderung 5.6). Die von einem PS angebotene Schnittstelle ist *lokationstransparent*, d. h. die Interaktion zwischen einem PS-Klient und einem PS (bzw. dem PS-Server auf dem der jeweilige PS betrieben wird) erfolgt stets unter Verwendung derselben Operationen der PS-Schnittstelle (Abschnitt 5.3). Dies gilt sowohl für (i) domänenübergreifende (z. B. zwischen $C_{2.1}$ und $PS_{1.1}$) als auch (ii) domäneninterne lokale (z. B. zwischen $C_{3.2}$ und $PS_{3.1}$) und (iii) domäneninterne entfernte Kommunikation (z. B. zwischen $C_{2.2}$ und $PS_{2.2.1}$).

Während der Ausführung ihrer Funktionalität können PS-Klienten, die die Funktion der INVOKE-Aktivität realisieren, mit Web-Services interagieren; analoges gilt für die nachrichtenempfangenden Aktivitäten PICK und RECEIVE. Ähnlich PS-Klienten werden auch hinsichtlich Web-Services unterschiedliche Möglichkeiten zum Deployment unterstützt. Durch den Mechanismus der Web-Service-Bindings (vgl. Abschnitt 2.1.4) kann der Aufruf eines Web-Service entweder (i) als lokaler Methodenaufruf[1] (z. B. der Dienstaufruf von $C_{2.1}$), (ii) über ein beliebiges Web-Service-Binding zum Aufruf entfernter Dienste (z. B. HTTP oder JMS, $C_{3.1}$) für die Integration bestehender Web-Service-Implementierungen

[1] Existierende Laufzeitumgebungen für Web-Services wie beispielsweise *Apache Axis 2* (`http://ws.apache.org/axis2`) oder das *Web Service Invocation Framework (WSIF)* (`http://ws.apache.org/wsif`) erlauben zu diesem Zweck die Verwendung effizienter maschinenlokaler Interaktion.

oder (iii) über die PS-Infrastruktur selbst erfolgen (vgl. Kapitel 6).

In Definition 4.24 wurde – im Kontext der Beschreibung des entwickelten Partitionierungsverfahrens – der Begriff der Partition als abstrakter logischer Bezeichner eingeführt. Setzt man diesen in Beziehung zu den einzelnen, im Verlauf dieses Abschnitts vorgestellten, Elementen der PS-Infrastruktur, so gilt die folgende Konkretisierung: Eine Partition kann entweder eine Domäne oder einen PS-Klienten identifizieren. Die Bedeutung einer Partitionszuordnung ist dabei vom Typ des jeweiligen Partitionierungsobjekts abhängig. Zeigt die Partition einer Aktivität (d. h. einer Transition eines EWFN) auf einen PS-Klienten, so wird die Funktionalität der jeweiligen Transition durch den jeweiligen Klienten ausgeführt (vgl. Anforderung 4.7). Zeigt die Partition einer Aktivität hingegen auf eine Domäne, so kann während eines Domänen-internen Deployment-Schritts (vgl. Abschnitt 5.8) ein PS-Klient innerhalb der Domäne für die Ausführung der Funktionalität der jeweiligen Aktivität bestimmt werden. Ein Instanzdatum (bzw. dessen entsprechende Stelle im EWFN-Modell eines Prozesses) wird immer auf den EWFN-PS der jeweiligen Domäne abgebildet.

5.2.1 Zusammenhang zwischen EWFN-Metamodell und PS-Infrastruktur-Komponenten

Der Zusammenhang zwischen den Elementen des in Abschnitt 3.2 einführend vorgestellten EWFN-Metamodells [Mar10] und der zu ihrer Ausführung verwendeten Tuplespace-basierten PS-Infrastruktur wird im Folgenden erläutert; die grundlegenden Konzepte der Tuplespaces wurden in Abschnitt 2.6 vorgestellt.

Die Marken des EWFN-Formalismus (Σ) werden auf der Ebene der PS-Infrastruktur als *Tupel* abgebildet. Analog einer EWFN-Marke ist ein Tupel eine geordnete Liste typisierter Felder [Gel85]. Die im Rahmen der Ausführung einer Instanz eines Prozesses kommunizierten Tupel folgen einer einheitlichen Struktur; ihr Aufbau und die Bedeutung und Verwendung ihrer Felder für die Ausführung von BPEL-Prozessen wird in Abschnitt 5.6.1 erläutert.

Ein Stelle (P) in einem EWFN beschreibt einen Puffer, welcher Marken von Transitionen entgegennehmen und an diese abgeben kann. Formal ist

der Inhalt einer Stelle, sein sogenanntes *Marking* (M), definiert als ein Multi-Set von Tupeln, d. h. eine Stelle kann mehrere identische Tupel enthalten. Eine Typisierung von Stellen (d. h. die Festlegung, dass eine Stelle nur Token eines bestimmten Typs aufnehmen darf) findet im EWFN-Formalismus nicht statt; d. h. eine Stelle kann beliebig viele Tokens unterschiedlichen Typs zum gleichen Zeitpunkt aufnehmen. Die Stellen eines EWFN werden durch *Process-Spaces* (PS) realisiert, die auf PS-Servern betrieben werden. Analog einer Stelle im EWFN-Formalismus kann ein PS beliebig strukturierte Tupel (ggf. auch mehrere identische Tupel) aufnehmen; diese Eigenschaft entspricht der eines Tuplespace. Aufgrund der Beziehung zwischen einer (logischen) Partition und einer Domäne (vgl. Abschnitt 5.2) können während der Ausführung des Partitionierungsverfahrens mehreren Stellen eines EWFN derselben Partition (bzw. deren EWFN-PS) zugeordnet werden. Dies bedeutet, dass die Tupel, die das Marking der jeweiligen Stellen repräsentieren, im selben PS abgelegt werden.

Die durch Transitionen (T) im EWFN-Modell repräsentierten funktionalen Einheiten werden auf Ebene der PS-Infrastruktur durch PS-Klienten realisiert, welche, entkoppelt durch PS, mit anderen Klienten durch den Austausch von Tupeln interagieren. Den EWFN-Pattern entsprechend gilt, dass eine BPEL-Aktivität im Allgemeinen mehr als eine Transition umfasst. Aufgrund der Eigenschaft der rekursiven Komponierbarkeit der EWFN-Pattern kann ein Pattern allerdings selbst wiederum als eine Transition repräsentiert werden. Aus Gründen der Übersichtlichkeit wird in den Abbildungen in diesem und den folgenden Abschnitten stets die Notation verwendet, dass ein PS-Klient sämtliche Transitionen innerhalb des EWFN-Pattern der jeweiligen BPEL-Aktivität zusammengefasst implementiert.

Abbildung 5.3 zeigt zwei unterschiedliche Bereiche, denen sich die Funktionalität eines PS-Klienten zuordnen lässt: Die *Koordinationslogik* (engl. *Coordination Logic*) des Klienten beinhaltet dessen Kommunikation mit seiner Umwelt; seine *Berechnungslogik* (engl. *Computation Logic*) alle Funktionen, die die Verarbeitung der konsumierten Daten in internen Verarbeitungsschritten betreffen [GC92].

Mit seiner Umwelt kommuniziert ein PS-Klient durch drei unterschiedliche

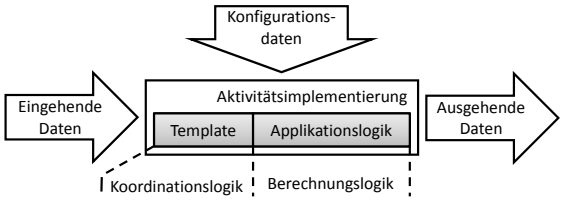


Abbildung 5.3: Konzeptuelle Sicht auf einen PS-Klienten.

Kanäle: er konsumiert *eingehende Daten* (engl. *Inbound Data*) zur Verarbeitung und produziert als Folge *ausgehende Daten* (engl. *Outbound Data*). Welche Daten ein Klient als Teil seiner Koordinationslogik konsumiert, wie er diese zu verarbeiten hat und welche Resultate er nach deren Verarbeitung erzeugen soll, regeln *Konfigurationsdaten* (engl. *Configuration Data*). Die ein- und ausgehend kommunizierten Tupel werden im EWFN-Formalismus durch ein- bzw. ausgehende Kanten repräsentiert. Diese beschreiben den Aufruf einer Operation eines PS-Klienten auf einem PS-Server zur Verarbeitung eines oder mehrerer Tupel. Die Semantik eines Operationsaufrufs ist abhängig vom Typ der EWFN-Kante, welche diesen beschreibt. So gilt beispielsweise, dass ausgehende Kanten einer Transition $(T \times P) \subseteq F$ in einem EWFN Schreiboperationen eines PS-Klienten auf einem PS repräsentieren, eingehende Kanten $(P \times T \times R) \subseteq F$ je nach Typ destruktive bzw. nicht-destruktive Leseoperationen. Das Tupel, das durch die Operation einer ausgehenden Kante geschriebenen wird, wird durch die Funktion $L_{write} : (T \times P) \rightarrow \Sigma$ beschrieben. Für die Ausführung seiner internen Verarbeitungsschritte kann ein PS-Klient mit beliebig vielen PS interagieren. Wie in Abbildung 5.1 dargestellt, können diese sowohl Teil der eigenen (z. B. $C_1, PS_{1.1}$) als auch einer fremden Domäne sein (z. B. $C_{2.1}, PS_{1.1}$).

Die Menge der eingehenden Kanten einer Transition, entlang welcher der entsprechende PS-Klient Kontrollflussmarken konsumiert (bzw. die Menge der entsprechenden PS-Operationen und Templates des PS-Klienten), bestimmen die Bedingung, unter der die durch die jeweilige Transition repräsentierte Funktion ausgeführt werden kann. Im EWFN-Formalismus werde diese durch

die Templates der Menge X repräsentiert. Entsprechend der Schaltsemantik einer EWFN-Transition ist die Ausführung eines PS-Klienten synchronisierend; d. h. die Ausführung der Anwendungslogik eines PS-Klienten wird so lange blockiert, bis die entsprechenden PS-Operationen aller eingehenden Kontrollflusskanten jeweils ein zum Template konformes Tupel zurückliefern können und damit die Startbedingung der entsprechenden Transition im EWFN-Modell erfüllt ist. Nach Ausführung der internen Verarbeitungsschritte des PS-Klienten (d. h. der durch diesen repräsentierten BPEL-Aktivität) signalisiert dieser seine erfolgreiche oder nicht erfolgreiche Ausführung an andere PS-Klienten durch die Produktion von Tupeln entsprechend der Funktion L_{write} des jeweiligen EWFN-Modells.

Sämtliche Daten, die ein PS-Klient für die Ausführung seiner Funktionalität benötigt, konsumiert er zu Beginn eines Ausführungszyklus, d. h. eines EWFN-Schaltvorgangs, aus den jeweiligen PS. Die Resultate der Ausführung veröffentlicht er am Ende des Ausführungszyklus. Beinhalten die jeweiligen Verarbeitungsschritte das Konsumieren bzw. Produzieren mehrere Aufrufe, müssen hierfür also mehrere Kanten in einem EWFN "navigiert" werden, so erfolgen diese entsprechend Anforderung 5.7 innerhalb einer atomaren Transaktion, so dass das System auch im Fall des Scheiterns einer der Operationen in einen konsistenten Zustand überführt wird (vgl. Anforderung 5.7 und Abschnitt 5.5).

Ein Erhalten des internen Zustands eines PS-Klienten über die Grenzen eines Schaltvorgangs hinweg ist nicht notwendig. Zur Berücksichtigung der Anforderungen nach Verfügbarkeit (Anforderung 5.4) und Skalierbarkeit (Anforderung 5.6) erlaubt diese Einschränkung, dem *Competing-Consumers*-Pattern [HW04] folgend, die redundante Vorhaltung identisch konfigurierter PS-Klienten. Somit bleibt zum einen selbst beim Ausfall eines Klienten das Gesamtsystem funktionsfähig, zum anderen kann die Auslastung einzelner PS-Klienten hierdurch gesenkt werden.

Zusätzlich gilt, dass zustandslose PS-Klienten (i) den Vorgang der Wiederherstellung eines konsistenten Zustands nach einem Systemfehler stark vereinfachen – dieser Vorgang wird in den Abschnitten 5.5 und 5.5.4 erläutert – sowie (ii) den Austausch ihrer Implementierung und damit die Berücksichtigung von Anforderung 5.5 vereinfachen. Analog dem Vorgehen zur Steigerung des Ver-

fügbarkeit, kann eine neue Version eines bestimmten PS-Klienten zunächst als *Competing Consumer* mit den entsprechenden PS verbunden werden. Befindet sich ein PS-Klient der Vorgängerversion nicht in Ausführung, so kann seine Implementierung vom PS getrennt werden und der PS-Klient der aktuellen Version übernimmt dessen Funktion. Ein Austausch der Implementierung eines PS-Servers bedingt ein manuelles Replizieren der in ihm enthaltenen Tupel in die neue Installation. Um einen konsistenten Zustand laufender Prozessinstanzen zu garantieren, müssen vor der Replikation des PS dessen externe Schnittstellen für den operativen Betrieb deaktiviert werden.

5.2.2 Veröffentlichung der Beschreibung der PS-Infrastruktur

In Abschnitt 4.4 wurde die maschinenlesbare Beschreibung der PS-Infrastruktur als eine Eingabe des Partitionierungsverfahrens identifiziert. Die Verwaltung und Veröffentlichung dieser Infrastrukturbeschreibung wird nachfolgend – ebenso wie die Erfassung der funktionalen und nicht-funktionalen Diensteigenschaften (vgl. Abschnitt 4.2.2.1) – am Beispiel einer UDDI-Registry erläutert. Abbildung 5.4 verdeutlicht die gewählte Vorgehensweise graphisch.

Die Abbildung der einzelnen Konzepte auf das *UDDI Registry Information Model* [oas02] ist dabei wie folgt:

Domänen

Eine *Domäne* wird durch das UDDI-RIM-Element BusinessEntity beschrieben, welches nach [oas02] einen "Dienstanbieter oder eine Entität beschreibt, die Informationen über sich selbst und über von sich angebotene Dienste bereitstellt".

Sollen mehrere Domänen zu einer Domäne aggregiert werden, so geschieht dies unter Verwendung des PublisherAssertion-Elements, welches die Herstellung einer bidirektionalen Verknüpfung zwischen BusinessEntity-Elementen erlaubt. Die Art der durch eine PublisherAssertion beschriebenen Relation zwischen zwei Domänen *A* und *B* wird durch eine KeyedReference vom Typ `uddi.org:relationships` näher qualifiziert. Für die Beschreibung einer Subdomänenbeziehung wird die

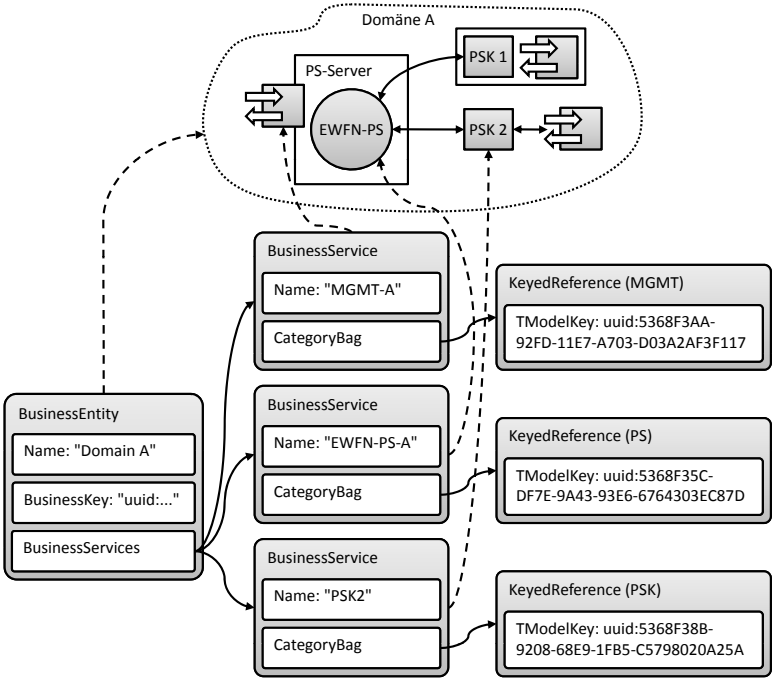


Abbildung 5.4: Beschreibung der PS-Infrastruktur mit den Elementen des *UDDI Registry Information Model*.

`parent-child`-Relation verwendet.

PS

Jede Domäne verfügt über genau einen *EWFN-PS*, der für die Ausführung von Prozessen verwendet wird (vgl. Abschnitt 5.2). Dieser wird durch das UDDI-RIM-Element BUSINESSSERVICE beschrieben, welches nach [oas02] eine "logische Dienstklassifizierung" beschreibt.

Die Zuordnung eines PS-BUSINESSSERVICE zu seiner Domäne erfolgt durch Referenzierung der BUSINESSENTITY der Domäne im Wert des BUSINESSKEY-Attributs. Die Klassifikation eines BUSINESSSERVICE als EWFN-PS

erfolgt durch eine KeyedReference in dessen CategoryBag mit einem speziellen TModelKey-Attribut-Wert[1].

Die URL des PS, z. B. `ps://organisation-a.org/ps1` wird im Wert des AccessPoint-Attributs eines BindingTemplate-Elements des PS-BusinessService abgelegt. Entsprechend der Definition der Elemente einer URI in [BLFM05] umfasst der Identifikator die Elemente *Scheme* (`ps`), *Authority* (`organisation-a.org`) und *Path* (`ps1`) relativ zur jeweiligen Authority.

Für einen PS ist die Angabe einer URL im AccessPoint des BindingTemplate obligatorisch.

PS-Klienten

Da ein *PS-Klient*, ähnlich einem PS, ebenfalls ein innerhalb einer Domäne angebotener Dienst ist, erfolgt dessen Abbildung analog dem PS durch das BusinessService-UDDI-RIM-Element. Im Gegensatz zum PS wird der PS-Klient allerdings durch einen anderen TModelKey-Wert [2] kategorisiert. In der AccessPoint-URL eines BindingTemplate des BusinessService des PS-Klienten wird das *Scheme* `psc` verwendet.

Die Angabe von PS-Klienten ist fakultativ (vgl. Abschnitt 5.2); die Angabe dieser Information hat Einfluss auf die mögliche Verwendung des PS-Klienten während des Partitionierungsvorgangs. Sind für eine Domäne keine PS-Klienten explizit definiert, so wird für die Partitionierung angenommen, dass in jeder Domäne mindestens ein PS-Klient für die Verarbeitung von Aktivitäten zur Verfügung steht. In diesem Fall erfolgt während der Partitionierung lediglich die Bestimmung der Domäne einer Aktivität. Beim Deployment wird durch das Domänen-interne Deployment (vgl. Abschnitt 5.8) bestimmt, welcher PS-Klient die jeweilige Funktion ausführen soll. Sind hingegen für eine Domäne PS-Klienten explizit definiert, so kann jeder von diesen im Rahmen der Partitionierung (beispielsweise in Form eines *Fixed Node* (vgl. Abschnitt 4.5)

[1] `uuid:5368F35C-DF7E-9A43-93E6-6764303EC87D`
[2] `uuid:5368F38B-9208-68E9-1FB5-C5798020A25A`

einzeln verwendet werden. In diesem Fall gilt, dass bereits während der Partitionierung bestimmt werden kann, welcher PS-Klient eine bestimmte Aktivität ausführt. Dessen Identifikator wird dann als Teil des Deployment-Schritts an die Deployment-Komponente des PS-Servers der jeweiligen Domäne übergeben, welche die Konfiguration des jeweiligen PS-Klienten entsprechend durchgeführt.

Gemäß Anforderung 4.7 des Partitionierungsverfahrens gilt, dass PS-Klienten, die die Funktion einer nachrichtenempfangenden Aktivität (d. h. RECEIVE oder PICK) realisieren, zum Partitionierungszeitpunkt statisch in Form eines *Fixed Node* bestimmt werden.

Management-Dienst einer Domäne

Sowohl für das Prozess-Deployment wie auch die Abfrage erfasster Protokolldaten ist eine Interaktion externer Werkzeuge mit den an der Ausführung einer bestimmten Prozesskonfiguration beteiligten PS-Servern notwendig (vgl. Abbildung 5.5).

Da dies in der Regel domänenübergreifender Interaktionen sind, muss die Adresse des Management-Diensts einer Domäne (bzw. des dieser Domäne zugeordneten PS-Servers) für externe Interaktionspartner dokumentiert werden. Analog der PS und der PS-Klienten erfolgt die Dokumentation in Form eines entsprechend kategorisierten[1] BUSINESSSERVICE, wobei die Adresse des Management-Diensts entsprechend der PS-Klienten im ACCESSPOINT eines BINDINGTEMPLATE-Elements erfasst wird.

5.2.3 Logischer interner Aufbau der PS-Server und PS-Klienten

Abbildung 5.5 zeigt eine Übersicht des internen Aufbaus der PS-Klienten und PS-Server [WML09b, MWL09]. Deren Interaktion ist durch eine Koordinationsschicht entkoppelt, welche die klientenseitigen Anteile der Koordinationslogik, z. B. das Konsumieren und Produzieren von Kontrollfluss- und Instanzdateninformationen unter Verwendung der in Abschnitt 5.6.1 beschriebenen Tupelstruktur, implementiert. Die serverseitigen Anteile dieser Koordinationslogik,

[1] uuid:5368F3AA-92FD-11E7-A703-D03A2AF3F117

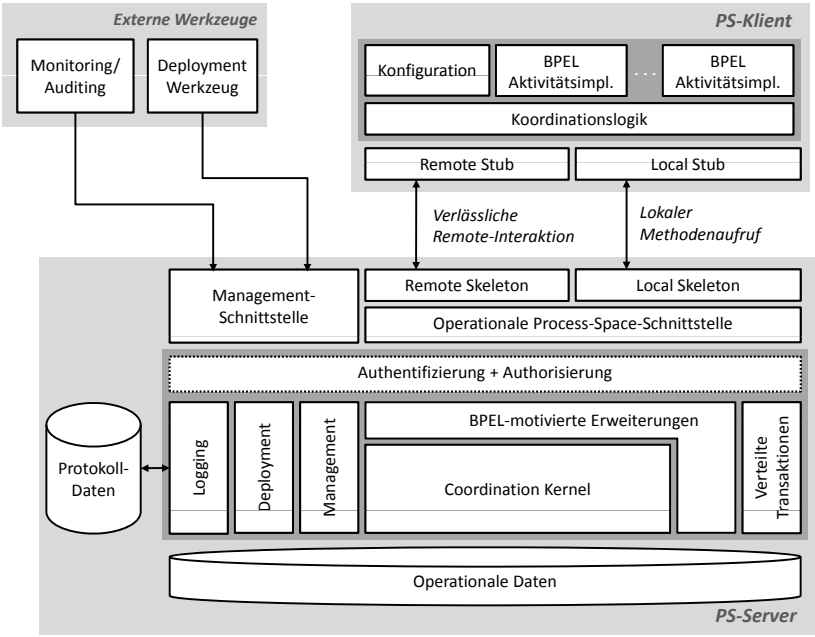


Abbildung 5.5: Übersicht der Gesamtarchitektur des *Process-Space*-Systems.

d. h. die Implementierungen der Operationen der PS-Schnittstelle, sind Teil des sogenannten *Coordination Kernel*[1] und werden in Abschnitt 5.3 erläutert.

Aus dem Ziel der PS-Infrastruktur, die verteilte Prozessausführung mittels dezentraler Navigation zu ermöglichen, resultiert die Forderung nach der Möglichkeit, PS und PS-Klienten auf unterschiedlichen physikalischen Rechnern betreiben zu können. Dies wiederum bedingt die Notwendigkeit von entfernter Kommunikation zwischen PS-Klienten und PS-Servern. Da das System allerdings auch größtmögliche Flexibilität zum Deployment von Klienten und Servern (vgl. hierzu Abbildungen 3.2 und 3.3) bei gleichzeitiger Gewährleistung bestmöglicher Ausführungseffizienz zum Ziel hat, werden die Operationen der PS-Schnittstelle zusätzlich durch maschinenlokale Kommunikationsmecha-

[1] Die Begriffswahl folgt hier dem *Coordination Kernel (CoK)* in CORSO [Küh94].

nismen nutzbar gemacht. Verwendet werden sie im Fall, dass PS-Klient und PS auf demselben physikalischen Rechner betrieben werden (z. B. $C_{3.2}$ und PSS_3 in Abbildung 5.1). Die beschriebene Funktionalität wird durch die Komponenten *Remote Stub* und *Local Stub* auf der Klienten- bzw. *Remote-Skeleton* und *Local-Skeleton* auf der Server-Seite realisiert. Während sich die in den beiden Ansätzen jeweils verwendete Kommunikationstechnologie unterscheidet, sind die Schnittstellen funktional (d. h. hinsichtlich ihrer Operationen und deren Signatur) äquivalent. Aus Sicht der Anwendungslogik auf Seite der PS-Klienten sind sämtliche Interaktionen zwischen PS-Klient und PS-Server sogenannte *Request-Response* Interaktionen, d. h. sie umfassen eine Anfragenachricht des Klienten, welche durch den Server empfangen und verarbeitet wird. Das Ergebnis der Verarbeitung der Anfragenachricht wird mittels einer Antwortnachricht an den Klienten zurück gesandt. Dabei gilt, dass der PS-Klient für den Zeitraum der Ausführung der Operation, d. h. vom Zeitpunkt des Sendens der Anfragenachricht bis zum Empfang des Resultats, blockiert wird. Im Fall entfernter Kommunikation bedingen eine Reihe möglicher Netzwerkeffekte (z. B. Nachrichtenverluste oder Nachrichtenduplikationen) eine gesonderte Adressierung im verwendeten Kommunikationsprotokoll. Im Fall *lokaler Kommunikation* sind diese zusätzlichen Protokollschritte nicht notwendig und können ausgelassen werden. Als Beispiel für die Notwendigkeit dieser zusätzlichen Interaktionsprotokollschritte wird als Teil des Beschreibung der prototypischen Realisierung der Stub- und Skeleton-Komponenten in Abschnitt 7.1.2 die Realisierung der oben genannten blockierenden Interaktion zwischen PS-Klient und PS-Server beim Konsumieren von Tupeln erläutert.

Der sogenannte *Coordination Kernel* bildet den funktionalen Kern der PS-Infrastruktur (vgl. Abbildung 5.5) und kapselt Funktionen, die für eine Ausführung von EWFN-Modellen notwendig sind. Den Kern dieser Schnittstelle bildet das Linda-Modell [Gel85] mit seinen Funktionen *rd*, *in*, *out*. Um diesen funktionalen Kern wurden zusätzliche Funktionen definiert, mit dem Ziel, eine effiziente Ausführung von BPEL-Prozessen in ihrer EWFN-Repräsentation zu erlauben. Diese sind realisiert entweder in Form von zusätzlichen Operationen der PS-Schnittstelle oder BPEL-motivierten Erweiterungen der regulären PS-Operationen. In der logischen Architektur der PS-Server werden diese Funk-

tionen durch die Komponente *BPEL-motivierte Erweiterungen* reflektiert.

Während der Ausführung einer Prozessinstanz auf der PS-Infrastruktur kann es vorkommen, dass ein PS-Klient für die Verarbeitung seiner Eingangsdaten mehr als ein Tupel konsumieren muss; dasselbe gilt für die Produktion der von diesem produzierten bzw. während seiner Ausführung modifizierten Tupel. Mögliche Szenarien, die die Notwendigkeit für derartige Operationen verdeutlichen, sind beispielsweise das Konsumieren (bzw. Produzieren) mehrerer Kontrollflusstupel beim Zusammenführen (bzw. Aufspalten) mehrerer paralleler Ausführungspfade beim Beenden (bzw. Beginn) einer FLOW-Aktivität oder das zusätzliche Konsumieren von Instanzdatentupeln als Teil der Ausführung einer Aktivität. Da beispielsweise auch ein Absturz eines Klienten während eines Ausführungszyklus (gemäß Anforderung 5.7) zu einem konsistenten Zustand des Gesamtsystems (und damit insbesondere zu einem konsistenten Zustand der Gesamtheit der in sämtlichen für die Ausführung der Prozesses verwendeten EWFN-PS enthaltenen Tupel) führen muss, werden derartige Operationen in einer verteilten Transaktion zwischen einem PS-Klient und potentiell mehreren PS-Servern ausgeführt, welche durch die *Verteilte Transaktionen*-Komponente in Abbildung 5.5 realisiert und vom *Coordination Kernel* verwendet werden. Das transaktionale Modell des Systems wird in Abschnitt 5.5 vorgestellt; die Beschreibung der Umsetzung ist Gegenstand von Abschnitt 7.1.3.

Anforderung 5.8 nach Wiederherstellbarkeit des letzten Systemzustands im Fall eines Wiederanlaufens des Systems nach einem fatalen Fehler, wird auf Basis eines transaktionalen und persistenten Speichers realisiert. In der Architektur wird dieser durch die *Operationale Daten*-Komponente reflektiert; deren Umsetzung wird in Abschnitt 5.4 auf Konzeptebene und in Abschnitt 7.1.1 auf Realisierungsebene beschrieben. Zur Realisierung ihrer Funktionalität nutzen sowohl die *Coordination Kernel*- als auch der *Transaktionen*-Komponente die Funktionen der *Operationale Daten*-Komponente.

Um, entsprechend Anforderung 5.3, eine Nachvollziehbarkeit von Prozessinstanzen zu gewährleisten, werden sämtliche ausgeführten PS-Operationen protokolliert. Dabei wird sichergestellt, dass die erfassten Protokolldaten eine vollständige Rekonstruktion des Ablaufs der einzelnen Prozessinstanzen erlau-

ben; man spricht hierbei vom sogenannten *Prozess-Monitoring* bzw. *Prozess-Auditing* [Mue01b]. Aufgrund der Zustandslosigkeit der PS-Klienten erfolgt die Erfassung und Vorhaltung der Protokolldaten lediglich auf Seite der PS-Server in Form einer *Protokoll-Datenbank*. Die Funktionen zur Erfassung der Protokolldaten werden durch die *Logging*-Komponente bereitgestellt. Diese wird vom *Coordination Kernel* über die Ausführung von PS-Operationen benachrichtigt. Neben dieser Protokollierung der ausgeführten PS-Operationen erlaubt die PS-Infrastruktur die Überwachung technischer Betriebsparameter (wie beispielsweise die aktuelle Systemauslastung) des WfMS; man spricht hierbei vom sogenannten *technischen Monitoring* [Mue01b]. Der Vorgang der Protokolldatenerfassung auf den PS-Servern für das Prozess-Auditing und die Rekonstruktion der Ausführung eines Prozesses anhand der erfassten Protokolldaten wird in Abschnitt 5.7 erläutert. Aufgrund der Umsetzungsnähe des technischen Monitoring wird dessen Realisierung in Kapitel 7 als Teil der Beschreibung der prototypischen Implementierung des Systems beschrieben.

Die Deployment-Komponente der PS-Server erlaubt, zur Erfüllung von Anforderung 5.2, die Konfiguration der PS-Klienten unter Verwendung ihrer Konfigurationsschnittstelle und dem, als Resultat der Prozesspartitionierung erzeugten, DDD. Der PS-Server kann zu diesem Zweck einen Teil des DDD eines Prozesses – ein sogenanntes DDD-Fragment (vgl. hierzu die Beschreibung des Deployment-Vorgangs sowie des Aufbaus des DDD in Abschnitt 5.8) – als Eingabe entgegennehmen, diesen ggf. durch weitere Informationen anreichern und die PS-Klienten seiner Domäne(n) entsprechend der Vorgaben des DDD unter Verwendung der Funktionen des *Coordination Kernel* konfigurieren.

Die *Management*-Komponente realisiert zum einen Verwaltungsfunktionen der PS-Server, wie beispielsweise Funktionen zur Erzeugung und Entfernung von PS, zum anderen stellt sie die für das oben genannte *technische Monitoring* notwendige Funktionalität bereit.

Für Gewährleistung von Autorisierung und Authentifizierung eingehender Nachrichten ist die Komponente *Authentifizierung + Autorisierung* vorgesehen. Wie in Abschnitt 5.1.2 erläutert, ist eine Berücksichtigung dieser Aspekte nicht Gegenstand der Dissertation, weswegen auf eine detaillierte Beschreibung dieser Komponente verzichtet wird.

5.3 PS-Schnittstelle und Coordination-Kernel

Durch die *Process-Space-Schnittstelle (PS-Schnittstelle)* bietet ein PS-Server PS-Klienten eine Menge von Funktionen an, unter deren Verwendung die PS-Klienten durch Austausch von Tupeln miteinander interagieren. Die Schnittstelle gliedert sich dabei in zwei Teile, die sogenannte *Operative PS-Schnittstelle* (in [Mar10] auch als *Coordination Interface* bezeichnet) und die *Management Schnittstelle*.

5.3.1 Operative PS-Schnittstelle

Abbildung 5.6 verdeutlicht graphisch die Funktionen der operativen PS-Schnittstelle sowie die Entitäten, auf welchen diese operieren.

Jeder *PS* kann konzeptuell beliebig viele *Tupel* enthalten. Eine Zuordnung eines Tupels zu mehr als einem PS ist nicht möglich. Ein Tupel besteht aus potentiell mehreren *Feldern*, von denen jedes wiederum genau einen Typ hat. *Templates* haben denselben Aufbau wie Tupel. Eine *produzierende Operation* erzeugt ein oder mehrere Tupel und publiziert diese in genau einem PS. Eine *konsumierende Operation* wendet ein oder mehrere Templates auf einen PS an und gibt eine beliebige Anzahl von Tupeln an den aufrufenden PS-Klienten zurück. Die hier genannten Kardinalitäten gelten jeweils innerhalb eines einzelnen Operationsaufrufs. Der Fall, dass mehrere Operationsaufrufe mittels einer Transaktion zu einem logischen (atomar ausgeführten) Operationsverbund zusammengefasst werden können (vgl. Abschnitt 5.3.2 und 5.5), wird hierbei nicht berücksichtigt.

Die Semantik der Funktionen der operativen PS-Schnittstelle wird nachfolgend erläutert; die Vorstellung der algorithmischen Realisierung der einzelnen Funktionen der operativen PS-Schnittstelle ist Teil der Beschreibung des *Coordination Kernel* in [Mar10].

Schreiben, *write* Die *write*-Operation[1] erlaubt die Veröffentlichung von Tupeln durch PS-Klienten. Im Gegensatz zur *out*-Operation der klassischen

[1] Die Namensgebung der PS-Operationen folgt soweit möglich der *JavaSpaces* Schnittstelle [sun03].

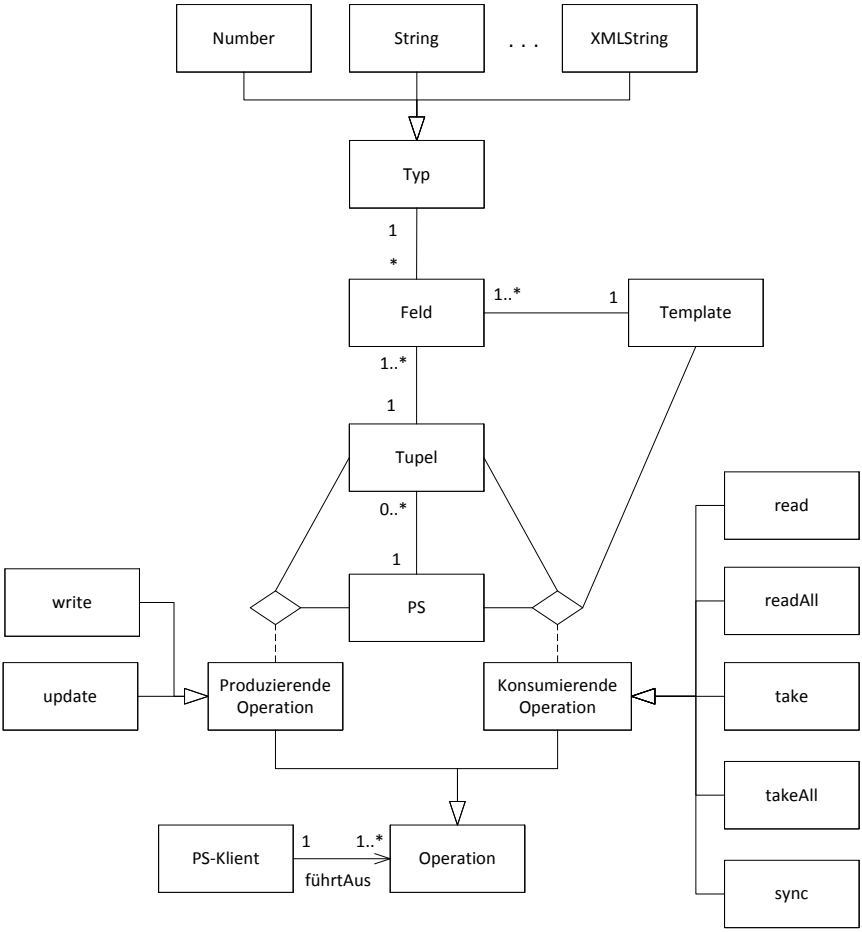


Abbildung 5.6: Funktionen der operativen PS-Schnittstelle.

Linda-Schnittstelle erlaubt *write* das Schreiben mehrerer Tupel in einem Funktionsaufruf zur Minimierung des zwischen PS-Klient und PS-Server notwendigen Kommunikationsaufwands. Vergleichbare Bulk-Operation zum Schreiben bieten beispielsweise die *JavaSpaces05*-Schnittstelle [sun03]

oder *XVSM* [KMKS09]. Analog der klassischen Linda-Semantik gilt, dass ein mittels *write* geschriebenes Tupel keine bereits in einem PS vorhandenen Tupel überschreibt; *write* ist also nicht idempotent. Werden mehrere identische Tupel (d. h. Tupel mit jeweils denselben Feld-Typen und -Werten) geschrieben, so existieren nach dem Schreibvorgang mehrere identische Tupel im PS. Dies entspricht der Multiset-Semantik einer Stelle im EWFN-Metamodell. Ein geschriebenes Tupel verbleibt so lange im jeweiligen PS, bis es von einem PS-Klienten destruktiv konsumiert wird. Ein automatisches Entfernen von Tupeln (wie beispielsweise mittels sogenannter *Leases* in JavaSpaces [sun03, FHA99]) findet in der PS-Infrastruktur keine Verwendung und wird dementsprechend durch den *Coordination Kernel* nicht unterstützt. Das Schreiben eines Tupels hat eine Neuevaluierung der Templates lesender und blockierter PS-Klienten zur Folge, um zu prüfen, ob die Anforderungen der von diesen spezifizierten Templates durch das (bzw. die) geschriebene(n) Tupel erfüllt werden können.

Lesen (nicht-destruktiv), *read* Die *read*-Operation erlaubt das nicht-destruktive Konsumieren von Tupeln, beschrieben durch Templates. Für das Template-Matching bei *read*-Operationen gilt die Linda-Semantik bezüglich Typ- bzw. Wertgleichheit und dem sogenannten *Wildcard*-Operator. Ebenfalls analog zur Linda-Semantik ist *read* aus Sicht des aufrufenden PS-Klienten blockierend, d. h. eine Ausführung der Operation blockiert die Ausführung der Anwendungslogik des PS-Klienten, bis ein dem *read*-Template entsprechendes Tupel konsumiert werden kann. Ein nichtblockierendes Verhalten von *read* kann durch Angabe eines Timeout-Werts erreicht werden, nach dessen Erreichen die Ausführung der Operation mit leerem Rückgabewert abbricht. Der Abbruch der Operationsausführung ist dabei unabhängig davon, ob bis zu diesem Zeitpunkt das Template des PS-Klienten erfüllt werden kann. Eine ähnliche Semantik von *read* findet sich beispielsweise in MARS [CLZ00] oder Jada [CR96].

Die Möglichkeit zur Angabe eines Timeout-Werts besteht neben *read* auch für die anderen *konsumierenden Operationen*.

Mehrfaches Lesen (nicht-destruktiv), *readAll* Die *readAll*-Operation erlaubt das nicht-destruktive Konsumieren aller Tupel in einem PS, die (hinsichtlich des Template-Matching-Algorithmus) die Anforderungen des vom PS-Klienten spezifizierten Templates erfüllen. *readAll* findet in Verbindung mit sogenannten *Wildcard*-Feldern in Templates Verwendung, welche Platzhalter für beliebige Typen und Werte sind. Bezüglich des sonstigen Verhaltens ist *readAll* identisch zu *read*. Ähnliche Funktionalität bieten beispielsweise Gigaspaces[1] in Form der Operation *readMultiple* oder XVSM [KMS08] in Form des `CNT_ALL`-Selektors.

Lesen (destruktiv), *take* Die *take*-Operation verhält sich hinsichtlich der Bestimmung eines, zum Template des Klienten konformen, Tupels identisch zu *read*. Im Gegensatz zu *read* wird dieses nach erfolgreichem Auffinden und der Übergabe an den jeweiligen PS-Klienten bei *take* allerdings dem PS entnommen; ein wiederholtes Konsumieren desselben Tupels ist mittels *take* somit nicht möglich.

Mehrfaches Lesen (destruktiv), *takeAll* Die *takeAll*-Operation ist die destruktive Entsprechung zu *readAll* für *take* und verhält sich zu dieser analog. Während die Nachbildung der Semantik von *takeAll* durch wiederholte Aufrufe von *take* möglich ist, gilt dies für die *readAll*-Operation aufgrund des sogenannten *Multiple-Read*-Problems [RW96] im Allgemeinen nicht.

Synchronisierendes Lesen, *sync* Der spezielle Fall der synchronisierenden Zusammenführung mehrerer Ausführungspfade in einem Prozess – der sogenannte *Synchronizing-Join* – wird durch die *sync*-Operation [MWL08c, MWL08d, MWL09] realisiert. Dabei erlaubt *sync* die Angabe mehrerer Templates und blockiert die weitere Ausführung des aufrufenden PS-Klienten so lange, bis für jedes, durch den PS-Klienten spezifizierte, Template ein entsprechendes Tupel im PS verfügbar ist. Weiterhin erlaubt *sync* die Verwendung sogenannter *Join-Variablen*, die eine Korrelation mehrerer Tupel durch identische Werte in potentiell unterschiedlichen Feldern erlauben. Der Wert für den Korrelator muss hier allerdings, im

[1] `http://www.gigaspaces.com`

Gegensatz zur Verwendung der "klassischen" Linda-Operationen, nicht im Vornherein bekannt sein. Um im Fall sich überschneidender Templates unterschiedlicher PS-Klienten mögliche Verklemmungen zu verhindern, implementiert *sync* Strategien zur Konfliktauflösung. Eine detaillierte Beschreibung der Realisierung des Template-Matching bei *sync* ist Teil von [Mar10].

Modifizieren, *update* Soll ein Tupel in einem Tuplespace modifiziert werden, so bedingt dies unter Verwendung der Linda-Operationen das destruktive Konsumieren des Tupels, die Modifizierung des konsumierten Tupels sowie das Schreiben des modifizierten Tupels in den Tuplespace durch einen Klienten. Zur Minimierung des Kommunikationsaufwands zwischen PS-Klient und PS-Server kapselt *update* diese Operationen in einen einzelnen Operationsaufruf. Die Parametrisierung von *update* erfolgt zum einen durch ein Template, welches das zu modifizierende Tupel spezifiziert, und zum anderen durch ein Tupel, mit welchem das zu modifizierende Tupel ersetzt werden soll. In den Operationen der Linda-Schnittstelle beschrieben, entspricht die Semantik von *update* einem *take*, einer Modifikation auf Seite des PS-Klienten und einem anschließenden *write* des modifizierten Tupels. Ähnliche Funktionen bieten beispielsweise *Gigaspaces* und *XVSM*. Aufgrund des Anwendungshintergrunds der Ausführung von BPEL-Prozessen, bietet die *update*-Operation der PS-Infrastruktur erweiterte Funktionen zur Modifikation von Tupeln mit XML-Feldern. Abschnitt 5.6.2.4 beschreibt die *update*-Operation im Detail.

Löschen, *destroy* Um das Entfernen eines oder mehrerer Tupel aus einem PS ohne die Notwendigkeit der Übermittlung deren Werte vom PS-Server an den aufrufenden PS-Klient zu erlauben, realisiert *destroy* die in [KMKS09] beschriebene Semantik eines destruktiven Konsumierens (*take*) mit anschließendem Verwerfen des Ergebnisses von *take* bereits auf Seite des PS-Server.

Weiterhin definiert die operative PS-Schnittstelle die nachfolgend vorgestellten Funktionen, die für die Umsetzung transaktionalen Verhaltens Verwendung

finden.

Transaktionserzeugung, *createTransaction* Zur Gewährleistung transaktionalen Verhaltens (Anforderung 5.7) erlaubt die Funktion *createTransaction* die Erzeugung eines Transaktionskontexts. Dieser Transaktionskontext wird allen Operationen, die innerhalb einer bestimmten Transaktion ausgeführt werden sollen, als Parameter übergeben. Das transaktionale Verhalten ist dabei unabhängig davon, ob die transaktional ausgeführten Operationen auf demselben PS (und damit insbesondere auf demselben PS-Server) oder auf unterschiedlichen PS auf unterschiedlichen PS-Servern verarbeitet werden. Die Semantik des transaktionalen Verhaltens und die Verwendung des Transaktionskontexts wird in Abschnitt 5.5 erläutert.

Erfolgreicher Transaktionsabschluss, *commit* Den aus Sicht eines Klienten erfolgreichen Abschluss der Ausführung einer Abfolge von PS-Operationen macht dieser durch den Aufruf der Operation *commit* kenntlich. Die Anwendung der Operation hat die Ausführung eines verteilten atomaren *Commit*-Protokolls zwischen den, von den ausgeführten PS-Operationen betroffenen, PS zur Folge. Können sämtliche PS die auf ihnen ausgeführten Operationen erfolgreich ausführen, so werden die Änderungen der Operationen für Nicht-Teilnehmer der Transaktion sichtbar. Die *commit*-Operation blockiert dabei so lange, bis die Transaktion entweder durch sämtliche Transaktionsteilnehmer erfolgreich abgeschlossen werden konnte oder aufgrund eines Fehlers eines Teilnehmers diese abgebrochen werden musste. Im Fall des Transaktionsabbruchs erfolgt eine Rücknahme der Effekte der als Teil der Transaktion ausgeführten PS-Operationen.

Expliziter Transaktionsabbruch, *abort* Neben dem impliziten Transaktionsabbruch, bedingt durch beispielsweise einen Fehler eines Transaktionsteilnehmers, hat auch ein PS-Klient selbst (als Teil seiner Anwendungslogik) die Möglichkeit, die Effekte der transaktional ausgeführten PS-Operationen vor dem *commit* explizit zurückzunehmen. Der explizite

klientengetriebene Abbruch einer Transaktion erfolgt durch die *abort*-Operation.

Eine zusätzliche Funktionalität der PS-Schnittstelle, die für die Realisierung unterschiedlicher Aspekte der Ausführung von EWFN-Prozessen zum Einsatz kommt, ist der sogenannte *Operationskontext*. Dies sind zusätzliche Parameter eines Aufrufs einer PS-Operation, welche ein Klient an einen PS-Server übergeben kann und welche die Verarbeitung der aufgerufenen Operation auf den PS-Servern beeinflussen können; *XVSM* bietet vergleichbare Funktionalität [Cra10] in Form des sogenannten *context xtree*. Verwendung findet dies beispielsweise während der Erfassung von Protokolldaten (Abschnitt 5.7) oder während der Protokollierung der Ausführung einer SCOPE-Aktivität zur Realisierung der *Default Compensation Order* (Abschnitt 5.6.3.2). Die Erläuterung der Funktionsweise des Operationskontexts ist Gegenstand des jeweiligen Abschnitts.

5.3.2 Management-Schnittstelle

Die operative PS-Schnittstelle wird ergänzt um die Funktionen der Management-Schnittstelle; diese adressieren die nachfolgend aufgeführten Teilbereiche.

Verwaltung von PS Zur logischen Gruppierung von Tupeln erlaubt die PS-Infrastruktur die Verwendung mehrerer benannter PS pro PS-Server. Die Struktur der PS ist flach, eine hierarchische Schachtelung von PS findet in der Navigation von EWFN-Modellen keine Verwendung und wird demnach nicht unterstützt. Bei ihrer Erzeugung durch die Funktion *createSpace* erhalten PS einen Namen, dieser wird in Verbindung mit der Adresse des PS-Servers, auf welchem der PS betrieben wird, in Form einer URL beschrieben. Der Zugriff auf den PS durch Aufruf einer PS-Operation bedingt immer die Angabe dieser PS-URL. Auf einem PS ausgeführte Operationen haben keine Auswirkung auf den Inhalt anderer PS auf demselben Server. Aufgrund der durch PS erzeugten logischen Gruppierung von Tupeln tragen PS zur Minimierung des Aufwands der Ausführung der PS-Operationen bei, indem sie beispielsweise die

Menge der auf einem PS-Server bereitgestellten Tupel, die für die Ausführung des Template-Matching-Algorithmus betrachtet werden müssen, eingrenzen.

Durch die Operation *removeSpace* wird ein PS mit sämtlichen in ihm enthaltenen Tupeln gelöscht. Nach Abschluss der Operation ist weder ein Zugriff auf den PS noch auf die in ihm enthaltenen Tupel möglich. Die Tupel werden aus dem physischen Speicher der PS-Server entfernt.

Verwaltung von Blacklist-Einträgen Jeder PS bietet eine sogenannte *Blacklist*, welche es erlaubt, Tupelmengen aus der Verwendung in den Template-Matching-Mechanismen des *Coordination Kernel* auszunehmen. Die Blacklist hat die folgenden Eigenschaften: ist ein Tupel, das als Resultat auf eine konsumierende PS-Operation eines Klienten gefunden wurde, auch zu einem Eintrag der Blacklist konform, so wird es als Resultat der Anfrage des Klienten ignoriert. Dies kann entweder zur Folge haben, dass der Klient ein anderes, zu dem von diesem spezifizierten Template konformes Tupel als Antwort erhält, das keinem Template in der Blacklist entspricht, oder – wenn kein derartiges Tupel vorhanden ist – die Operation des Klienten blockiert wird. Zur Manipulation ihrer Werte bietet eine Blacklist die Operationen *addEntry* zum Anlegen eines neuen Eintrags, *removeEntry* zum Entfernen eines bestimmten Eintrag und *clear* zur Entfernung aller Einträge einer Blacklist. Da eine Blacklist immer an einen bestimmten PS gebunden ist, wird diese beim Entfernen des entsprechenden PS durch die Operation *removeSpace* ebenfalls gelöscht. Verwendung findet die Blacklist zur Realisierung der sogenannten TERMINATION in BPEL (vgl. Abschnitt 5.6.3.1).

Weitere Funktionen Wie in Abschnitt 5.2.3 erläutert, kapselt die Management-Schnittstelle weiterhin Funktionen für (i) die Installation bzw. die Deinstallation von Prozessmodellen auf der PS-Infrastruktur, (ii) die Abfrage von während der Ausführung von Prozessinstanzen erfassten Protokolldaten, sowie (iii) die Überwachung technischer Betriebsparameter. Da die Beschreibung dieser Funktionen jeweils Gegenstand nachfolgender

Abschnitte ist, wird auf deren Erläuterung an dieser Stelle verzichtet.

5.3.3 BPEL-motivierte Erweiterungen der PS-Schnittstelle

Generell gilt, dass die Funktionen der operativen PS-Schnittstelle bzw. des *Coordination Kernel* aus der Motivation heraus gewählt wurden, eine möglichst effiziente Ausführung von EWFN-Modellen zu erlauben. Sowohl die Funktionen des Linda-Modells als auch bestimmte neu eingeführte Operationen, wie beispielsweise *sync*, sind dabei unabhängig von der konkreten Sprache, die als Eingabe für die EWFN-Transformation verwendet wurde.

Einige wenige Funktionen wurden, motiviert durch das Ziel der effizienten Ausführung von BPEL-Prozessen, eingeführt. Dies sind Mechanismen

- zum selektiven Konsumieren einzelner Teile eines Tupels mit umfangreichen XML-Daten (Abschnitt 5.6.2.5)
- zur effizienten Manipulation großer Daten direkt auf Seite des PS (Abschnitt 5.6.2.4)
- zur Realisierung der Terminierung von SCOPE- Aktivitäten und ganzen Prozessen (Abschnitt 5.6.3.1).

5.3.4 Adressierung bekannter Linda-Probleme

Aufgrund der konzeptuellen Nähe der PS-Infrastruktur zu den Konzepten der Tuplespaces werden in der Folge einige klassische Probleme Linda-basierter Kommunikation vorgestellt und es wird diskutiert in welcher Weise diese innerhalb der PS-Infrastruktur adressiert werden.

Die Operationen *rd* und *in* des Linda-Modells [Gel85] blockieren den konsumierenden Klienten so lange, bis ein zum spezifizierten Template konformes Tupel konsumiert werden kann. Wird durch die über den Tuplespace interagierenden Anwendungen nicht sichergestellt, dass ein entsprechendes Tupel im Verlauf der Ausführung einer Anwendung erzeugt wird, so bleibt der jeweilige Klient potentiell unendlich lange blockiert, man spricht daher auch vom *endlosen Blockieren*. Dasselbe gilt für den Fall, dass mehrere Klienten versuchen, mit demselben Template Daten aus demselben Tuplespace zu konsumieren. In

diesem Szenario kann – aufgrund des Nichtdeterminismus der konsumierenden Operationen – der Fall eintreten, dass stets derselbe Klient Daten konsumiert, während andere, ebenfalls selbst zum Konsumieren bereite Klienten, endlos blockiert bleiben. Wie aus der Beschreibung des Problems hervorgeht, handelt es sich beim *endlosen Blockieren* eher um ein Problem der unter Verwendung eines Tuplespace realisierten Anwendung, als um ein Problem der Linda-basierten Kommunikation. Andere Methoden zur Kommunikation – wie beispielsweise *Messaging* [HW04] – zeigen hinsichtlich *Endlosem Blockieren* äquivalentes Verhalten; d. h. erhält beispielsweise eine *Message-Driven Bean* [MKDM03] keine Nachricht, so wird diese ebenfalls nicht ausgeführt. Ebenso gelten für die sogenannten *Competing Consumers* [HW04] im Allgemeinen keine Garantien hinsichtlich der Verteilung der Nachrichten an die unterschiedlichen Konsumenten. In der PS-Infrastruktur gelten hinsichtlich *Endlosen Blockierens* die folgenden Eigenschaften. Durch die BPEL-EWFN-Transformation und die in Abschnitt 5.6.1 vorgestellte Tupelstruktur wird sichergestellt, dass PS-Klienten lediglich Tupel erzeugen, die wiederum als Eingabe für weitere PS-Klienten dienen können. Da PS-Klienten keinen Zustand über einen Ausführungszyklus hinweg vorhalten müssen, ist im Fall des Vorhaltens mehrerer äquivalent konfigurierten PS-Klienten für ein Einhalten der Ausführungssemantik von BPEL nicht relevant, welcher PS-Klient seine Eingangsbedingung erfüllt und einen Verarbeitungsschritt durchführt.

Ein Kerngedanke des Linda-Kommunikationsparadigmas ist die Konsumieren von Daten mittels Template-Matching. Die potentiellen Konsumenten eines Tupels beschreiben Struktur und bekannte Werte des zu konsumierenden Tupels in Form des Template. Erfüllen mehrere in einem Tuplespace befindliche Tupel die Anforderungen eines Konsumenten, so legt die Semantik der ursprünglichen Linda-Funktionen nicht fest, welches Tupel ein Konsument zur Verarbeitung erhält. Im Fall eines nicht-destruktiven Konsumieren eines Tupels durch einen Konsumenten bedeutet dies, dass dieser unter Umständen bei wiederholtem Aufruf der konsumierenden Operation dasselbe Tupel erhält, auch bekannt als das sogenannte *Multiple Rd* Problem [RW96]. Als Folge ist es einem Klienten nicht möglich, globales Wissen über alle im Tuplespace befindliche Tupel zu erlangen, ohne diese dem Tuplespace zunächst durch destruktives Konsumieren

zu entnehmen (was ggf. konkurrierende Konsumenten blockieren würde). Dem Problem mehrfachen Lesens eines Klienten wird in der PS-Infrastruktur durch den Aufbau der kommunizierten Tupel begegnet. Der Aufbau der für die Prozessausführung kommunizierten Tupel ist dergestalt, dass eine eindeutige Identifikation jedes Tupels möglich ist (vgl. Abschnitt 5.6.1). Weiterhin gilt, dass das Konsumieren der Kontrollflusstupel (d. h. die Tupel, deren Konsumieren die Ausführung eines PS-Klienten zur Folge hat) destruktiv erfolgt; ein wiederholtes Konsumieren desselben Tupels ist daher ausgeschlossen.

Als Folge der Linda zugrunde liegenden Kommunikation durch "ungerichtete" Publikation von Daten [MWSL07] existiert zwischen einem Tupel in einem Tuplespace und einem Produzenten oder Konsumenten des Tupels kein direkter Bezug, d. h. ein Tupel kann keinem Klienten des Tuplespace zugeordnet werden. Zur Entfernung nicht mehr benötigter Tupel aus einem Space existieren unterschiedliche Ansätze: In *JavaSpaces* wird diese Funktionalität beispielsweise durch einen *Lease*-Mechanismus realisiert. Beim Schreiben eines Tupels spezifiziert der schreibende Klient einen Zeitraum, nach dessen Ende das Tupel automatisch aus dem Space entfernt wird (sofern der Lease nicht erneuert wird). Im Gegensatz hierzu stützt sich *CORSO* [Küh94] auf einen auf *Reference Counting* basierenden *Garbage Collection*-Mechanismus [KN98]. Da im EWFN-Modell eines Prozesses alle Zeitpunkte, an denen ein Entfernen bestimmter Tupel erfolgen muss, explizit ausmodelliert sind, wird eine entsprechende Funktion in der PS-Infrastruktur nicht benötigt und dementsprechend nicht unterstützt.

In Szenarien, in denen ein Klient *A* fortlaufend Tupel in einen Tuplespace schreibt, während ein anderer Klient *B* diese fortlaufend konsumiert, ist in der ursprünglichen Linda-Semantik nicht sichergestellt, dass *B* die Tupel in derselben Reihenfolge konsumiert, in welcher diese von *A* produziert werden. Dies kann unter bestimmten Umständen zum sogenannten *Verhungern* führen, also zur Existenz von Tupeln, die im Vergleich zu anderen (hinsichtlich der Templates laufender Operationen) identischen Tupeln unverhältnismäßig lange im Tuplespace verbleiben. In der PS-Infrastruktur ist dieses Verhalten beispielsweise für eine gleichpriorisierte Behandlung unterschiedlicher Instanzen eines Prozessmodells relevant. Die Lösung des Problems wird durch FIFO-Semantik

des Template-Matching-Algorithmus erreicht. Für diesen gilt, dass er stets das Tupel als Resultat einer konsumierenden Operation zurückliefert, welches (i) zum Template des Klienten konform ist und (ii) bereits die längste Zeit im Tuplespace verbracht hat. Die Semantik ist vergleichbar dem FIFO-Koordinator in XVSM [KRJ05, KMKS09].

5.4 Persistente Speicherung operationaler Daten

Die Komponente *Persistenter, transaktionaler Speicher* der logischen Architektur der PS-Server (Abbildung 5.5) bildet die Grundlage für die Realisierung von transaktionalem Verhalten (Anforderung 5.7) und Wiederherstellbarkeit (Anforderung 5.8).

Für die Umsetzung dieser Persistenzschicht kommen unterschiedliche Realisierungsalternativen aus dem Bereich der Datenbanken in Betracht. Diese umfassen *Relationale Datenbank-Management-Systeme (RDBMS)* (beispielsweise verwendet in früheren Versionen von *Mozartspaces*[1], eine Realisierung der XVSM-Technologie [KRJ05]), *Objekt-Datenbanken*, *XML-Datenbanken* oder *Record*-basierte persistente Datenspeicher (in *Blitz Javaspaces*[2] kommt beispielsweise *Berkeley DB*[3] zum Einsatz).

Für sämtliche Realisierungsalternativen muss eine Abbildung (i) eines Tupels, (ii) eines PS als logische Gruppierung von Tupeln, (iii) der Template-Matching-Operatoren und (iv) der PS-Operationen auf die verwendete Persistenz-Technologie erfolgen; aufgrund dieser Anforderungen können die oben genannten Realisierungsalternativen wie folgt bewertet werden.

Aufgrund ihrer Eigenschaft der Vorgabe eines starren Schema für erfasste Daten eigenen sich RDBMS gut für Daten, die einer vorab bekannten Struktur folgen. Obwohl die Ausführung von EWFN-Modellen unter Verwendung einer vorab definierten Tupelstruktur (vgl. Abschnitt 5.6.1) erfolgt und damit eine Realisierung auf Basis eines RDBMS möglich wäre, ist die PS-Infrastruktur nicht auf den Anwendungsfall der EWFN-Ausführung festgelegt. Sie kann darüber

[1] http://www.mozartspaces.org
[2] http://www.dancres.org/blitz
[3] http://www.oracle.com/technology/products/berkeley-db/index.html

hinaus als generische Implementierung des Tuplespace-Konzepts betrachtet werden und damit als Plattform für weitere Anwendungen bieten. Ein Beispiel hierfür ist das in Kapitel 6 vorgestellte Web-Service-Binding für Tuplespaces. Die hieraus resultierende potentiell beliebige Struktur der kommunizierten Tupel sowie die Schemalosigkeit der PS sprechen damit gegen ein RDBMS als Realisierungsplattform. XML-Datenbanken erlauben zwar eine relativ einfache Ablage beliebig strukturierter Daten, verlangen allerdings in Fällen wie der PS-Infrastruktur, in denen XML nicht das native Datenformat der manipulierten Daten ist, eine relativ aufwändige (De-) Serialisierung der Daten nach bzw. vor dem Persistierungsvorgang. Die relativ einfachen Template-Matching-Operationen der PS-Infrastruktur allein motivieren weder die Verwendung eines RDBMS noch einer XML-Datenbank in ausreichendem Maße, um die oben genannten Nachteile aufzuwiegen.

In Record-basierte Datenbanken, welche eine persistente Ablage von Schlüssel-Wert-Paaren erlauben, ist die Abbildung der Konzepte Tupel und PS einfach realisierbar. Record-basierte Datenbanken haben allerdings die Eigenschaft, dass der Wert eines Eintrags der Datenbank für das DBMS im Allgemeinen opak ist, ein Zugriff auf den Wert damit ausschließlich durch seinen Schlüssel erfolgen kann. Die Realisierung des Template-Matching PS-Operationen (welches potentiell auf den Werten sämtlicher Felder eines Tupel operieren kann) verlangt somit im Fall Record-basierter Datenbanken die Umsetzung entsprechender Datenstrukturen auf Anwendungsebene.

Aus Implementierungssicht einfacher gestaltet sich die Realisierung des Template-Matching-Verfahrens bei der Verwendung einer Persistenz-Technologie, die die eine direkte Ablage von Objekten erlaubt, da in diesem Fall die abgelegten Daten für das DBMS transparent sind und damit beispielsweise ein direkter Zugriff auf die Werte der einzelnen Felder eines Tupels möglich wird.

Aus den oben genannten Gründen kommt für die Realisierung des PS-Server-Prototyps (vgl. Abschnitt 7.1) daher ein, auf dem Konzept der *Object Prevalence* [BJW87] basierender, transaktionaler und persistenter Objekt-Speicher zum Einsatz. Dessen Funktionsweise wird im Folgenden erläutert.

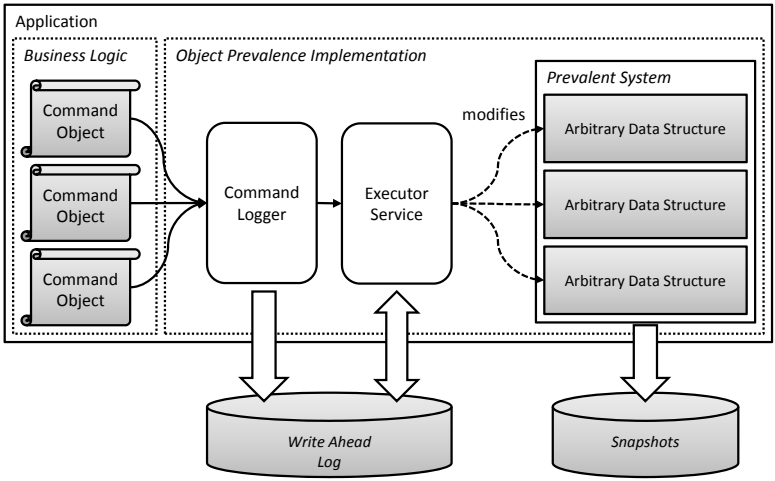


Abbildung 5.7: Schematische Darstellung der *Object-Prevalence*-Konzepte.

5.4.1 Realisierung durch *Object Prevalence*

Das von *Object Prevalence* adressierte Problem ist die transaktionale Modifikation von Daten. Abbildung 5.7 veranschaulicht die Konzepte von *Object Prevalence* graphisch. Kernelement des Ansatzes bildet das sogenannte *Prevalent System*, ein Container, der beliebige Datenstrukturen beherbergen kann. Lesender oder modifizierender Zugriff auf das Prevalent-System erfolgt dabei ausschließlich unter Verwendung von *Command*-Objekten [GHJV95], die die Anwendungslogik kapseln, die auf das Prevalent-System angewendet werden soll. Die Command-Objekte werden durch einen *Command Logger*, vor ihrer Verarbeitung durch den sogenannten *Executor-Service*, in Form eines *Write Ahead Logs (WAL)* [HR01] serialisiert und auf einem persistenten Medium ablegt. Die Verarbeitung der serialisierten Command-Objekte erfolgt durch den Executor-Service transaktional atomar, so dass das Scheitern der Ausführung der Anwendungslogik eines Command-Objekts dazu führt, dass bereits am Prevalent-System vorgenommene Änderungen wieder zurückgenommen werden. Somit ist zu jedem Zeitpunkt dessen konsistenter Zustand gewährleistet.

Erfolgt nach einem fatalen Systemfehler der *Recovery*-Vorgang des PS-Servers (vgl. Abschnitt 5.5.4), so werden als Teil des Wiederherstellungsvorgangs die Command-Objekte im WAL des Servers erneut ausgeführt und damit der Zustand des Prevalent-System zum Zeitpunkt des Systemfehlers wiederhergestellt. Um den Zeitraum eines eventuellen Wiederherstellungsvorgangs zu minimieren und das WAL in seiner Größe zu beschränken, werden in regelmäßigen Abständen sogenannte *Snapshots* des Prevalent-System erzeugt und diese ebenfalls serialisiert und persistent abgelegt. Sämtliche WAL-Einträge bis zum Snapshot können somit verworfen werden; der Recovery-Vorgang beginnt mit der Wiederherstellung des letzten Snapshots und führt lediglich die restlichen, im WAL protokollierten, Command-Objekte erneut aus.

Für die Anwendung der Konzepte der *Object Prevalence* in der PS-Infrastruktur werden die Operationen der PS-Schnittstelle sowie die Template-Matching-Funktionen in Form von *Command*-Objekten realisiert, welche auf den jeweiligen PS als Prevalent-System angewendet werden. Die Umsetzung der Persistenzschicht auf Basis der *Object Prevalence*-Implementierung *Prevayler*[1] ist in Abschnitt 7.1 beschrieben.

5.5 Realisierung transaktionalen Verhaltens

Hinsichtlich der Transaktionsunterstützung (Anforderung 5.7) der PS-Infrastruktur gelten eine Reihe unterschiedlicher Anforderungen und Eigenschaften, welche zum Teil in Kapitel 3 und im Verlauf dieses Kapitels bereits genannt wurden. Sie lassen sich wie folgt zusammenfassen:

- Die PS der PS-Infrastruktur haben die Eigenschaften transaktionaler und persistenter Nachrichtenspeicher. PS-Klienten müssen keinen Zustand über die Grenzen eines Ausführungszyklus hinaus halten, d. h. sämtliche für die Prozessausführung benötigten Informationen sind in den Tupeln der einzelnen PS manifestiert.
- Bedingt durch die dezentrale Prozessnavigation in der PS-Infrastruktur existiert keine zentraler Koordinator der Prozessausführung; die ein-

[1] http://www.prevayler.org

zelnen PS-Klienten koordinieren ihre Ausführung untereinander. Die Koordination erfolgt dabei nicht direkt, sondern mittels *Token-Passing* über einen (oder mehrere) PS.

- Sämtliche Verarbeitungsschritte, die ein PS-Klient während eines Ausführungszyklus durchführt, sind durch die EWFN-Patterns [Mar10] explizit beschrieben; Seiteneffekte bestehen nicht.
- Atomare Operationen in der Hinsicht, dass sämtliche Ausführungsteilnehmer eines Prozesses zu einem bestimmten Zeitpunkt einen synchronen atomaren Commit ausführen müssen, sind in der PS-Infrastruktur, aufgrund der Interaktion zwischen einzelnen PS-Klienten, nicht erforderlich. Dies ist eine Folge des oben genannten *Token-Passing*-Verfahrens zur Kommunikation von PS-Klienten sowie der Persistenz- und Recovery-Eigenschaften der PS (vgl. Erläuterungen im nachfolgenden Abschnitt).
- Da sich die Ausführung eines Prozesses potentiell über große Menge unterschiedlicher Ausführungsteilnehmer (sowohl im Hinblick auf PS-Server als auch PS-Klienten) verteilen kann, muss gewährleistet sein, dass die gegenseitige Beeinflussung der unterschiedlichen Ausführungsteilnehmer minimiert wird, d. h. soweit möglich nur unmittelbar betroffene PS-Klienten und PS-Server Teil derselben Transaktion sind.
- Als Teil der Wiederherstellung des Systems nach einem Fehler (Anforderung 5.8) sollen nur diejenigen PS-Operationen erneut ausgeführt werden, die von der fehlgeschlagenen Transaktion unmittelbar betroffen sind. Eine Beeinflussung von PS-Servern, die an der Transaktion nicht unmittelbar Teilnehmen müssen, ist zu vermeiden.
- Es muss jedoch gelten, dass sich das Gesamtsystem, d. h. die Gesamtheit des PS-Klienten, der PS-Server und der auf den PS-Servern abgelegten Tupel, jederzeit in einem konsistenten Zustand befindet.

Aufgrund der oben genannten Eigenschaften stützt sich das transaktionale Verhalten der PS-Infrastruktur auf das Modell der sogenannten *Stratifizierten Transaktionen*.

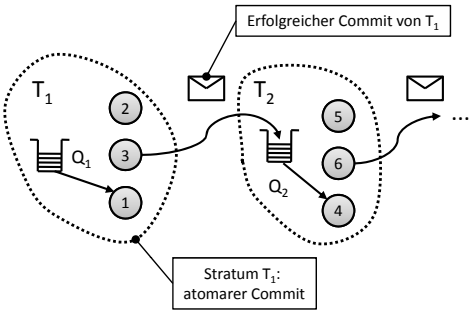


Abbildung 5.8: Verkettung von Strata durch Nachrichtenaustausch über persistente, transaktionale Message Queues.

5.5.1 Stratifizierte Transaktionen

Die Grundannahme *Stratifizierter Transaktionen (SA)* [LR99, Ley97] ist es, dass nicht in allen Szenarien verteilter Transaktionen der Commit aller, an einer Gesamttransaktion $\mathcal{T}$ beteiligter, Teiltransaktionen $T_1, \cdots, T_n$ zeitlich synchron erfolgen muss. Für viele Anwendungsfälle ist die Zusicherung ausreichend, dass zu irgendeinem Zeitpunkt in der Zukunft die Gesamttransaktion erfolgreich abgeschlossen ist. Unter der Annahme dieser "Aufweichung" der Atomizitäts-Eigenschaft ermöglichen *Stratifizierte Transaktionen* eine effiziente Ausführung komplexer verteilter Transaktionen durch Verkettung mehrerer Transaktionsverbunde (sogenannter *Strata*), welche intern ein atomares Commit-Protokoll verwenden, durch den Austausch von Nachrichten über persistente und transaktionale *Message Queues*.

Abbildung 5.8 veranschaulicht die Verkettung einzelner Strata graphisch. Transaktionen $1, 2, 3$ in Stratum T_1 synchronisieren sich über ein atomares Commit-Protokoll, ebenso wie Transaktionen $4, 5, 6$ in Stratum T_2. Nach erfolgtem Commit der Transaktionen in Stratum T_1 legt Transaktion 3 eine Nachricht transaktional in die Eingangswarteschlange Q_2 von Stratum T_2. Als Teil der Ausführung von Transaktion 4 in Stratum T_2 wird diese Nachricht transaktional aus Q_2 konsumiert und damit die Ausführung von Stratum T_2 gestartet.

Scheitert die Ausführung einer der Transaktionen in Stratum T_2, so resultiert dies in einem *Abort* von T_2 und damit sämtlicher Teil-Transaktionen $4, \ldots, 6$. Dieser *Abort* hat damit insbesondere das Zurückrollen der Transaktion 4 zur Folge, welche die Nachricht, die die Ausführung von Stratum T_2 ausgelöst hatte, aus Q_2 konsumiert hat. Da dieses Konsumieren selbst als Teil von T_2 erfolgt, resultiert das Zurückrollen von Transaktion 4 in einem Zurückschreiben der transaktionsauslösenden Nachricht in Q_2. Beim Wiederanlaufen des Systems kann die entsprechende Nachricht wieder aus Q_2 konsumiert und damit die Ausführung der Teil-Transaktionen von T_2 erneut gestartet werden. Im Besonderen ist zu bemerken, dass der Abbruch der Teil-Transaktion 4 in T_2 keine Auswirkungen auf den erfolgreichen Commit von T_1 hat.

Für die Teil-Transaktionen einer *Stratifizierten Transaktion* gilt, aufgrund der eventuell wiederholten Ausführung der Transaktionen im Rahmen des Neustarts des Stratums, die Auflage, dass diese entweder keine Seiteneffekte haben (bzw. diese idempotent sind) oder die Seiteneffekte kompensierbar sind und deren Kompensation durch den *Abort* des Stratums ausgelöst wird. Ein Problem stratifizierter Transaktionen sind sogenannte *Poisoned Messages* [HW04], Nachrichten die bei wiederholter Verarbeitung zu immer derselben Fehlersituation führen. Wie oben beschrieben wird durch die Transaktionssemantik der Strata im Fehlerfall (d. h. *Abort* des Stratums) die transaktionsauslösende Nachricht wieder in die Eingangswarteschlange des Stratums zurückgelegt. Führt ein wiederholtes Konsumieren der Nachricht zur selben Fehlersituation, wiederholt sich dieser Zyklus fortlaufend und blockiert damit die weitere Verarbeitung der Gesamttransaktion. Eine Lösung dieses Problems stellt die Verwendung eines sogenannte *Invalid Message Channels* [HW04] dar. Dieser beschreibt in einer MQ-basierten Applikation eine Queue, in die fehlerhafte Nachrichten zur Durchsicht durch einen Administrator eingestellt werden können. Die Entscheidung, wann eine Nachricht in den Invalid-Message-Channel eingestellt werden soll, kann beispielsweise auf Grundlage der Anzahl der Verarbeitungsversuche einer Nachricht erfolgen.

[LR99] gibt die folgende formale Definition für eine *Stratifizierte Transaktion*. Als Grundlage für die Beschreibung der Anwendung stratifizierter Transaktionen auf EWFN-Modelle wird diese an dieser Stelle wiederholt.

Definition 5.9 (Stratifizierte Transaktion)**.** *Eine* Stratifizierte Transaktion $\mathcal{Z} = (\mathcal{T}, \mathcal{M})$ *ist nach [LR99] beschrieben durch die Mengen* $\mathcal{T}$ *und* $\mathcal{M}$*.*

Für die Menge der Strata $\mathcal{T}$ *gilt*

$$\mathcal{T} = \{S_1, \cdots, S_k\} \text{ mit } S_i = \{s_1, \cdots, s_{n_i}\}, 1 \leq i \leq k, n_i \in \mathbb{N}$$

Wobei S_i *eine Menge von Transaktionen – ein sogenanntes* Stratum *– bezeichnet*[1]*. Es gilt weiterhin:*

(i) $\forall 1 \leq i \leq k : S_i \neq \emptyset$*, d. h. jedes* S_i *besteht aus mindestens einer Transaktion,*

(ii) $\forall 1 \leq i, j \leq k : i \neq j \Rightarrow S_i \cap S_j = \emptyset$*, d. h. dass keine Teiltransaktion in mehr als einem Stratum vorkommen darf,*

(iii) $\bigcup_{i=1}^{k} S_i = \mathcal{T}$*, d. h. die Gesamtheit aller Strata entspricht der Gesamttransaktion.*

(iv) Weiterhin gilt $\forall 1 \leq i \leq k : S_i$ *hat eine Eingangswarteschlange* Q_i *und* $\forall 1 \leq i \leq k : S_i$ *ist eine globale Transaktion , d. h. ein durch ein atomares Commit-Protokoll verknüpfter Verbund von Transaktionen.*

(v) $\mathcal{M} \subseteq \mathcal{T} \times \mathcal{T}$ *definiert eine Ordnung der Strata über die Menge von* $\mathcal{T}$*.*

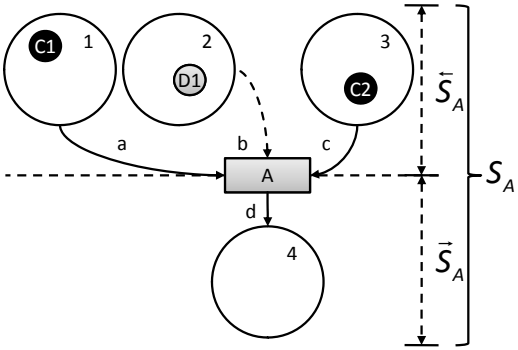
(vi) Es gilt $(S_i, S_j) \in \mathcal{M} \iff \exists s \in S_i : s$ *legt eine Nachricht in* $Q_j \wedge \exists t \in S_j :$ t *liest eine Nachricht aus* Q_j*.*

In der Folge wird erläutert, wie die Stratifizierung von Transaktionen auf die Ausführung eines EWFN-Modells angewendet werden kann.

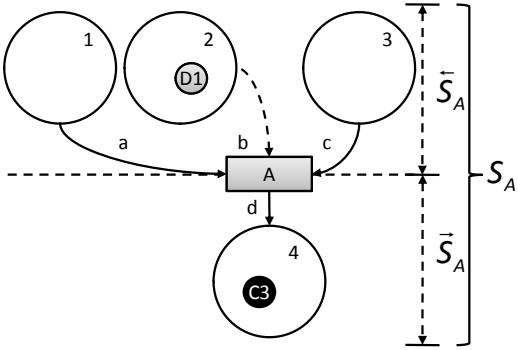
5.5.2 Stratifizierte Transaktionen für EWFN-Modelle

Abbildung 5.9(a) zeigt exemplarisch ein Szenario für einen Schaltvorgang einer Transition eines EWFN, in welchem ein PS-Klient mit vier PS interagiert. Der PS-Klient *A* konsumiert hierbei Tupel aus den PS 1, 2 und 3 während der Ausführung der Operationen *a*, *b* und *c*. Die Operationen *a* und *c* sind destruktiv, Operation *b* ist nicht-destruktiv. Nach dem Konsumieren führt A einen internen Verarbeitungsschritt (d. h. die von ihm implementierte Funktionalität) durch

[1] Um den Bezeichnerkonflikt zwischen einem Stratum T in [LR99] mit den Transitionen im EWFN-Formalismus zu lösen, wird ein Stratum hier durch den Bezeichner S repräsentiert.



(a) Teil-Transaktion T_1: Erfüllung der Eingangsbedingung



(b) Teil-Transaktion T_2: Erzeugung der Ausgangseffekte

Abbildung 5.9: Transaktionale Ausführung eines PS-Klienten.

und signalisiert die Beendigung seiner Ausführung durch die Produktion eines Kontrollflusstupels in den PS 4 (Abbildung 5.9(b)).

Vergleicht man die "Navigation" eines EWFN mit der in Abschnitt 5.5.1 aufgeführten Definition stratifizierter Transaktionen, so zeigt sich, dass in beiden Modellen die transaktionale Ausführung einer komplexen Gesamttransaktion (die Ausführung eines EWFN-Modells) dadurch erfolgt, dass diese in einzelne Teilschritte (die Ausführung der einzelnen Transitionen, hier A) aufgespalten wird, welche durch relativ lokale Eingangsbedingungen (das Konsumieren

einer entsprechenden Nachricht aus einer Eingangswarteschlange bzw. die erfolgreiche Ausführung der Operationen a, b und c) und Ausgangseffekte (das Schreiben einer Nachricht in eine Ausgangswarteschlange bzw. die erfolgreiche Ausführung der Operation d) gekennzeichnet sind.

Die Ausführung eines Teilschritts der Gesamttransaktion durch den PS-Klienten A erfolgt dabei als atomare Transaktion S_A, welche die Mengen $\overleftarrow{S_A}$ und $\overrightarrow{S_A}$ von transaktional ausgeführten Operationen für die Erfüllung der Eingangsbedingung bzw. Ausgangseffekte umfasst.

Definition 5.10 (Stratifizierte Transaktionen zur Ausführung von EWFN-Modellen)**.** *Angelehnt an die Definition stratifizierter Transaktionen (Definition 5.9) gelten bei Anwendung auf die Ausführung von $EWFN_{Linda}$-Modellen für eine* Stratifizierte EWFN-Transaktion $\mathcal{Z}_{EWFN} = (\mathcal{T}_{EWFN}, \mathcal{M}_{EWFN})$ *die folgenden Eigenschaften*[1]*:*

(i) Ein Stratum $S_t \in \mathcal{T}_{EWFN}$ beinhaltet sämtliche Operationen, welche der PS-Klient, der die Transition $t \in T$ realisiert, während eines Schaltvorgangs transaktional ausführt. Die für die Erfüllung der Kommunikation und Koordination notwendigen Operationen des PS-Klienten lassen sich zusammenfassen als: $S_t = \overleftarrow{S_t} \cup \overrightarrow{S_t}$. Dabei gilt für die Menge $\overleftarrow{S_t}$ der Operationen zur Erfüllung der Eingangsbedingung gilt: $\overleftarrow{S_t} = \{f \in F | \pi_2(f) = t\}$
Analog gilt für die Menge $\overrightarrow{S_t}$: $\overrightarrow{S_t} = \{f \in F | \pi_1(f) = t\}$

(ii) Die Operationen eines Stratums S_t können sich auf unterschiedliche PS beziehen. Für die Erfüllung der Eingangsbedingung sind dies die PS, die die Stellen der Menge $\bullet t$ mit $\bullet t = \{p | (p, t, r) \in F\}$, repräsentieren. Für die PS, in welchen die Materialisierung der Ausgangseffekte erfolgt $t\bullet$ gilt $t\bullet = \{p | (t, p) \in F\}$.

(iii) Die Vereinigung sämtlicher Strata entspricht der stratifizierten Transaktion, d. h. der Gesamtheit der transaktional ausgeführten Operationen sämtlicher Strata (repräsentiert durch die Kanten des EWFN). Es gilt also: $\mathcal{T}_{EWFN} = \{S_t | t \in T\} = F$

[1] Für die folgende Darstellung wird die in Kapitel 4 von [Mar10] definierte Notation für $EWFN_{Linda}$ verwendet. Dies Umfasst die Mengen T, F und die Multiset-Operationen + und − sowie den Template-Matching-Operator $\approx \subseteq \Sigma \times X$

(iv) $\forall S \in \mathcal{T}_{EWFN} : |\overleftarrow{S}| \geq 1 \wedge |\overrightarrow{S}| \geq 1$, *d. h. jedes Stratum umfasst jeweils mindestens die Navigation einer Kante für die Realisierung der Eingangsbedingung und der Ausgangseffekte.*

(v) $\forall S_i, S_j \in \mathcal{T}_{EWFN}, i \neq j : S_i \cap S_j = \emptyset$, *d. h. die durch eine Kante repräsentierte Operation ist immer Teil genau eines Stratums.*

(vi) *Am Beispiel der Schaltsemantik von* $EWFN_{Linda}$ *gilt, dass Stratum* $S_t \in \mathcal{T}_{EWFN}$ *des PS-Klienten* $t \in T$ *genau dann ausführungsbereit ist, wenn für jede eingehende Kante* $\overleftarrow{f} \in F$ *mit* $\pi_2(\overleftarrow{f}) = t$ *eine Marke in der jeweiligen Stelle aus* $\bullet t$ *enthalten ist, welche zum Template von* $\overleftarrow{f}$ *konform ist. Formal gilt:*

$$\forall \overleftarrow{f} \in \overleftarrow{S_t} : L_{read}(A(\overleftarrow{f}), M^{MS}(\pi_1(\overleftarrow{f}))) \neq \varepsilon$$

(vii) *Bei dem erfolgreichen Commit eines Stratums wird das Konsumieren der Eingangsbedingungen und das Erzeugen der Ausgangseffekte für andere Strata "sichtbar".*
Gilt vor der Ausführung einer Kante $\overleftarrow{f} \in \overleftarrow{S_t}$ *das Marking* M_1^{MS}*, so gilt nach der erfolgreichen Ausführung von* $\overleftarrow{f}$ *das Marking* M_2^{MS} *mit:*

$$M_2^{MS}(\pi_1(\overleftarrow{f})) = M_1^{MS}(\pi_1(\overleftarrow{f})) - L_{read}(A(\overleftarrow{f}), M_1^{MS}(\pi_1(\overleftarrow{f})))$$

Nach der Ausführung einer Kante $\overrightarrow{f} \in \overrightarrow{S_t}$ *gilt entsprechend:*

$$M_2^{MS}(\pi_2(\overrightarrow{f})) = M_1^{MS}(\pi_2(\overrightarrow{f})) + L_{write}(\overrightarrow{f})$$

Für den Schaltvorgang einer Transition, d. h. die Ausführung eines Stratums S_t *gilt somit die in [Mar10] definierte Regel:*

$$\forall p \in \bullet t : M_2^{MS}(p) = M_1^{MS}(p) - L_{read}(A(p, t, \text{"take"}), M_1^{MS}(p))$$
$$\forall p \in t\bullet : M_2^{MS}(p) = M_1^{MS}(p) + L_{write}(t, p)$$

(viii) $\mathcal{M}_{EWFN} \subseteq \mathcal{T}_{EWFN} \times \mathcal{T}_{EWFN}$ *definiert eine Ordnung über die Menge der Strata. Es gilt* $(S_A, S_B) \in \mathcal{M}_{EWFN}$ *genau dann, wenn die erfolgreiche Ausführung von* S_A *zur Herstellung der Eingangsbedingung von* S_B *beiträgt, d. h.* S_A *Tupel*

in PS publiziert, die Teil von $\overleftarrow{S_B}$ sind und diese zusätzlich zu einem von B spezifizierten Template konform sind. Formal gilt:

$$(S_A, S_B) \in \mathcal{M}_{EWFN} \Leftrightarrow \exists \overrightarrow{f_A} \in \overrightarrow{S_A}, \overleftarrow{f_B} \in \overleftarrow{S_B} : \\ \pi_2(\overrightarrow{f_A}) = \pi_1(\overleftarrow{f_B}) \wedge \\ L_{write}(\overrightarrow{f_A}) \approx A(\overleftarrow{f_B})$$

Die Realisierung des atomaren Commit der Transaktionen innerhalb eines Stratums erfolgt unter Verwendung des *Zwei-Phasen-Commit* (engl. *Two Phase Commit (2PC)*) [SS83, GR92] Protokolls. Die Implementierung des 2PC-Protokolls in der prototypischen Umsetzung des Systems ist in Abschnitt 7.1.3 näher erläutert.

5.5.3 Transaktionssemantik der PS-Operationen

Werden PS-Operationen innerhalb einer Transaktion ausgeführt, so gelten die nachfolgend aufgeführten Tupel-Sichtbarkeitsregeln für transaktionsexterne PS-Klienten.

Beim transaktionalen Schreiben von Tupeln durch die *write*-Operation werden die durch den Operationsaufruf publizierten Tupel erst nach erfolgreichem *Commit* der Transaktion extern sichtbar. Wird die Transaktion abgebrochen, so werden die vom PS-Klienten geschriebenen Tupel aus dem PS wieder entfernt. Durch diese Semantik wird sichergestellt, dass das Gesamtsystem auch nach nicht erfolgreicher Ausführung eines Klienten wieder in einen konsistenten Zustand überführt wird; d. h. die Ausgangseffekte einer Transition werden entweder vollständig oder nicht erzeugt. Als Resultat dieser Sichtbarkeitsregeln ist das transaktionale Verhalten frei von sogenannten *Cascading Aborts* [CDK05].

Bei der transaktionalen Ausführung der *read*-Operation bestimmt der PS-Server zunächst ein zu dem, vom jeweiligen Klienten spezifizierten, Template konformes Tupel. Dieses Tupel wird für destruktive Leseoperationen anderer Klienten so lange gesperrt, bis die Transaktion entweder erfolgreich abgeschlossen oder abgebrochen wurde. Mit dieser Einschränkung wird der Fall ausgeschlossen, dass ein parallel ausgeführter Klient das, für die Befriedigung

der *read*-Operation bestimmte, Tupel im Zeitraum der Ausführung der Transaktion destruktiv konsumiert. Ein paralleles Lesen durch andere Klienten ist von der Sperrung nicht betroffen. Ein analoges Verhalten gilt auch für die *readAll*-Operation, welche eine Blockierung sämtlicher Template-konformen Tupel im jeweiligen PS für destruktive Leseoperationen im Zeitraum der Transaktionsausführung zur Folge hat.

Für die transaktionale Ausführung der *take*- und *takeAll*-Operationen gilt, dass ein Template-konformes Tupel (bzw. alle Template-konformen Tupel) über den gesamten Zeitraum der Transaktion für sämtliche konsumierenden Operationen (d. h. *take*, *read*, *readAll* und *sync*) gesperrt ist. Entsprechendes gilt für die *destroy*-Operation.

Während der transaktionalen Ausführung der *update*-Operation wird das zu modifizierende Tupel für sämtliche lesenden und schreibenden Operationen gesperrt. Die durch *update* durchgeführten Änderungen werden für nicht an der Transaktion teilnehmende PS-Klienten erst nach erfolgreichem *Commit* der Transaktion sichtbar.

5.5.4 Wiederanlaufen von PS-Klienten und PS-Servern nach einem Systemfehler

Da die PS-Infrastruktur ein *Client-Server*-System ist, umfasst der Vorgang des Wiederanlaufens des Systems nach einem fatalen Systemfehler die Ausführung des Wiederherstellungsvorgangs, sowohl auf den PS-Klienten (die sogenannte *Client Recovery*) als auch den PS-Servern (die sogenannte *Server Recovery*).

Wie in Abschnitt 5.5.2 erläutert, basiert das transaktionale Modell der PS-Infrastruktur auf stratifizierten Transaktionen. Es gilt somit, dass sowohl die Erfüllung der Eingangsbedingung eines Klienten, die Ausführung seines internen Verarbeitungsschritts und die Erzeugung seiner Ausgangseffekte innerhalb einer atomaren Transaktion erfolgen. Ein Abbruch der Ausführung eines PS-Klienten führt zum *Abort* des jeweiligen Stratums, was wiederum sowohl ein Rückschreiben bereits konsumierter Eingangsdaten als auch eine Rücknahme bereits erzeugter Ausgangseffekte zur Folge hat. Entsprechend der in Abschnitt 5.5.3 beschriebenen Transaktionssemantik der PS-Operationen gilt, dass ge-

schriebene Tupel erst dann für nicht an der Transaktion teilnehmende Klienten sichtbar werden, wenn die Transaktion erfolgreich abgeschlossen wurde. Hierdurch wird einerseits eine vorzeitige Ausführung von im EWFN-Modell nachfolgenden PS-Klienten ausgeschlossen, andererseits wird sichergestellt, dass die Eingangsbedingungen des jeweiligen Stratums vor dessen Ausführungsbeginn wieder vollständig hergestellt werden. Der Vorgang des Wiederanlaufens eines PS-Klienten umfasst somit das erneute Konsumieren der Eingangsdaten und der damit verbundenen Wiederholung der Verarbeitung der Nachricht unter Einhaltung der in Abschnitt 5.5.1 genannten Vorgehensweise zur Verarbeitung von *Poisoned-Messages*.

Ein weiterer Aspekt des Wiederanlaufens eines PS-Klienten nach einem Fehler ist die Wiederherstellung seiner Konfiguration. Diese können PS-Klienten bei der Deployment-Komponente des PS-Servers ihrer Domäne erfragen. Das Erfragen der Konfiguration erfolgt dabei entsprechend des in Abschnitt 5.8 beschriebenen Verfahrens.

Aufgrund der gewählten Realisierung der Persistenzschicht stützt sich der Vorgang der *Server Recovery* – entsprechend der Beschreibung in Abschnitt 5.4.1 – auf die Rekonstruktion des letzten Snapshots des Prevalent-System des jeweiligen Servers und die erneute Ausführung der im WAL protokollierten *Command*-Objekte.

5.6 Verwendung des Coordination Kernel zur Ausführung von BPEL-Prozessen

Der *Coordination Kernel*, dessen Schnittstelle in Abschnitt 5.3 vorgestellt wurde, erlaubt die dezentrale Koordination der Ausführung verteilter PS-Klienten und bietet die nötigen Funktionen für den Zugriff auf die (expliziten und impliziten, vgl. Abschnitt 4.2.3) Instanzdaten, die für die Ausführung der Funktionalität der Klienten notwendig sind. Anknüpfend an die Erläuterungen in [Mar10] wird in den folgenden Abschnitten dargelegt, wie (i) eine Ausführung von BPEL-Prozessen auf Basis der BPEL-EWFN-Patterns unter Verwendung des Coordination Kernel erfolgen kann und wie (ii) das Laufzeitverhalten der Pro-

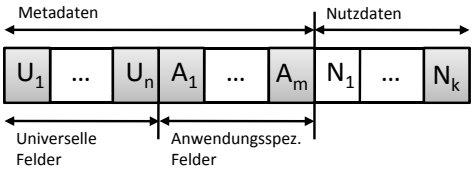


Abbildung 5.10: Generischer Aufbau der für die Kommunikation der PS-Klienten verwendeten Tupel.

zessausführung gegenüber diesem durch speziell auf die effiziente Ausführung von BPEL-Prozessen ausgerichtete Funktionen der PS-Server verbessert werden kann.

5.6.1 Tupelstruktur

Die Ausführung von BPEL-Prozessen unter Verwendung der PS-Infrastruktur erfordert die Präsenz einer Reihe von Informationen in den Tupeln, die zwischen den einzelnen PS-Klienten kommuniziert werden. Die Vorstellung dieser Struktur sowie die Erläuterung der Bedeutung dieser Informationen für die Prozessausführung ist Gegenstand dieses Abschnitts.

Abbildung 5.10 zeigt eine Übersicht der Tupelstruktur, wie sie für die in dieser Dissertation vorgestellten Anwendungen auf der PS-Infrastruktur, d. h. die Navigation von EWFN-Modellen sowie das in Kapitel 6 vorgestellte Web-Service-Binding für Tuplespaces, Verwendung findet. Die PS-Infrastruktur ist allerdings generisch insofern, als dass diese Struktur nicht bindend ist, sondern für andere Anwendungen eine andere Struktur gewählt werden kann.

Entsprechend der in MOM [HW04] gebräuchlichen Terminologie können die Felder eines Tupels einem *Metadaten*- und einem *Nutzdaten*-Anteil zugeordnet werden. Die Felder des Metadaten-Anteils umfassen zum einen die sogenannten *Universellen Felder* $(U_1, \ldots, U_n)$, die anwendungsübergreifend Verwendung finden; zum anderen beinhalten die sogenannten *anwendungsspezifischen Felder* $(A_1, \ldots, A_m)$ alle Felder, die innerhalb einer bestimmten Anwendung der PS-Infrastruktur (z. B. die Ausführung von EWFN-Modellen) definiert sind.

Feld	Typ	Beschreibung
U-AID	URI	Anwendungsidentifikator
U-AVER	Number	Versionsidentifikator
U-CONST	Boolean	Konstantes Tupel
U-UNPROC	Boolean	Tupel kann nicht verarbeitet werden
U-TIMESTAMP	Timestamp	Zeitpunkt der Erzeugung des Tupels

Tabelle 5.1: Universelle Tupelfelder

Die Felder des Nutzdaten-Anteils eines Tupels $(N_1, \ldots, N_k)$ beinhalten die eigentlichen anwendungsspezifischen Nutzdaten des Tupels. Die universellen und anwendungsspezifischen Felder sowie deren Bedeutung wird in der Folge erläutert, die Nutzdatenfelder für jedes Template bzw. jedes produzierte Tupel der PS-Klienten der einzelnen BPEL-Aktivitäten sind aus den Tabellen in Kapitel 5 von [Mar10] ersichtlich.

5.6.1.1 Universelle Felder

Tabelle 5.1 zeigt eine Übersicht der universellen Tupelfelder.

Das Feld U-AID identifiziert die Applikation, welche das jeweilige Tupel erzeugt und in den PS geschrieben hat, in Form einer URI [BLFM05]; im Fall der Ausführung eines EWFN-Modells ist dessen Wert:

```
http://iaas.uni-stuttgart.de/ns/2008/process-space
```

Um eine Weiterentwicklung der Klienten der PS-Middleware bei gleichzeitiger Gewährleistung von Abwärtskompatibilität von Anwendungen zu erlauben (vgl. Anforderung 5.5), identifiziert das Feld U-AVER die Versionsnummer der Anwendung, die das jeweilige Tupel erzeugt hat.

Das Boolesche Feld U-CONST markiert ein Tupel als konstant. Ein konstantes Tupel hat die Eigenschaft, dass ein PS-Klient, nachdem er ein als konstant markiertes Tupel ein erstes Mal konsumiert hat, bei einem späteren Zugriff auf das Tupel nicht nochmals prüfen muss, ob das Tupel in dieser Form noch im PS verfügbar ist oder inzwischen durch einen anderen Klienten aus dem

PS entfernt oder modifiziert wurde. Die PS-Operationen verhalten sich auf einem konstanten Tupel allerdings äquivalent zu ihrer Anwendung auf nichtkonstante Tupel, d. h. entnimmt ein PS-Klient *A* ein konstantes Tupel mittels einer *take*-Operation, so wird dieses aus dem jeweiligen PS entfernt. Hat ein anderer Klient *B* das konstante Tupel bereits vor dessen Entnahme durch *A* (nicht-destruktiv) konsumiert, so wird bei wiederholtem lesenden Zugriff von *B* das Tupel auch nach dessen Entnahme durch *A* dieses aus dem klientenseitigen Zwischenspeicher an *B* zurückgeliefert. Die Interpretation von U-CONST erfolgt also (im Gegensatz zu anderen Umsetzungen konstanter Tupel, beispielsweise der CONST-Tupel in CORSO [Küh94, KN98]) ausschließlich auf Seite der PS-Klienten und bildet die Grundlage für eine Optimierung des Verfahrens zur klientenseitigen Zwischenspeicherung einmal konsumierter Tupel, welches in Abschnitt 5.6.2 erläutert wird.

Für den Fall, dass die Verarbeitung eines Tupels durch einen PS-Klienten mehrfach in einem fatalen Fehler resultiert (welcher nicht Teil seiner Anwendungslogik ist) und dieser Fehler auf ein fehlerhaftes Datum in einem konsumierten Tupel zurückzuführen ist (beispielsweise durch einen Fehler während des Parsens des Nutzdatenanteils eines Tupels), kann der Klient dessen Feld U-UNPROC auf ein Boolesches `true` setzen, das Tupel in den jeweiligen PS zurückschreiben und einen Administrator über den aufgetretenen Fehler benachrichtigen. Damit wird dieses Tupel für die weitere Verarbeitung durch diesen und andere Klienten der jeweiligen Anwendung gesperrt[1]. Das U-UNPROC-Feld erfüllt somit die Funktion eines *Invalid Message Channel* [HW04, MWSL07].

Das Feld U-TIMESTAMP beinhaltet den Zeitpunkt der Veröffentlichung des jeweiligen Tupels. Für die Belegung des Felds gilt in der PS-Infrastruktur eine besondere Semantik. Wird von einem Klient beim Schreiben eines Tupels kein Wert für im Feld U-TIMESTAMP vorgegeben, so setzt der verarbeitende PS-Server dessen Wert auf den Zeitpunkt (relativ zur lokalen Uhr des jeweiligen PS-Servers), an welchem die Operation durch die PS-Infrastruktur verarbeitet wurde. Somit werden die Zeitstempelwerte auch bei mehreren, von unterschiedlichen Klienten (mit potentiell nicht synchronisierten Uhren) geschriebenen

[1] Die Templates jedes regulären Klienten verlangen den Wert `false` im U-UNPROC-Feld eines zu konsumierenden Tupels.

Feld	Typ	Beschreibung
A-TYPE	String	Typ des Tupels
A-PID	URI	Identifikator des Prozessmodells
A-PCONF	UID	Identifikator einer Prozesskonfiguration, z. B. eine UUID
A-PIID	UID	Identifikator einer Instanz einer Konfiguration

Tabelle 5.2: Anwendungsspezifische Felder zur Ausführung von EWFN-Modellen auf der PS-Infrastruktur.

Tupeln, in Relation zur selben PS-Server-Zeit erfasst. Verwendung findet das Feld U-TIMESTAMP beispielsweise für die Bestimmung der Reihenfolge der Beendigung von SCOPE-Aktivitäten zur Realisierung der *Default Compensation Order* (vgl. Abschnitt 5.6.3.2). Welches Feld eines geschriebenen Tupels ein PS-Server in der oben beschriebenen Art bearbeiten soll, kann ein Klient durch den Operationskontext einer PS-Operation bestimmen (vgl. Abschnitt 5.3.1).

Weitere mögliche universelle Felder sind solche, die Informationen bereitstellen, die die Signatur und Verschlüsselung einzelner Tupelfelder erlauben, um die in Abschnitt 5.1.2 formulierten nicht-funktionalen Anforderungen nach Nachrichtenintegrität und -vertraulichkeit umzusetzen. Beispiele hierfür sind die Felder *Partition Field* und *Asymmetric Partition Field* in *SecSpaces* [GLZ06].

5.6.1.2 Anwendungsspezifische Felder zur Ausführung von EWFN-Modellen

Ergänzend zu den, im vorigen Abschnitt erläuterten, universellen Tupelfeldern, zeigt Abbildung 5.2 die sogenannten *anwendungsspezifischen* Tupelfelder, die all jene Informationen beinhalten, die für die Ausführung von EWFN-Modellen benötigt werden.

Das Feld A-TYPE erlaubt eine anwendungsspezifische Typisierung von Tupeln. Für die Ausführung von BPEL-EWFN-Modellen werden die Typen `ControlFlow` und `InstanceData` unterschieden. Das Feld A-PID identifiziert das BPEL-Prozessmodell, welches als Eingabe für die BPEL-EWFN-Transformation verwendet wurde, durch eine URI-Repräsentation von dessen *QName*. Der Wert bestimmt sich aus dem Namensraum des Prozesses, verbunden mit einem #

Feld	Typ	Beschreibung
C-CFType	String	Typ des Kontrollflusstupels
X-AId	UID	Kontrollflusskantenidentifikator
X-AIId	UID	Kontrollflusskanteninstanzidentifikator
C-SContext	List(X-SId: UID, X-SIId: UID)	Scope-Kontext
C-IScopeId	UID	*Isolated Scope*-Kontext

Tabelle 5.3: Felder für Tupel mit A-Type = `ControlFlow`.

und seinem lokalen Namen. Um mehrere parallel in Ausführung befindliche Konfigurationen desselben Prozessmodells bei Gewährleistung der Nachvollziehbarkeit jeder einzelnen Konfiguration zu erlauben, identifiziert das Feld A-PConf die jeweilige Konfiguration eines Prozesses durch einen Bezeichner, der in Verbindung mit A-PId eindeutig ist. Generell können für die Bestimmung eines Bezeichners vom Typ UID (hier und im weiteren Verlauf der Arbeit) unterschiedliche Verfahren zum Einsatz kommen; in der prototypischen Umsetzung der PS-Infrastruktur werden für sämtliche Felder vom Typ UID global eindeutige Identifikatoren in Form von *Universally Unique Identifiers (UUID)* [LMS05] verwendet.

Die Zuordnung eines Tupels zu einer bestimmten Instanz einer Konfiguration eines Prozesses erfolgt durch das Feld A-PIId. Hinsichtlich der Eindeutigkeit der Identifikatoren muss gelten, dass Kombination aus A-PId, A-PConf und A-PIId jedes Tupel, das Teil der Ausführung eines EWFN ist, eindeutig einer Instanz einer bestimmten Prozesskonfiguration zuordnet.

5.6.1.3 Felder für Kontrollflusstupel

Abbildung 5.3 zeigt die für die Repräsentation von Kontrollflussinformationen, in Form der sogenannten *Kontrollflusstupel* (A-Type = `ControlFlow`), benötigten Felder; diese ergänzen die in Abschnitt 5.6.1.2 aufgeführten anwendungsspezifischen Felder zur Ausführung von EWFN-Modellen.

Ein Kontrollflusstupel kann entweder positiven (`CF`), negativen (`CFE`, mnemonisch “exceptional”) oder toten (`DP`) Kontrollfluss beschreiben. Der Kontroll-

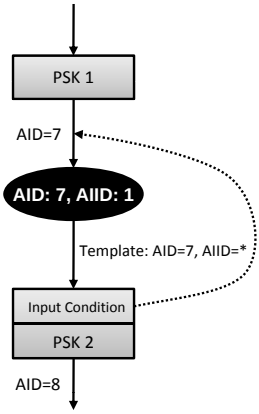


Abbildung 5.11: Sequentielle Weitergabe der Kontrolle über die Prozessausführung.

flusstyp eines Tupels wird durch den Wert des Felds C-CFTYPE beschrieben. Die unterschiedlichen Tupeltypen finden zur Realisierung der *Dead Path Elimination* oder des *Fault- und Compensation-Handling* (vgl. Abschnitt 5.6.3) Verwendung.

Für die Realisierung der dezentralen Ausführung von EWFN-Modellen ist – neben Möglichkeiten zur Identifikation von Modell, Konfiguration und Instanz des ausgeführten Prozesses – auch die Bereitstellung geeigneter Mittel zur Abbildung der durch das Prozessmodell definierten Ausführungsordnung der Prozessaktivitäten (bzw. den entsprechenden PS-Klienten) erforderlich. Abbildung 5.11 zeigt einen Ausschnitt eines EWFN in Form einer Interaktion zwischen zwei PS-Klienten $PSK1$ und $PSK2$. Bei der Transformation eines BPEL-Prozesses in dessen EWFN-Repräsentation erhalten sämtliche Kanten des EWFN einen innerhalb des jeweilige Prozessmodells eindeutigen Kantenidentifikator (engl. *arc identifier*) X-AID [Mar10]. Jede Kante eines EWFN repräsentiert dabei eine zur Navigation des Modells ausgeführte PS-Operation. Die Identifikatoren dieser Kanten sind Teil der entlang der jeweiligen Kanten kommunizierten Tupel im Feld X-AID und werden zur Beschreibung der Eingangsbedingungen der PS-Klienten verwendet. Die Schreiboperation, mit

der $PSK1$ die Beendigung seiner Ausführung an $PSK2$ signalisiert, erfolgt aus Sicht von $PSK1$ entlang der EWFN-Kante 7. Dies kommuniziert der PS-Klient durch den Wert 7 im X-AID-Feld eines Kontrollflusstupels und publiziert dieses in den von $PSK2$ überwachten PS[1].

Zur Koordination der Ausführung von $PSK1$ und $PSK2$ verwendet $PSK2$ den Wert 7 im Feld X-AID des Templates seiner Eingangsbedingung. Die Information, welcher Wert für X-AID zu verwenden ist, wird anhand des EWFN-Modells bestimmt und $PSK2$ während des Deployment-Vorgangs (Abschnitt 5.8) übermittelt. $PSK2$ verfährt analog zu $PSK1$ hinsichtlich der Erzeugung seiner Ausgangseffekte.

Um auch die mehrfache Kommunikation von Kontrollflusstupeln entlang derselben bestimmten Kante erfassen zu können – notwendig beispielsweise für die Ausführung von Schleifen (Abbildung 5.13) – identifiziert X-AIID eine Instanz einer Kante. Die Kombination aus X-AID und X-AIID identifiziert jedes entlang einer bestimmten EWFN-Kante kommunizierte Tupel eindeutig. Da derselbe PS-Klient stets sämtliche Tupel konsumiert, die entlang einer bestimmten Kante produziert werden, findet X-AIID in den Templates der Klienten im Allgemeinen keine Verwendung und wird mit dem Wildcard-Symbol ("$\star$") belegt.

Ist die Ausführung einer Aktivität durch die Beendigung der Ausführung mehrerer vorangehender Aktivitäten bedingt – realisiert diese also einen *Join* oder *Synchronizing Merge* [RHAM06] – so wird ebenfalls entsprechend der oben erläuterten Vorgehensweise verfahren (Abbildung 5.12). In diesem Fall gilt, dass die Identifikatoren aller eingehenden Kontrollflusskanten die Templates der zur Erfüllung der Eingangsbedingung ausgeführten PS-Operationen von $PSK3$ bestimmen. Für das Zusammenführen mehrerer Ausführungspfade einer Prozessinstanz gilt weiterhin die Forderung der Gleichheit der (in Abbildung 5.12 aus Übersichtlichkeitsgründen nicht dargestellten) Identifikatoren einer Instanz einer bestimmten Prozesskonfiguration A-PID, A-PCONF sowie

[1] Zur Vereinfachung der Darstellung wurde auf die Einzeichnung der zur Kommunikation der Tupel verwendeten PS verzichtet. Da diese auf die folgende Erläuterung keinen Einfluss haben, gilt für das abgebildete Beispiel – ohne Beschränkung der Allgemeinheit – eine Kommunikation aller Tupel über denselben PS.

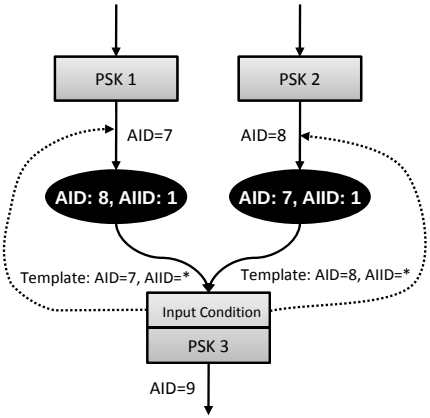


Abbildung 5.12: Zusammenführung mehrerer Ausführungspfade in einem Prozess.

A-PIID erforderlich. Realisiert wird diese Gleichheit durch die Join-Variablen der *sync*-Operation [MWL08c].

Jeder PS-Klient, der eine SCOPE-Aktivität implementiert (d. h. auch jeder implizite SCOPE) wird durch einen SCOPE-Identifikator (X-SID) identifiziert. Die Bestimmung des Werts für X-SID erfolgt statisch zum Zeitpunkt der BPEL-EWFN-Transformation und ist eindeutig innerhalb des jeweiligen Prozessmodells. Unter bestimmten Bedingungen ist diese statische Vorgabe für X-SID nicht ausreichend für eine vollständige Bestimmung der Instanz eines SCOPE zur Ausführungszeit. So besteht beispielsweise die Möglichkeit, dass während der Ausführung einer Iterationsaktivität derselbe SCOPE mehrfach durchlaufen wird. Um dennoch eine eindeutige Identifikation der unterschiedlichen Instanzen eines SCOPE zu erlauben, werden diese durch das anwendungsspezifische Feld X-SIID beschrieben; die Kombination aus dem statisch bestimmten Wert für X-SID und dem zur Laufzeit bestimmten Wert für X-SIID ist eindeutig für jede Instanz eines SCOPE während der Ausführung einer Prozessinstanz. Erzeugt wird der X-SIID-Wert eines SCOPE durch den PS-Klienten, der die Anwendungslogik der jeweiligen SCOPE-Aktivität implementiert. Um einen Instanzdatenzugriff

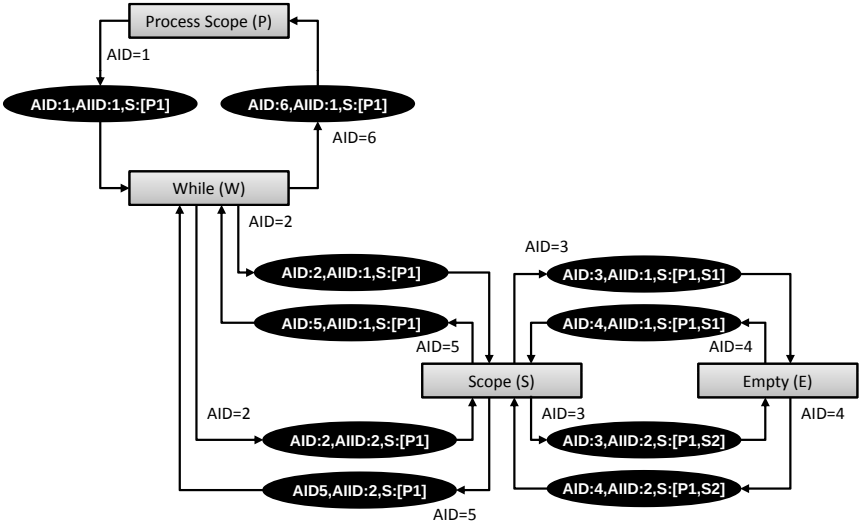


Abbildung 5.13: Notwendigkeit der Repräsentation hierarchisch verschachtelter SCOPE-Informationen in kommunizierten Tupeln (basierend auf [Var09]).

(vgl. Abschnitt 5.6.2) zu erlauben, der der SCOPE-Hierarchie des jeweiligen Prozessmodells Rechnung trägt, wird der SCOPE-Kontext einer Aktivität durch das anwendungsspezifische Feld C-SCONTEXT, welches eine Liste von X-SIID- und X-SIID-Wertpaaren beinhaltet, zwischen PS-Klienten kommuniziert.

Abbildung 5.13 verdeutlicht die Anwendung des C-SCONTEXT-Felds (im dargestellten Beispiel kurz: S) anhand eines Beispiels. Die Abbildung zeigt den sogenannten *Root Scope P* eines Prozesses. Diesem untergeordnet ist eine WHILE-Schleife, deren Kind-Aktivität ein SCOPE S ist, welcher wiederum eine EMPTY-Aktivität als Kind hat. Für die Bedingung der WHILE-Schleife gelte im Beispiel eine zweifache Ausführung des Schleifenrumpfs. Die Navigation des dargestellten EWFN verläuft wie folgt.

Während seiner Ausführung erzeugt PS-Klient P ein Tupel $(\text{AID} : 1, \text{AIID} : 1, S : [(P1)])$. Da die PROCESS-Aktivität einen impliziten SCOPE darstellt, bein-

haltet das Tupel (neben den Identifikatoren von Absender und Empfänger) den zusammengefassten symbolischen Wert ($P1$) im Feld C-SCONTEXT, welcher den SCOPE und die SCOPE-Instanz des Absenders identifiziert.

Dieses Tupel hat die Ausführung von W zur Folge, welche wiederum die erste Ausführung von S durch das Tupel $(\text{AID} : 2, \text{AIID} : 1, S : [(P1)])$ auslöst. Da S einen SCOPE implementiert, fügt S in das Feld C-SCONTEXT des an E kommunizierten Tupels $(\text{AID} : 3, \text{AIID} : 1, S : [(P1), (S1)])$ den Wert $(S1)$ an, wobei S die SCOPE-Aktivität und 1 deren Instanz identifiziert. PS-Klient E kommuniziert die Beendigung seiner Ausführung in Schritt 4 an S zurück. Da damit die Ausführung der ersten Iteration von S ebenfalls abgeschlossen ist (E ist die einzige Aktivität in S), entfernt S den Wert $(S1)$ aus dem Feld C-SCONTEXT und erzeugt ein entsprechendes Tupel, welches die Ausführung der nächsten Iteration von W zur Folge hat.

Die zweite Ausführung von S, ausgelöst durch Tupel $(\text{AID} : 2, \text{AIID} : 2, S : [(P1)])$, führt zur Vergabe einer neuen X-SIID mit dem Wert 2, welche in Form von $[(P1), (S2)]$ als Wert des Felds C-SCONTEXT E kommuniziert wird. Zu bemerken ist an dieser Stelle, dass die X-AID-Werte der zwischen W und S ausgetauschten Tupel in der ersten und der zweiten Iteration von W identisch sind. In jeder Iteration wird allerdings ein neuer X-AIID-Wert vergeben. Die folgenden Schritte verlaufen entsprechend der ersten Iteration der WHILE-Schleife.

Führt ein PS-Klient eine Aktivität in einem *Isolated Scope* aus, so beinhaltet das Feld C-ISCOPEID die Kombination aus X-SID und X-SIID einer isolierten SCOPE-Aktivität. Der C-ISCOPEID-Identifikator wird zur Laufzeit einer Prozessinstanz von dem PS-Klienten vergeben, der die jeweilige SCOPE-Aktivität implementiert, und mit den Kontrollflusstupeln zwischen den einzelnen PS-Klienten kommuniziert. Er kommt zur Realisierung von Variablenzugriffen innerhalb einer isolierten SCOPE-Aktivität zum Einsatz. Wird die Ausführung einer SCOPE-Aktivität selbst durch ein `CF`-Tupel ausgelöst, welches innerhalb einer isolierten SCOPE-Aktivität erzeugt wurde (und welches demnach einen Wert im Feld C-ISCOPEID trägt), so erzeugt dieser selbst keinen neuen C-ISCOPEID-Identifikator, sondern reicht den existierenden Identifikator an seine Kind-Aktivitäten weiter. In diesem Fall findet eine sogenannte "Infizierung" [Org07] eines SCOPE mit

der *isolated*-Eigenschaft statt. Entsprechend der BPEL-Spezifikation gilt, dass eine isolierte SCOPE-Aktivität keine weiteren SCOPE-Aktivitäten beinhalten darf, welche selbst die Eigenschaft *isolated* haben (vgl. SA00091 in [Org07]).

Im Hinblick auf *Isolated Scopes* verlangt die BPEL-Spezifikation weiterhin, dass die *Link Status* für SCOPE-übergreifende Synchronisationskanten einer isolierten SCOPE-Aktivität erst bei Beenden des SCOPE für SCOPE-externe Aktivitäten sichtbar werden. Realisiert wird diese Eigenschaft durch Transition $t3$ des *Link Source*-EWFN [Mar10]. Handelt es sich (i) um einen SCOPE-übergreifenden Link und ist dieser weiterhin (ii) Teil eines *Isolated Scope*, so prüft $t3$ auf den erfolgreichen Abschluss der Ausführung des *Isolated Scope*, dem die jeweilige Aktivität zugeordnet ist, durch Zugriff auf dessen *Scope State*-Tupel. Gilt eine der oben genannten Bedingungen nicht, so ist dieses Verhalten nicht erforderlich.

Abbildung 5.15 in Abschnitt 5.6.2 verdeutlicht den Instanzdatenzugriff in isolierten SCOPE-Aktivitäten anhand eines Beispiels.

5.6.2 Instanzdatenzugriff

Während der Ausführung seiner Funktionalität kann ein PS-Klient auf Instanzdaten des Prozesses entweder lesend oder modifizierend zugreifen. Aufgrund des der Arbeit zugrunde liegenden Szenarios der Ausführung von Produktionsprozessen gilt die Annahme, dass potentiell große Instanzdaten während der Prozessausführung manipuliert werden müssen. Da im Fall einer verteilten Ausführung ein Instanzdatenzugriff potentiell auch ein Netzwerkzugriff auf einen entfernten PS-Server (d. h. einen PS-Server, der nicht Teil der Domäne des aufrufenden PS-Klienten ist) zur Folge haben kann, ist die Minimierung der (partitionsübergreifend) zu kommunizierenden Datenmenge ein essentielles Ziel der PS-Infrastruktur.

Gegenstand dieses Abschnitts ist die Beschreibung der Struktur der Instanzdaten-Tupel, die Realisierung eines isolierten Instanzdatenzugriffs für die Verarbeitung von *Isolated Scopes*, der Lebenszyklus von Instanzdaten und verschiedene Mechanismen, um einen effizienten Instanzdatenzugriff zur Laufzeit einer Prozessinstanz zu erlauben.

Feld	Typ	Beschreibung
I-ID	UID	Identifikator des Instanzdatums
X-SID	UID	SCOPE-Identifikator
X-SIID	UID	SCOPE-Instanz-Identifikator

Tabelle 5.4: Anwendungsspezifische Felder für Instanzdatentupel (A-TYPE = `InstanceData`).

5.6.2.1 Aufbau der Instanzdatentupel

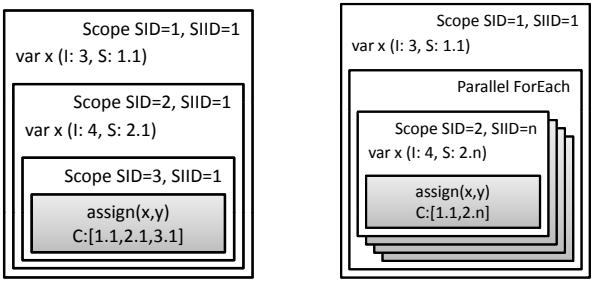
Der Aufbau der Instanzdatentupel umfasst sowohl die universellen und die generischen Felder für die EWFN-Ausführung, als auch die in Abbildung 5.4 aufgeführten instanzdatenspezifischen Felder.

Das Feld I-ID identifiziert ein Instanzdatum in einem Prozessmodell. Der Wert I-ID-Wert jedes Instanzdatums wird zum Zeitpunkt der BPEL-EWFN-Transformation statisch bestimmt.

Zur Berücksichtigung von Sichtbarkeitsregeln innerhalb eines SCOPE identifiziert die Kombination der Felder X-SID und X-SIID die SCOPE-Instanz, der ein bestimmtes Instanzdatum zugeordnet ist; für sie gelten die Abschnitt 5.6.1.3) für das Feld C-SCONTEXT erläuterten Eigenschaften.

Abbildung 5.14 verdeutlicht die Notwendigkeit der Felder X-SID und X-SIID sowie deren Verwendung in Verbindung mit dem Feld C-SCONTEXT eines Kontrollflusstupels für den Zugriff auf ein Instanzdatum am Beispiel einer Variable; für andere Instanzdaten, die den *Scoping*-Regeln von BPEL unterliegen, gilt dieses Verfahren ebenso.

Abbildung 5.14(a) zeigt drei hierarchisch ineinander geschachtelte SCOPE-Aktivitäten sowie eine ASSIGN-Aktivität als Kind des SCOPE mit X-SID $= 3$ (kurz: SID) und X-SIID $= 1$ (kurz: SIID). Die beiden umgebenden SCOPE-Aktivitäten deklarieren jeweils die Variable x, die entsprechend obiger Erläuterung zum Zeitpunkt der BPEL-EWFN-Transformation beide einen jeweils eindeutigen Wert für I-ID (kurz: I) erhalten. Durch die, im EWFN-Modell des Prozesses spezifizierten Informationen, die der PS-Klient während seiner Konfiguration in Form eines Teils des DDD erhalten hat (vgl. Abschnitt 5.2.1 und 5.8.1) kennt



(a) Zur Transformationszeit bekannte Anzahl von SCOPE-Instanzen.

(b) Zur Transformationszeit unbekannte Anzahl von SCOPE-Instanzen .

Abbildung 5.14: Variablenzugriff bei geschachtelten SCOPE-Aktivitäten (basierend auf [Var09]).

dieser sowohl den Wert I-ID der Variable als auch die URL des PS, in welchem x abgelegt ist. Er verfügt damit über alle Informationen, die in diesem Beispiel für die Identifikation und den Zugriff auf das Instanzdatentupel der im SCOPE mit X-SID $= 2$ deklarierten Variable x mit dem Wert I-ID $= 4$ notwendig sind.

Abbildung 5.14(b) zeigt ein Abbildung 5.13 ähnliches Szenario, in welchem einer PARALLEL FOREACH-Schleife ein (expliziter) SCOPE einbeschrieben ist. Die Anzahl der Laufzeitinstanzen dieser SCOPE-Aktivität ist zum Zeitpunkt der BPEL-EWFN-Transformation im Allgemeinen nicht bekannt. Demzufolge wird durch die BPEL-EWFN-Transformation für die Variable x auch nur eine I-ID $= 4$ vergeben, die für sämtliche Instanzen der Variable gilt. Die Sicherstellung, dass die ASSIGN-Aktivität in Iteration n des SCOPE auch auf die in diesem deklarierte Instanz der Variable zugreift und nicht auf die in einer parallelen SCOPE-Instanz m mit $m \neq n$ deklarierte Variable, erfolgt unter Verwendung des letzten Eintrags des Felds C-SCONTEXT (kurz: C) des Kontrollflusstupels, welches die

Ausführung der Aktivität ausgelöst hat[1]. Bei Ausführung der ASSIGN-Aktivität in dem SCOPE mit X-SID $= 2$ und X-SIID $= n$ ist somit ein Instanzdatentupel gesucht, für das gilt: I-ID $= 4$, X-SID $= 2$ und X-SIID $= n$. Da alle Instanzdatentupel mit demselben I-ID-Wert im selben PS abgelegt werden, erfolgt das Konsumieren der Variable x durch die ASSIGN-Aktivität sämtlicher Iterationen der Schleife aus demselben PS.

5.6.2.2 Realisierung von isoliertem Instanzdatenzugriff

Die korrekte Verarbeitung sogenannter *Isolated Scopes* [Org07] verlangt eine Isolation des Zugriffs auf Instanzdaten, auf welche aus einem *Isolated Scope* heraus zugegriffen wird. Die BPEL-Spezifikation schreibt in diesem Fall eine Serialisierung des Variablenzugriffs vor. Es muss also gelten, dass sämtliche Zugriffe von mehreren Klienten gemeinsam genutzten Instanzdaten in einer Weise erfolgen, die in ihrem Ergebnis dem einer seriellen Ausführung der *Isolated Scopes* entspricht. Die Sicherstellung dieser Serialisierbarkeit erfolgt durch ein sogenanntes *Mutex*-Tupel für die jeweilige Variable. Abbildung 5.15 verdeutlicht den Instanzdatenzugriff in *Isolated Scopes* graphisch anhand des Beispiels eines Variablenzugriffs auf die Variable V. Aus Gründen der Übersichtlichkeit wurde in der Abbildung auf eine Darstellung der Erzeugung der Variablen- und *Mutex*-Tupel verzichtet; diese erfolgt während der Ausführung des PS-Klienten, der die SCOPE-Aktivität implementiert, in welcher das jeweilige Instanzdatum deklariert wurde.

Für das dargestellte Beispiel gelte ohne Beschränkung der Allgemeinheit, dass A_1 in S_2 seine Ausführung vor A_3 in S_4 beginnt. Ausgelöst wird die Ausführung von A_1 durch ein entsprechendes von S_2 erzeugtes Kontrollflusstupel. Entsprechend der Beschreibung in Abschnitt 5.6.1.3 wird durch das Feld C-ISCOPEID (kurz: I) von S_2 an seine Kind-Aktivitäten kommuniziert, dass eine isolierte Ausführung im Kontext von S_2 zu erfolgen hat. Zur Erreichung eines

[1] Im Fall sogenannter *scope crossing links*, d. h. LINK-Elemente, die die SCOPE-Grenzen kreuzen, kann es vorkommen, dass die C-SCONTEXT-Listen der eingehenden Kontrollflusstupel unterschiedliche Länge haben (d. h. beispielsweise der LINK seine Quelle in einem übergeordneten SCOPE hat). In diesem Fall gilt, dass das C-SCONTEXT-Feld größerer Länge für die Bestimmung des SCOPE-Kontexts herangezogen wird.

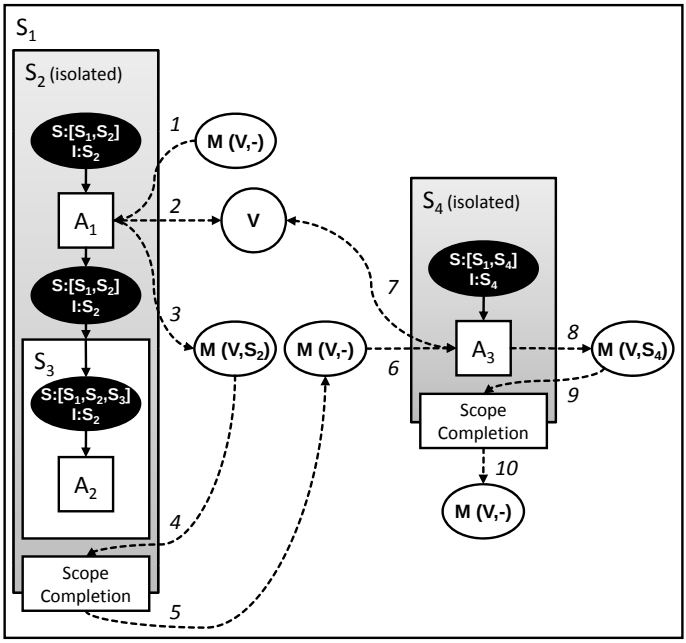


Abbildung 5.15: Instanzdatenzugriff in *Isolated Scopes*.

gegenseitigen Ausschlusses paralleler Zugriffe auf Instanzdaten gilt in diesem Fall, dass jede Aktivität, deren Anwendungslogik einen Instanzdatenzugriff beinhaltet, zunächst das dem Instanzdatum entsprechende Mutex-Tupel destruktiv konsumiert. Dabei gilt weiterhin, dass jeder PS-Klient lediglich das sogenannte *freie* Mutex-Tupel – im Beispiel dargestellt durch $M(V,-)$ – oder das *gebundene* Mutex-Tupel seines umgebenden *Isolated Scope* – im Beispiel $M(V,S_2)$ – konsumiert; niemals allerdings ein an einen anderen *Isolated Scope* gebundenes Tupel. Beim ersten Zugriff auf eine Variable in einem *Isolated Scope* konsumiert der PS-Klient deren freies Mutex-Tupel (1) und hat damit das Zugriffsrecht auf das jeweilige Instanzdatum erworben (2). Um dieses Zugriffsrecht an weitere Aktivitäten im *Isolated Scope* zu kommunizieren, bindet der jeweilige PS-Klient das Mutex-Tupel an seinen SCOPE und macht dieses

den anderen PS-Klienten (durch Publikation in denselben PS, in welchem sich auch das entsprechende Instanzdatentupel befindet) zugänglich (3). Die Beendigung der Ausführung einer isolierten SCOPE-Aktivität konsumiert sämtliche durch den SCOPE gebundenen Mutex-Tupel (4), erzeugt entsprechende freie Mutex-Tupel und gibt damit das jeweilige Instanzdatum für die Verarbeitung in einem anderen, parallelen *Isolated Scope* frei (5).

Die Verarbeitung von S_4 erfolgt nach Freigabe der gebundenen Mutex-Tupel durch S_2 analog durch die Schritte: Konsumieren des freien Mutex-Tupels (6), Instanzdatenzugriff (7), Binden des Mutex-Tupels (8) und Freigabe des Mutex-Tupels (9, 10).

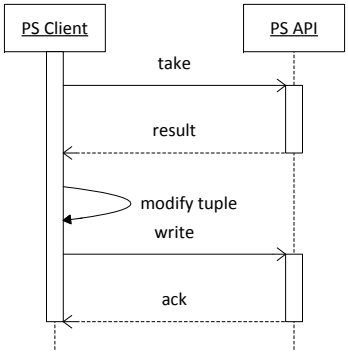
5.6.2.3 Lebenszyklus von Instanzdaten

Wie im vorangehenden Abschnitt erwähnt, sind Instanzdaten in BPEL einem SCOPE (bzw. einer Instanz eines SCOPE) zugeordnet. Da allerdings nicht unbedingt gilt, dass jede SCOPE-Aktivität eines Prozesses auch tatsächlich in jeder Instanz ausgeführt wird, gilt für die Ausführung in der PS-Infrastruktur, dass einem Instanzdatum zwar zum Zeitpunkt der BPEL-EWFN-Transformation ein eindeutiger Wert im Bezeichner I-ID sowie zum Partitionierungszeitpunkt eine entsprechende Partition zugewiesen wird, das jeweilige Instanzdatentupel allerdings erst zum Ausführungszeitpunkt erzeugt wird. Konkret geschieht dies während der Initialisierung eines SCOPE durch die SCOPE-Aktivität, nachdem diese ihren X-SIID-Wert bestimmt hat.

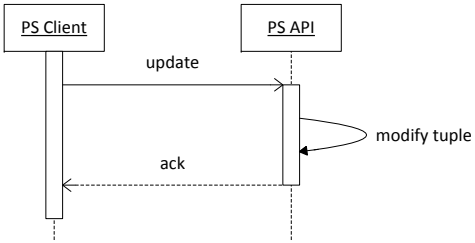
Instanzdaten verbleiben während der gesamten Ausführungszeit in dem ihnen zugewiesenen PS. Sie werden entweder von einem Klienten (nichtdestruktiv) konsumiert oder durch die *update*-Operation im jeweiligen PS modifiziert.

Auch nach der erfolgreichen Ausführung des SCOPE dem ein Instanzdatum zugeordnet ist, verbleibt dieses als Grundlage für die sogenannte *Compensation* im jeweiligen PS. Die Gesamtheit sämtlicher impliziter und expliziter Instanzdaten eines SCOPE zum Zeitpunkt der Beendigung seiner Ausführung bildet dessen sogenannten *Snapshot* [Org07].

Entfernt werden die Instanzdaten einer Prozessinstanz erst zum Zeitpunkt



(a) Aktualisierung des Werts eines Tupels durch Konsumieren, Modifizieren und Schreiben.



(b) Aktualisierung des Werts eines Tupels mittels *In-Place Update*.

Abbildung 5.16: Modifikation von Tupelwerten

der Beendigung ihrer Ausführung.

5.6.2.4 Verbesserung von Instanzdatenzugriffen durch die *update*-Operation

Wie in Abschnitt 5.3 erläutert, erlaubt die *update*-Operation die Aktualisierung des Werts eines in einem PS befindlichen Tupels direkt auf der Seite des PS-Servers. Abbildung 5.16 verdeutlicht den Unterschied einer Änderung eines Tupels mit den Operationen der Linda-Schnittstelle einerseits (Abbildung 5.16(a)) und unter Verwendung der *update*-Operation andererseits (Abbildung 5.16(a)). Die Realisierung durch Konsumieren, klientenseitigem Modifizieren und Rückschreiben des Tupels in den PS bedingt nicht nur vier kommunizierte Nachrichten zwischen PS-Klient und PS-Server, sondern darüber hinaus die Übermittlung des Tupelwerts jeweils vor und nach der Modifikation.

Die *update*-Operation verlangt die Angabe dreier Parameter: (i) eine PS-URL, (ii) ein Template, das das zu modifizierende Tupel beschreibt und (iii) ein Tupel, durch das ein zum Template konformes Tupel ersetzt werden soll. Es gilt dabei, dass *update* bei jedem Aufruf nur maximal ein Tupel im jeweiligen PS ersetzt. Sollen die Werte einzelner Felder des modifizierten Tupels in ihrer

bestehenden Form beibehalten und nicht modifiziert werden, so besteht die Möglichkeit, diese durch einen bestimmten Feld-Typ im Template entsprechend zu markieren.

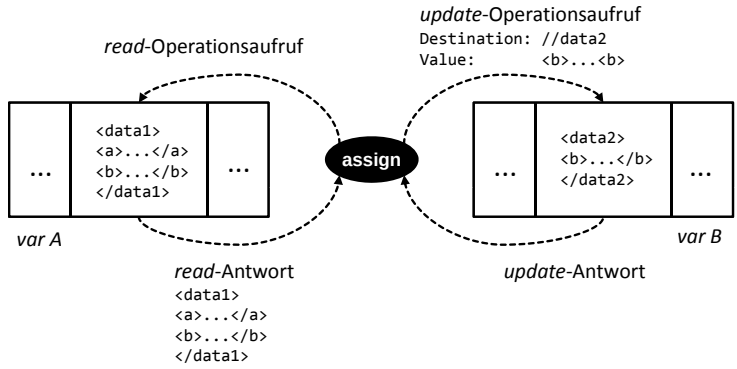
In der Ausführung von BPEL-Prozessen findet die *update*-Operation, in Verbindung mit der serverseitigen Evaluierung von XPath-Ausdrücken, welche im nachfolgenden Abschnitt erläutert wird, beispielsweise für die selektive Aktualisierung von Teilen eines XML-Baums (z. B. einem *Part* einer Variable) als Teil einer ASSIGN-Aktivität, Verwendung.

5.6.2.5 Verbesserung von Instanzdatenzugriffen mittels XML-Template-Matching

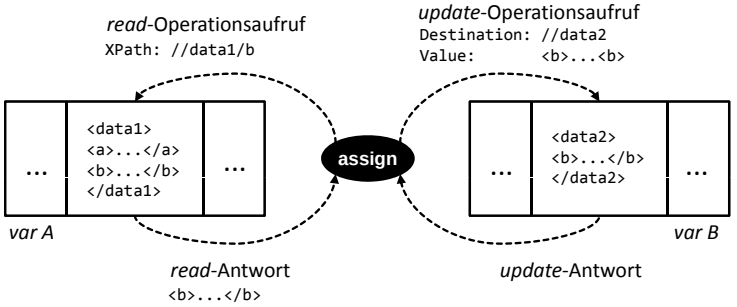
Obwohl die BPEL-Spezifikation mittels ihrer Erweiterungsmechanismen die Verarbeitung beliebig typisierter Daten erlaubt, ist XML das typischerweise verwendete Format für Instanzvariablen; seine Unterstützung ist durch die BPEL-Spezifikation für jedes BPEL-WfMS gefordert.

Als Folge der Zuordnung eines Instanzdatums zu genau einer Partition (vgl. Abschnitt 3.6) kommt es während der dezentralen Prozessausführung zu domänenübergreifenden Instanzdatenzugriffen. Da eine Minimierung des domänenübergreifend zu kommunizierenden Datenvolumens zu einer Verbesserung des Laufzeitverhaltens der Prozessausführung führt, unterstützen die konsumierenden Operationen der PS-Schnittstelle die Angabe von *XPath*-Ausdrücken [CD+99] in den Feldern ihrer Templates. Hinsichtlich der Semantik konsumierender PS-Operation mit einem XPath-Feld im Template gilt, dass das Resultat des Operationsaufrufs ein Tupel ist, in welchem der Wert des entsprechenden Feldes im konsumierten Tupel durch das Resultat der Anwendung des XPath-Ausdrucks ersetzt wurde. Dies bedeutet, dass das vom PS-Server an den PS-Klienten zurückgegebene Tupel nicht dem im PS abgelegten Tupel entspricht, sondern bereits auf Seite des PS-Servers vorverarbeitet wird.

Abbildung 5.17 veranschaulicht dies anhand eines Beispiels. Die Abbildung zeigt eine ASSIGN-Aktivität, welche einen Teilbaum des in der Variablen *A* enthaltenen XML-Dokuments, spezifiziert durch den XPath-Ausdruck `//data1/b`, der Variablen *B* zuweist. Abbildung 5.17(a) zeigt dies unter Verwendung der



(a) Konsumieren des gesamten Datums.



(b) Konsumieren des relevanten Teils des Datums.

Abbildung 5.17: Reduzierung konsumierter Daten durch serverseitige Evaluierung von XPath-Ausdrücken.

"regulären" PS-Operationen *read* und *update*. In diesem Fall, wird zunächst das gesamte Variablentupel A vom ASSIGN-Klienten konsumiert, dann auf Seite des Klienten der relevante Teil des XML-Dokuments durch Evaluierung des XPath-Ausdrucks extrahiert, ein entsprechendes Variablentupel erzeugt und mit diesem Wert die *update*-Operation für das Variablentupel B aufgerufen. Offensichtlich gilt in diesem Fall, dass die durch die *read*-Operation konsumierte Datenmenge ($|a| + |b|$) unnötigerweise größer ist, als die durch die *update*-Operation geschriebene Datenmenge ($|b|$). Abbildung 5.17(b) zeigt

dasselbe Szenario unter Verwendung der beschriebenen serverseitigen Evaluierung von XPath-Ausdrücken. Der XPath-Ausdruck wird als Teil des *read*-Operationsaufrufs vom ASSIGN-Klienten an den jeweiligen PS-Server übermittelt. Als Resultat des Operationsaufrufs wird vom Server das Variablentupel A zurückgeliefert, wobei der Wert des Felds, dessen korrespondierendes Template-Feld den XPath-Ausdruck beinhaltete, durch das Resultat der Evaluierung des Ausdrucks auf Seite des PS-Servers ersetzt wird. Durch das selektive Konsumieren der tatsächlich für die Ausführung der ASSIGN-Aktivität benötigten Daten reduziert sich das zu lesende Datenvolumen somit auf $|b|$.

5.6.2.6 Verbesserung von Instanzdatenzugriffen durch Caching-Mechanismen

In Szenarien, in denen durch denselben PS-Klienten wiederholt konsumierend auf den Wert eines bestimmten Instanzdatums zugegriffen wird, kann das übertragene Datenvolumen durch die Anwendung von Caching-Mechanismen auf Seite der PS-Klienten reduziert werden. Ein Szenario, an welchem sich die durch Caching-Mechanismen erzielbare Verbesserung verdeutlichen lässt, ist beispielsweise der mehrfache lesende Zugriff eines Klienten auf eine Variable, welche einer fremden Domäne zugeordnet ist und einen umfangreichen Nutzdatenanteil aufweist, innerhalb einer Iteration. Gilt weiterhin, dass sich der Wert der Variablen über den Zeitraum der mehrfachen Aufrufe hinweg nicht ändert, so ist die mehrfache Übermittlung des Werts des Variablentupels nicht notwendig; der jeweilige PS-Klient kann den Wert des Instanzdatums lokal zwischenspeichern und spätere konsumierende Operationen desselben Instanzdatums aus seinem lokalen Zwischenspeicher (engl. *Cache*) bedienen.

Da, bedingt durch nebenläufige Sprachkonstrukte wie die FLOW-Aktivität, allerdings die Möglichkeit besteht, dass der Wert des Instanzdatums während des mehrfachen Konsumierens durch einen nebenläufigen Prozessstrang modifiziert wird, ist im Allgemeinen die Überprüfung der sogenannten *Freshness* des im lokalen Speicher enthaltenen Werts des Instanzdatums notwendig; eine Ausnahme hierzu bilden lediglich als konstant gekennzeichnete Daten (vgl. Feld U-CONST in Abschnitt 5.6.1.1).

Abbildung 5.18 zeigt die Vorgehensweise zur Realisierung der Zwischen-

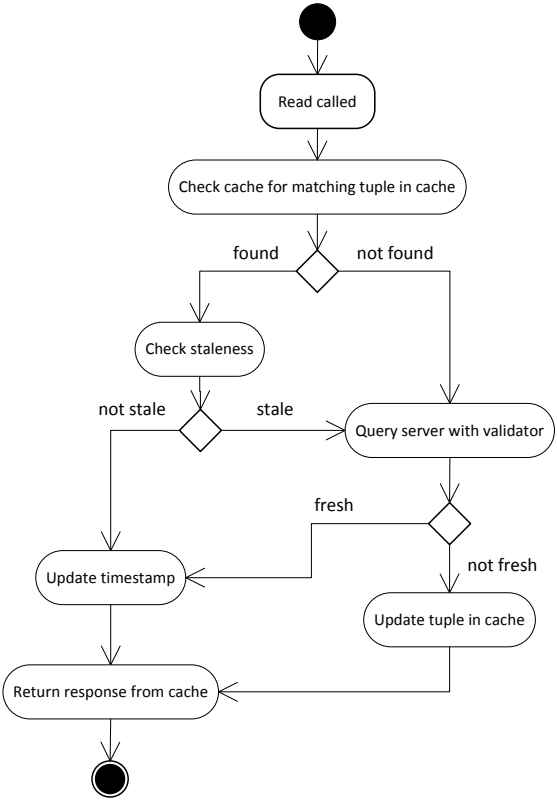


Abbildung 5.18: Realisierung der klientenseitigen Zwischenspeicherung von Tupelwerten am Beispiel der *read*-Operation (Klientensicht).

speicherung von Instanzdaten auf der Seite des PS-Klienten in Form eines UML-Aktivitätsdiagramms.

Beim Aufruf einer *read*-Operation unter Verwendung des Caching-Mechanismus übergibt der PS-Klient als Teil des Operationskontexts einen sogenannten *Validator*. Der Validator erlaubt dem Klienten eine Festlegung, welches Kriterium für die Bestimmung der *Freshness* eines im Cache befindlichen Werts gelten soll. Die Implementierung des Cache nimmt die Anfrage des PS-Klienten

entgegen und prüft lokal, ob ein zwischengespeichertes Tupel vorhanden ist, das zu dem, vom PS-Klient spezifizierten, Template konform ist.

Findet der Cache kein entsprechendes Tupel, so leitet er die Anfrage an den betreffenden PS-Server weiter. Die Ausführung der Operation auf dem PS-Server führt entweder zu einem Tupel, das zum spezifizierten Template konform ist, oder zum leeren Tupel nach Erreichen eines spezifizierten Timeout. Für beide Fälle gilt, dass diese das Kriterium *not fresh* im Sinne des Cache erfüllen. Der Klient übernimmt das vom Server erhaltene Tupel in den lokalen Cache und übergibt dieses an den Klienten als Antwort auf seine ursprüngliche *read*-Anfrage.

Findet der Cache hingegen bei seiner Prüfung ein zum Template des Klienten konformes Tupel, so überprüft er, ob dieses entsprechend des definierten Validators als *stale* zu bezeichnen ist und somit mittels einer Abfrage beim Server sichergestellt werden muss, ob das Tupel noch "aktuell" ist, d. h. beim Server noch in genau dieser Form vorliegt. Ergibt die Überprüfung der Staleness ein negatives Ergebnis, so gilt das Tupel als *fresh* und kann direkt aus dem Cache an den Klienten zurückgegeben werden. Ergibt die Überprüfung der Staleness hingegen, dass eine Überprüfung des Tupels erforderlich ist, so wird die Anfrage des Klienten mit einer Prüfsumme des im Cache befindlichen Tupels an den Server weitergeleitet.

Um der operationalen Semantik von BPEL bei zentraler Ausführung zu entsprechen, gilt, dass für jedes im Cache befindliche Tupel prinzipiell die Staleness-Eigenschaft angenommen wird und somit bei jedem Datenzugriff eine entsprechende Überprüfung stattfinden muss. Dies ist notwendig, da bei jedem Zugriff durch einen PS-Klienten gilt, dass sich der Wert des Instanzdatums seit dem letzten Zugriff, bedingt durch eine nebenläufige Operation, geändert haben kann[1].

Bei Eingang des Operationsaufrufs prüft der Server, ob ein zum Template konformes Tupel im angefragten PS vorliegt. Ist dies nicht der Fall, so verhält

[1] Abschnitt 5.9.2 identifiziert die Einschränkung dieser erzwungenen Freshness-Überprüfung durch Vorgabe eines entsprechenden Validators als eine Möglichkeit, das Laufzeitverhalten von Prozessinstanzen – unter Aufgabe der *zentralen Ausführungssemantik* von BPEL – weiter zu verbessern.

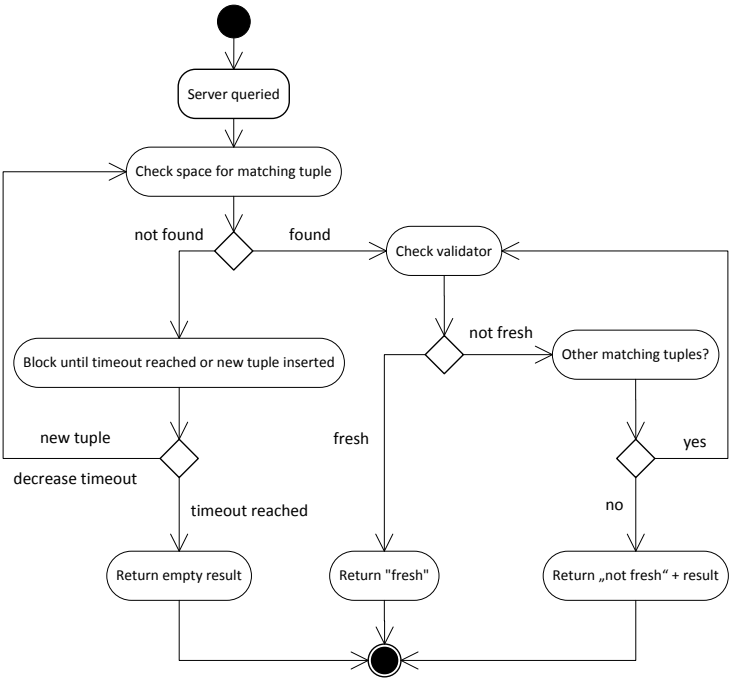


Abbildung 5.19: Realisierung der klientenseitigen Zwischenspeicherung von Tupelwerten am Beispiel der *read*-Operation (Serversicht).

sich der Server entsprechend der Verarbeitung einer regulären *read*-Operation und blockiert so lange, bis entweder der vom Klient spezifizierte Timeout erreicht ist oder bis ein neues Tupel in den PS geschrieben wird, woraufhin die Überprüfung auf Tupel-Verfügbarkeit wiederholt wird (Abbildung 5.19) und der neue Timeout-Wert um die bereits gewartete Zeit vermindert wird.

Findet der Server hingegen ein oder mehrere zum Template konforme Tupel, so bestimmt er für jedes dieser Tupel eine Prüfsumme und vergleicht diese mit der vom Klienten übergebenen Prüfsumme. Hat eines der gefundenen Tupel eine identische Prüfsumme, so übergibt der Server die Antwort *fresh* an den Klienten, ohne das gefundene Tupel als Teil der Antwort zu übergeben.

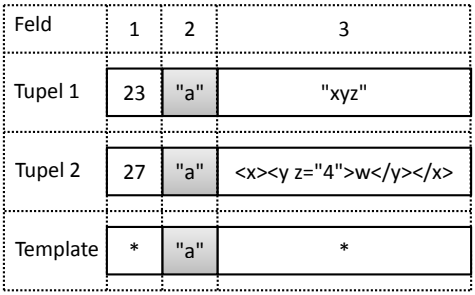


Abbildung 5.20: Beispiel einer *read*-Operation mit Verwendung des klientenseitigen Zwischenspeichers.

Hat keines der gefundenen Tupel eine Prüfsumme identisch der vom Klienten angegebenen, so wählt der Server eines der gefundenen Tupel und übergibt dieses mit der Antwort *not fresh* an den Klienten.

In beiden Fällen aktualisiert der PS-Klient seinen lokalen Cache. Im Fall der Server-Antwort *fresh* wird der letzte Aktualisierungszeitpunkt des Cache-Eintrags des Tupels vermerkt, im Fall der Antwort *not fresh* wird das vom Server übergebene Tupel mit seinem Aktualisierungszeitpunkt in den Cache übernommen.

Zu bemerken ist an dieser Stelle, dass die Einführung von Caching für die *read*-Operation die Folge hat, dass bei wiederholter Ausführung einer *read*-Operation dasselbe Tupel "priorisiert" behandelt wird, d. h. immer dasselbe Tupel auch dann an den Klienten zurückgegeben wird, wenn inzwischen neue, zum Template des Klienten konforme, Tupel in den PS geschrieben wurden.

Abbildung 5.20 verdeutlicht dies anhand eines Beispiels. Angenommen, in einem PS liegt das dargestellte Tupel 1. In diesem Zustand des PS wird eine *read*-Operation mit dem dargestellten Template ausgeführt. Das $\star$-Symbol beschreibt als sogenanntes *Wildcard*-Feld ein Feld eines beliebigen Werts eines beliebigen Typs. Da Tupel 1 zum Template konform ist, wird das Tupel vom PS-Server an den PS-Klient übergeben, welcher dieses in seinen lokalen Cache übernimmt. Angenommen, nach Abschluss dieser *read*-Operation wird Tupel 2, welches

ebenfalls zum angegebenen Template konform ist, in den PS geschrieben. Ruft nun in diesem Fall der PS-Klient erneut eine *read*-Operation mit demselben Template auf, so wird entsprechend obiger Beschreibung vom Cache eine Prüfsumme bestimmt und (im Fall einer angenommenen Staleness des Eintrags im Cache) mit der Anfrage an den Server übergeben. Resultat der Verarbeitung dieser Suchanfrage auf dem PS-Server ist sowohl Tupel 1 als auch Tupel 2. Den nachfolgend ausgeführten Prüfsummenvergleich besteht allerdings lediglich Tupel 1, woraufhin der Zustand *fresh* an den Klienten signalisiert wird und dieser daraufhin das im Cache befindliche Tupel als Resultat des Operationsaufrufs an den Klienten zurückgibt. Tupel 2 wird somit, trotz der Entsprechung zum Template, nicht als Resultat der *read*-Operation in Betracht gezogen.

Da aufgrund des in Abschnitt 5.6.2.1 vorgestellten Aufbaus der Instanzdatentupel ein Zugriff auf ein Instanzdatum stets durch dessen Identifikator(en) erfolgt, ist das oben beschriebene Verhalten erwünscht.

Eine weitere Optimierung des vorgestellten *Caching*-Mechanismus ist das universelle Feld U-CONST (vgl. Abschnitt 5.6.1.1), das ein Tupel als konstant markiert. Hat dieses Feld eines Tupels den Wert eines Booleschen `true`, so kann, aufgrund der Eigenschaft der Konstantheit des Tupels, auf die Überprüfung der *Freshness* beim jeweiligen PS-Server verzichtet, eine wiederholte Anfrage also vollständig auf Seite des PS-Klienten aus dem Cache beantwortet werden.

5.6.3 Realisierung der PS-Klienten

Abbildung 5.21 zeigt den Aufbau eines PS-Klienten. Entsprechend Abbildung 5.3 umfasst dieser die zur Interaktion mit seiner Umgebung notwendige Koordinations- und Kommunikationslogik sowie eine Umsetzung der Funktionalität der BPEL-Aktivität(en), die von diesem PS-Klienten implementiert werden.

Die Überprüfung eines Klienten, ob seine (durch das EWFN spezifizierten) Eingangsbedingungen hinsichtlich des Kontrollflusses erfüllt sind, erfolgt durch *take*- bzw. *sync*-Operationen mit entsprechenden Templates; die Templates stützen sich dabei auf die in Abschnitt 5.6.1.3 vorgestellte Tupelstruktur für

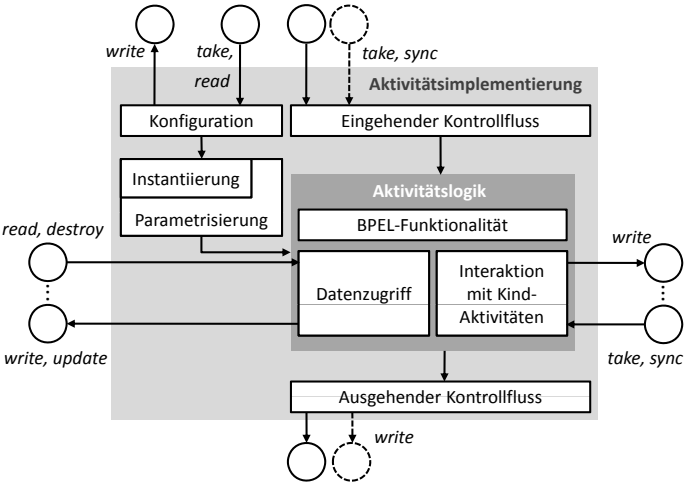


Abbildung 5.21: Interner Aufbau eines PS-Klienten und Interaktion mit seiner Umgebung.

Kontrollflusstupel.

Als Teil dieser Ausführung kann sowohl das (nicht-destruktive) Konsumieren oder Modifizieren von Instanzdaten durch *read* bzw. *update*, als auch (bei strukturierten Aktivitäten, z. B. einer WHILE-Schleife) die Interaktion mit Kind-Aktivitäten erfolgen[1]. Wie in der Abbildung dargestellt, kann ein PS-Klient die Funktionalität mehrerer BPEL-Aktivitäten implementieren.

Ist die Ausführung der Funktionalität eines Klienten abgeschlossen, so erzeugt er eine Menge von Tupeln, die seine Ausgangseffekte repräsentieren und schreibt diese in die entsprechenden PS. Da die Konfiguration eines Klienten ebenfalls unter Verwendung der PS-Infrastruktur erfolgt, ist diese in Abbildung 5.21, analog der Eingangsbedingungen des Klienten, durch ein entsprechendes Konfigurations-Template reflektiert. Das Konsumieren entsprechender Konfigurationsdaten kann entweder die Instantiierung von Aktivitätsimplementie-

[1] In Abbildung 5.22 wurde auf eine gesonderte Darstellung dieser Ausführungsteile verzichtet, da sich der PS-Klient in diesen entsprechend der Erfüllung der Eingangsbedingung bzw. der Erzeugung der Ausgangseffekte verhält.

rungen auf dem jeweiligen PS-Klienten (sofern diese noch nicht existieren) bzw. eine Änderung der Konfiguration bereits instantiierter Klienten zur Folge haben.

Abbildung 5.22 zeigt eine Übersicht über den Lebenszyklus eines PS-Klienten als UML-Aktivitätsdiagramm. Der Start eines PS-Klienten ist abhängig davon, ob es sich bei diesem um einen regulären Start oder um ein Wiederanlaufen nach einem Systemfehler handelt (vgl. Abschnitt 5.5.4). Handelt es sich beim Start um ein Wiederanlaufen, so erfrägt der PS-Klient seine Konfiguration aktiv bei der Deployment-Komponente seiner Domäne (vgl. Abschnitt 5.8). Andernfalls meldet sich dieser bei der Deployment-Komponente seiner Domäne an und wartet daraufhin passiv auf Konfigurationsdaten.

Erfolgt die Verarbeitung der Konfigurationsdaten erfolgreich, ist der Klient konfiguriert und bereit zur Verarbeitung von Anfragen. Verläuft die Konfiguration nicht erfolgreich, so kehrt der Klient in den konfigurationslosen Zustand zurück. Handelt es sich bei einem empfangenen Konfigurationsdatum um die Aufforderung zur Terminierung des Klienten, so führt dieser abhängig von der Art der Terminierung die folgenden Operationen aus: Handelt es sich bei der Terminierung des Klienten um eine Terminierung im Sinn eines regulären Anhaltens des Klienten (ausgelöst durch den Administrator des Systems), so führt dieser zunächst alle noch laufenden Verarbeitungen aus; danach beendet sich der PS-Klient. Erfolgt eine Terminierung im Sinn der BPEL-TERMINATE-Aktivität, so werden in Ausführung befindliche Verarbeitungsschritte (die zur jeweiligen Prozess- bzw. SCOPE-Instanz gehören) sofort abgebrochen und deren Transaktionen (als Teil des Schritts *Terminate Activity Implementation*) zurückgesetzt.

Nach erfolgreicher Konfiguration versucht der Klient fortlaufend, seine Eingangsbedingung durch das Konsumieren entsprechender Tupel zu erfüllen. Tritt bei der Verarbeitung der Eingangs-Tupel ein Fehler auf, so werden die Tupel als *unprocessable* markiert und in die jeweiligen PS zurückgeschrieben. Bei erfolgreicher Verarbeitung der Eingangs-Tupel gilt die folgende Fallunterscheidung: Repräsentieren die konsumierten Tupel negativen Prozesskontrollfluss (sind sie also vom Typ `DP`), so wird die von der Aktivität implementierte Anwendungslogik (d. h. die Funktion der implementierten BPEL-Aktivität) nicht ausgeführt;

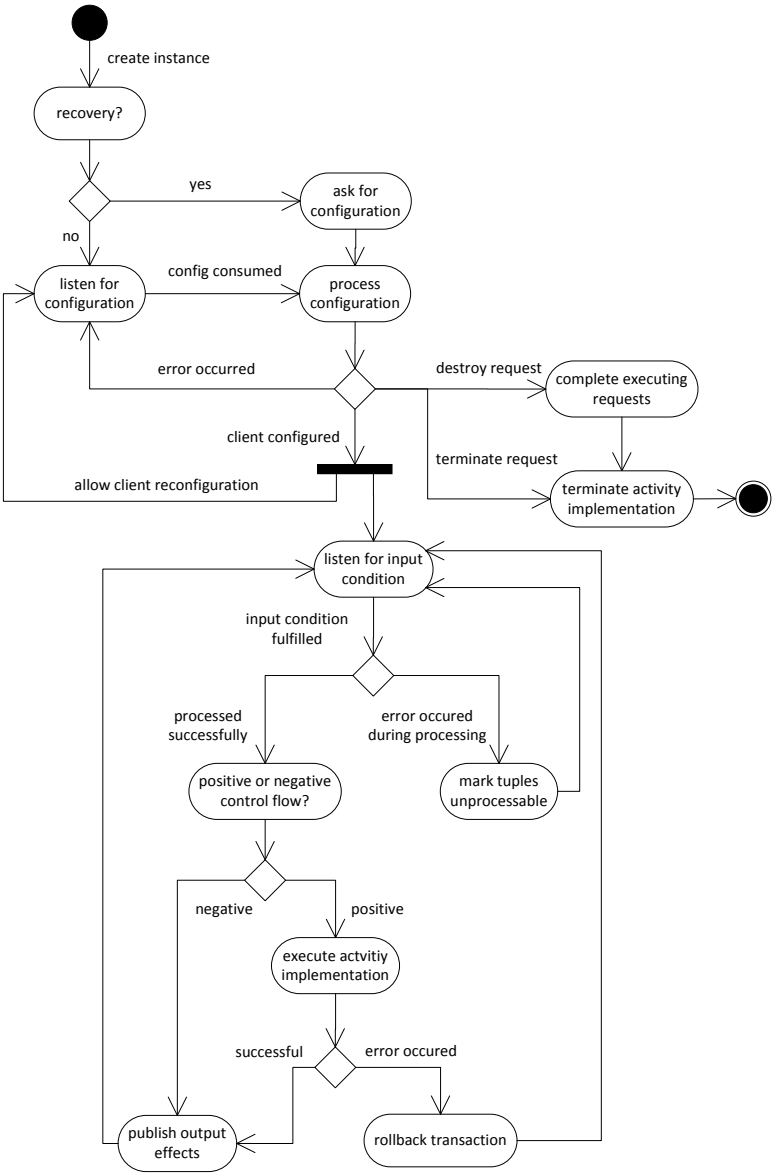


Abbildung 5.22: Lebenszyklus eines PS-Klienten

in diesem Fall kommuniziert der jeweilige PS-Klient dies – entsprechend der in [Mar10] beschriebenen EWFN-Ausführungssemantik – durch die Produktion entsprechender DP-Tupel an nachfolgende PS-Klienten. Repräsentieren die konsumierten Tupel hingegen positiven Prozesskontrollfluss (sind sie also vom Typ CF oder CF-E), so wird die Anwendungslogik der jeweiligen BPEL-Aktivität ausgeführt. Während dieser Ausführung können Fehler auftreten. Diese Fehler lassen sich zwei Kategorien zuordnen: ein Fehler kann entweder *erwartet* oder *unerwartet* sein.

Ein *erwarteter Fehler* ist ein Fehler auf Prozessebene, beispielsweise ein *SOAP-Fault* als Resultat der Ausführung einer INVOKE-Aktivität oder die explizite Auslösung eines Fehlers durch die THROW-Aktivität. Da diese Fehler auf einen BPEL-*Fault* abgebildet werden kann, gilt die Verarbeitung der Verarbeitungszyklus des PS-Klienten dennoch als *successful*. Die Verarbeitung erwarteter Fehler ist als Teil der BPEL-Spezifikation beschrieben; demnach wird diese durch die BPEL-EWFN-Patterns [Mar10] reflektiert und bedarf keine gesonderte Behandlung durch die PS-Infrastruktur. Betreffende PS-Klienten signalisieren ihre fehlerhafte Ausführung – entsprechend den im EWFN-Modell definierten Kanten – an die jeweiligen FAULTHANDLER durch die Produktion entsprechender Kontrollflusstupel. Die Verarbeitung derartiger Fehler hat als ersten Schritt die Terminierung der SCOPE-Aktivität (sowie von dieser eingeschlossener SCOPE-Aktivitäten), in welcher der Fehler ausgelöst wurde, zur Folge. Die Terminierung bedingt die Entfernung sämtlicher Kontrollflusstupel der jeweiligen SCOPE-Aktivität (vgl. Abschnitt 5.6.3.1). Da allerdings die Ausführung der PS-Klienten, die die Logik der *Fault- und Compensation-Handler* implementieren, ebenfalls durch das Konsumieren von Kontrollflusstupeln ausgelöst wird, unterscheidet sich die Koordination der Ausführung von PS-Klienten zur Verarbeitung von Fehlern, der Terminierung und der Kompensation von der "regulären" Koordination der Ausführung von PS-Klienten (im Sinne des sogenannten *forward control flow*). Dies geschieht durch Wahl des Typs A-CFTYPE = CFE in den kommunizierten Kontrollflusstupeln. Die Übersetzung von Tupeln mit A-CFTYPE = CF zu CFE erfolgt durch die Transition $t1$ im EWFN-Pattern des FAULTHANDLER [Mar10].

Im Gegensatz dazu ist ein sogenannter *unerwarteter Fehler* ein Fehler auf

Ebene der Aktivitätsimplementierung, der nicht auf einen BPEL-FAULT abgebildet werden kann. In diesem Fall wird – entsprechend des in Abschnitt 5.5.2 beschriebenen transaktionalen Verhaltens der PS-Klienten – die jeweilige globale Transaktion des PS-Klienten mit seinen, ihn umgebenden PS, zurückgerollt, was ein Rückschreiben bereits (destruktiv) konsumierter bzw. ein Entfernen bereits geschriebener Tupel zur Folge hat. Nach Wiederherstellung der Situation vor Beginn der Ausführung des PS-Klienten kann dieser einen weiteren Versuch unternehmen, seine Anwendungslogik auszuführen. Scheitert dies zum wiederholten Mal, so können die auslösenden Kontrollfluss-Tupel – ähnlich dem oben erläuterten Fall der fehlerhaften Verarbeitung der Eingang-Tupel – über das globale Feld U-UNPROC als fehlerhaft markiert werden und eine Benachrichtigung eines Administrators erfolgen. Dieser Vorgang führt dazu, dass die durch den jeweiligen PS-Klienten implementierte Aktivität in dieser Prozessinstanz ohne manuellen Eingriff nicht zur Ausführung kommt; von dieser Aktivität unabhängige Aktivitäten (d. h. Aktivitäten, die in keiner Kontrollflussbeziehung zur betroffenen Aktivität stehen) können ihre Verarbeitung allerdings regulär fortsetzen.

Eine Erläuterung der prototypischen Umsetzung des vorgestellten Aufbaus und Verhaltens der PS-Klienten ist Gegenstand von Abschnitt 7.2.

5.6.3.1 Terminierung von Prozessen und SCOPE-Aktivitäten

Die Terminierung eines BPEL-Prozesses bezeichnet den Vorgang der vorzeitigen Beendigung entweder einzelner SCOPE-Aktivitäten (und damit der in ihnen enthaltenen Kind-Aktivitäten) oder des gesamten Prozesses durch die EXIT-Aktivität. Der Terminierungsvorgang kann dabei aus unterschiedlichen Gründen erfolgen. Dies sind zum einen die explizite Terminierung von SCOPE-Aktivitäten durch die Aktivitäten TERMINATE und TERMINATESCOPE, zum anderen ist es die Terminierung aller Kind-Aktivitäten einer SCOPE-Aktivität, während deren Verarbeitung ein Fehler aufgetreten ist, zu Beginn der Ausführung des FAULTHANDLER der SCOPE-Aktivität.

In beiden Fällen der Terminierung verlangt die BPEL-Spezifikation den schnellstmöglichen Abbruch von in Ausführung befindlichen Aktivitäten und

damit insbesondere die Verhinderung der Ausführung nachfolgender Aktivitäten. Angewendet auf die PS-Infrastruktur sind drei unterschiedliche Szenarien hinsichtlich Terminierung von PS-Klienten zu unterscheiden:

PS-Klient nicht in Ausführung (S1)
: Die für die Ausführung eines PS-Klienten notwendigen Eingangsbedingungen sind entweder nicht oder nur teilweise erfüllt, seine Ausführung hat dementsprechend noch nicht begonnen, oder seine Eingangsbedingung sind zwar erfüllt, ein Verarbeitungszyklus wurde allerdings dennoch bisher nicht gestartet; d. h. sämtliche Eingangs-Tupel befinden sich noch in ihren jeweiligen PS.

PS-Klient in unterbrechbarer Ausführung (S2)
: Der PS-Klient hat seine Eingangs-Tupel bereits ganz oder teilweise konsumiert und befindet sich momentan in Ausführung. Weiterhin gilt, dass die Anwendungslogik des PS-Klienten eine Unterbrechung erlaubt.

PS-Klient in nicht unterbrechbarer Ausführung (S3)
: Auch in diesem Szenario gilt (ähnlich S2) dass der PS-Klient seine Eingangs-Tupel bereits ganz oder teilweise konsumiert hat. Allerdings gilt in diesem Fall weiterhin, dass die Anwendungslogik des PS-Klienten entweder nicht unterbrechbar ist oder der PS-Klient die Ausführung seiner Anwendungslogik abgeschlossen hat, bevor er in der Lage ist, das Kommando zum Abbruch der Anwendungslogik zu verarbeiten.

Die Umsetzung der Anforderungen der BPEL-Spezifikation in den oben genannten Fällen wird in der nachfolgend beschriebenen Weise erreicht.

Da in Szenario (S1) die Ausführung des zu terminierenden PS-Klienten noch nicht begonnen hat, kann seine Ausführung verhindert werden, indem die Tupel, die dessen Eingangsbedingung ausmachen, aus ihren jeweiligen PS entfernt werden.

Bedingt durch die speziellen Anforderungen der Szenarien (S2) und (S3) ist die Entfernung der jeweiligen Tupel hier allein nicht ausreichend für die Realisierung der Terminierung. In Szenario (S2) hat der PS-Klient die Tupel,

die seine Eingangsbedingung ausmachen, bereits teilweise oder vollständig konsumiert und befindet sich bereits in einem Ausführungszyklus. In diesem Fall erfolgt die Terminierung des Klienten unter Verwendung seiner Konfigurationsschnittstelle (vgl. Abbildung 5.22). Die Terminierung hat den Abbruch der laufenden Ausführung des Klienten zur Folge (vgl. Abschnitt 5.6.3), was ein Rücksetzen seiner laufenden Transaktion zur Folge hat. Da ein weiteres Konsumieren der Eingangs-Tupel durch den PS-Klienten verhindert werden muss, werden in der *Blacklist* (vgl. Abschnitt 5.3.2) der betroffenen PS entsprechende Einträge für die jeweils zu terminierenden SCOPE-Instanz(en) erzeugt.

Szenario 3 ähnelt Szenario 2 insoweit, als sich auch hier der jeweilige PS-Klient bereits in Ausführung befindet. Da hier seine Ausführung allerdings nicht oder nicht mehr unterbrechbar ist, schließt er seine Ausführung regulär ab. Durch die Anwendung der Blacklist-Funktionalität stehen die von ihm produzierten Tupel allerdings nachfolgenden PS-Klienten nicht mehr zum Konsumieren zur Verfügung.

Zusammenfassend beinhaltet die Realisierung der Terminierung (i) das Entfernen der betroffenen Kontrollflusstupel, (ii) das Anlegen entsprechender Blacklist-Einträge durch Anlegen entsprechender Blacklist-Einträge bei den jeweiligen PS-Server und (iii) das Schreiben eines entsprechenden Konfigurations-Tupels (*write*) zur Terminierung der betroffenen PS-Klienten.

Da eine Terminierung eines gesamten Prozesses die Interaktion des terminierenden PS-Klienten mit den PS-Servern sämtlicher, an der Ausführung des Prozesses teilnehmenden, Domänen bedingt, werden die zur Terminierung notwendigen Verarbeitungsschritte aus Gründen der Minimierung des domänenübergreifenden Kommunikationsaufwands durch die BPEL-motivierten PS-Operationen *exitProcess* bzw. *terminateScope* direkt auf der Seite der PS-Server umgesetzt.

Die Operation *exitProcess* nimmt als Parameter Werte entgegen, die den Werten der Felder A-PID, A-PCONF, A-PIID der zur terminierenden Prozessinstanz entsprechen. Die Ausführung von *exitProcess* umfasst die oben genannten Verarbeitungsschritte. Der terminierende PS-Klient ruft *exitProcess* auf sämtlichen, an der Ausführung des Prozesses beteiligten, PS auf.

Die Operation *terminateScope* verlangt zusätzlich zu den Parametern von

exitProcess die Angabe von Werten für X-SID und X-SIID der zu terminierenden SCOPE-Aktivitäten. Die Angabe von X-SIID erlaubt die Terminierung einer bestimmten Instanz einer *Scope*-Aktivität. Um die gleichzeitige Terminierung mehrerer SCOPE-Aktivitäten, beispielsweise eines bestimmten SCOPE und sämtlicher von diesem eingeschlossenen SCOPE-Aktivitäten, zu erlauben, kann *terminateScope* mit einer Liste von X-SID- und X-SIID-Wertpaaren aufgerufen werden. Zur korrekten Verarbeitung von Links, die über die Grenzen des terminierten SCOPE hinausreichen, werden `CF`-Tokens im Gegensatz zu *exitProcess* nicht entfernt, sondern durch entsprechende `DP`-Tokens ersetzt [Mar10]. Dies gilt wohlgemerkt nur für `CF`, nicht aber für `CFE`-Tokens, die für die Koordination der Ausführung der Terminierungsaktivität selbst verwendet werden.

5.6.3.2 Kompensation von Fehlern

Der Mechanismus der sogenannten *Compensation* in BPEL hat eine Zurücknahme der Effekte der Kind-Aktivitäten bereits erfolgreich ausgeführter SCOPE-Aktivitäten zum Ziel. Die BPEL-Spezifikation verlangt hierbei, dass eine Kompensation eines SCOPE nur dann erfolgt, wenn dieser seine Ausführung zuvor erfolgreich beendet hat. Weiterhin gilt im Fall der Nicht-Existenz eines expliziten COMPENSATIONHANDLER die sogenannte *Default Compensation Order* [Org07], welche die Reihenfolge der Ausführung der COMPENSATIONHANDLER der Kind-SCOPE-Aktivitäten eines SCOPE spezifiziert.

Die Ausführung der COMPENSATIONHANDLER ist durch das EWFN-Modell eines Prozesses explizit ausspezifiziert. Lediglich die Bestimmung der *Default Compensation Order* bedarf einer gesonderten Behandlung auf Ebene der PS-Infrastruktur. Eine in der BPEL-Spezifikation dargestellte Methode, um eine in jedem Fall korrekte *Default Compensation Order* zu gewährleisten, ist die Kompensierung von SCOPE-Aktivitäten in umgekehrter Reihenfolge ihrer Beendigung während ihrer regulären Ausführung anhand eines Zeitstempels. Im vorliegenden Fall der dezentralen Prozessausführung ist dieses Verfahren allerdings aufgrund unter Umständen nicht präzise synchronisierter Uhren nicht ohne weiteres anwendbar. Um dennoch die korrekte *Default Compensation Order* zu gewährleisten, gelten für SCOPE-Aktivitäten die folgenden

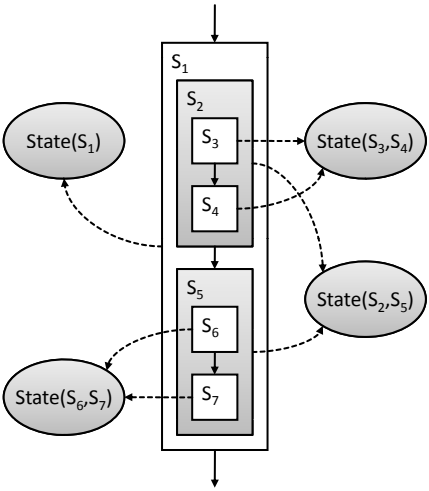


Abbildung 5.23: Protokollierung des Zustands von *Peer Scope*-Aktivitäten

Eigenschaften.

Jede SCOPE-Aktivität protokolliert in Form der Transition *Finalize Scope (t4)* [Mar10] die Beendigung ihrer Ausführung in Form eines sogenannten *Scope State*-Tupels. Es gilt, dass sämtliche *Peer Scopes* eines Prozesses (d. h. sämtliche SCOPE-Aktivitäten, die denselben Vater-SCOPE haben) ihren *Scope State* in demselben PS erfassen; für SCOPE-Aktivitäten, die keine *Peer Scopes* sind, gilt diese Einschränkung nicht.

Abbildung 5.23 zeigt dies beispielhaft an den SCOPE-Aktivitäten $S_1, \cdots, S_7$. Durchgezogene Pfeile zwischen SCOPE-Aktivitäten bezeichnen Kontrollflussabhängigkeiten; gestrichelte Pfeile bezeichnen das Schreiben von *Scope State*-Informationen. Die Abbildung zeigt die maximal mögliche Verteilung des *Scope State* für die gegebene SCOPE-Hierarchie. So wird beispielsweise der Zustand der SCOPE-Aktivitäten S_3 und S_4 aufgrund ihrer *Peer Scope*-Relation in demselben PS erfasst; für S_6 und S_2 gilt dies hingegen nicht.

Entsprechend der Erläuterung des universellen Tupel-Felds U-TIMESTAMP (vgl. Abschnitt 5.6.1.1) erhalten sämtliche in einen PS publizierte Tupel einen

Zeitstempel des Zeitpunkts der Schreiboperation. Da sämtliche *Peer Scope*-Aktivitäten ihren Status, gemäß den EWFN-Pattern, in demselben PS protokollieren, werden diese relativ zur selben Uhrzeit (des PS-Servers) protokolliert, was eine Anwendung der in der Spezifikation beschriebenen Vorgehensweise zur Kompensation in umgekehrter Reihenfolge ihrer Beendigung erlaubt. Hierfür konsumiert die PS-Klienten-Implementierung einer COMPENSATE-Aktivität sämtliche *ScopeState*-Tupel der Kind-SCOPE-Aktivitäten aus deren *Scope State*-PS, bestimmt anhand der Werte der U-TIMESTAMP-Felder die Reihenfolge, in der die SCOPE-Aktivitäten zu kompensieren sind und führt die Kompensation entsprechend des COMPENSATIONHANDLER-EWFN durch. Für die Kompensation der einzelnen Kind-SCOPE-Aktivitäten wird (in rekursivem Abstieg) analog verfahren.

5.7 Protokollierung der Prozessausführung

Entsprechend Anforderung 5.3 ist die Protokollierung der Ausführung von Prozessinstanzen eine essentielle Anforderung für eine Laufzeitumgebung für Produktionsprozesse, da diese die Grundlage für sowohl das *Monitoring* als auch das *Auditing* von Prozessinstanzen bilden. Die Vorgehensweise zur Erfassung dieser Protokolldaten, so dass durch diese eine vollständige Rekonstruktion des Ausführungsablaufs des EWFN-Modells eines Prozesses (und damit dessen BPEL-Repräsentation) möglich ist, wird im weiteren Verlauf dieses Abschnitts erläutert.

Hinsichtlich der Vorgehensweise zur Erfassung der Protokolldaten gelten die nachfolgend aufgeführten Überlegungen. Da PS-Klienten ausschließlich über den Austausch von Tupeln miteinander interagieren und selbst zustandslos sind (d. h. sie keinen internen Zustand über die Grenzen eines Ausführungszyklus hinweg vorhalten) gilt, dass der Zustand einer Prozessinstanz in der Gesamtheit der geschriebenen bzw. gelesenen Tupel, die dieser Instanz durch die Werte der Tupelfelder A-PMID, A-PCONF, A-PIID zugeordnet sind, manifestiert ist. Dieser Argumentation folgend gilt, dass die Protokollierung aller, während der Verarbeitung einer Prozessinstanz ausgeführten, PS-Operationen, die mit der Erzeugung, Modifikation und Entfernung dieser Tupel betraut sind, sämtliche

Informationen bietet, die notwendig sind, um den Ablauf einer Prozessinstanz auch nach deren Ausführung rekonstruieren zu können.

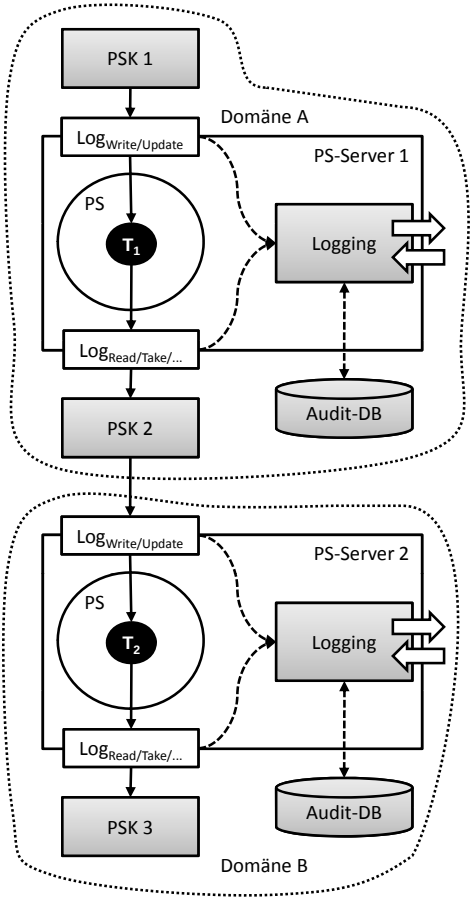


Abbildung 5.24: Verteilte Erfassung von Protokolldaten durch Protokollierung des Aufrufs der PS-Operationen auf den jeweiligen PS-Servern.

Abbildung 5.24 zeigt die für die PS-Infrastruktur gewählte Vorgehensweise der Erfassung der Protokolldaten, welche den oben formulierten Überlegun-

gen Rechnung trägt. Protokolldatenerfassung erfolgt ausschließlich durch die *Logging*-Komponente desjenigen PS-Servers (vgl. Abbildung 5.5) auf welchem eine PS-Operation ausgeführt wird. Dabei gilt, dass für jede auf einem PS-Server ausgeführte PS-Operation ein entsprechender Protokolleintrag in der Protokoll-Datenbank des jeweiligen PS-Servers erzeugt wird; die Erfassung eines Protokolleintrags erfolgt also lokal, eine Kommunikation mit anderen PS-Servern oder -Klienten zur Ausführungszeit einer Prozessinstanz ist daher nicht erforderlich.

Ein Protokolldatum umfasst Protokolldatenfelder, die generisch für sämtliche, auf der PS-Infrastruktur ausgeführte, Anwendungen erfasst werden; diese sind in Tabelle 5.5 dargestellt. Weiterhin ist die *Logging*-Komponente der PS-Server-Architektur erweiterbar für die Erfassung anwendungsspezifischer Protokolldatenfelder, um eine Erfassung dieser Informationen bei Verarbeitung der jeweiligen PS-Operation direkt auf dem PS-Server vorzunehmen; die anwendungsspezifischen Protokolldatenfelder für die Ausführung von EWFN-Modellen sind in Tabelle 5.6 dargestellt.

Da sich die Ausführung einer Prozessinstanz im Allgemeinen auf mehrere Domänen verteilt und die Erfassung der Protokolldaten jeweils lokal auf den PS-Servern dieser Domänen erfolgt, ist für eine vollständige Auswertung des Ausführungsablaufs einer Prozessinstanz die Aggregation der erfassten Protokolldaten aller an der Ausführung dieser Instanz beteiligten PS-Server erforderlich. Aus diesem Grund bietet die *Logging*-Komponente der PS-Server Operationen, die eine entfernte Abfrage der erfassten Protokolldaten durch ein Monitoring-Werkzeug erlauben. Die Struktur der erfassten Protokolldaten wird im nachfolgenden Abschnitt vorgestellt.

5.7.1 Struktur der erfassten Protokolldaten

Tabelle 5.5 zeigt die Felder der durch die PS-Infrastruktur erfassten generischen Protokolldaten.

Das Feld L-Op benennt die Operation (vgl. Abschnitt 5.3.1), deren Ausführung durch den jeweiligen Protokolleintrag beschrieben wird.

Das Feld L-PS beinhaltet die URL des PS, auf dem die Operation L-Op ausge-

Feld	Typ	Beschreibung
L-OP	String	Ausgeführte Operation
L-PS	URI	Identifikator des PS, auf de L-OP angewendet wird
L-CID	UID	Identifikator des Klienten, der L-OP ausgeführt hat
L-TUPLE	List(Tuple)	Liste der Tupel, die von L-OP betroffen waren
L-TIMESTAMP	Timestamp	Zeitpunkt der Ausführung von L-OP

Tabelle 5.5: Aufbau der generischen Protokolldaten

führt wird.

Das Feld L-CID identifiziert den PS-Klienten, der die Ausführung der Operation ausgelöst hat, welche durch den jeweiligen Protokolleintrag protokolliert wird; sie ist global eindeutig und wird vom Klienten im Operationskontext des Operationsaufrufs an den Server übertragen.

Das Feld L-TUPLE beinhaltet die Tupel, die von der protokollierten Operation betroffen waren.

Das Feld L-TIMESTAMP gibt den Zeitpunkt an, zu welchem die Ausführung der protokollierten Operation begonnen bzw. beendet wurde. Für die PS-Infrastruktur wird angenommen, dass die Systemzeit der PS-Server innerhalb gewisser Toleranzen synchron ist; für die Uhrensynchronisation finden existierende Verfahren und Protokolle, wie beispielsweise das *Network Time Protocol (NTP)* [Mil92] Verwendung. Da allerdings in einem verteilten System keine exakt synchronisierten Uhren vorausgesetzt werden können, stützt sich die Protokollierung der Abfolge der einzelnen Navigationsschritte einer EWFN-Ausführung zusätzlich auf deren *kausale Abhängigkeit*, reflektiert durch das anwendungsspezifische Protokolldatenfeld L-CONTEXT (vgl. Tabelle 5.6).

Tabelle 5.6 zeigt die Felder der durch die PS-Infrastruktur erfassten anwendungsspezifischen Protokolldaten, die während der Ausführung von EWFN-Modellen zusätzlich zu den generischen Protokolldatenfeldern (Tabelle 5.5) erfasst werden.

Die Felder A-PID, A-PCONF und A-PIID entsprechen den jeweiligen anwendungsspezifischen Tupelfeldern (vgl. Abschnitt 5.6.1.2); sie werden zur Ein-

Feld	Typ	Beschreibung
A-PID	URI	Identifikator des Prozessmodells
A-PCONF	UID	Identifikator einer Prozesskonfiguration, z. B. eine UUID
A-PIID	UID	Identifikator einer Instanz einer Konfiguration
L-ARC	X-AID, X-AIID, A-TYPE	Kante des EWFN die durch L-OP "navigiert" wird
L-CONTEXT	List(X-AID, X-AIID)	Identifikator der Kanten, deren Navigation "Auslöser" für die Ausführung von L-OP waren

Tabelle 5.6: Aufbau der anwendungsspezifischen Protokolldaten für die Ausführung von EWFN-Modellen

schränkung der Abfrage der Protokolldaten verwendet.

Das Feld L-ARC beschreibt die von der protokollierten PS-Operation unmittelbar "navigierte" EWFN-Kante, deren Instanz und den Typ des Tupels bzw. der Tupel, welche(s) entlang der jeweiligen Kante kommuniziert wurde(n). Verwendung findet L-ARC (bzw. die enthaltenen Werte von X-AID und X-AIID) zur Herstellung eines Bezugs zwischen der Verarbeitung einer PS-Operation während der Ausführung eines EWFN-Modells und einer Aktivität desjenigen Prozessmodells, das als Eingabe für den EWFN-Transformationsschritt diente (vgl. Bedeutung des Attributs `ArcID` im DDD, Abschnitt 5.8.1). Dies ist notwendig, da es einem Nutzer der PS-Infrastruktur möglich sein soll, die Ausführung eines Prozesses anhand des von ihm vorgegebenen BPEL-Prozessmodells und nicht anhand des tatsächlich auf der PS-Infrastruktur ausgeführten EWFN-Modells nachvollziehen zu können.

Das Feld L-CONTEXT beschreibt den Kontext, in welchem die protokollierte Operation ausgeführt wurde. Der Wert von L-CONTEXT ist eine Liste der Werte X-AID und X-AIID (vgl. Abschnitt 5.6.1) der EWFN-Kanten, deren Navigation die Schaffung der Voraussetzungen des protokollierten Operationsaufrufs zur Folge hatte.

Für die Bestimmung des Werts des *L-Context*-Felds des Ausführungsprotokolls einer PS-Operation, die während der Navigation eines EWFN-Modells

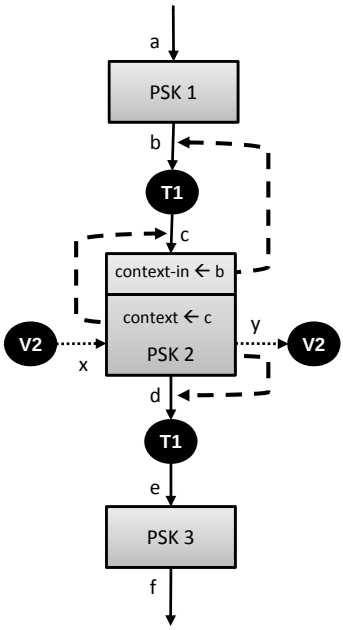


Abbildung 5.25: Beschreibung kausaler Abhängigkeiten zwischen Ausführungen von PS-Klienten durch das Feld L-CONTEXT.

ausgeführt wird, gelten die folgenden – anhand Abbildung 5.25 erläuterten – Fallunterscheidungen.

- Beschreibt eine Operation das Konsumieren eines Kontrollflusstupels durch *take* oder *sync*, so beinhaltet das Feld L-CONTEXT die X-AID- und X-AIID-Identifikatoren der Kante bzw. Kanten, die zur Erfüllung des bzw. der Templates der Operation beigetragen haben. Dies wird als der sogenannte *Startkontext context-in* bezeichnet.
  Im dargestellten Beispiel ist dies der Identifikator der Kante b.
  Der Wert von X-AID kann durch anwendungsspezifische Protokollierungsfunktionen auf Seite des PS-Servers direkt dem Template des jeweiligen Operationsaufrufs entnommen werden. Da das X-AIID-Feld im Template

mit dem Wildcard-Operator belegt ist, wird dessen Wert aus dem X-AIID-Feld des mit der Operation jeweils konsumierten Tupels übernommen.

- Beschreibt eine Operation das Erzeugen eines Kontrollflusstupels durch *write*, so beinhaltet das Feld L-CONTEXT die X-AID- und X-AIID-Identifikatoren der Kante(n), die die Operationen repräsentieren, die zum Start des Ausführungszyklus des jeweiligen PS-Klienten geführt haben, d. h. dessen eingehende Kontrollflusskanten. Dies wird als der sogenannte *Ausführungskontext context* bezeichnet.
  Im dargestellten Beispiel ist dies der Identifikator der Kante c.
  Diese Werte übergibt der ausführende PS-Klient als Teil des Operationskontexts des Aufrufs der jeweiligen PS-Operation an den PS-Server.
- Beschreibt eine Operation das Lesen oder Modifizieren eines Instanzdatentupels, so erfolgt die Bestimmung des Werts des Felds L-CONTEXT wie beim Erzeugen von Kontrollflusstupeln.

Nachfolgend wird die Verwendung der Felder L-ARC und L-CONTEXT zur Abbildung der kausalen Abhängigkeit der Ausführung mehrerer PS-Operationen anhand von zwei Beispielen erläutert.

Abbildung 5.26 zeigt ein Beispiel einer Interaktion zwischen mehreren PS-Klienten, in welchem die in Abbildung 5.25 dargestellten Konzepte umgesetzt werden. Tabelle 5.7 zeigt die Felder L-OP, L-CID, L-CONTEXT und L-ARC der während der Ausführung des dargestellten EWFN-Modells erfassten Protokolldaten; jede Zeile der Tabelle repräsentiert einen erfassten Protokolleintrag. Zur Verdeutlichung des Beispiels werden zusätzlich jeweils die Namen der Quell- und Ziel-PS-Klienten der ausgeführten Operation genannt. Die Ellipsis in der letzten Spalte der Tabelle soll verdeutlichen, dass hier jeweils das (bzw. die) in der Operation produzierten bzw. konsumierten Tupel als Teil des jeweiligen Protokolleintrags abgelegt werden.

Anhand des Beispiels wird die Bedeutung des Felds L-CONTEXT in Verbindung mit L-ARC ersichtlich. Zeile 7 zeigt beispielsweise die Erfüllung der Eingangsbedingung von D durch die Ausführung der *take*-Operation entlang Kante 8 auf einem Tupel, das entlang der Kante 5 erzeugt wurde. Da beide Kanten jeweils zum ersten mal navigiert wurden, gilt für beide der Instanzidentifikator

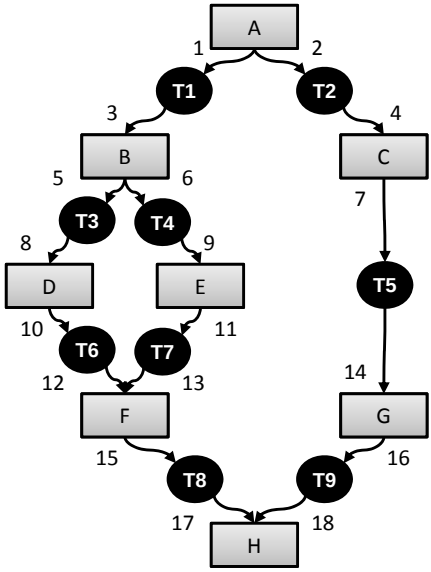


Abbildung 5.26: Beispiel: Ausschnitt einer zyklenfreien Interaktion von PS-Klienten und kommunizierte Kontrollflusstupel.

1. L-CONTEXT bestimmt sich somit zu $(5, 1)$, L-ARC zu $(8, 1, \texttt{CF})$. Die Navigation von Kante 10, die den Abschluss der Ausführung von D signalisiert, führt – entsprechend obiger Beschreibung – zur Protokollierung eines Eintrags mit diesem Wert $(8, 1)$ im Feld L-CONTEXT. Ein interessanter Punkt in diesem Protokoll ist der *Join* bei F (Zeile 11 und 12), d. h. das Resultat der Interaktionen $D \rightarrow F$ mit dem L-ARC-Wert $(12, 1, \texttt{CF})$ und $E \rightarrow F$ mit dem L-ARC-Wert $(13, 1, \texttt{CF})$. Als Resultat des Konsumierens zweier Kontrollflusstupel entlang unterschiedlicher eingehender Kanten resultiert die Beendigung der Ausführung von F (Zeile 13) im L-CONTEXT-Wert $(12, 1), (13, 1)$.

Abbildung 5.27 zeigt eine weitere Anwendung der beschriebenen Vorgehensweise. In diesem Fall beinhaltet das dargestellte EWFN eine Schleife; die Kanten $3, 4, 7, 8$ werden somit mehrfach navigiert. Weiterhin zeigt das Beispiel den Instanzdatenzugriff entlang der Kanten 5 und 6, der ebenfalls in jeder

	Protokolldatenfeld						A-Tuple
	L-Op	L-CId	L-Context	L-Arc	Quelle	Ziel	...
1	*write*	A	-	(1,1,CF)	A	B	
2	*write*	A	-	(2,1,CF)	A	C	
3	*take*	B	[(1,1)]	(3,1,CF)	A	B	
4	*take*	C	[(2,1)]	(4,1,CF)	A	C	
5	*write*	B	[(3,1)]	(5,1,CF)	B	D	
6	*write*	B	[(3,1)]	(6,1,CF)	B	E	
7	*take*	D	[(5,1)]	(8,1,CF)	B	D	
8	*take*	E	[(6,1)]	(9,1,CF)	B	E	
9	*write*	D	[(8,1)]	(10,1,CF)	D	F	
10	*write*	E	[(9,1)]	(11,1,CF)	E	F	
11	*take*	F	[(10,1)]	(12,1,CF)	D	F	
12	*take*	F	[(11,1)]	(13,1,CF)	E	F	
13	*write*	F	[(12,1), (13,1)]	(15,1,CF)	F	H	
14	*write*	C	[(4,1)]	(7,1,CF)	C	G	
15	*take*	G	[(7,1)]	(14,1,CF)	C	G	
16	*write*	G	[(14,1)]	(16,1,CF)	G	H	
17	*take*	H	[(15,1)]	(17,1,CF)	F	H	
18	*take*	H	[(16,1)]	(18,1,CF)	G	H	

Tabelle 5.7: Beispiel: Ausschnitt der während der Ausführung von Abbildung 5.26 erfassten Protokolldaten.

Iteration der dargestellten Schleife ausgeführt wird.

Wie im Beispiel dargestellt, gilt, dass die Instanzdatenzugriffe (Zeile 5 und 6 bzw. 11 und 12) stets (siehe Abbildung 5.25) im Ausführungskontext des jeweiligen PS-Klienten, d. h. mit den Identifikatoren dessen eingehender Kanten, protokolliert werden; diese PS-Operationen werden somit als kausale Folge der Navigation der Eingangskanten der jeweiligen Transition identifiziert.

Es ist zu bemerken, dass in der zweiten Iteration der Schleife, die mit der Navigation von Kante 3 (Zeile 9) beginnt, ein neuer Wert für X-AIId sowohl in L-Context als auch in L-Arc vergeben wird, um eine eindeutige Identifikation der jeweiligen Kante zu erlangen.

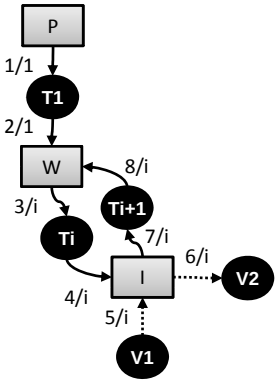


Abbildung 5.27: Beispiel: Ausschnitt einer zyklischen Interaktion zwischen PS-Klienten.

5.7.2 Bezug zwischen Navigationsschritt im EWFN und der Ausführung einer BPEL-Aktivität

Abbildung 5.28 zeigt einen Ausschnitt eines EWFN-Modells, anhand welchem beispielhaft erläutert werden soll, wie ein Bezug zwischen der Ausführung einer PS-Operation und der Ausführung einer BPEL-Aktivität auf Basis der erfassten Protokolldaten und dem EWFN-Pattern der jeweils ausgeführten Aktivität erfolgen kann. Zu diesem Zweck wurden die Kontrollflusskanten (in der Abbildung als schwarze, durchgezogene Linien dargestellt) aufsteigend durchnummeriert. Die Nummern geben symbolisch den X-AID-Wert der jeweiligen Kante wider (vgl. Tabelle 5.3).

Ein PS-Klient beginnt einen Verarbeitungszyklus bei Ausführung derjeniger PS-Operation, die die Navigation der Kante von der *Start*-Stelle eines EWFN-Pattern realisieren und beendet diesen durch die Navigation der Kanten zur *Ended*- bzw. *Failed*-Stelle; der Typ der konsumierten bzw. produzierten Tupel ist hier jeweils `CF`. Welche Kanten dies sind, ist abhängig von der jeweils ausgeführten BPEL-Aktivität. So gilt beispielsweise für die dargestellte ASSIGN-Aktivität ein Ausführungsbeginn durch Navigation der Kante zwischen der

	Protokolldatenfeld						A-Tuple	
	L-Op	L-CId	L-Context	L-Arc	Quelle	Ziel	I-Id	...
1	*write*	P	-	(1,1,CF)	P	W	-	
2	*take*	W	[(1,1)]	(2,1,CF)	P	W	-	
3	*write*	W	[(2,1)]	(3,1,CF)	W	I	-	
4	*take*	I	[(3,1)]	(4,1,CF)	W	I	-	
5	*read*	I	[(4,1)]	(5,1,DATA)	-	-	V1	
6	*update*	I	[(4,1)]	(6,1,DATA)	-	-	V2	
7	*write*	I	[(4,1)]	(7,1,CF)	I	W	-	
8	*take*	W	[(7,1)]	(8,1,CF)	I	W	-	
9	*write*	W	[(8,1)]	(3,2,CF)	W	I	-	
10	*take*	I	[(3,2)]	(4,2,CF)	W	I	-	
11	*read*	I	[(4,2)]	(5,2,DATA)	-	-	V1	
12	*update*	I	[(4,2)]	(6,2,DATA)	-	-	V2	
13	*write*	I	[(4,2)]	(7,2,CF)	I	W	-	
14	*take*	W	[(7,2)]	(8,2,CF)	I	W	-	

Tabelle 5.8: Beispiel: Ausschnitt der während der Ausführung von Abbildung 5.27 erfassten Protokolldaten.

Stelle *Start* und der Transition *t1*. Legt man die in Tabelle 5.6 eingeführte Protokolldatenstruktur zugrunde, so wird die Navigation dieser Kante durch einen Protokolleintrag beschrieben, für welchen gilt:

- L-Op = `TAKE`
- X-AId in L-Arc = 1
- A-Type = `CF`

Ein erfolgreiches Ausführungsende wird durch Navigation der Kante zwischen *t1* und *Ended* signalisiert. Entsprechend obiger Vorgehensweise gelten für den entsprechenden Protokolleintrag die folgenden Werte:

- L-Op = `WRITE`
- X-AId in L-Arc = 2
- A-Type = `CF`

Für die abgebildete Reply-Aktivität gilt ein Ausführungsbeginn bei Navigation der Kante zwischen *Start* und *t1*, ein erfolgreiches Ausführungsende

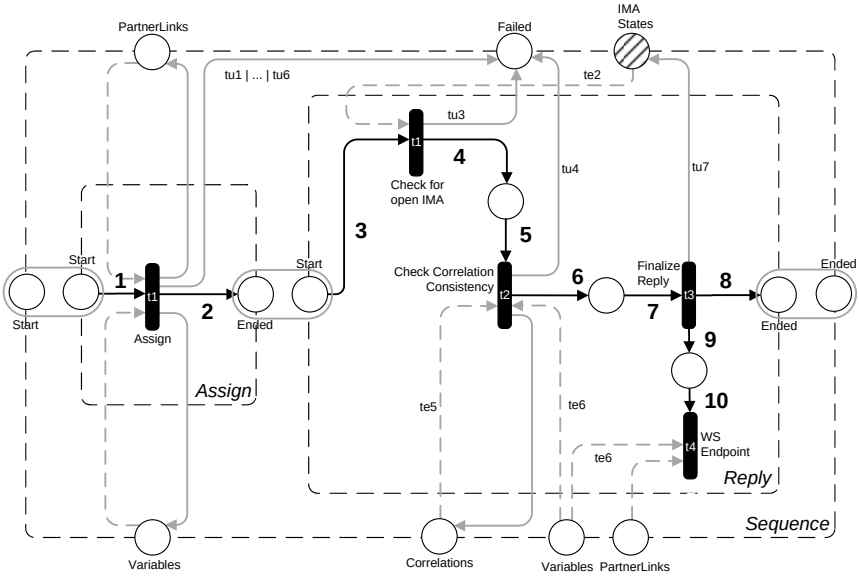


Abbildung 5.28: Ausschnitt eines EWFN-Modells.

bei Navigation der Kante zwischen *t3* und *Ended* und ein fehlerhaftes Ausführungsende bei der Kante zwischen *t1* bzw. *t2* und *Failed*. Die Bestimmung des Aufbaus der entsprechenden Protokolleinträge erfolgt analog dem obigen Verfahren.

5.7.3 Verwendung erfasster Protokolldaten für die Parametrisierung der Partitionierung

In Abschnitt 3.5.2.4 wurden die Protokolldaten ausgeführter Prozessinstanzen als eine mögliche Einflussgröße des Prozesspartitionierungsvorgangs vorgestellt. Auf Grundlage der Erläuterungen im vorangehenden Abschnitt kann die Bestimmung der Wahrscheinlichkeitswerte wie folgt realisiert werden.

Die Ausführungswahrscheinlichkeit P_{Exec} einer Aktivität in einer bestimmten Prozessinstanz berechnet sich durch den Quotient der *Anzahl der Navigationen*

der Start-Kante der jeweiligen Aktivität und der *Anzahl der Prozessinstanzen*. Für die Bestimmung der Wahrscheinlichkeit der erfolgreichen Ausführung einer Aktivität P_{Succ} gilt analog der Quotient der *Anzahl der Navigationen der Ended-Kante der jeweiligen Aktivität* und der *Anzahl der Prozessinstanzen*. Da sämtliche Instanzdatenzugriffe im EWFN-Modell durch Kanten explizit repräsentiert sind, kann eine Bestimmung der durchschnittlichen Größe von Instanzdatenzugriffen analog erfolgen; auf eine gesonderte Darstellung der Berechnung wird daher verzichtet.

5.8 Deployment

In der PS-Infrastruktur wird das Prozess-Deployment (Anforderung 5.2), durch die Konfiguration der an der Ausführung eines Prozesses teilnehmenden PS-Klienten realisiert.

Abbildung 5.29 zeigt die Vorgehensweise zur Installation eines Prozesses auf der PS-Infrastruktur. Unter Verwendung des Partitionierungswerkzeugs erzeugt die *Deployer*-Rolle (vgl. Abschnitt 3.5.2.3) auf Basis des ursprünglichen

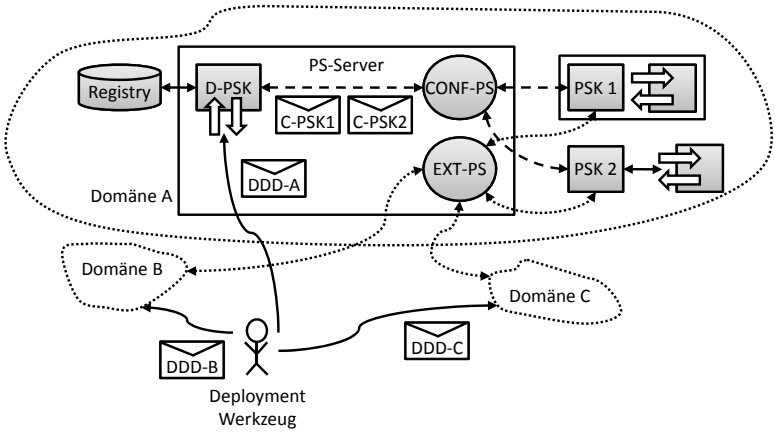


Abbildung 5.29: Deployment von Prozessen auf der PS-Infrastruktur.

BPEL-Prozesses, dessen korrespondierender EWFN-Transformation und der Parametrisierung der Partitionierung einen sogenannten *Distributed Deployment Descriptor (DDD)*, welcher die für die Konfiguration sämtlicher PS-Klienten notwendigen Informationen beinhaltet.

Der erzeugte DDD wird in sogenannte DDD-Fragmente für die einzelnen Domänen (im Beispiel A, B, C) dekomponiert. Der Deployer übermittelt das jeweilige DDD-Fragment (*DDD-A*, *DDD-B*, *DDD-C*) sowie das ursprüngliche BPEL-Modell und zugehörige WSDL-Dokumente (zur Beschreibung der Schnittstelle des Prozesses selbst sowie der Schnittstelle aufgerufener Dienste) an die Deployment-Komponente des PS-Servers (vgl. Abbildung 5.5) der entsprechenden Domäne. Die Deployment-Komponente nimmt das empfangene DDD-Fragment entgegen, legt dieses persistent ab, dekomponiert es in weitere Fragmente (*C-PSK1*, *C-PSK2*) für die Konfiguration einzelner Domänen-interner PS-Klienten und publiziert diese Fragmente nebst dem BPEL-Prozessmodell in den Konfigurations-PS (*CONF-PS*) der Domäne. Wurde im Rahmen der Partitionierung lediglich die Partition einer Aktivität, nicht aber der ausführende PS-Klient bestimmt, so bestimmt die Deployment-Komponente des PS-Servers der Domäne im Rahmen eines sogenannten *Domänen-internen* Deployment-Schritts anhand lokaler Information, z. B. aktueller Verfügbarkeit oder Auslastung der in der Domäne angemeldeten (vgl. Abschnitt 5.6.3) PS-Klienten, welcher PS-Klient eine bestimmte Aktivität implementieren soll und ergänzt diese Information vor der Weitergabe der Konfigurationsinformationen an die einzelnen PS-Klienten.

Die einzelnen PS-Klienten überwachen nach ihrem Start den Konfigurations-PS der Domäne auf die Präsenz von DDD-Fragmenten, die an sie gerichtet sind (vgl. Abbildung 5.22). Findet ein Klient ein derartiges Konfigurationsdatum, so konsumiert er dieses, verarbeitet es und bestätigt der Deployment-Komponente die erfolgreiche Verarbeitung.

Nach erfolgter Bestätigung der Verarbeitung sämtlicher Konfigurations-Elemente des DDD-Fragments der jeweiligen Domäne bestätigt die Deployment-Komponente dies dem Deployer. Hat der Deployer entsprechende Bestätigungen von sämtlichen Domänen erhalten, so ist die Konfiguration der PS-Klienten abgeschlossen und der Prozess bereit zur Instantiierung. Die Deinstallation

eines Prozesses verläuft analog; der Deployer übersendet eine entsprechende Deinstallations-Nachricht an sämtliche Domänen, welche die Identifikatoren A-PID und A-PCONF der zu deinstallierenden Prozesskonfiguration beinhaltet. Die Deinstallation eines Prozesses hat zur Folge, dass keine neuen Instanzen des Prozesses mehr erzeugt werden können. In Ausführung befindliche Prozessinstanzen sind hiervon nicht betroffen.

5.8.1 Distributed Deployment Descriptor

Abbildung 5.30 zeigt einen exemplarischen Ausschnitt eines DDD zur Konfiguration der PS-Klienten der Domäne mit dem `Domain`-Identifikator `ps://organization-a.org`. Aufgrund besserer Lesbarkeit werden im dargestellten Beispiel symbolische Bezeichner für beispielsweise Aktivitäten-, Kanten- und Instanzdatenidentifikatoren verwendet.

Durch die Attribute `Process` und `Configuration` des DDD-XML-Elements wird die jeweilige Konfiguration eines Prozessmodells identifiziert, auf die sich der DDD bezieht. Das Element `Domain` beinhaltet die Konfigurationsdaten sämtlicher PS-Klienten einer Domäne. Es bildet damit die Grundlage für die Zerlegung eines vollständigen DDD eines Prozesses in seine domänenspezifischen DDD-Fragmente. Das Attribut `ID` des `Domain`-Elements identifiziert die Domäne global eindeutig. Das Element `ExternalPS` beinhaltet die URL des extern sichtbaren PS zur Ausführung von EWFN-Modellen in der jeweiligen Domäne (vgl. Abbildung 5.2 und 5.29), durch welchen zur Laufzeit von Instanzen der Prozesskonfiguration die Interaktion mit PS-Klienten anderer Domänen erfolgt.

Jedes `Activity`-Kind-Element des `Activities`-Container-Elements beschreibt eine Implementierung einer BPEL-Aktivität durch einen PS-Klienten. Da als maximale Verteilungsgranularität der in Kapitel 7 vorgestellten Umsetzung (und damit auch des hier abgebildeten DDD) eine BPEL-Aktivität gewählt wurde, werden interne Transitionen eines EWFN einer BPEL-Aktivität hier demnach nicht gesondert betrachtet.

Eine Aktivität eines BPEL-Prozesses wird identifiziert durch den XPath-Ausdruck im Attribut `Path` des `Activity`-Elements. In Verbindung mit dem

```
<DDD
  xmlns="http://www.iaas.uni−stuttgart.de/ns/2009/process−space/ddd"
  xmlns:example="http://example.org"
  Process="example:LoanApproval"
  Configuration="urn:uuid:bb10ed11−939f−012b−0912−00166f1aa24f">
  <Domain ID="ps://organization−a.org">
    <ExternalPS>ps://organization−a.org/extps</ExternalPS>
    <Activities>
      <Activity Path="//process/flow/invoke[@name='invoke2']">
        <PSClient>psc://organization−a.org/psk1</PSClient>
        <Operation PlaceID="Start"
            ArcID="invoke2−start"
            OperationType="take"
            TupleType="CF"
            TransitionID="t1"
            PS="ps://organization−a.org/extps">
          <LinkID>invoke1−ended</LinkID>
        </Operation>
        <Operation PlaceID="Ended"
            ArcID="invoke2−ended"
            OperationType="write"
            TupleType="CF"
            TransitionID="t6"
            PS="ps://organization−a.org/extps"/>
        <Operation PlaceID="Variables−In"
            ArcID="invoke2−input−variables"
            DataID="invoke2−input−variable"
            OperationType="read"
            TupleType="DATA"
            TransitionID="t1"
            PS="ps://organization−a.org/extps"/>
        <Operation PlaceID="Variables−Out"
            ArcID="invoke2−output−variables"
            DataID="invoke2−output−variable"
            OperationType="update"
            TupleType="DATA"
            TransitionID="t6"
            PS="ps://organization−a.org/extps"/>
        <ServiceBinding>
          <ServiceWSDL Name="sns:Service1" Port="Service1HTTPPort"/>
        </ServiceBinding>
      </Activity>
    </Activities>
    <PSParticipants>
      <PSParticipant>ps://organization−a.org</PSParticipant>
      <PSParticipant>ps://organization−b.org</PSParticipant>
      <PSParticipant>ps://organization−c.org</PSParticipant>
    </PSParticipants>
  </Domain>
</DDD>
```

Abbildung 5.30: Ausschnitt eines *Distributed Deployment Descriptor (DDD)*.

BPEL-Modell, welches mit dem DDD zu den Deployment-Komponenten der an der Prozessausführung teilnehmenden Domänen (bzw. deren PS-Server) übermittelt wird, erlaubt der `Path`-Wert die Bestimmung der durch den jeweiligen PS-Klienten implementierten BPEL-Aktivität sowie deren Parameter in der BPEL-Repräsentation des Prozesses.

Über das Element `PSClient` wird der Identifikator desjenigen PS-Klienten bestimmt, der für die Ausführung der Funktionalität der jeweiligen `Activity` konfiguriert wird. Das `PSClient`-Element bildet damit die Grundlage für die Aufteilung des DDD-Fragments in Konfigurationsinformation für die einzelnen PS-Klienten innerhalb einer Domäne. Aufgrund der Möglichkeit einer Domänen-internen Bestimmung des Werts des Elements `PSClient` ist dieser zum Zeitpunkt der Übermittlung des DDD zwischen Deployer und Deployment-Komponente einer Domäne optional. Zum Zeitpunkt der Konfiguration der einzelnen PS-Klienten ist der Wert obligatorisch.

Jedes `Operation`-Element beschreibt eine von der jeweiligen Aktivität auszuführende PS-Operation. Das Attribut `PlaceID` identifiziert die Stelle des EWFN des Prozesses, auf dem die jeweilige PS-Operation ausgeführt wird. Das Attribut `ArcID` identifiziert die Kante eines EWFN, die durch die jeweilige PS-Operation "navigiert" wird. Der Typ der jeweiligen Operation wird durch das Attribut `OperationType` beschrieben; der Typ des mit der Operation verarbeiteten Tupels durch das Attribut `TupleType`. Hierbei identifiziert der Wert `CF` die Operation als eine, den Kontrollfluss betreffende, Operation, der Wert `DATA` eine, den Datenzugriff betreffende, Operation. Die Transition eines EWFN-Pattern, welche die durch ein `Operation`-Element repräsentierte PS-Operation durchführt, wird mittels des Attributs `TransitionID` beschrieben. Das Attribut `PS` beinhaltet die URL des PS, auf dem die jeweilige PS-Operation ausgeführt werden soll.

Bei konsumierenden Kontrollflusskanten identifiziert `LinkID` die ausgehende Kante des vorangehenden PS-Klienten durch deren X-AID-Wert. Da durch die *sync*-Operation ein Konsumieren von mehr als einem `CF`-Tupel möglich ist, erfolgt die Repräsentation von `LinkID` in Form eines XML-Elements. Der Wert des `LinkID`-Elements findet für die Bestimmung der Templates der Eingangsbedingung des jeweiligen PS-Klienten Verwendung (vgl. Abschnitt 5.6.1.3).

Bei `Operation`-Elementen deren `TupleType`-Feld den Wert `DATA` hat, beinhaltet das `DataID`-Attribut den Wert I-ID des Instanzdatums, auf das als Teil der jeweiligen Operation zugegriffen werden soll.

Zur Unterstützung der Terminierung eines Prozesses beinhaltet der DDD eine Liste der EWFN-PS sämtlicher Ausführungsteilnehmer des Prozesses. Diese werden in Form der `PSParticipant`-Kind-Elemente des `PSParticipants`-Elements repräsentiert.

Zusätzlich zu den oben genannten Konfigurationsinformation erfordert die Konfiguration der Interaktionsaktivitäten eines Prozesses – INVOKE, RECEIVE und PICK – zusätzliche Informationen zum Service-Binding. Für RECEIVE- und PICK-Aktivitäten ist dies eine Beschreibung des empfangenden Endpunkts, für INVOKE eine Beschreibung des Binding des aufzurufenden Web-Service. Die Bestimmung dieser Information erfolgt durch das Kind-Element `ServiceBinding` einer entsprechenden `Activity` und kann, entsprechend der Anforderungen des jeweiligen Einsatzszenarios, auf unterschiedlichen Wegen, z. B. in Form einer statischen Vorgabe eines PORT in einem WSDL-Dokument oder dynamisch zur Laufzeit durch einen ESB virtualisiert erfolgen. Die in Abschnitt 7.2 vorgestellte prototypische Implementierung unterstützt das statische Binden von Dienstimplementierungen durch das in Abbildung 5.30 dargestellte Element `ServiceWSDL` mit seinen Attributen NAME und PORT.

5.9 Qualitative Bewertung der Eigenschaften der entwickelten Infrastruktur

Gegenstand dieses Abschnitts ist eine qualitative Bewertung der Eigenschaften der in den vorangehenden Abschnitten vorgestellten PS-Infrastruktur. In die Diskussion fließen ein: (i) die Betrachtung der, aus der Notwendigkeit der Weitergabe der Kontrolle über die Prozessausführung resultierenden, Prozesslaufzeiteigenschaften (Abschnitt 5.9.1), (ii) eine Auflistung des sogenannten *Shared State*[1], der für das Erhalten der Ausführungssemantik von BPEL not-

[1] Der Begriff *Shared State* bezeichnet all jene Informationen, welche von mehreren Ausführungsteilnehmern eines Prozesses konsumiert bzw. manipuliert werden müssen.

wendig ist (Abschnitt 5.9.2), sowie (iii) die durch das vorgestellte Verfahren erreichte Autonomie der Teilnehmer einer Prozessausführung (Abschnitt 5.9.3).

Die in diesem Abschnitt vorgestellten Erkenntnisse bilden damit die Grundlage für mögliche weitere, an die Resultate dieser Dissertation anknüpfende, Arbeiten.

5.9.1 Koordination der Aktivitätsausführung

Die Koordination der Aktivitätsausführung, d. h. die Weitergabe der Kontrolle über die Prozessausführung zwischen PS-Klienten, erfolgt ausschließlich entlang der in der EWFN-Repräsentation eines Prozesses explizit spezifizierten Kontrollflusskanten. Bedingt durch die in Abschnitt 5.6.1.3 vorgestellte Struktur der Kontrollflusstupel gilt, dass diese durch Identifikatoren mit der Kanten, entlang welchen sie produziert werden, in Bezug stehen. Diese Identifikatoren bilden die Basis für die Formulierung der Templates konsumierender PS-Operationen. Ein EWFN ist deterministisch insofern, als dass zu jedem Zeitpunkt der Prozessausführung ein existierendes Kontrollflusstupel jeweils nur durch genau einen PS-Klienten konsumiert werden kann. Konkurrierende Zugriffe mehrerer PS-Klienten um dieselben Kontrollflusstupel, und damit verbundene konfliktierende Templates, sind somit ausgeschlossen.

Die einzelnen EWFN-Pattern spezifizieren eine überwiegend sequentielle Kontrollflussweitergabe[1]. Die Realisierung einer rein sequentiellen Koordination von PS-Klienten bedingt die Erzeugung eines entsprechenden Tupels auf Seite der vorangehenden PS-Klienten und das Konsumieren des Tupels auf Seite des nachfolgenden Klienten. Hinsichtlich der Komplexität der Koordination der Ausführung von PS-Klienten gilt damit eine vergleichbare Komplexität zum Koordinationsaufwand in MOM-basierten Anwendungen. Ausnahmen hiervon, also Aktivitäten, deren realisierender PS-Klient in seiner Eingangsbedingung mehr als ein Tupel erwartet, sind Klienten, die mehrere parallele Ausführungspfade eines Prozesses synchronisieren. Beispiele hierfür sind die Transition $t1$

[1] Da die EWFN-Repräsentation eines Prozesses sämtliche möglichen alternativen Ausführungspfade einer Aktivitätsausführung beinhaltet, handelt es sich bei der Mehrzahl anscheinend paralleler Ausführungspfade im EWFN tatsächlich um sich gegenseitig ausschließende Ausführungspfade.

im LinkTarget-EWFN, $t2$ bei Pick, If, EventHandler und FaultHandler, $t3$ bei Compensate und im TerminiationHandler und $t5$ bei HandleTermination. Die Adressierung dieser Fälle erfolgt durch die *sync*-Operation auf Ebene der Infrastruktur (vgl. Abschnitt 5.3.1) oder durch das *sync*-Pattern [MWL08c] auf Ebene des EWFN-Modells.

Explizite Synchronisation der Ausführung von PS-Klienten durch gegenseitigen Ausschluss mittels sogenannter *Mutex-Tupel* erfolgt lediglich an zwei Stellen: (i) zur Sicherstellung der korrekten Instantiierung von Prozessen und (ii) zur Realisierung einer korrekten Ausführungssemantik bei der Ausführung von *Isolated Scopes*. Hinsichtlich der Instantiierung von Prozessen verlangt die BPEL-Spezifikation die Sicherstellung, dass sich jeweils nur eine instanzerzeugende Receive- oder Pick-Aktivität in Ausführung befinden darf. Realisiert wird dieser gegenseitige Ausschluss durch ein, von sämtlichen instanzerzeugenden Aktivitäten eines Prozesses, gemeinsam genutztes Mutex-Tupel, welches als Teil der Eingangsbedingung entsprechender Aktivitäten destruktiv konsumiert wird. Als Teil der Ausgangsbedingung der Aktivitäten wird dieses wieder veröffentlicht und steht damit anderen Aktivitäten zum Konsumieren zur Verfügung. Die Ausführung paralleler *Isolated Scopes* muss laut BPEL-Spezifikation in einer Weise erfolgen, die hinsichtlich konkurrierender Instanzdatenzugriffe der einer seriellen Ausführung entspricht. Die Synchronisation dieser Zugriffe erfolgt durch ein, von den entsprechenden *Isolated Scopes* gemeinsam genutztes, variablenspezifisches Mutex-Tupel (vgl. Abschnitt 5.6.2.2).

5.9.2 Instanzdaten

Interessant für die Diskussion des Laufzeitverhaltens der EWFN-Ausführung ist neben der Koordination der PS-Klienten auch deren Zugriff auf Instanzdaten (aus potentiell unterschiedlichen Domänen); in der Folge wird hierfür der Begriff *verteilter (Instanz-) Zustand* bzw. *Shared Instance State* verwendet. Dabei werden abhängig davon, in welcher Relation die jeweiligen PS-Klienten zueinander stehen, folgende Arten von *Shared Instance State* unterschieden: *Shared Scope State*, *Shared Partner State*, *Shared Interaction State*, *Shared Accessor State* und *Shared Global State*.

Shared Accessor State sind Variablen (im EWFN-Modell repräsentiert durch den *Variables*-Platz) und Informationen über den Typ aufgetretener Fehler (repräsentiert durch den *Faults*-Platz). *Shared Accessor State* erfordert einen Zugriff sämtlicher PS-Klienten, die das entsprechende Instanzdatum während der Ausführung ihrer Funktionalität verwenden. Im Fall der Variablen sind dies zum einen sämtliche Aktivitäten, welche die jeweilige Variable lesen oder ihren Wert modifizieren, zum anderen die SCOPE-Aktivität, welcher die Variable zugeordnet ist. Im Fall eines Fehler-Datums ist ein Austausch zwischen der Aktivität, die den Fehler ausgelöst hat und deren FAULTHANDLER erforderlich. Wird im CATCH bzw. CATCHALL des FAULTHANDLER ein RETHROW ausgeführt, so greift auch dieses lesend auf die Fehlerinformationen zu.

Bedingt durch die spezifizierte Ausführungssemantik von BPEL erfordert ein Instanzdatenzugriff in BPEL bei jeder Operation auf dem Instanzdatum (auch bei lesenden Operationen) eine Überprüfung, ob der bearbeitende PS-Klient auf dem aktuellen Wert des Instanzdatums operiert. Durch die in Abschnitt 5.6.2.6 vorgestellten Caching-Mechanismen, die *update*- und *destroy*-Funktionen der operativen PS-Schnittstelle sowie die Möglichkeit des selektiven Konsumierens eines Teils eines Instanzdatums (Abschnitt 5.6.2.5), wird zwar eine Senkung der zwischen PS-Server und PS-Klient ausgetauschten Datenmenge erreicht, die Notwendigkeit der Überprüfung der *Freshness* eines Instanzdatums (welche jeweils eine Interaktion zwischen PS-Klient und dem PS, in welchem das jeweilige Instanzdatum abgelegt ist, erfordert) muss allerdings bei jedem Instanzdatenzugriff bestehen bleiben.

Um einen negativen Einfluss auf das Laufzeitverhalten von Prozessen mit *Shared Accessor State* so weit wie möglich zu minimieren, sind sowohl die erwartete Anzahl an Ausführungen einer Aktivität (und damit auch die Durchführung der mit der Ausführung der Aktivitäten verbundenen Variablenzugriffe) als auch die erwartete durchschnittliche Größe eines Instanzdatums Einflussgrößen des in Abschnitt 4.4 vorgestellten Verfahrens zur Partitionierung von BPEL-Prozessen. In Verbindung mit dem

Ziel des Verfahrens führt dies zu einer möglichst hohen *Lokalität*, d. h. einer relativen "Nähe" eines Instanzdatums zu den Aktivitäten, die dieses nutzen. Auf Grundlage der Annahme, dass domäneninterne Kommunikation – und damit unter Umständen sogar lokale Interaktionen auf einem physikalischen Rechner – im Allgemeinen (sowohl hinsichtlich Latenz als auch Bandbreite) effizienter als domänenübergreifende Kommunikation ist, resultiert dies in einem relativ effizienten Instanzdatenzugriff.

Auch auf Ebene des BPEL-Modells eines Prozesses kann zur Minimierung der negativen Auswirkung von *Shared Accessor State* beigetragen werden. So gilt beispielsweise, dass globale Variablen mit vielen Zugriffen aus unterschiedlichen Domänen soweit möglich zu vermeiden sind. Granularität der Instanzdatenpartitionierung ist ein Instanzdatum. Wenige Variablen haben somit eine Einschränkung der möglichen Freiheitsgrade des Partitionierungsverfahrens zur Folge. Werden hingegen Variablen lokal, entsprechend ihres Verwendungsbereichs, deklariert und verwendet, so ist eine, hinsichtlich eines effizienteren Laufzeitverhaltens, bessere Partitionierung möglich.

Auch auf Ebene der BPEL-Sprachspezifikation besteht durch Lockerung der verlangten Laufzeitgarantien die Möglichkeit einer Verbesserung des Laufzeitverhaltens. So könnte beispielsweise eine Annotation von Aktivitäten (z. B. SCOPE-Aktivitäten) dergestalt erfolgen, dass für sämtliche lesende Instanzdatenzugriffe eines Klienten in einem bestimmten Zeitintervall keine *Freshness*-Überprüfung stattfinden muss. Die Erforschung der sich durch Änderungen an der Ausführungssemantik von BPEL ergebenden Optimierungspotentiale bietet Raum für zukünftige Arbeiten im Bereich der dezentralen Ausführung von Geschäftsprozessen auf Basis des EWFN-Metamodells und der in dieser Arbeit vorgestellten Laufzeitinfrastruktur.

Shared Interaction State sind Instanzdaten, die von mehreren PS-Klienten gemeinsam genutzt werden müssen, um bestimmte Interaktionsaktivitäten ausführen zu können. Dies sind Informationen, die den aktuellen Wert eines PARTNERLINK (im EWFN-Modell durch den *PartnerLinks*-Platz

repräsentiert) beschreiben oder zur Korrelation von Nachrichten (repräsentiert durch den *Correlations*-Platz) verwendet werden.

Hinsichtlich der PARTNERLINK-Information gilt, dass diese von sämtlichen Interaktionsaktivitäten gemeinsam verwendet werden müssen, die eine Interaktion über den jeweiligen PARTNERLINK ausführen. Weiterhin gilt dies für alle ASSIGN-Aktivitäten, deren Quelle oder Ziel der jeweilige PARTNERLINK ist.

Hinsichtlich der CORRELATIONSET-Daten gilt, dass diese zwischen allen PS-Klienten, die eine Interaktion unter Verwendung des jeweiligen CORRELATIONSET ausführen, gemeinsam verwendet werden müssen. Dabei gilt weiterhin, dass, wenn ein CORRELATIONSET einmal mit einem bereits initialisierten Wert gelesen wurde, sein Wert nicht mehr geändert werden darf. Folglich kann dieses bei mehrfacher Abfrage durch denselben Klienten aus dessen klientenseitigem Zwischenspeicher bedient werden. Realisiert wird dies auf Ebene der Infrastruktur durch Markierung des CORRELATIONSET-Datentupels durch den Wert `true` im U-CONST-Feld.

Shared Partner State beschreibt Daten, die von potentiell sämtlichen PS-Klienten einer Domäne gemeinsam verwendet werden. Eine Notwendigkeit eines Domänen-externen Zugriffs besteht jedoch nicht. Aus Gründen der effizienten domäneninternen Zugriffsmöglichkeit (im Vergleich zum domänenübergreifenden Zugriff) hat *Shared Partner State* nur relativ geringe Auswirkungen auf das Prozesslaufzeitverhalten.

Shared Parter State sind Zustandsinformationen über (i) aktuell in Ausführung befindliche RECEIVE-Aktivitäten (im EWFN durch den *ReceiveStates*-Platz repräsentiert) und (ii) "offene" Interaktionen mit WfMS-externen Kommunikationspartnern (repräsentiert durch den *IMAStates*-Platz).

Informationen über in Ausführung befindliche RECEIVE-Aktivitäten werden zur Sicherstellung der BPEL-Eigenschaft verwendet, wonach sich zu keinem Zeitpunkt mehr als eine Aktivität zum Nachrichtenempfang mit derselben Kombination aus PARTNERLINK, OPERATION und CORRELATIONSET in Ausführung befinden darf. Da ein PARTNERLINK immer an einen

bestimmten (wenn auch über den Zeitraum der Ausführung einer Prozessinstanz variablen) Endpunkt gebunden ist (welcher wiederum einer bestimmten Domäne zugeordnet ist) gilt, dass alle potentiell zueinander in Konflikt stehenden Aktivitäten in derselben Domäne liegen müssen. Um diesen Fall auszuschließen genügt eine domäneninterne Verwaltung der *ReceiveState*-Informationen.

Aufgrund Eigenschaft 4.8, die eine Zuordnung von RECEIVE/PICK- und zugehörigen REPLY-Aktivitäten zu derselben Partition erfordert, wenn die Interaktion in synchronem Kommunikationsmodus erfolgt, gilt hinsichtlich dieser ein entsprechendes Verhalten.

Shared Scope State bezeichnet Informationen, die zur Instanzlaufzeit (i) von SCOPE-Aktivitäten und den zugehörigen Handler-Aktivitäten bzw. (ii) von *Peer Scopes* gemeinsam verwendet werden müssen.

Für die Realisierung der korrekten Installation der COMPENSATIONHANDLER und dem Deaktivieren der EVENTHANDLER muss diesen Aktivitäten der Zustand ihrer zugehörigen SCOPE-Aktivität bekannt sein; im EWFN-Modell wird dieser durch den *Scope State*-Platz beschrieben.

Die Umsetzung der *Default Compensation Order* erfordert weiterhin die Dokumentation des Zustands von *Peer Scope*-Aktivitäten (insbesondere die Beendigung von deren Ausführung) in einem PS (vgl. Abbildung 5.23).

Shared Global State bezeichnet Informationen, die von sämtlichen PS-Klienten eines Prozesses gemeinsam verwendet werden. Die einzige Information dieser Klasse ist die EWFN-Stelle *ProcessMetadata*, über welche allen an der Ausführung eines Prozesses teilnehmenden PS-Klienten Informationen über das Prozessmodell zugänglich gemacht werden. Für das Laufzeitverhalten eines EWFN spielt die *ProcessMetadata*-Stelle keine Rolle; realisiert wird diese entsprechend der Beschreibung in Abschnitt 5.8, durch die Veröffentlichung der entsprechenden Daten im Konfigurations-PS der jeweiligen Domäne.

5.9.3 Bewertung der durch den Ansatz erreichten Autonomie

Ein Nebeneffekt der dezentralen Ausführung von BPEL unter Verwendung des EWFN-Modells [Mar10] und der entwickelten Laufzeitinfrastruktur ist die Entkopplung der PS-Klienten durch die als Kommunikationsplattform verwendeten PS und die hieraus resultierende Autonomie der einzelnen Ausführungsteilnehmer.

Durch die in den Abschnitten 4.2.2.1 und 5.2.2 erläuterten Verfahrensweisen zur Dokumentation (i) der von den einzelnen Ausführungsteilnehmern angebotenen Dienste und (ii) der von diesen eingebrachten PS-Klienten, steht den Ausführungsteilnehmern ein breites Spektrum hinsichtlich der Granularität der Dokumentation der von diesen angebotenen Ausführungsumgebung zur Verfügung. So ist es beispielsweise möglich, dass die Dienstdokumentation ausschließlich auf der Ebene funktionaler Diensteigenschaften erfolgt. Dies erlaubt es einem Domänenbetreiber, lediglich die Information bereitzustellen, dass er einen bestimmten Dienst für die Prozessausführung auf Basis der PS-Infrastruktur anbietet. Ebenso kann hinsichtlich der Beschreibung von PS-Klienten verfahren werden. Auch in diesem Fall reicht das Spektrum von keinerlei Dokumentation der betriebenen PS-Klienten nach außen (was bedeutet, dass die Deployment-Komponente des PS-Servers der Domäne nach Übermittlung des DDD die PS-Klienten für die Ausführung der einzelnen im DDD spezifizierten Aktivitäten im Rahmen eines Domänen-internen Deployment-Schritts bestimmt) bis hin zu vollständiger Dokumentation sämtlicher PS-Klienten, was zur Folge hat, dass diese Informationen bereits zum Partitionierungszeitpunkt mit einbezogen werden können.

Wird eine möglichst weitreichende Autonomie der einzelnen Domänen angestrebt, so ist ein Deployment zu bevorzugen, das den Deployment-Komponenten der einzelnen Domänen größtmögliche Freiheit in der konkreten Auswahl von PS-Klienten lässt. Änderungen der lokalen Dienst-Infrastruktur in einzelnen Domänen (sogenannte *Changes in Environment* [HSB98, Kar06]) können somit allein durch Rekonfiguration der PS-Klienten innerhalb einer Domäne kompensiert werden, so dass von außerhalb der Domäne keine Änderungen sichtbar sind.

5.10 Zusammenfassung

In den vorangehenden Abschnitten wurde die PS-Infrastruktur vorgestellt, ein verteiltes WfMS zur dezentralen Ausführung von Geschäftsprozessen in BPEL. Grundlage für die Beschreibung der PS-Infrastruktur bildete eine Beschreibung des zugrunde liegenden Szenarios der Ausführung von Produktionsprozessen in BPEL und die Identifikation einer Reihe entsprechender, sowohl funktionaler als auch nicht-funktionaler Anforderungen an das System. Hieran anknüpfend wurde zunächst eine grobe Übersicht über die Elemente der, an die Konzepte der Tuplespaces angelehnte, PS-Infrastruktur gegeben und deren Beziehung zu den Elementen des EWFN-Metamodells [Mar10] erläutert. Im Anschluss wurde die Bedeutung und Funktion der einzelnen Komponenten im Detail vorgestellt. Neben der Beschreibung der Realisierung der Elemente der PS-Infrastruktur wurden sowohl unterschiedliche Aspekte der Ausführung von BPEL-Prozessen – wie z. B. der Aufbau der während der Prozessausführung kommunizierten Tupel, die Verarbeitung auftretender Fehler und die Realisierung der Kompensationsmechanismen von BPEL – als auch angeschlossene Bereiche, wie die Protokollierung und Überwachung ausgeführter Prozessinstanzen und das Prozess-Deployment erläutert. Den Abschluss der Beschreibung der PS-Infrastruktur bildeten die Diskussion der durch den vorgestellten Ansatz und die zugehörige Ausführungsinfrastruktur erreichbaren Verteilungsgrad der Prozessausführung sowie die Bewertung der Autonomie der Ausführungsteilnehmer.

KAPITEL 6

TUPLESPACE-BASIERTE KOMMUNIKATION ZWISCHEN WEB-SERVICES

Aufgrund der funktionalen und nicht-funktionalen Eigenschaften der in Kapitel 5 vorgestellten PS-Infrastruktur eignet sich diese nicht nur als Laufzeitumgebung für die dezentrale Ausführung von BPEL-Prozessen, sondern kann darüber hinaus auch als Kommunikationsplattform für die Interaktion von Web-Services verwendet werden. Begünstigt wird dieser Anwendungsfall weiterhin durch den hohen Entkopplungsgrad Tuplespace-basierter Kommunikation in den Dimensionen Raum, Zeit und Referenz [Ley06, MWSL07]. In den folgenden Abschnitten wird ein sogenanntes *Web-Service-Binding für Tuplespaces* [WML08a, WML08c] vorgestellt, welches – aufbauend auf die Web-Service-Standards SOAP und WSDL – ein Verfahren zur Tuplespace-basierten Kommunikation zwischen Web-Service-Nutzern und Web-Service-Anbietern erlaubt.

Ein Ziel des entwickelten Bindings ist die Umsetzbarkeit auf einer möglichst großen Anzahl existierender Implementierungen des Tuplespace-Konzepts. Aus

diesem Grund stützt sich das Binding lediglich auf die Operationen *in*, *out* und *rd* der traditionellen Linda-Schnittstelle [Gel85]. Aus Gründen der Konsistenz der Erläuterungen in diesem Abschnitt mit denen der vorangehenden Abschnitte, werden allerdings die Operationsnamen von *JavaSpaces* [sun03] verwendet. Da die PS-Infrastruktur die Linda-Operationen mit entsprechender operationaler Semantik implementiert, ist eine Umsetzung des Bindings auf dieser ohne zusätzliche Erweiterungen möglich (vgl. Abschnitt 7.4).

Die Struktur des Kapitels ist wie folgt: in Abschnitt 6.1 wird zunächst diskutiert, welche Aspekte der Tuplespace-Technologie deren Einsatz als Kommunikationsplattform für Web-Service-Interaktionen motivieren und wie sich diese zu existierenden Binding-Implementierungen unterscheiden. In Abschnitt 6.2 wird eine Abbildung von SOAP-Nachrichten auf Tupel beschrieben, die deren Tuplespace-basierte Übermittlung zwischen Nachrichtensender und -empfänger erlaubt. In Abschnitt 6.3 wird erläutert, wie unterschiedliche, sowohl standardisierte [CGM+04, CHL+07], als auch komplexere, nicht-standardisierte MEP [WCL+08, Sch07] unter Verwendung der Linda-Operationen und der in Abschnitt 6.2 definierten Tupelstruktur realisiert werden können. Abschließend ist die Erläuterung der für die Unterstützung des Tuplespace-Binding in den WSDL-Beschreibungen der Dienste notwendigen Informationen Gegenstand von Abschnitt 6.4.

6.1 Eigenschaften Tuplespace-gestützter Kommunikation

Das *SOAP Messaging Framework* [GHM+07a] (vgl. Abschnitt 2.1.4), eine der Basistechnologien unter den etablierten Web-Service-Standards [WCL+05], definiert (i) ein standardisiertes Nachrichtenformat für Interaktionen zwischen Web-Service-Anbietern und Web-Service-Nutzern, (ii) ein Regelwerk das die Verarbeitung von Nachrichten reglementiert und (iii) einen Mechanismus, welcher es erlaubt, festzulegen, wie SOAP-Nachrichten über verschiedene (Netzwerk-) Transportprotokolle zwischen Nachrichtensender und Nachrichtenempfänger ausgetauscht werden können, die sogenannten *SOAP Bindings* (vgl. Abschnitt 2.1.4).

Ein gegenwärtig weit verbreitetes Transportprotokoll für SOAP-Nachrichten

ist das *Hypertext Transfer Protocol (HTTP)* [GHM+07b, FGM+99]. Obwohl dieses aufgrund relativ einfachen Deployments und geringen Infrastrukturannahmen für ein breites Spektrum an möglichen Einsatzgebieten gut geeignet ist[1], weist es in einem Einsatz für die Kommunikation zwischen Web-Services auch einige Nachteile auf; diese werden im Folgenden erläutert.

Interaktionen über HTTP sind *synchron*, d. h. eine potentielle Antwortnachricht des Adressaten einer Nachricht wird in der Regel über denselben, vom Sender der ursprünglichen Nachricht aufgebauten, Kommunikationskanal verschickt, über den auch die Anfragenachricht übermittelt wurde. Ein Initiieren einer Verbindung seitens des Servers ist aufgrund oben genannter Infrastrukturannahmen im Allgemeinen nicht möglich bzw. hat Auswirkungen auf die Konfiguration der Netzwerkverbindung.

Selbst in Fällen, in welchen HTTP in einem asynchronen Interaktionsmodus verwendet wird, d. h. der Sender einer Antwort-Nachricht diese in Form einer eigenen HTTP-Anfrage an den Dienstnutzer übermittelt, gelten Einschränkungen hinsichtlich *zeitlicher* und *referentieller* Kopplung [AADH05].

Unter zeitlicher Kopplung versteht man die Notwendigkeit, dass Sender und Empfänger einer Nachricht für das Zustandekommen eines Nachrichtenaustauschs gleichzeitig verfügbar sein müssen. Dies ist bei HTTP eine Folge der direkten und ungepufferten Nachrichtenübermittlung zwischen Klient und Server unter Verwendung des *Transmission Control Protocol (TCP)* [Pos81b]. Ist zum Zeitpunkt des Versendens einer Nachricht der Empfänger nicht zum Nachrichtenempfang bereit, so kann die Nachricht nicht zugestellt und demzufolge auch nicht durch diesen verarbeitet werden.

Unter referentieller Kopplung versteht man, dass der Absender einer Nachricht direkt den Adressaten der Nachricht als deren Ziel verwendet; im Fall einer Interaktion über HTTP ist dies ein *Uniform Resource Locator (URL)* [BLMM94]. Ein Mechanismus für referentielle Entkopplung ist beispielsweise MOM, in welcher eine Nachricht nicht direkt an einen bestimmten Empfänger gerichtet

[1] Bei Nutzung von HTTP als Transportprotokoll gelten relativ geringe Anforderungen an einerseits die verwendete Middleware (ein HTTP-Prozessor der die `POST`-Methode implementiert) und andererseits an die Netzwerkverbindung zwischen Dienstanbieter als auch Dienstnutzer. Für die Abwicklung einer *Request-Response*-Interaktion ist beispielsweise lediglich der Nachrichtenempfang auf einem TCP-Port des Dienstanbieters erforderlich.

ist, sondern an einen Kanal in Form einer *Queue* oder eines *Topic*. Enge referentielle Kopplung hat zur Folge, dass ein Dienst – und dementsprechend auch sämtliche Nutzer dieses Diensts – immer an eine bestimmte Adresse gebunden sind. Ändert sich, beispielsweise bedingt durch infrastrukturelle Änderungen beim Dienstanbieter, die Adresse des Diensts, so ist gleichfalls eine Anpassung sämtlicher Klienten erforderlich, die diesen Dienst nutzen. Da Dienste oft von einer Vielzahl unterschiedlicher Klienten aus unterschiedlichen administrativen Domänen genutzt werden, ist eine derartige Anpassung aller Klienten im Allgemeinen nicht praktikabel.

Um die oben beschriebenen Nachteile einer engen zeitlichen und referentiellen Kopplung bei synchronen Transportprotokollen wie HTTP zu eliminieren, existieren eine Reihe von Web-Service-Binding-Implementierungen, die sich zur Kommunikation auf nachrichtenbasierte Protokolle stützen; Beispiele hierfür sind unter anderem SMTP [Kle08], XMPP [SA04] oder JMS [sun08]. Alle diese Transportprotokolle basieren auf dem aus der Netzwerktechnik bekannten *Store-and-Forward*-Prinzip [Com08, Tan02]. Diesem zufolge wird eine Nachricht, die an einen bestimmten Empfänger gerichtet ist, zunächst einem Mittelsmann übergeben. Dieser nimmt die Nachricht entgegen, speichert sie zwischen und gibt sie anschließend entweder an den Adressaten der Nachricht oder aber einen weiteren Mittelsmann weiter. Durch die Nachrichtenübermittlung über potentiell mehrere Mittelsmänner (welche eine zu übermittelnde Nachricht bei Nicht-Verfügbarkeit des nachfolgenden Kommunikationspartners so lange zurückstellen können, bis dieser wieder verfügbar ist), in Verbindung mit der Beschreibung des Adressaten durch den logischen Identifikator seines Kanals, wird eine Entkopplung von Sender und Empfänger sowohl in zeitlicher als auch in referentieller Sicht erreicht.

Tuplespaces sind MOM in vielerlei Hinsicht ähnlich [Ley06, MWSL07, FKLT07, AADH05]: Tuplespace-basierte Kommunikation ist (i) inhärent asynchron, (ii) zeichnet sich durch eine relativ schwache Kopplung zwischen Absender und Empfänger hinsichtlich der drei oben genannten Kopplungsdimensionen aus, und bietet (iii) aufgrund der Möglichkeit des nicht-destruktiven Konsumierens und der Template-basierten Adressierung eine Grundlage für Unterstützung erweiterter *Message Exchange Pattern (MEP)* (z. B. *One-To-Many* Interaktionen

im *Master-Worker*-Stil [FHA99]) direkt auf der Transportschicht.

Während in MOM Absender und Empfänger Nachrichten über explizit festgelegte (und durch logische Bezeichner identifizierte) Queue- oder Topic-Kanäle austauschen, findet bei Tuplespace-basierter Kommunikation Datenaustausch durch Publizieren und Konsumieren von Daten über einen, von allen Teilnehmern der Interaktion (und potentiell weiteren Klienten) gemeinsam genutzten, Tuplespace statt. Durch die in Tuplespaces verwendete Form von assoziativer Adressierung (d. h. der Beschreibung der zu konsumierenden Daten durch ein Template) können mehrere unterschiedliche und voneinander unabhängige Interaktionen über denselben Tuplespace abgewickelt werden. Man spricht in Verbindung mit Tuplespace-gestützter Kommunikation auch von sogenannter *Pull*-Adressierung [MWSL07], d. h. der Empfänger beschreibt die Beschaffenheit der Daten die er konsumieren will und "zieht" diese aus dem jeweiligen Tuplespace. Dies steht im Gegensatz zum *Push*-Verfahren, welches MOM-basierter Kommunikation zugrunde liegt, d. h. der Absender einer Nachricht "drückt" diese in den jeweiligen Kanal des Empfängers. Diese "ungerichtete" Publikation von Daten im Fall Tuplespace-basierter Kommunikation führt dazu, dass ein veröffentlichtes Tupel prinzipiell allen, am Tuplespace hörenden, Nachrichtenkonsumenten gleichermaßen zur Verfügung steht, was auch den Absender der Nachricht mit einschließt. Dies ermöglicht die relativ einfache Realisierung einer Reihe von MEP, deren Umsetzung unter Verwendung *Push*-basierter Kommunikationstechnologien komplexer ist. Beispiele hierfür sind das *Request-for-bid*- oder das *Update*-MEP. Die Realisierung entsprechender MEP, die diese Eigenschaft von Tuplespaces (sowie die Möglichkeit zum nicht-destruktiven Konsumieren von Nachrichten) nutzen, werden in Abschnitt 6.3.6 bzw. 6.3.7 detailliert vorgestellt.

Eine weitere, dem MOM ähnliche Eigenschaft von Tuplespaces ist inhärente Unterstützung des sogenannten *Replicated-Worker*-Pattern [FHA99], demzufolge eine Nachricht an eine Reihe potentieller Empfänger gerichtet wird, die um diese Nachricht konkurrieren, wobei jede Nachricht aber jeweils genau einmal verarbeitet wird; in [HW04] wird dies als das *Competing-Consumers*-Pattern bezeichnet. Im Fall einer hohen Auslastung einer Implementierung eines bestimmten Diensts erlaubt dies die einfache Lastverteilung durch Starten

weiterer Dienstimplementierungen, die dieselben Templates spezifizieren und denselben Tuplespace auf eingehende Anfragen hin überwachen.

Neben den oben erläuterten funktionalen Eigenschaften der Tuplespaces begünstigen auch die nicht-funktionalen Eigenschaften einiger Tuplespace-Implementierungen, wie transaktionales Verhalten und persistente Vorhaltung publizierter Tupel, eine Verwendung eines Tuplespace als Plattform für die Abwicklung von Web-Service-Interaktionen.

6.2 Aufbau der Tupel zur Kapselung von SOAP-Nachrichten

Die Übermittlung von SOAP-Nachrichten, sogenannter *SOAP-Envelopes* (vgl. Abschnitt 2.1.4), über ein bestimmtes Transportprotokoll erfordert unter anderem eine Serialisierung des *SOAP-Information-Sets*. Für diese Serialisierung müssen die folgenden Eigenschaften gelten: (i) Sie muss über das gewählte Transportprotokoll übertragen und beim Empfänger wieder in die ursprünglich versendete Nachricht rekonstruiert werden können. (ii) Weiterhin muss es den beteiligten Web-Service-Laufzeitumgebungen möglich sein, aus der gewählten Serialisierung sämtliche Informationen zu erhalten, die für die Verarbeitung einer Nachricht während ihrer Übermittlung notwendig sind.

Im Fall von Tuplespace-gestützter Übermittlung von SOAP-Nachrichten bedeutet dies, dass eine zu übermittelnde SOAP-Nachricht in Form eines Tupels gekapselt wird. Ein derartiges *SOAP-Tupel* beinhaltet – neben der übermittelten SOAP-Nachricht selbst – eine Reihe zusätzlicher Felder, welche für die Übermittlung der Nachricht notwendigen Informationen (z. B. die Adressierung des Nachrichtenempfängers) beinhalten, um diese für die Template-Matching-Mechanismen der Tuplespace-Implementierung verwertbar zu machen. Bestimmt werden diese Informationen entweder durch die Tuplespace-Binding-Implementierung der jeweiligen Web-Service-Laufzeitumgebung oder durch Extraktion aus dem Envelope der übermittelten SOAP-Nachricht (präzise dessen *SOAP-Header*). Die Felder eines derartigen SOAP-Tupels sind in Tabelle 6.1 dargestellt.

Aufgrund der Eigenschaft der Template-basierten Kommunikation können mehrere Web-Services denselben Tuplespace für die Interaktion mit ihren

Feld	Typ	Beschreibung
B-DESTINATION	URI	Identifikator des adressierten Diensts
B-ACTION	URI	Identifikator der *SOAP-Action* der durch das Tupel gekapselten SOAP-Nachricht
B-MSGID	String	Identifikator der Nachricht
B-MEP	URI	Identifikator des MEP welchem die Nachricht zuzuordnen ist
B-CORID	String	Identifikator einer korrelierenden Nachricht
B-RTYPE	URI	Beziehung zur korrelierenden Nachricht
B-VERSION	URI	Verwendete Version des Bindings
B-ISFAULT	URI	Identifiziert eine Nachricht als *SOAP-Fault*
B-CONTENT	String	SOAP-Envelope
B-CONTENTTYPE	String	Verfahren zur Serialisierung von B-CONTENT

Tabelle 6.1: Aufbau der Tupel für die Kommunikation zwischen den Teilnehmern einer Web-Service-Interaktion.

Kommunikationspartnern nutzen. Um es einem Dienstnutzer auch in diesem Fall zu erlauben, einen bestimmten Dienst zu adressieren, identifiziert das Feld B-DESTINATION eines SOAP-Tupels den aufzurufenden Dienst in Form einer URI. Dienstanbieter verwenden das Feld B-DESTINATION in ihren Templates für das Konsumieren von an sie gerichteten Dienstaufrufen. Der Wert des B-DESTINATION-Felds entspricht dem Wert des DESTINATION-Elements des *WS-Addressing*-Header-Blocks [GHR06] der übermittelten SOAP-Nachricht. Neben dieser "traditionellen" Bestimmung des Adressaten einer SOAP-Nachricht besteht eine andere, speziell im Fall einer Tuplespace-basierten *Pull*-Kommunikation mögliche, Adressierungsalternative. Diese stützt sich auf die Selektion der zu verarbeitenden SOAP-Nachrichten durch den Web-Service mittels eines Template auf den Nutzdatenanteil der Nachricht (den sogenannten *SOAP-Body*). Da dies allerdings mächtigere Template-Matching-Mechanismen verlangt als die durch die Linda-Schnittstelle angebotene Identitäts- oder Wildcard-Prüfung – wie beispielsweise XPath [CD+99, BBC+07] oder XQuery [BCF+07] – und damit die Auswahl zur Realisierung des Bindings verwendbarer Tuplespace-

Implementierungen einschränken würde, wird diese Art der Adressierung von Kommunikationspartnern in der Folge nicht weiter betrachtet.

Um eine korrekte Verarbeitung von SOAP-Tupeln auf Seite des Nachrichtenempfängers zu ermöglichen (beispielsweise für eine Bestimmung der auszuführenden Operation), beinhaltet das B-ACTION-Feld einen Identifikator der Intention einer Nachricht in Form einer URI. Der Wert des B-ACTION-Felds entspricht dem Wert des ACTION-Elements des *WS-Addressing*-Header-Blocks der übermittelten SOAP-Nachricht.

Jedes SOAP-Tupel ist eindeutig durch den Wert des Felds B-MSGID identifiziert; diese Identifikation kann beispielsweise in Form einer UUID [LMS05] erfolgen. Beinhaltet die eingeschlossene SOAP-Nachricht einen *WS-Addressing*-Header-Block, welcher einen optionalen MESSAGEID-Header enthalten kann, so wird dessen Wert in das B-MSGID-Feld des Tupels propagiert.

Das Feld B-MEP identifiziert das *Message Exchange Pattern*, dem die SOAP-Nachricht im jeweiligen Tupel zugehört in Form einer URI. Mögliche Werte für dieses Feld sind die in der WSDL-2.0-Spezifikation [CGM+04, CHL+07] definierten URIs oder andere URIs für eigene, nicht standardisierte MEP (vgl. Abschnitt 6.3.6 oder 6.3.7 für zwei Beispiele derartiger MEP).

Eine Web-Service-Interaktion kann mehrere Nachrichten umfassen; so besteht beispielsweise das oben genannte *In-Out*-MEP aus einer Anfrage- und einer Antwortnachricht. Um eine Korrelation dieser Nachrichten zu ermöglichen (d. h. sicherzustellen, dass der Sender einer Anfragenachricht auch die der Anfrage entsprechende Antwortnachricht konsumiert), referenziert der Wert des B-CORID-Felds die B-MSGID einer Nachricht, welche "in Relation" zur dieser Nachricht steht. Im Fall einer Interaktion, die dem *In-Out*-MEP folgt, hat das B-CORID-Feld der Antwortnachricht den Wert des B-MSGID-Felds der Anfragenachricht, die die Erzeugung der Antwortnachricht ausgelöst hat.

Die Art der Relation zweier Nachrichten wird durch den Wert des B-RTYPE-Felds qualifiziert. Zulässige Werte für dieses Feld sind abhängig von dem MEP, dem die jeweilige Nachricht zugehört. Für die in der WSDL-2.0-Spezifikation definierten MEP werden die in der WS-Addressing-Spezifikation vordefinierten Werte verwendet. So ist beispielsweise im Fall eines *In-Out*-MEP der B-RTYPE-Wert der Antwortnachricht definiert als `http://schemas.xmlsoap.org/`

ws/2004/03/addressing#reply.

Der Wert des Felds B-VERSION beschreibt die für die Interaktion verwendete Version des Web-Service-Bindings für Tuplespaces; dies ermöglicht eine kontinuierliche Weiterentwicklung der Binding-Spezifikation unter Bewahrung der Abwärtskompatibilität mit bestehenden Implementierungen.

Im Fall, dass die im SOAP-Tupel enthaltene SOAP-Nachricht einen Anwendungsfehler in Form eines sogenannten *SOAP-Fault* darstellt, ist der Wert des B-ISFAULT-Felds ein Boolesches `true`, andernfalls `false`.

Das B-CONTENT-Feld beinhaltet eine Repräsentation des kompletten SOAP-Envelope bestehend aus dessen *SOAP-Header-* und *SOAP-Body*-Anteil.

Das Feld B-CONTENTTYPE identifiziert das zur Serialisierung des Werts des B-CONTENT-Felds verwendete Verfahren. Die zur Serialisierung von SOAP-Nachrichten typischerweise verwendete Repräsentation ist ihre XML-Serialisierung [GHM+07a] in *UTF-8-Encoding*, identifiziert durch den String "application/soap+xml".

6.3 Tuplespace-basierte Umsetzung von Web-Service-MEP

Im Folgenden wird die Umsetzung einer Reihe von Web-Service-MEP unter Verwendung der Operationen der Linda-Schnittstelle vorgestellt. Die vorgestellten MEP umfassen dabei sowohl die, in der WSDL-2.0-Spezifikation beschriebenen, standardisierten MEP [CGM+04, CHL+07] als auch drei Beispiele zusätzlicher MEP [WCL+08], die unter Verwendung der Operationen eines Tuplespace besonders einfach realisiert werden können. Auf eine gesonderte Behandlung der *WSDL 1.1 Operation Types* [CCMW01] wird an dieser Stelle verzichtet, da diese durch die standardisierten WSDL-2.0-MEP wie folgt abgedeckt sind: das WSDL-2.0-MEP *In-Only* entspricht dem *One-Way* Operation-Type in WSDL 1.1, *In-Out* entspricht *Request-Response*, *Out-In* entspricht *Solicit-Respone* und *Out-Only* entspricht dem *Notification* Operation-Type.

Die Beschreibung der Umsetzung der MEP umfasst jeweils zwei Teile: In einem ersten informellen Teil wird die Realisierung des MEP graphisch in Form eines UML-Sequenzdiagramms dargestellt und in Textform erläutert. In einem zweiten Teil wird das jeweilige MEP unter Verwendung des π-Kalkül

formal beschrieben. Die notwendigen Konzepte des π-Kalküls – soweit diese für die Beschreibung der MEP erforderlich sind – werden im Folgenden kurz erläutert, eine detailliertere Darstellung des π-Kalküls findet sich in [Mil99] und [SW03].

6.3.1 Der π-Kalkül und seine Verwendung zur Beschreibung von Web-Service-Message-Exchange-Pattern

Ziel des π-Kalküls ist die Beschreibung der Interaktion von Prozessen [Mil99]. Diese Interaktion erfolgt durch den Austausch von Nachrichten, sogenannten *Namen*, über sogenannte *Kanäle*. Ein Name kann dabei sowohl ein fixes Datum, eine Variable oder einen Kanal selbst beinhalten. Aus der Übertragbarkeit eines Kanals resultiert die sogenannte *Link Passing Mobility* [Puh07]. Diese ermöglicht eine Interaktion zwischen einander vor Beginn der Interaktion unbekannten Kommunikationspartnern, nachdem ein Kanal zwischen diesen durch einen, beiden Partnern bekannten, Vermittler ausgetauscht wurde.

Für die Formalisierung der MEP im π-Kalkül gelten nachfolgend beschriebene Konventionen, welche sich weitestgehend an [Puh07] orientieren:

- Die Menge $\mathcal{K} = \{0, P, R, T, \ldots\}$ ist die Menge der an einer Interaktion beteiligten Prozesse, d. h. der aktiven Interaktionsteilnehmer; sie werden durch Majuskel bezeichnet. Der nicht-interagierende oder abgeschlossene Prozess wird durch das Symbol 0 dargestellt.
- Die Menge $\mathcal{N} = \{n, m, o, p, \ldots\}$ ist die Menge der Namen; bezeichnet durch Minuskel. Falls ein $n \in \mathcal{N}$ einen Kanal identifiziert, gilt folgende Konvention: wird eine Nachricht über Kanal n versendet, so wird dies durch die Notation $\overline{n}$ gekennzeichnet, bei einer Leseoperation wird lediglich der Kanalbezeichner verwendet.
- Sende- und Empfangsaktionen werden durch Klammerpaare gekennzeichnet, wobei im Fall eines Sendens spitze und beim Empfang runde Klammerpaare verwendet werden. Diese umschließen jeweils die gesendeten Namen und folgen auf den Identifikator des Kanals. Bedingt durch das Voranstehen der Kanal-Identifikatoren werden diese auch als

Output Präfix im Fall eines Sendens und *Input Präfix* im Fall eines Empfangs bezeichnet. So wird beispielsweise das Versenden der Nachricht a über den Kanal b durch die Notation $\overline{b}\langle a\rangle$ beschrieben; analog wird $b(a)$ für den Empfang der Nachricht a über einen Kanal b verwendet. Leseoperationen sind dabei immer blockierend. Das bedeutet, dass eine Leseoperation die weitere Ausführung eines Prozesses solange blockiert, bis ein entsprechender Name über den jeweiligen Kanal empfangen wurde. Zur Vereinfachung der Darstellung können die Klammerpaare um gesendete bzw. empfangene Nachrichten weggelassen werden.

- Da der π-Kalkül die Interaktionen eines Prozesses mit seiner Umgebung beschreibt, werden interne Verarbeitungsschritte der Prozesse (d. h. Aktionen, in denen der Prozess nicht mit seiner Umgebung interagiert) nicht betrachtet. Um einen oder mehrere interne Arbeitsschritte zu beschreiben, wird die sogenannte *stille Aktion* τ verwendet.
- Der π-Kalkül erlaubt die Beschreibung einer sequentiellen Ausführungsabfolge der Prozesse P und Q durch die Notation $P.Q$. Parallele Ausführung wird durch die Notation $P|Q$ beschrieben. Synchronisation paralleler Prozesse wird durch (blockierende) Leseoperationen auf Kanälen erreicht. Soll eine alternative Ausführung von Prozessen beschrieben werden, so wird dies durch $P + Q$ dargestellt.
- Eine Einschränkung des Gültigkeitsbereichs eines Namens auf eine Menge von Prozessen wird durch die Notation $(\nu x)P$ erreicht. In diesem Beispiel ist die Gültigkeit von x auf P eingeschränkt. So gilt beispielsweise für den Ausdruck $(\nu x)P.(\nu x)Q$, dass x zwei unterschiedliche, jeweils auf P oder Q eingeschränkte, Namen identifiziert.
- Eine Überprüfung auf Gleichheit von Namen wird durch die Notation $[a = b]P$ dargestellt. Im Fall einer Gleichheit verhält sich der dargestellte Prozess wie P, sonst wie 0. Analoges gilt im Fall der Ungleichheit für $[a \neq b]P$.
- Folgende abkürzende Schreibweisen werden verwendet: eine Abfolge $a(b).a(b).a(b)$ kann durch $\{a(b)\}_1^3$ dargestellt werden. Soll beispielsweise das zweite Element dieser Abfolge ausgelassen werden, so wird die

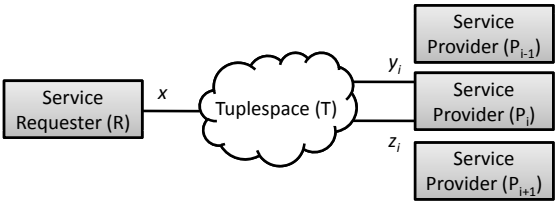


Abbildung 6.1: Benennung der Teilnehmer einer Tuplespace-basierten Web-Service-Interaktion.

Notation $\{a(b)\}_1^{3\backslash 2}$ verwendet. Für $P_1|P_2|P_3$ gilt die Abkürzung $\prod_{i=1}^{3} P_i$.

Für die nachfolgende Beschreibung der Web-Service-MEP gelten weiterhin die folgenden Konventionen. Der Dienstnutzer (im Allgemeinen der Initiator des MEP) wird durch den *Requester*-Prozess R beschrieben. Er interagiert mit einem oder mehreren Dienstanbietern, die durch die *Provider*-Prozesse $P_1, \ldots, P_n$ beschrieben werden. Die Rolle des zwischen R und P_i vermittelnden Tuplespace wird zur Verdeutlichung der Interaktion in den MEP-Formalisierungen ebenfalls explizit beschrieben; dieser wird durch den Prozess T bezeichnet. Die an einer Interaktion beteiligten Rollen sind in Abbildung 6.1 graphisch zusammengefasst. Zur Kommunikation mit dem Tuplespace nutzt R den Kanal x. Ein Dienstanbieter P_i nutzt für die Kommunikation mit dem Tuplespace den Kanal y_i. Der Kanal z_i wird für die Kommunikation von Steuerdaten – z. B. für die Anmeldung (engl. *Subscription*) in *Publish-Subscribe*-Interaktionen – verwendet. In Fällen, in denen ein Anfragesender eine zu seiner Anfrage korrelierende Antwort erwartet, empfängt er diese über einen "privaten" Kanal r, den er als Teil der Anfrage übergibt. Zur Unterscheidung der privaten Kanäle der einzelnen Interaktionsteilnehmer wird zwischen den Kanälen r_R und r_P bzw. r_{P_i} unterschieden.

Für die formale Beschreibung der MEP gilt zusätzlich die Annahme, dass über den Zeitraum einer Instanz des jeweiligen MEP hinweg, dieselben Interaktionspartner über den Tuplespace miteinander kommunizieren, also kein Teilnehmer der Interaktion diese vor deren Ende verlässt.

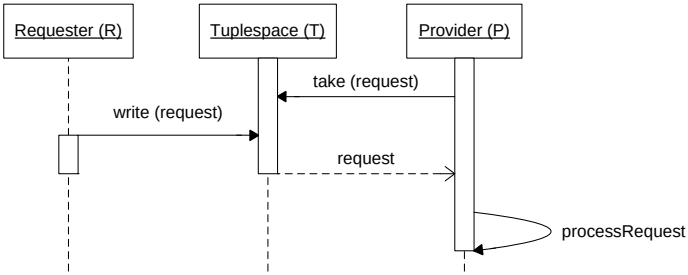


Abbildung 6.2: Graphische Darstellung des *In-Only*-MEP mit den Operationen der Linda-Schnittstelle.

6.3.2 In-Only-MEP

Das in Abbildung 6.2 schematisch dargestellte *In-Only*-MEP besteht aus einem Nachrichtenaustausch zwischen einem Anfragesender und einem Anfrageempfänger. Der Anfragesender richtet seine Anfragenachricht durch die URI im Feld B-DESTINATION an einen bestimmten Anfrageempfänger. Die URI besteht aus einem Protokoll-Präfix, dem Identifikator des Tuplespace, den der Anfrageempfänger auf Nachrichten hin überwacht und dem Identifikator des aufgerufenen Anfrageempfängers (vgl. Abschnitt 6.4).

Der Anfragesender veröffentlicht das SOAP-Tupel in dem vom Anfrageempfänger überwachten Tuplespace unter Verwendung der *write*-Operation. Der Anfrageempfänger spezifiziert in seinem Template seinen B-DESTINATION-Wert und entnimmt das vom Anfragesender geschriebene SOAP-Tupel durch die *take*-Operation. Die Gleichheit der B-DESTINATION-Felder im geschriebenen Tupel auf der einen und im Template auf der anderen Seite führt zur Bildung eines virtuellen Kanals zwischen Sender und Empfänger [MWSL07]. Im Fall des *In-Only*-MEP wird für die Entnahme von SOAP-Tupeln auf der Seite des Anfrageempfängers die destruktive *take*-Operation (und nicht die nichtdestruktive *read*-Operation) verwendet, um sicherzustellen, dass jede Anfrage genau einmal vom Anfrageempfänger konsumiert und durch diesen verarbeitet wird.

Nach Empfang der Nachricht deserialisiert die Web-Service-Binding-Imple-

$$
\begin{aligned}
\mathcal{K} &= \{R, P_i, T\}, 1 \leq i \leq n, n \in \mathbb{N} \\
\mathcal{N} &= \{x, y_i, m, \varepsilon\}, 1 \leq i \leq n, n \in \mathbb{N} \\
R &\stackrel{def}{=} \nu m(\overline{x}\langle m\rangle)).0 \\
T &\stackrel{def}{=} \nu m(x(m).\overline{y_k}\langle m\rangle).\{\overline{y_i}\langle\varepsilon\rangle\}_{i=1}^{n\setminus k}.0, 1 \leq k \leq n, n \in \mathbb{N} \\
P_i &\stackrel{def}{=} \nu m(y_i(m).([m = \varepsilon]0 + [m \neq \varepsilon]\tau_{P_i}.0)) \\
\textit{In-Only}_{Linda} &\equiv R|T|\prod_{i=1}^{n} P_i, n \in \mathbb{N}
\end{aligned}
$$

Abbildung 6.3: Formalisierung des *In-Only*-MEP.

mentierung die im Feld B-CONTENT enthaltene SOAP-Nachricht entsprechend des im Feld B-CONTENTTYPE definierten Serialisierungsverfahrens, bestimmt daraufhin unter Verwendung des Werts des B-ACTION-Felds die aufzurufende Web-Service-Implementierung und führt diese aus.

Abbildung 6.3 zeigt die Formalisierung des *In-Only*-MEP im π-Kalkül entsprechend oben erläuterter Syntaxkonventionen.

Auf eine detaillierte Darstellung und Diskussion des *Out-Only*-MEP wird an dieser Stelle verzichtet, da es sich analog zum *In-Only*-MEP verhält, lediglich die Rollen von Anfragesender R und Anfrageempfänger P sind im *Out-Only*-MEP vertauscht.

6.3.3 In-Out-MEP

Das in Abbildung 6.4 dargestellte *In-Out*-MEP, dessen formale Repräsentation in Abbildung 6.5 beschrieben ist, besteht aus einem bidirektionalen Nachrichtenaustausch. Zunächst sendet der Anfragesender eine Anfragenachricht an den Empfänger; diese Nachrichtenübermittlung erfolgt analog zum *In-Only*-MEP.

Nachdem die Anfragenachricht vom Anfrageempfänger verarbeitet wurde, erzeugt dieser eine entsprechende Antwortnachricht (welche auch einen Fehler auf Anwendungsebene repräsentieren kann). Um die Antwortnachricht in

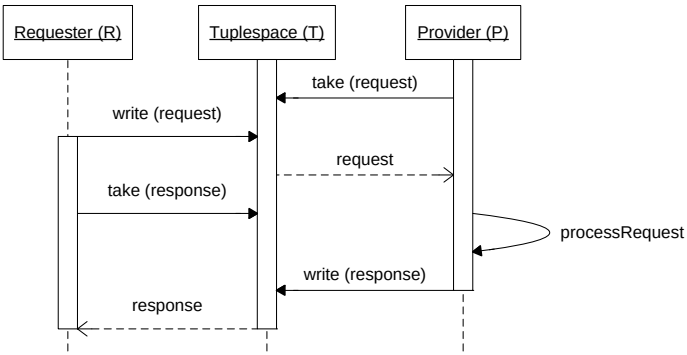


Abbildung 6.4: Graphische Darstellung des *In-Out*-MEP mit den Operationen der Linda-Schnittstelle.

Relation zur ursprünglichen Nachricht des Anfragesenders zu setzen (und es diesem damit zu ermöglichen, die Antwortnachricht als solche zu erkennen), extrahiert der Anfrageempfänger den Wert des Felds B-MsgId (und einen ggf. vorhandenen WS-Addressing *Reply-Endpoint* aus dem *WS-Addressing*-Header-Block der SOAP-Nachricht im Feld B-Content, der den Tuplespace identifiziert, in welchem der Anfragesender eine Antwort auf die von ihm gesendete Anfrage erwartet aus der Anfragenachricht; dieser wird als Wert des B-CorId-Felds des Antwort-Tupels gesetzt.

Zur Identifikation der Nachricht als Antwortnachricht im *In Out*-MEP wird der Wert deren B-RType-Felds auf den Wert `http://schemas.xmlsoap.org/ws/2004/03/addressing#reply` gesetzt [GHR06]. Hinsichtlich der Übermittlung der Antwortnachricht gelten zwei Möglichkeiten. Spezifiziert die Anfragenachricht einen *Reply-Endpoint*, so wird die Antwortnachricht mittels der *write*-Operation in den durch diesen identifizierten Tuplespace publiziert. Wird kein *Reply-Endpoint* explizit spezifiziert, so wird die Antwortnachricht in den Tuplespace publiziert, aus welchem die Anfragenachricht konsumiert wurde.

Der ursprüngliche Anfragesender konsumiert diese Nachricht aus dem Tuplespace durch eine *take*-Operation; das hierfür verwendete Template erwartet

$$
\begin{aligned}
\mathcal{K} &= \{R, P_i, T\}, 1 \leq i \leq n, n \in \mathbb{N} \\
\mathcal{N} &= \{x, y_i, m, r_R, r_{P_i}, q, \varepsilon\}, 1 \leq i \leq n, n \in \mathbb{N} \\
R &\stackrel{def}{=} \nu m \nu q \nu r_R(\overline{x}\langle m, r_R\rangle.r_R(q)).0 \\
T &\stackrel{def}{=} \nu m \nu q \nu r_R \nu r_{P_i}(x(m, r_R).\overline{y_k}\langle m, r_{P_i}\rangle.\{\overline{y_i}\langle\varepsilon, \varepsilon\rangle\}_{i=1}^{n\setminus k}.r_{P_i}(q).\overline{r_R}\langle q\rangle).0, \\
&\quad 1 \leq k \leq n, n \in \mathbb{N} \\
P_i &\stackrel{def}{=} \nu m \nu q \nu r_{P_i}(y_i(m, r_{P_i}).([m = \varepsilon]0 + [m \neq \varepsilon]\tau_{P_i}.\overline{r_{P_i}}\langle q\rangle.0) \\
\textit{In-Out} &\equiv R|T|\prod_{i=1}^{n} P_i, n \in \mathbb{N}
\end{aligned}
$$

Abbildung 6.5: Formalisierung des *In-Out*-MEP.

den Wert des B-MsgId-Felds der ursprünglichen Anfragenachricht als Wert des Felds B-CorId Felds und den entsprechenden Identifikator für eine Antwortnachricht im *In-Out*-MEP als Wert des B-RType-Felds. Da sowohl auf Sender- als auch auf Empfängerseite blockierende Operationen (oder aber, wie oben angedeutet, Publish-Subscribe-Mechanismen) verwendet werden, hat die zeitliche Abfolge der Operationen auf beiden Seiten keine Auswirkungen auf die Ausführung des MEP. Analog zum *In-Only*-MEP werden im *In-Out*-MEP lediglich destruktive Operationen zum Konsumieren von Nachrichten verwendet, da auch hier sowohl Anfrage- als auch Antwortnachricht nur einmal verarbeitet werden sollen.

Abbildung 6.5 zeigt die Formalisierung des *In-Out*-MEP. Im Vergleich zum *In-Only*-MEP übermittelt der Dienstnutzer neben der Anfragenachricht m auch einen Rückkanal r_R über Kanal x an den Tuplespace. Dieser reicht die Anfragenachricht mit einem entsprechenden Rückkanal r_{P_i} an eine Dienstimplementierung über den Kanal y_K weiter und wartet dann auf die Antwort q des Diensts über auf dem Kanal r_{P_i}. Die Antwort leitet T daraufhin über Kanal r_R an R weiter.

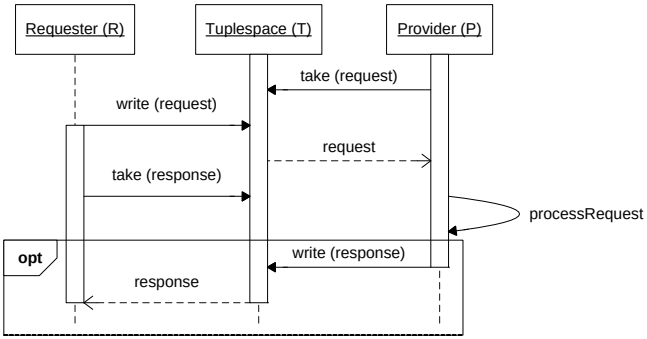


Abbildung 6.6: Graphische Darstellung des *In-Optional-Out*-MEP mit den Operationen der Linda-Schnittstelle.

$$R \stackrel{def}{=} \nu m \nu q \nu r_R(\overline{x}\langle m, r_R\rangle.(r_R(q).0 + 0))$$
$$P_i \stackrel{def}{=} \nu m \nu q \nu r_{P_i}(y_i(m, r_{P_i}).([m = \varepsilon]0 + [m \neq \varepsilon]\tau_{P_i}.(\overline{r_{P_i}}\langle q\rangle.0 + 0)))$$

Abbildung 6.7: Notwendige Änderungen an der Formalisierung des *In-Out*-MEP (Abbildung 6.5) für die formale Beschreibung des *In-Optional-Out*-MEP.

6.3.4 In-Optional-Out-MEP

Das *In-Optional-Out*-MEP folgt im Wesentlichen dem Aufbau des *In-Out*-MEP und wird demzufolge größtenteils analog umgesetzt. Der einzige Unterschied der MEP ist, dass im Fall des *In-Optional-Out*-MEP der Anfrageempfänger nicht in allen Fällen eine Antwortnachricht erzeugen muss, sondern dies "nach Bedarf" entscheidet. Eine graphische Darstellung ist in Abbildung 6.6 dargestellt.

Da sich seine formale Repräsentation nicht wesentlich vom *In-Out*-MEP unterscheidet, werden in Abbildung 6.7 lediglich die Unterschiede des MEP auf Seite des Anfragesenders (R) und des Anfrageempfängers (P) dargestellt.

In diesem MEP kann der Dienstanbieter, der die Anfrage verarbeitet, ent-

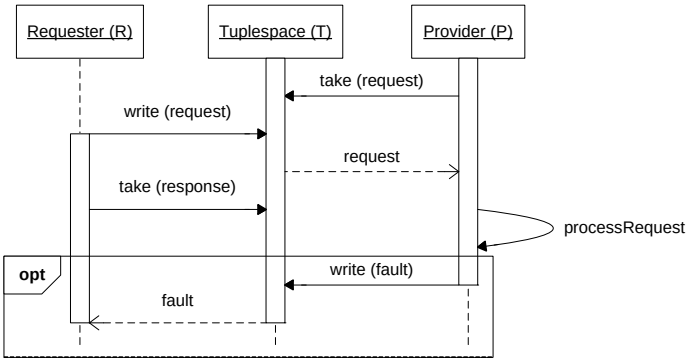


Abbildung 6.8: Graphische Darstellung des *Robust-In-Only*-MEP mit den Operationen der Linda-Schnittstelle.

scheiden, ob er die Verarbeitung der Anfrage mit einer Antwort abschließen möchte ($\overline{r_{P_i}}\langle q\rangle.0$) oder nicht (0). Analog hierzu kann der Anfragesender eine eingehende Antwort verarbeiten ($r_R(q).0$) oder die Interaktion ohne diese abschließen (0).

Entschließt sich P_i gegen ein Übersenden einer Antwortnachricht, so wartet R potentiell unendlich lange auf eine Antwortnachricht. Da die WSDL-2.0-Spezifikation keine Vorgabe eines Warteintervalls vorsieht, muss in diesem Fall (wie auch im Fall von *Robust-In-Only* im folgenden Abschnitt) R das Warten auf eine Antwortnachricht selbst abbrechen, sobald R an dieser nicht mehr interessiert ist.

6.3.5 Robust-In-Only-MEP

Das in Abbildung 6.8 dargestellte *Robust-In-Only*-MEP hat, im Hinblick auf den Nachrichtenaustausch zwischen Anfragesender und Anfrageempfänger betrachtet, exakt denselben Aufbau und dementsprechend dieselbe Realisierung wie das *In-Optional-Out*-MEP.

Das MEP unterscheidet sich lediglich im Typ der Nachricht, die vom Anfrageempfänger an den ursprünglichen Anfragesender zurückgeschickt wird. Während dies im Fall des *In-Optional-Out*-MEP sowohl eine Fehlernachricht

als auch eine “reguläre” Antwortnachricht sein kann, wird im *Robust-In-Only*-MEP lediglich im Fall eines Fehlers auf Seite des Anfrageempfängers eine Fehlernachricht an den ursprünglichen Anfragesender zurückgeschickt. Da sich diese allerdings hinsichtlich der zu ihrer Übertragung verwendeten Tuplespace-Operationen nicht von einer “regulären” Antwortnachricht unterscheidet, ist auch die Formalisierung des MEP identisch zu der des *In-Optional-Out*-MEP (Abbildung 6.7).

6.3.6 One-To-Many- bzw. Replicated-Worker-MEP

Konzeptuell ähnelt das *One-To-Many*-MEP einer Broadcast- bzw. Multicast-Operation [Com08], in welcher eine Nachricht an eine Menge von Nachrichtenempfängern gerichtet wird. Von diesem MEP existieren zwei Varianten: (i) entweder erhalten alle am Tuplespace auf (entsprechende) Anfragenachrichten hörende Anfrageempfänger eine Kopie der Nachricht oder (ii) ausschließlich ein Anfrageempfänger. Der Eindeutigkeit wegen wird in der Folge der Begriff *One-To-Many*-MEP für Variante (i) verwendet, für Variante (ii) der Begriff *Replicated-Worker*- oder *Master-Worker*-MEP[1] [FHA99].

Abbildung 6.9 zeigt die Realisierung des *One-To-Many*-MEP. Analog zum *In-Only*-MEP publiziert hier der Anfragesender eine Anfragenachricht in einen Tuplespace. Im Fall des *One-To-Many*-MEP konsumieren die einzelnen Anfrageempfänger die Anfragenachricht allerdings nicht mit der destruktiven *take*-Operation, sondern mit der nicht-destruktiven *read*-Operation. Folglich steht eine einmal konsumierte Nachricht anderen Anfrageempfängern noch zum (ebenfalls nicht-destruktiven) Konsumieren und damit zur Verarbeitung bereit.

Das *Replicated-Worker*-MEP folgt im Wesentlichen der Realisierung des *One-To-Many*-MEP, zum Konsumieren von Nachrichten auf Seite der Anfrageempfänger wird allerdings die destruktive *take*-Operation verwendet. Dies hat zur Folge, dass eine einmal konsumierte Nachricht nur einmal (durch den

[1] Identifiziert wird das *Replicated-Worker*-MEP durch die URI `http://iaas.uni-stuttgart.de/ns/2009/process-space/replicated-worker` im Feld B-MEP eines SOAP-Tupels; für das *One-To-Many*-MEP gilt die URI `http://iaas.uni-stuttgart.de/ns/2009/process-space/one-to-many`.

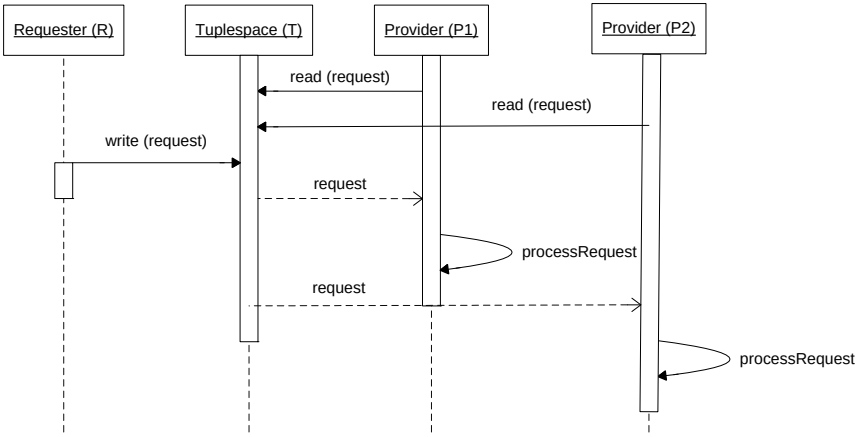


Abbildung 6.9: Graphische Darstellung des *One-To-Many*-MEP mit den Operationen der Linda-Schnittstelle.

$$T \stackrel{def}{=} vm(x(m).\{\overline{y_i}\langle m\rangle\}_{i=1}^{n}).0, 1 \leq k \leq n, n \in \mathbb{N}$$
$$P_i \stackrel{def}{=} vm(y_i(m).\tau_{P_i}).0$$

Abbildung 6.10: Notwendige Änderungen an der Formalisierung des *In-Only*-MEP (Abbildung 6.3) für die formale Beschreibung des *One-To-Many*-MEP.

konsumierenden Anfrageempfänger) verarbeitet wird und anderen Anfrageempfängern damit nicht mehr zur Verfügung steht. Eine alternative Interpretation ist die Beschreibung des *Replicated-Worker*-MEP als eine Sonderform des *In-Only*-MEP, in welchem mehrere Implementierungen desselben Web-Service denselben Tuplespace auf Anfragenachrichten (durch ein Template mit demselben Wert im Feld B-DESTINATION) hin überwachen.

Abbildung 6.10 zeigt die an der Formalisierung des *In-Only*-MEP notwendigen Änderungen zur Beschreibung des *One-To-Many*-MEP.

Die Formalisierung des *Replicated-Worker*-MEP entspricht bei Tuplespace-

basierter Realisierung der des *In-Only*-MEP (Abbildung 6.3). Dies ergibt sich als Folge der Publikation der Nachricht in den Tuplespace, aus welchem diese von sämtlichen Interaktionspartnern konsumiert werden kann.

6.3.7 Request-For-Bid-MEP

Das in Abbildung 6.11 dargestellte *Request-For-Bid*-MEP[1] stellt eine Kombination des *In-Out*- und des *One-To-Many*-MEP dar. Hier konsumieren potentiell mehrere Anfrageempfänger nicht-destruktiv (*read*) die Anfragenachricht eines Anfragesenders (analog *One-To-Many*). Jeder Anfrageempfänger verarbeitet die Anfragenachricht und schickt daraufhin eine Antwortnachricht an den ursprünglichen Anfragesender (analog *In-Out*). Beendet wird das MEP mit der destruktiven Entnahme der Anfragenachricht durch den ursprünglichen Anfragesender.

Die Formalisierung des *Request-For-Bid*-MEP ist in Abbildung 6.12 dargestellt.

[1] Identifiziert wird das *Request-For-Bid*-MEP durch die URI `http://iaas.uni-stuttgart.de/ns/2009/process-space/request-for-bid` im Feld B-MEP eines SOAP-Tupels.

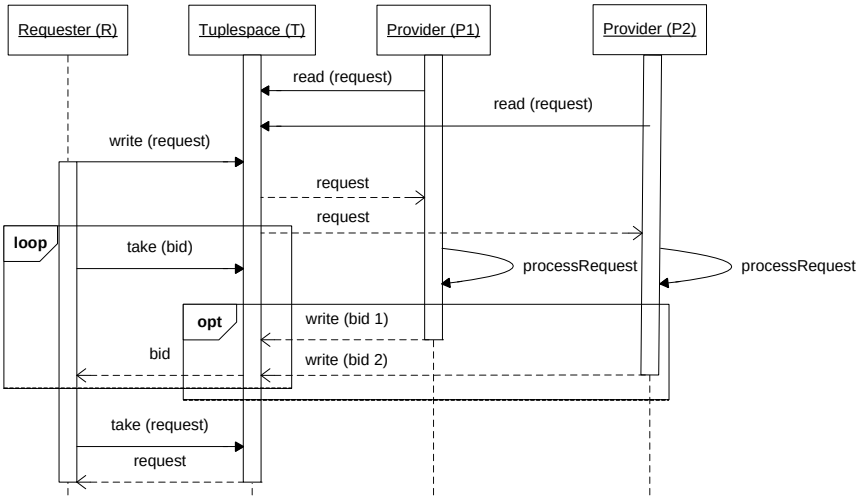


Abbildung 6.11: Graphische Darstellung des *Request-For-Bid*-MEP mit den Operationen der Linda-Schnittstelle.

$$
\begin{aligned}
\mathcal{K} &= \{R, P_i, T\}, 1 \leq i \leq n, n \in \mathbb{N} \\
\mathcal{N} &= \{x, y_i, m, r_R, r_{P_i}, \varepsilon\}, 1 \leq i \leq n, n \in \mathbb{N} \\
R &\stackrel{def}{=} \nu m \nu q \nu r_R (\overline{x}\langle m, r_R\rangle.\{r_R(q_i) + 0\}_{i=1}^{n}).0 \\
T &\stackrel{def}{=} \nu m \nu q \nu r_R \nu r_{P_i} (x(m, r_R).\{\overline{y_i}\langle m, r_{P_i}\rangle\}_{i=1}^{n}. \\
&\quad \{r_{P_i}(q_i).\overline{r_R}\langle q_i\rangle.0 + 0\}_{i=1}^{n}), \\
&\quad n \in \mathbb{N} \\
P_i &\stackrel{def}{=} \nu m \nu q \nu r_{P_i} (y_i(m, r_{P_i}).\tau_{P_i}.(\overline{r_{P_i}}\langle q\rangle.0 + 0)) \\
\textit{Request-for-Bid} &\equiv R|T|\prod_{i=1}^{n} P_i, n \in \mathbb{N}
\end{aligned}
$$

Abbildung 6.12: Formalisierung des *Request-For-Bid*-MEP.

6.4 WSDL-Beschreibung Tuplespace-basierter Web-Services

Um potentiellen Nutzern eines Web-Service die für einen Aufruf über ein bestimmtes Kommunikationsprotokoll notwendigen Informationen zu bieten, müssen diese als Teil der WSDL-Beschreibung des Diensts erfasst werden. In der Folge wird am Beispiel der WSDL-2.0-Spezifikation erläutert, wie dies für das Tuplespace-Web-Service-Binding in standardkonformer Weise erfolgen kann.

Die Adressierung eines über einen Tuplespace angebotenen Diensts erfolgt durch eine URI (vgl. Abschnitt 6.2). Diese folgt entsprechend [BLFM05] dem nachfolgend erläuterten Aufbau. Der Bezeichner des *Scheme* der URI ist `ts`. Die *Authority* der URI identifiziert die Adresse des Tuplespace-Servers. Der *Path*-Anteil der URI identifiziert optional den lokalen Namen des Tuplespaces auf dem jeweiligen Tuplespace-Server; das letzte Element des *Path*-Anteils der URI ist ein in diesem Kontext eindeutiger Name des Web-Service. Die URI eines Web-Service wird als Wert des Attributs `address` des `endpoint`-Kind-Elements des WSDL-`service` dokumentiert. Ein Beispiel einer URI, die den Dienst `StockQuoteService` am Tuplespace `ps1` an Tuplespace-Server `ps.organization-a.org` identifiziert, ist:

```
ts://ps.organization-a.org/ps1/StockQuoteService
```

Die standardkonforme Beschreibung eines über einen Tuplespace angebotenen Web-Service erfordert zusätzlich zur Adresse des Web-Service selbst die Spezifikation des für das SOAP-Nachrichten-Encoding zu verwendende Verfahrens. Dieses wird durch das Attribut `wsoap:protocol` des `Binding`-Elements eines Diensts in Form eines URI spezifiziert. Die für das in Abschnitt 6.2 beschriebene Nachrichten-Encoding zu verwendende URI ist:

```
http://iaas.uni-stuttgart.de/ns/2009/process-space/
ts-binding
```

Abbildung 6.13 zeigt einen Ausschnitt aus der WSDL-2.0-Beschreibung eines über das Web-Service-Binding für Tuplespaces angebotenen Web-Service.

```
<wsdl20:binding
   name="StockQuoteSoapTuplespaceBinding"
   interface="tns:StockQuoteInterface"
   type="http://www.w3.org/2006/01/wsdl/soap"
   wsoap:protocol="http://iaas.uni−stuttgart.de/ns/2009/process−space/ts−binding"
   xmlns:wsoap="http://www.w3.org/ns/wsdl/soap"
   xmlns:wsdl20="http://www.w3.org/ns/wsdl"/>

<wsdl20:service
   name="StockQuoteService"
   interface="tns:StockQuoteInterface"
   xmlns:wsdl20="http://www.w3.org/ns/wsdl">
   <wsdl20:endpoint
      name="Tuplespace"
      binding="tns:StockQuoteSoapTuplespaceBinding"
      address="ts://ps.organization−a.org/ps1/StockQuoteService"/>
</wsdl20:service>
```

Abbildung 6.13: Ausschnitt einer WSDL-2.0-Beschreibung eines über das Web-Service-Binding für Tuplespaces angebotenen Web-Service.

6.5 Zusammenfassung

In diesem Kapitel wurde erläutert, wie Kommunikation zwischen Web-Services auf Basis einer Tuplespace-Middleware erfolgen kann. Kernelemente des entwickelten Verfahrens sind die Definition einer generischen Tupelstruktur zur Repräsentation kommunizierter Nachrichten sowie einer Tuplespace-basierten Realisierung diverser Web-Service-MEP. Als Motivation für die Tuplespace-basierte Kommunikation zwischen Web-Services wurden sowohl die starke Entkopplung der Interaktionspartner in den Dimensionen Zeit, Raum und Referenz als auch die in Tuplespaces verwendete *Pull*-Adressierung identifiziert. Insbesondere letzteres erlaubt in Verbindung mit der Eigenschaft eines Tuplespace, Funktionen und Eigenschaften sowohl von Datenbanken als auch MOM zu vereinen, eine einfache und elegante Umsetzung auch komplexer MEP. Aufgrund der getroffenen Einschränkungen des Bindings – das Vorhandensein der *in*-, *out*- und *rd*-Operationen der Linda-Schnittstelle in Verbindung mit Template-Matching mit den Operatoren Identitätsprüfung und *Wildcard* – wird eine Umsetzung auf einer großen Anzahl von Implementierungen des Tuplespace-Konzepts möglich.

Zu Demonstrationszwecken wurde das entwickelte Binding unter Verwen-

dung zweier unterschiedlicher Tuplespace-Implementierungen – *JavaSpaces* und *Triple Space*[1] [WML08c, TNW+08] – prototypisch umgesetzt; die Erläuterung einer dieser Realisierungen wird in Abschnitt 7.4 vorgestellt.

[1] http://tripcom.org

KAPITEL 7

PROTOTYPISCHE UMSETZUNG

In diesem Abschnitt werden prototypische Realisierungen der in den vorangehenden Kapiteln vorgestellten Konzepte präsentiert. Dies umfasst sowohl die entwickelte PS-Infrastruktur, bestehend aus PS-Servern (Abschnitt 7.1) und PS-Klienten (Abschnitt 7.2), als auch das angeschlossene Werkzeug zur Partitionierung von BPEL-Prozessen (Abschnitt 7.3). Ein Überblick über eine prototypische Implementierung des Web-Service-Binding für Tuplespaces auf Basis der Web-Service-Laufzeitumgebung *Apache Axis 2*[1] ist Gegenstand von Abschnitt 7.4. Die Beschreibung der einzelnen Komponenten des Prototyps fokussiert sich auf einige wesentliche Teilaspekte der Umsetzung und deckt demnach nicht den vollen Umfang des Prototyps ab.

7.1 Process-Space-Server

Abbildung 7.1 zeigt eine schematische Übersicht der prototypischen Implementierung der PS-Server und derjenigen Komponenten der PS-Klienten, die für einen entfernten Zugriff auf die von den PS-Servern angebotenen Funktionen

[1] http://ws.apache.org/axis2

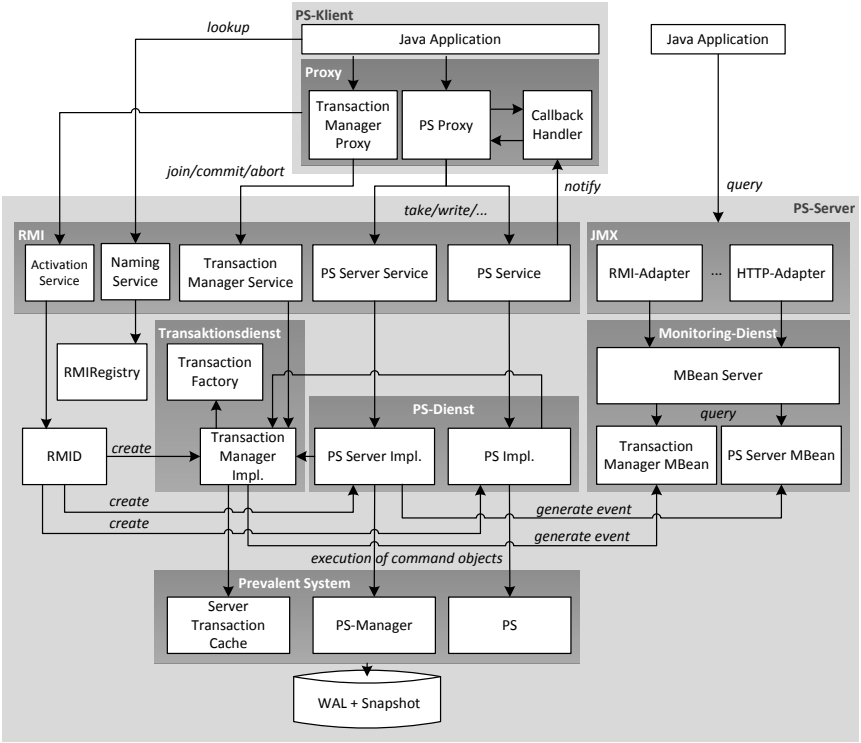


Abbildung 7.1: Übersicht über die Realisierung des PS-Server-Prototyps.

notwendig sind. Die Beschreibung der internen Realisierung der PS-Klienten ist Teil von Abschnitt 7.2.

Realisierungsplattform für sämtliche Bestandteile der prototypischen Implementierung ist *Java SE*. Die internen Komponenten des PS-Server-Prototyps lassen sich den sechs in der Abbildung grau hinterlegten Bereichen zuordnen. Netzwerkgestützte Kommunikation zwischen PS-Klient und PS-Server erfolgt über *Remote Method Invocation (RMI)* [sun06b] bzw. die *Java Management Extension (JMX)* [sun06a]. Die JMX-Schnittstelle erlaubt die Überwachung der technischen Betriebsparameter der PS-Server über unterschiedli-

che Protokolle. Die RMI-Schnittstelle erlaubt den Zugriff auf die durch die PS-Server angebotenen Operationen (vgl. Abschnitt 5.3). Diese Operationen werden auf den PS-Servern in Form verschiedener Dienste angeboten. Der *Naming-* und der *Activation-Service* sind von RMI angebotene Dienste zur Realisierung des Findens bzw. der Bestimmung von Objektreferenzen und zur serverseitigen Aktivierung von Objektinstanzen über die Grenzen einer Interaktion hinweg. Implementiert werden diese Dienste durch die *RMI Registry* bzw. den *Activation System Daemon (RMID)*, beide Implementierungen sind Teil der *J2SE*-Plattform und werden dementsprechend in der Folge nicht weiter diskutiert. Die Anwendungslogik des PS-Servers, d. h. die Implementierung der Funktionen der PS-Schnittstelle, wird durch den *PS-Dienst* realisiert. Die Funktionen des PS-Diensts können dessen Komponenten *PS-Server-Implementierung* und *PS-Implementierung* zugeordnet werden. Diese implementieren die Dienste *PS-Server-Service* bzw. *PS-Service* der RMI-Schnittstelle. Die PS-Server-Implementierung realisiert die Funktionen zur Verwaltung von PSs, die PS-Implementierung die eigentlichen PS-Operationen. Beide Komponenten bedienen sich hierzu den, in Form eines *Prevalent System* (vgl. Abschnitt 5.4.1) persistent vorgehaltenen, Komponenten *PS-Manager* und *PS* und manipulieren diese durch die Ausführung sogenannter *Command*-Objekte [GHJV95]; eine Beschreibung der Realisierung des Persistenzmechanismus ist Thema von Abschnitt 7.1.1. Für die Realisierung transaktionalen Verhaltens der PS-Operationen nutzt der PS-Dienst die Funktionen des *Transaktionsdiensts*. Dessen Kernfunktionalität wird durch die *Transaction-Manager-Implementierung* realisiert. In Bearbeitung befindliche Transaktionen werden ebenfalls im Prevalent-System verwaltet (vgl. Abschnitt 7.1.3). Sowohl der Transaktions- als auch des PS-Dienst werden bei der ersten Nutzung durch einen Klienten mittels des RMID instantiiert und existieren ab diesem Zeitpunkt bis zu ihrer Terminierung bei der Beendigung der Ausführung der PS-Server. Entsprechend der *RMI-Activation*-Semantik wird jeder Dienst in einer eigenen *Java Virtual Machine (JVM)* auf dem PS-Server ausgeführt. PS-Server und Transaction-Manager exponieren ihre technischen Betriebsparameter (z. B. Anzahl der PS, die auf einem Server betrieben werden oder die Anzahl in Ausführung befindlicher Transaktionen) in Form sogenannter *MBeans* [sun06a], deren Werte über einen

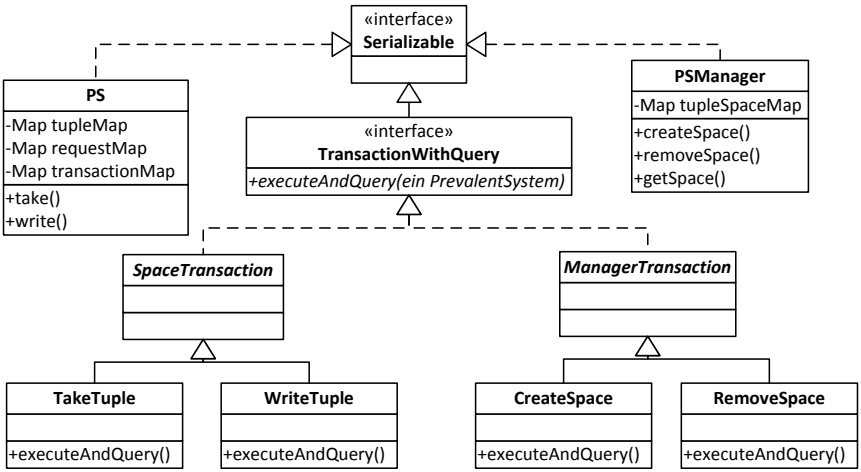


Abbildung 7.2: Vereinfachte Darstellung der Persistierung von Daten mittels Prevayler.

MBeanServer von PS-Server-externen Klienten abgefragt werden können.

Eine detaillierte Beschreibung der einzelnen Komponenten und ihrer Interaktionen untereinander ist Teil von [Wu08].

7.1.1 Persistenzschicht

Abbildung 7.2 zeigt die Realisierung des Persistenzmechanismus der PS-Server unter Verwendung von *Prevayler*[1], einer Java-Implementierung des *Object Prevalence*-Konzepts in Form eines (vereinfachten und daher unvollständigen) UML-Klassendiagramms.

Die Klasse `PSManager` ist die Repräsentation der *PS-Server*-Komponente in Abbildung 7.1 im Prevalent-System und implementiert sämtliche Funktionen für die Erzeugung und Zerstörung von PS, sowie den Zugriff auf existierende PS. Der `PSManager` ist ein Singleton für jeden PS-Server. Eine Instanz der `PS`-Klasse realisiert einen PS für die Ablage von Tupeln; als solcher bietet er

[1] `http://www.prevayler.org/`

Implementierungen sämtlicher PS-Operationen mit der in Abschnitt 5.3.1 vorgestellten Semantik. Die Beschreibung der prototypischen Realisierung dieser Operationen ist Gegenstand der Vorstellung der Implementierung des *Coordination Kernel* in [Mar10]. Sowohl `PS` als auch `PSManager` implementieren das Interface `Serializable`, was die Serialisierung und Deserialisierung der Instanzen der Klassen zur Erstellung von Prevalent-System-Snapshots erlaubt (vgl. Abschnitt 5.4.1).

Die transaktionale Manipulation der `PS`- und `PSManager`-Klassen erfolgt durch die Klassen `SpaceTransaction` und `ManagerTransaction`. Die Implementierung des *Prevayler*-Interface `TransactionWithQuery` identifiziert die Stellvertreter der einzelnen PS-Operationen `TakeTuple`, `WriteTuple` als *Command*-Objekte für die Ausführung durch *Prevayler*.

Die Verarbeitung eines Aufrufs einer PS-Operation durch einen PS-Server verläuft wie folgt: Nach der Übermittlung des Operationsaufrufs an den PS-Server (vgl. Abschnitt 7.1.2) erzeugt dieser ein entsprechendes Prevayler-Transaktionsobjekt vom Typ `SpaceTransaction` bzw. `ManagerTransaction`. Das Transaktionsobjekt wird durch die Prevayler-Klasse `CentralPublisher` serialisiert (daher die Implementierung von `Serializable`) und in einem `PersistentJournal` synchron, persistent und mit einem Zeitstempel versehen, abgelegt. Nach Ablage des Transaktionsobjekts im Journal wird dieses durch einen Prevayler-`TransactionSubscriber` konsumiert und verarbeitet. Als Teil der Verarbeitung führt dieser die Funktion des Transaktionsobjekts – implementiert in Form von dessen `executeAndQuery`-Methode – aus und liefert ggf. das Resultat der Operationsausführung an den Aufrufer zurück. Parallel zur Verarbeitung von Transaktionsobjekten durch den `TransactionSubscriber` löst eine Instanz der Klasse `SnapshotThread` die periodische Serialisierung von Abbildern des Prevalent-System und deren anschließende persistente Ablage aus. Kommt es während der Ausführung eines Transaktionsobjekts zu einem Systemabsturz, so wird der letzte verfügbare Snapshot des Prevalent-System durch den `GenericSnapshotManager` bestimmt und mittels Deserialisierung rekonstruiert. Nach dieser Rekonstruktion des letzten Snapshot werden die verbliebenen Einträge im `PersistentJournal` deserialisiert und entsprechend ihrer Reihenfolge im Journal auf dem rekonstruierten

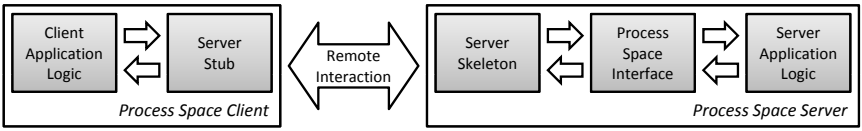


Abbildung 7.3: Realisierung der entfernten Nutzung der *Process Space* Schnittstellen.

Snapshot ausgeführt. Nach deren erneuter Verarbeitung ist der letzte Zustand des Systems vor dessen Absturz wiederhergestellt.

7.1.2 Entfernte Kommunikation zwischen PS-Klienten und PS-Servern

Implementierungsplattform für die Kommunikation zwischen PS-Klienten und PS-Servern ist *Java Remote Method Invocation (RMI)* [sun06b]. RMI ist eine Technologie für die – sowohl aus Sicht des Aufrufers als auch aus Sicht des Aufgerufenen – weitgehend transparente Interaktion mit Java-Objekten.

Abbildung 7.3 veranschaulicht die Realisierung der Interaktion zwischen PS-Klient und PS-Server unter Verwendung von RMI. In der Anwendungslogik der Klienten interagiert dieser mit einem *Proxy*-Objekt [GHJV95] in Form eines lokalen *Server Stub*. Dieser interagiert, für die Anwendungslogik des Klienten transparent, mit einem *Server Skeleton* auf Seite des PS-Servers, durch welchen die Implementierungen der Operationen der PS-Schnittstelle aufgerufen werden. Unterschiedliche Gründe motivieren die Verwendung von RMI für die Prototypenumsetzung. Unter anderem sind dies: (i) ein aus der nahtlosen Integration in die gewählte Realisierungsplattform resultierendes, einfaches Entwicklungsmodell, (ii) eine universelle Verfügbarkeit der notwendigen Bibliotheken und Dienste als Teil von J2SE, und (iii) die relativ geringen Anforderungen hinsichtlich des Deployments der Anwendung. Weiterhin erleichtern die als Teil von RMI verfügbaren Mechanismen zur Umsetzung von Callbacks und der Möglichkeit zur serverseitigen Aktivierung und Deaktivierung zustandsbehafteter Objekte die Umsetzung.

Aus Sicht der Umsetzung der Anwendungslogik eines PS-Klienten verläuft

ein Aufruf einer PS-Operation wie folgt. Der PS-Klient interagiert mit regulären Java-Objekten vom Typ `PSServer`, `PSProxy` und `TransactionManager`, welche jeweils das Java-Interface `Remote` implementieren. Zur Erzeugung eines PS verwendet ein PS-Klient den RMI *Naming*-Dienst, um eine Referenz auf das `PSServer`-Objekt zu erhalten; für das `TransactionManager`-Objekt (vgl. Abschnitt 7.1.3) wird analog verfahren. Um Operationen auf einem PS ausführen zu können, erwirbt ein PS-Klient analog dem oben beschriebenen Verfahren eine entsprechende PS-Objektreferenz durch Aufruf der `PSServer`-Methoden `createSpace`, falls ein neuen PS erzeugt werden soll, bzw. `getSpace`, falls eine Referenz zu einem bereits existierenden PS erzeugt werden soll. Ein PS wird repräsentiert durch ein entsprechendes `PSImpl`-Objekt (vgl. Abbildung 7.7), welches die Funktionen der operativen PS-Schnittstelle bereitstellt, sowie ein zugehöriges `PS`-Objekt im *Prevalent System* (vgl. Abbildung 7.2). Die Ausführung der PS-Operationen erfolgt allerdings nicht direkt auf dem `PS`-Objekt, sondern auf dem sogenannten `PSProxy`. Dieser kapselt die Methodenaufrufe auf dem `PS`-Objekt und ergänzt diese um zusätzliche Verarbeitungsschritte. Diese sind notwendig für beispielsweise die Umsetzung der blockierenden Semantik der PS-Operationen. Eine naive Realisierung durch das Vorhalten einer geöffneten (TCP-) Netzwerkverbindung über den gesamten Zeitraum einer Request-Response-Interaktion ist im Fall eines langen Blockierens nicht praktikabel, da ein PS-Server bei Abbruch dieser Verbindung (beispielsweise durch ein kurzes Abreißen der Netzwerkverbindung) nicht mehr in der Lage wäre, das Resultat der Operationsausführung an den Klienten zu übermitteln. Aus diesem Grund erfolgt die Realisierung blockierender Operationsaufrufe asynchron; Abbildung 7.4 veranschaulicht dies graphisch anhand eines UML-Sequenzdiagramms der Ausführung der *read*-Operation.

Der `PSProxy` auf Seite des PS-Klienten nimmt den Aufruf der PS-Operation von der Anwendungslogik des PS-Klienten entgegen und interagiert daraufhin stellvertretend für diesen mit dem PS-Server. Die Realisierung der Übermittlung der Antwortnachricht des PS-Servers an den PS-Klienten erfolgt unter Verwendung eines *Callback* beim `PSProxy`, welchen dieser als Teil des eigentlichen Operationsaufrufs an den PS-Server übermittelt. Der PS-Server führt daraufhin

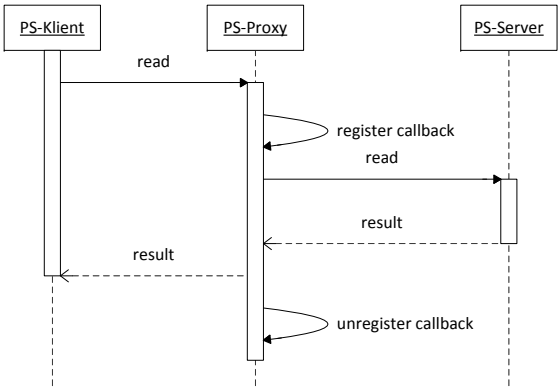


Abbildung 7.4: Ablauf einer nicht-blockierenden *Read* Operation.

die Operation des Klienten aus. In Abbildung 7.4 ist ein Fall dargestellt, in welchem ein zur *read*-Operation des Klienten konformes Tupel im abgefragten PS enthalten ist; die Anfrage des Klienten kann also zum Zeitpunkt des Operationsaufrufs sofort erfüllt werden. Der PS-Server übermittelt in diesem Fall die Antwort sofort an den Klienten unter Verwendung des von diesem übergebenen Callback. Der `PSProxy` übergibt das Resultat der Interaktion an den PS-Klienten und beendet damit die Interaktion.

Abbildung 7.5 zeigt einen aus Sicht des PS-Klienten gleichen Operationsaufruf. Im hier dargestellten Fall befindet sich allerdings im abgefragten PS zum Zeitpunkt des Operationsaufrufs kein zum spezifizierten Template konformes Tupel. Der PS-Server signalisiert dies dem `PSProxy`. Zu diesem Zeitpunkt ist die Interaktion zwischen `PSProxy` und PS-Server zunächst abgeschlossen und die Netzwerkverbindung zwischen diesen kann geschlossen werden. Auf Seite des PS-Servers wird die Anfrage des Klienten in Verbindung mit dessen Callback abgelegt. Wird zu einem späteren Zeitpunkt ein zur Anfrage des Klienten konformes Tupel in den abgefragten PS geschrieben, so benachrichtigt der PS-Server den `PSProxy` über diesen Zustand unter Verwendung von dessen Callback (und eine entsprechende, vom PS-Server initiierte Netzwerkverbindung). Bei Empfang einer derartigen Benachrichtigung kann der PS-Proxy

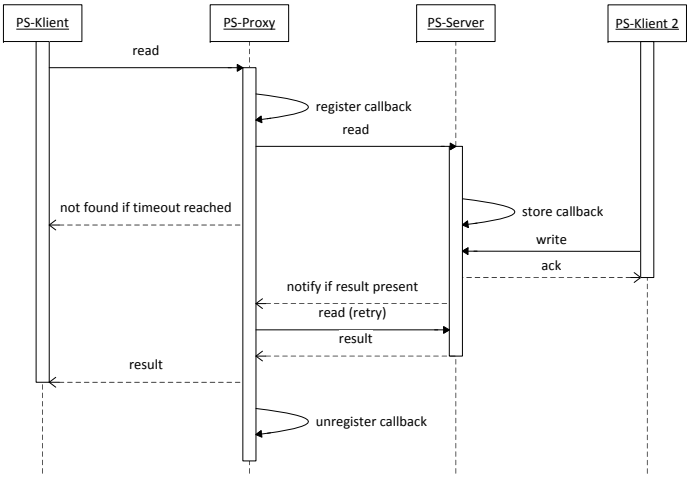


Abbildung 7.5: Ablauf einer blockierenden *Read* Operation.

entscheiden, ob das Tupel noch konsumiert werden soll und ggf. den Aufruf der *read*-Operation wiederholen. Hat im Zeitraum von Übermittlung und Verarbeitung der Benachrichtigung bereits ein anderer Klient das entsprechende Tupel konsumiert, so erhält der `PSProxy` auf die wiederholte Anfrage hin wiederum ein negatives Ergebnis, woraufhin der Proxy erneut in den Modus des blockierenden Wartens übergeht, bis der vom PS-Klient spezifizierte Timeout-Wert erreicht ist. Ist nach Erreichen des vom PS-Klienten spezifizierten Timeout-Werts keine weitere Information über ein vorhandenes Tupel beim `PSProxy` eingegangen, liefert dieser das leere Resultat an den PS-Klienten. Nach Beendigung der Interaktion durch entweder die Übergabe eines Resultats an den PS-Klienten oder das Erreichen des spezifizierten Timeout-Werts wird der registrierte Callback verworfen; erhält der PS-Proxy also nach Erreichen des Timeout-Werts ein Ergebnis vom PS-Server, so wird dieses nicht mehr verarbeitet.

Auf die oben beschriebene Weise bleibt der Klient über den gesamten Zeitraum der Interaktion blockiert, Verbindungen zwischen Klient und Server

werden allerdings nur unidirektional verwendet; lang laufende synchrone Interaktionen, die oben genannte Probleme aufweisen, bestehen nicht.

7.1.3 Realisierung transaktionaler Operationsausführung

Die Umsetzung des zur Ausführung von EWFNs verwendeten transaktionalen Modells stratifizierter Transaktionen (Abschnitt 5.5.2) erfordert, dass die Erfüllung der Eingangsbedingungen einer Transition, ihr interner Verarbeitungsschritt und die Erzeugung ihrer Ausgangseffekte innerhalb einer atomaren Transaktion erfolgen.

Die Umsetzung dieser Transaktionssemantik auf Basis von 2PC [Ber97] umfasst die folgenden Rollen: zwei oder mehr Transaktionsteilnehmer (von denen einer die Rolle des Initiators der Transaktion übernimmt) und einen Transaktionskoordinator, wobei gilt, dass eine Transaktion immer zwischen genau einem PS-Klienten und einem oder mehreren PS-Servern ausgeführt wird. Es gelten die folgenden Annahmen: (i) jeder Protokollschritt wird bei den Teilnehmern der Transaktion und dem Koordinator in Form eines WAL [HR01] persistiert; (ii) fatale Fehler, die ein Wiederanlaufen eines Transaktionsteilnehmers in endlicher Zeit verhindern, treten nicht auf; (iii) die Möglichkeit der Kommunikation zwischen den Teilnehmern der Transaktion und dem Koordinator ist gegeben.

Abbildung 7.6 zeigt den Ablauf einer Transaktion zwischen einem PS-Klienten, zwei PS-Servern und einem Koordinator. Anders als in der konzeptuellen Darstellung in Abbildung 7.6 erfolgt das Deployment des Koordinators in der Regel nicht isoliert, sondern in Verbindung mit jedem PS-Server; d. h. jeder PS-Server kann Koordinator einer globalen Transaktion sein. Nähere Informationen zur Umsetzung sind Teil der Erläuterungen in Abschnitt 7.1.

Der koordinierende PS-Server verwaltet den Zustand der Transaktion; zu diesem Zweck interagiert er mit den einzelnen Transaktionsteilnehmern. Das 2PC-Protokoll ist wie folgt aufgebaut: ein PS-Klient (d. h. der Initiator der Transaktion) kann beim Koordinator die Erzeugung einer neuen Transaktion durch die Protokollnachricht *Create Transaction* anfordern, welche in der Rückgabe eines Transaktionskontexts vom Koordinator an den Klienten resultiert. Diesen Transaktionskontext übergibt der Klient den innerhalb der Transaktion

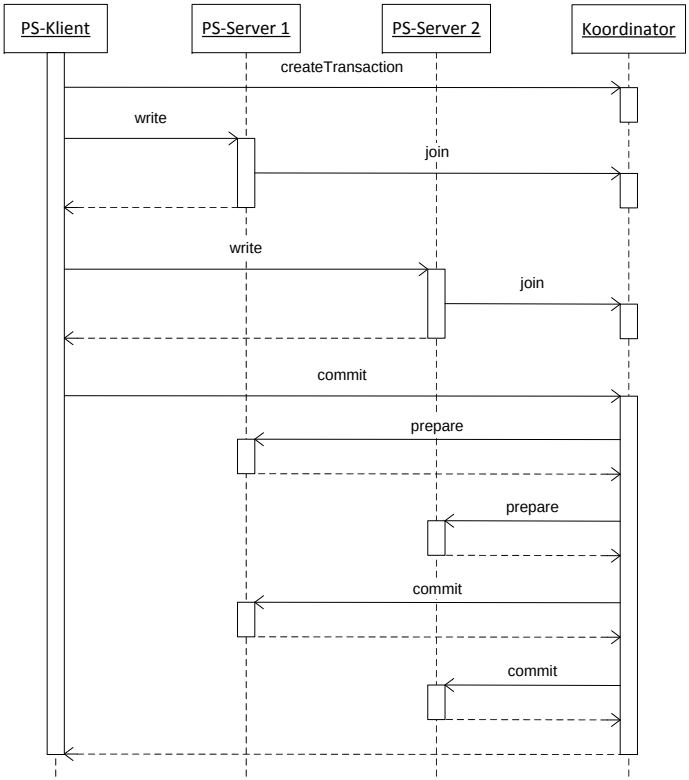


Abbildung 7.6: Sequenzdiagramm der Interaktion zwischen PS-Klienten, PS-Servern und dem Koordinator der globalen Transaktion.

auszuführenden PS-Operationen als Parameter (vgl. Abschnitt 5.3.2); im Beispiel sind dies jeweils eine Schreiboperation auf den PS-Servern 1 und 2. Die einzelnen PS-Server (d. h. die Teilnehmer der Transaktion) signalisieren ihre Teilnahme durch das Senden der Protokollnachricht *Join* an den Koordinator. Verlangt der initiirende PS-Klient den Abschluss der Transaktion durch die Übermittlung der Nachricht *Commit* an den Koordinator, so startet dieser das atomare Commit-Protokoll durch das Senden der Protokollnachricht *Prepare*

To Commit an sämtliche Teilnehmer der Transaktion. Jeder Teilnehmer der Transaktion persistiert ein sogenanntes *Undo Log* seiner Änderungen (um eine Rücknahme der Änderungen im Fall eines Zurückrollens der Transaktion im Fehlerfall zu ermöglichen) und signalisiert sein Einverständnis mit dem Commit durch die Übermittlung eines *Agreement* bzw. seine Ablehnung durch die Nachricht *Abort*. Mit der *Agreement*-Nachricht als Antwort auf die *Prepare To Commit*-Nachricht sichert der Teilnehmer zu, dass ein Commit erfolgreich ausgeführt werden kann.

Erhält der Koordinator von jedem Teilnehmer der Transaktion eine *Agreement*-Nachricht – und ist damit sichergestellt, dass sämtliche Teilnehmer der Transaktion diese mit einem Commit abschließen können – so wird der Abschluss der globalen Transaktion durch Senden der *Commit*-Nachricht an die PS-Server ausgelöst. Im Fall eines erfolgreichen Commits signalisiert ein Teilnehmer dies dem Koordinator durch Übersendung einer *Success*-Nachricht. Erhält der Koordinator hingegen als Antwort auf die *Prepare To Commit*-Nachricht von mindestens einem Teilnehmer der Transaktion eine *Abort*-Nachricht (oder innerhalb eines Timeout-Werts keine Antwort), so initiiert er, statt des Abschlusses der Transaktion durch ein Commit, das Zurückrollen der Transaktion bei den einzelnen Teilnehmern durch Übersenden der *Rollback*-Nachricht. Als Folge des Empfangs der *Rollback*-Nachricht setzen die Teilnehmer der Transaktion mittels des Eintrags in ihrem *Undo Log* die im Rahmen der Transaktion ausgeführten Änderungen zurück und signalisieren dem Koordinator das erfolgreiche Zurückrollen der Transaktion durch die *Success*-Nachricht. Im Beispiel von Abbildung 7.6 bedeutet dies, dass, sollte beispielsweise PS-Server 2 die *write*-Operation nicht erfolgreich ausführen können, dieser mit einem *Abort* auf die *Commit*-Anfrage des Koordinators antwortet, woraufhin der Koordinator mit einem *Rollback* die in derselben Transaktion ausgeführte Schreiboperation auf PS-Server 1 ebenfalls zurücknimmt, das geschriebene Tupel also niemals für PS-Klienten sichtbar wird.

Abbildung 7.7 zeigt die Umsetzung des oben beschriebenen Protokolls in der PS-Infrastruktur. Kernkomponenten der Realisierung sind die Implementierungen der Interfaces `TransactionManager` und `TransactionParticipant`.

Vor einer transaktionalen Operation eines Klienten erzeugt dieser über den

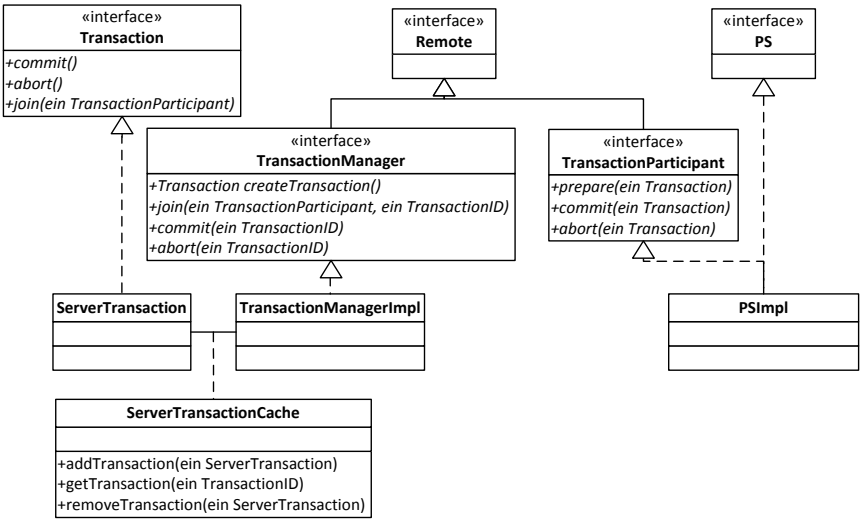


Abbildung 7.7: Realisierung globaler Transaktionen zwischen einem PS-Klient und mehreren PS-Servern.

Koordinator der verteilten Transaktion, den `TransactionManager`, einen Transaktionskontext vom Typ `ServerTransaction`, welcher an den erzeugenden PS-Klienten zurückgegeben und von diesem als Parameter sämtlicher, innerhalb der Transaktion auszuführender, Operationen übergeben wird. Die Kontexte in Ausführung befindlicher Transaktionen werden als Teil des Prevalent-System persistent im `ServerTransactionCache` verwaltet.

Abbildung 7.8 zeigt das Zustandsübergangsdiagramm eines PS-Klienten während der Ausführung einer Operation. Nach der Erzeugung eines Transaktionskontexts durch den `TransactionManager` befindet sich der PS-Klient im Zustand ACTIVE, in welchem dieser transaktional mit unterschiedlichen PS-Diensten interagieren kann. Den Abschluss der Transaktion signalisiert der Klient durch Aufruf der `commit()`-Methode seines lokalen PS-Proxy. Der PS-Proxy leitet den Aufruf über RMI an den Koordinator der Transaktion weiter, weswegen dieser das Interface `Remote` implementiert. Der Zustand des Kli-

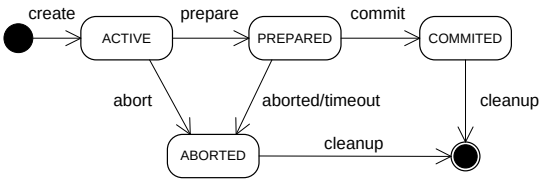


Abbildung 7.8: Zustandsübergangsdiagramm der Realisierung globaler Transaktionen auf den PS-Klienten (aus [Wu08]).

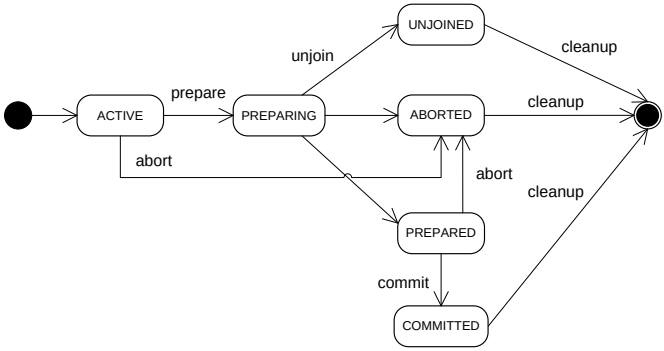


Abbildung 7.9: Zustandsübergangsdiagramm der Realisierung globaler Transaktionen auf den PS-Servern (aus [Wu08]).

enten ändert sich in der Folge zu PREPARED. Eine erfolgreiche Verarbeitung des Commit durch den PS-Server führt zu einem Übergang in den Zustand COMMITTED, ein Scheitern des Commit zu einem Übergang in den Zustand ABORTED.

Abbildung 7.9 zeigt die Transaktionsverarbeitung auf Seite der PS-Dienst-Teilnehmer der Transaktion. Dieser implementiert als Teil des Interface `TransactionPartiticpant` die Methoden `prepare()`, `commit()` und `abort()`. Erhält ein PS-Dienst einen Aufruf mit einem Transaktionskontext, dessen zugehöriger Transaktion er noch nicht beigetreten ist, so tritt er diesem durch Aufruf der Methode `join()` des jeweiligen `TransactionManager` bei, was ihn in

den Transaktionszustand ACTIVE überführt. Ein Aufruf der Methode `commit()` des Transaktionsteilnehmers durch den `TransactionManager` der Transaktion überführt jeden an der Transaktion teilnehmenden PS-Dienst in den Zustand PREPARING. In diesem prüft ein PS-Dienst, ob die auf ihm innerhalb der Transaktion ausgeführten PS-Operationen, erfolgreich verarbeitet werden können. Je nach Verlauf des PREPARING-Schritts kann ein PS-Dienst entweder signalisieren, dass an ihm im Rahmen der Transaktion keine Änderungen durchgeführt wurden und er sich dementsprechend bei der Abstimmung "enthält" (UNJOIN) oder die Änderungen erfolgreich (PREPARED) oder nicht erfolgreich ausgeführt werden können (ABORTED). Sein jeweiliges Abstimmungsergebnis übermittelt jeder PS-Dienst an den `TransactionManager` der Transaktion. Hat der `TransactionManager` von jedem `TransactionParticipant` eine Antwort erhalten und keiner von diesen mit einem negativen Ergebnis abgestimmt, so führt dieser den atomaren Commit der Transaktion durch Aufruf der `commit()`-Methode der einzelnen `TransactionParticipant`-Objekte herbei, was eine Überführung in den Zustand COMMITED zur Folge hat. Andernfalls wird die Transaktion auf allen teilnehmenden PS-Diensten zurückgenommen und diese werden in den Zustand ABORTED überführt; derselbe Effekt wird beim expliziten Abbruch der Transaktion durch den Initiator (Aufruf von `abort()` auf dem lokalen `TransactionManager`-Proxy und entsprechender Weiterleitung) erreicht.

7.2 Process-Space-Klienten

Wie in Abschnitt 5.6.3 erläutert, realisieren die PS-Klienten die Semantik der einzelnen BPEL-Aktivitäten. Abbildung 7.10 zeigt eine Übersicht der prototypischen Realisierung der PS-Klienten in Form der sogenannten *PS-Engine*, deren Umsetzung zwei wesentliche Aspekte umfasst. Die Funktionalität zur Verarbeitung des DDD-Fragments einer Domäne während des Prozess-Deployment-Schritts wird durch die *Deployment*-Komponente realisiert. Die Umsetzung der Anwendungslogik der PS-Klienten erfolgt in Form der sogenannten *PS-Runtime*. Entsprechend der Beschreibung in Abschnitt 5.2 gilt, dass in derselben *PS-Runtime* (d. h. insbesondere auf demselben physikalischen Rechner) mehr als

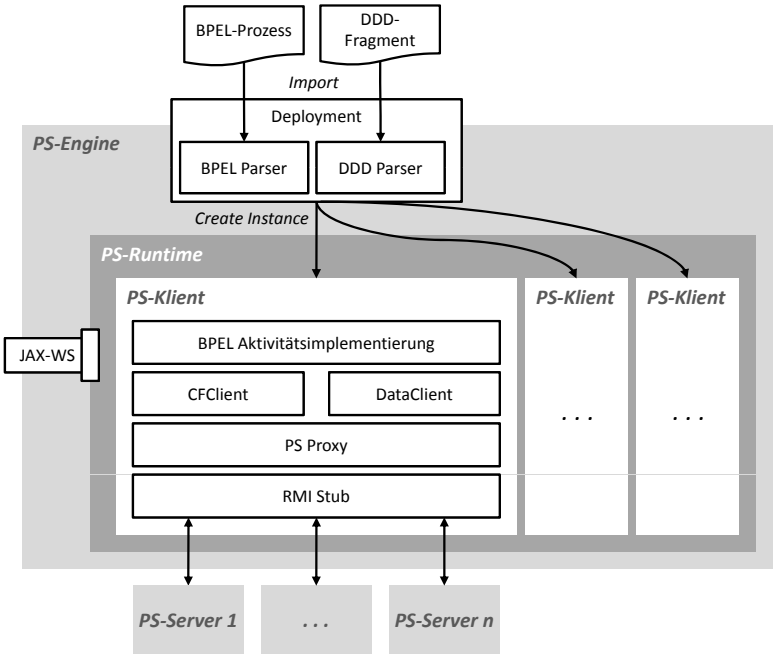


Abbildung 7.10: Übersicht der Realisierung der PS-Klienten.

ein *PS-Klient* betrieben werden kann. Während der Verarbeitung des DDD-Fragments wird eine Instanz von jedem in diesem definierten PS-Klienten erzeugt und entsprechend den Vorgaben im DDD konfiguriert. Die Konfigurationsdaten parametrisieren die durch den PS-Klienten ausgeführten PS-Operationen. Um eine möglichst weitgehende Entkopplung der Umsetzung der Anwendungslogik der einzelnen BPEL-Aktivitäten von den auszuführenden PS-Operationen zu erreichen, wurde zwischen der Anwendungslogik und dem PS-Proxy eine Abstraktionsschicht, bestehend aus `CFClient` und `DataClient`, eingezogen, durch welche die Erfüllung der Eingangsbedingung des PS-Klienten und die Erzeugung von dessen Ausgangseffekten bzw. deren Instanzdatenzugriffen realisiert wird. Der `CFClient` und der `DataClient` interagieren unter Verwendung des in Abschnitt 7.1.2 vorgestellten Kommunikationsmechanismus –

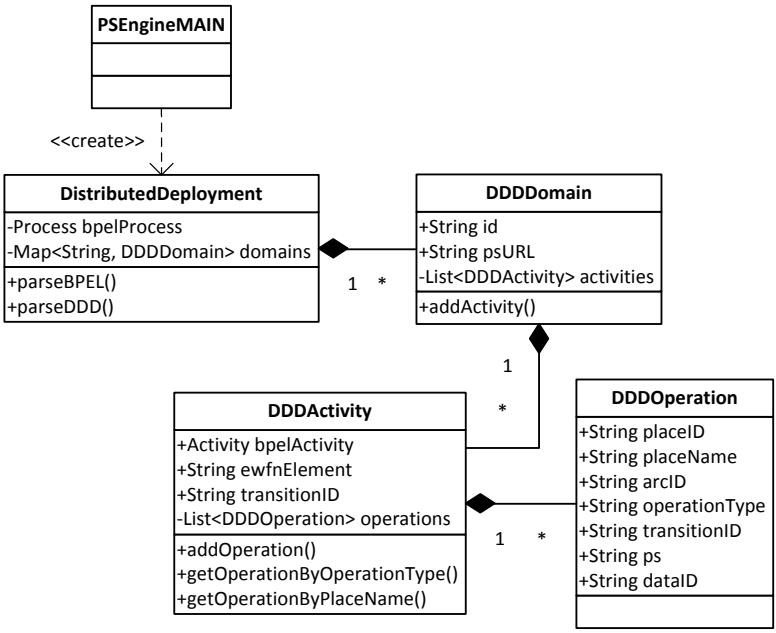


Abbildung 7.11: Verarbeitung des DDD durch die PS-Klienten.

entsprechend der Vorgaben im DDD – mit den einzelnen PS-Servern.

7.2.1 Verarbeitung der DDD-Fragmente

Abbildung 7.11 zeigt die an der Realisierung des Prozess-Deployment-Vorgangs beteiligten Klassen in Form eines vereinfachten UML-Klassendiagramms. Die Funktionalität zur Verarbeitung des auszuführenden BPEL-Prozesses und des DDD ist als Teil der Methoden `parseBPEL` bzw. `parseDDD` der Klasse `DistributedDeployment` realisiert. Instantiiert wird `DistributedDeployment` zum Start-Zeitpunkt der PS-Engine, realisiert durch die Klasse `PSEngineMain`.

Für die Verarbeitung des BPEL-Prozessmodells wird der BPEL-Parser und das zugehörige BPEL-Objektmodell des *Eclipse BPEL Designer*[1] verwendet. Durch

[1] `http://www.eclipse.org/bpel`

dieses wird die XML-Repräsentation des BPEL-Prozesses eingelesen und für sämtliche, in diesem definierten, Aktivitäten werden entsprechende Instanzen der Oberklasse `Activity` erzeugt, die die Parameter der jeweiligen Aktivitäten im Prozessmodell beinhalten. Die Verarbeitung des DDD erfolgt durch die Methode `parseDDD` der Klasse `DistributedDeployment`. Diese liest den DDD (bzw. das DDD-Fragment der Domäne der jeweiligen PS-Engine) ein und erzeugt eine entsprechende Instanz der Klasse `DDDDomain`. Das `DDDDomain`-Objekt beinhaltet einerseits die URL des EWFN-PS der betrachteten Domäne (vgl. Abschnitt 5.2) und eine Liste von Instanzen der Klasse `DDDActivity` für jeden PS-Klienten, der durch die Domäne bereitgestellt werden soll. Jede `DDDActivity`-Instanz beinhaltet einen Verweis auf die Aktivität im Objektmodell des *Eclipse BPEL Designer*. Die PS-Operationen, die von einem PS-Klienten während eines Verarbeitungszyklus ausführt werden, werden durch Instanzen der Klasse `DDDOperation` abgebildet. Die Attribute von `DDDOperation` orientieren sich an den Parametern des `Operation`-Elements im DDD (vgl. Abschnitt 5.8.1).

7.2.2 Ausführung der PS-Klienten in der PS-Runtime

Nach dem Deployment eines Prozesses, d. h. der Erzeugung der entsprechenden `DDDActivity`-Objekte, werden für die einzelnen Aktivitäten PS-Klienten-Implementierungen gestartet. Der Start der PS-Klienten erfolgt durch eine Instanz des Singleton `PartitionedBPELRuntime` (welches wiederum beim Start der PS-Engine erzeugt wird). Der Zusammenhang der an der Ausführung der PS-Klienten beteiligten Klassen ist in Abbildung 7.12 dargestellt.

Die `PartitionedBPELRuntime` verwaltet sämtliche auf dem jeweiligen physikalischen Rechner zu einer Prozesskonfiguration betriebenen PS-Klienten in der Liste `activityThreads`. Die Ausführung eines PS-Klienten erfolgt jeweils in einem eigenen Thread; erzeugt werden diese durch die private Methode `startActivityThreadsForPartition` des `PartitionedBPELRuntime`-Singletons. Ein PS-Klient wird dabei repräsentiert durch eine Instanz einer Unterklasse der abstrakten Klasse `ACTIVITY`. Diese referenziert die der Aktivität zugehörigen Informationen des DDD durch einen Verweis auf die ent-

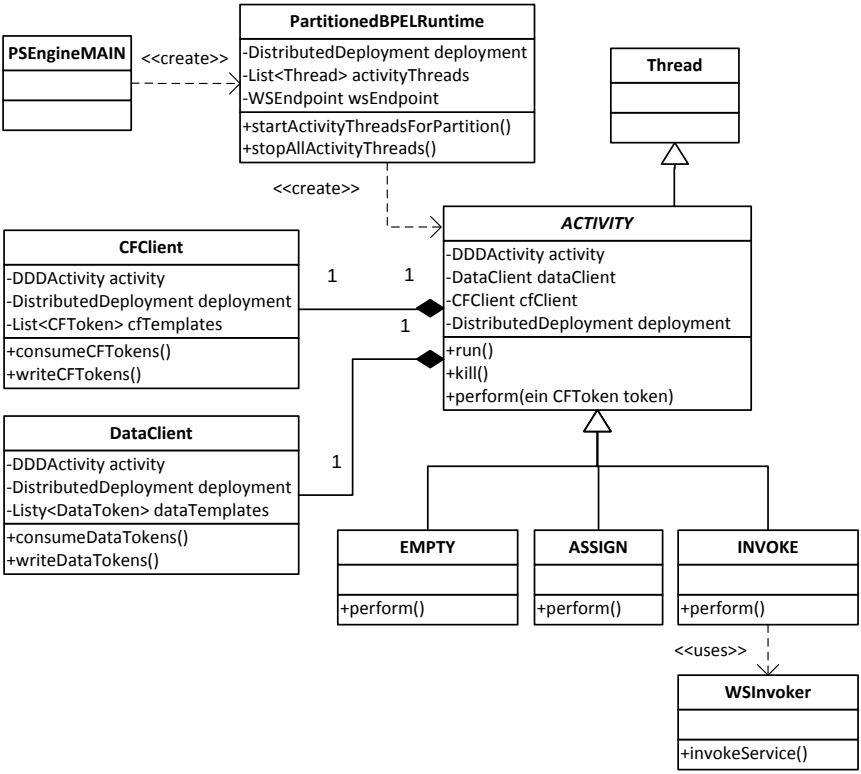


Abbildung 7.12: Laufzeitumgebung für die Ausführung von PS-Klienten.

sprechende `DDDActivity`. Die verwendeten Instanzen der Klassen `CFClient` und `DataClient` implementieren die Koordination der Ausführung eines PS-Klienten sowie dessen Instanzdatenzugriffe.

Gestartet wird ein PS-Klient durch Ausführung seiner `run`-Methode. Der Aufbau dieser Methode ist für sämtliche PS-Klienten gleich. Durch Aufruf der Methode `consumeCFTokens` versucht der PS-Klient seine Eingangsbedingung zu erfüllen, d. h. die entsprechenden `CFToken` aus den jeweiligen PS zu konsumieren. Die Repräsentation von Kontrollfluss- und Instanzdatentupeln (entsprechend der in Abschnitt 5.6.1 und 5.7.1 vorgestellten Tupelstruktur)

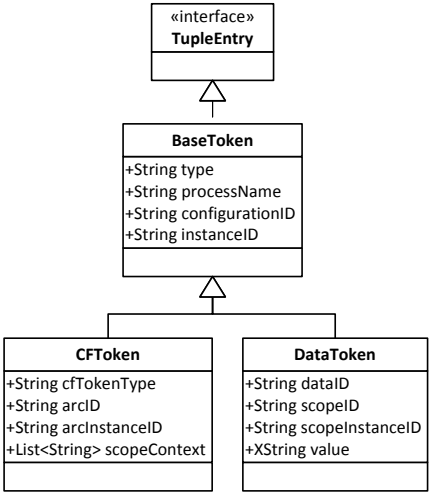


Abbildung 7.13: Repräsentation von Kontrollfluss- und Instanzdatentupeln.

erfolgt durch die Klassen `CFToken` bzw. `DataToken`, welche ihre generischen Felder von `BaseToken` erben; die Umsetzung ist in Abbildung 7.13 in Form eines vereinfachten UML-Klassendiagramms dargestellt. Nach erfolgreichem Konsumieren sämtlicher benötigter `CFToken` führt der jeweilige PS-Klient die von diesem implementierte BPEL-Aktivität aus. Realisiert ist diese in Form der Methode `perform` der jeweiligen Unterklasse der abstrakten Klasse `ACTIVITY`. Als Teil der Ausführung der `perform`-Methode kann ein PS-Klient unter Verwendung einer Instanz der Klasse `DataClient` Instanzdaten konsumieren und manipulieren. Ist die Ausführung der Anwendungslogik eines PS-Klienten beendet, so erzeugt dieser seine Ausgangsbedingung durch Aufruf der Methode `writeCFTokens` seiner `CFClient`-Instanz.

Implementiert ein PS-Klient die Funktionalität einer INVOKE-Aktivität, so hat die Ausführung seiner `perform`-Methode eine Interaktion mit einer Web-Service-Implementierung zur Folge. Realisiert wird diese durch eine Instanz der Klasse `WSInvoker`, welche (i) mit den für die Interaktion mit der Dienstimplementierung notwendigen Informationen aus dem jeweiligen `DDDActivity`-

Objekt parametrisiert wird, (ii) das zugehörige INPUTVARIABLE-Tupel durch den `DataClient` von `INVOKE` konsumiert, (iii) den Dienstaufruf unter Verwendung von *JAX-WS* durchführt und (iv) eine ggf. empfangene Antwort des Aufrufs in Form eines entsprechenden OUTPUTVARIABLE-Tupels erzeugt. Die Realisierung nachrichtenempfangender Nachrichten (z. B. RECEIVE) verläuft analog in Form der, in der Abbildung aus Gründen der Übersichtlichkeit nicht dargestellten, Instanz der `WSReceiver`-Klasse.

Die Vorstellung einer alternativen prototypischen Implementierung in Form einer Erweiterung des quelloffenen BPEL-WfMS *Apache ODE* ist Gegenstand von [Var09].

7.3 Umsetzung des Partitionierungsverfahrens

Abbildung 7.14 zeigt eine Übersicht der prototypischen Umsetzung des Partitionierungswerkzeugs, welches das in Abschnitt 4.4 vorgestellte Partitionierungsverfahren von BPEL-Prozessen umsetzt. Die Umsetzung ist eine Erweiterung der in [Dem08] vorgestellten Realisierung.

Eingabe des Partitionierungswerkzeugs bilden (i) der zu partitionierende BPEL-Prozess, (ii) die Partitionierungsparametrisierung des Prozesses, (iii) dessen EWFN-Repräsentation, (iv) die WSDL-Beschreibungen der Web-Services in der Dienstumgebung sowie (v) der Ausführungsumgebung des Prozesses. Sämtliche Eingabedaten liegen in Form von XML-Dateien im Dateisystem des zur Partitionierung verwendeten Systems vor. Die Abfrage einer UDDI-Registry ist nicht Teil der prototypischen Umsetzung. Ein Beispiel für eine Verwendung der Abfrage einer UDDI-Registry zur Bestimmung funktional kompatibler Dienstimplementierungen zu einer INVOKE-Aktivität findet sich in [Wut06]. Ebenso wird für die prototypische Realisierung auf eine Überprüfung nicht-funktionaler Kompatibilität von INVOKE-Aktivität und entsprechender Dienstimplementierung verzichtet; [Dem08] stellt ein Verfahren und eine Implementierung vor, wie der Abgleich nicht-funktionaler Anforderungen und Eigenschaften unter Verwendung eines proprietären Formats zur Beschreibung nicht-funktionaler Dienstmerkmale erfolgen kann. Vor der Verarbeitung durch das Partitionierungswerkzeug werden die Eingabedaten unter Verwendung entsprechender

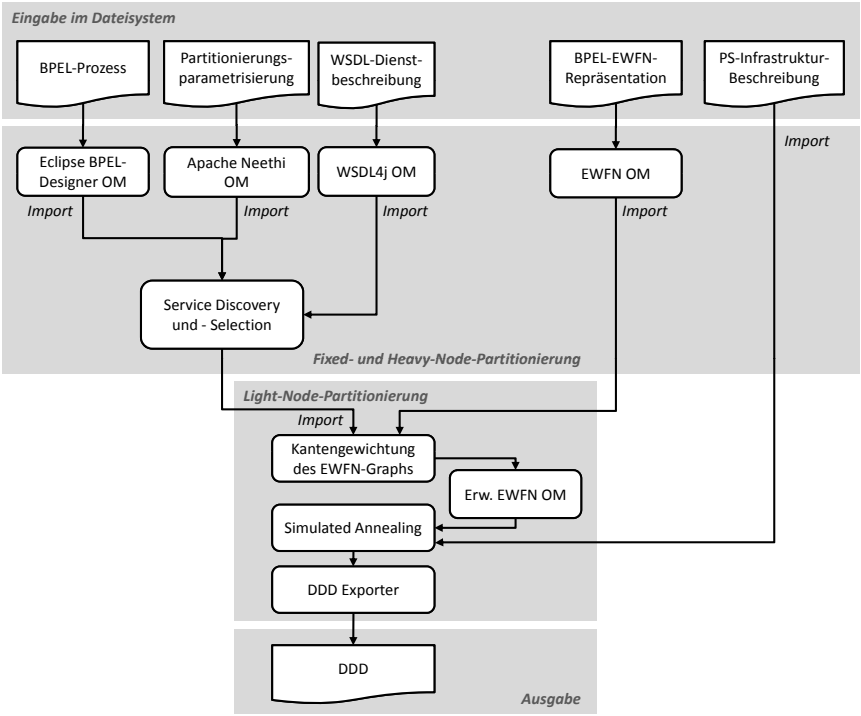


Abbildung 7.14: Prototypische Umsetzung des Partitionierungsverfahrens.

Objektmodelle eingelesen. Als Objektmodell für das zu partitionierende Prozessmodell kommt das BPEL-2.0-Objektmodell des *Eclipse BPEL Designer*[1] zum Einsatz. Die Repräsentation der Partitionierungsparametrisierung erfolgt unter Verwendung des WS-Policy-Objektmodells von *Apache Neethi*[2], die Repräsentation der Beschreibungen der funktionalen Diensteigenschaften unter Verwendung des *WSDL4j*-Objektmodells[3].

Entsprechend des Partitionierungsverfahrens umfasst auch die prototypi-

[1] http://www.eclipse.org/bpel/
[2] http://ws.apache.org/commons/neethi
[3] http://sourceforge.net/projects/wsdl4j/

sche Realisierung unterschiedliche Verarbeitungsschritte. Der BPEL-Prozess und die Partitionierungsparametrisierung sowie die funktionalen und nichtfunktionalen Dienstbeschreibungen dienen als Eingabe für die *Service Discovery und Selection*- Komponente. Diese implementiert Phase 1 (Abschnitt 4.5) und Phase 2 (Abschnitt 4.6) des entwickelten Partitionierungsverfahrens. Zu diesem Zweck werden zunächst anhand der statischen Partitionsvorgaben eventuelle *Fixed Nodes* bestimmt, d. h. insbesondere sämtliche nachrichtenempfangenden Aktivitäten. Anschließend wird – als Teil der Partitionierung der *Heavy Nodes* – für jede INVOKE-Aktivität des Prozesses die Menge der funktional kompatiblen Dienste durch einen Vergleich der Elemente PORTTYPE und OPERATION der Prozess-Beschreibung und der WSDL-Beschreibungen der bekannten Dienst-Implementierungen bestimmt. Die Auswahl der zu verwendenden Dienstimplementierungen für die einzelnen INVOKE-Aktivitäten erfolgt auf Grundlage des Kriteriums der Minimierung der Anzahl der an der Prozessausführung teilnehmenden Partner.

Das Resultat des *Service Discovery und Selection*-Schritts dient als Eingabe für die Umsetzung des Verfahrens zur Partitionierung der *Light Nodes* in Phase 3 des Partitionierungsverfahrens. Da das Verfahren auf Grundlage der in der EWFN-Repräsentation des Prozesses explizit spezifizierten Datenabhängigkeiten operiert, bildet diese, neben der bestimmten Partitionierung der *Heavy* und *Fixed Nodes*, eine Eingabe des ersten Schritts der *Light Node*-Partitionierung, der Bestimmung der Kantengewichte des EWFN-Graphen durch Anwendung der in Abschnitt 4.7.2 vorgestellten Algorithmen und der Partitionierungsparametrisierung. Die EWFN-Repräsentation eines Prozessmodells wird dabei unter Verwendung des in [Mar10] vorgestellten *BPEL-EWFN-Transformators* erstellt. Das Resultat des Gewichtungsvorgangs wird in Form eines erweiterten EWFN-Objektmodells repräsentiert, welches als Grundlage für die Graphpartitionierung durch *Simulated Annealing* (vgl. Abschnitt 4.7.3) dient. Das Verfahren berücksichtigt dabei die Partitionen bereits in vorangehenden Phasen bestimmter Partitionen von Aktivitäten bzw. EWFN-Transitionen und -Stellen.

Der nun partitionierte gewichtete EWFN-Graph dient wiederum als Eingabe eines DDD-Exporters, welcher anhand diesem einen DDD entsprechend dem in Abschnitt 5.8.1 vorgestellten Format erzeugt.

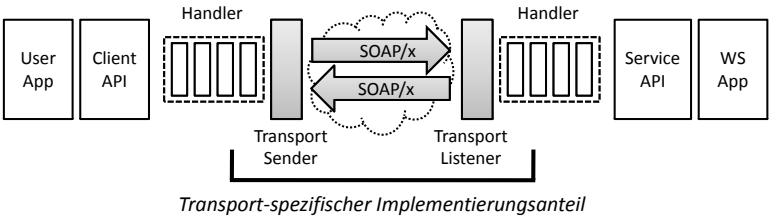


Abbildung 7.15: Abstrakte Übersicht über die Architektur von *Apache Axis 2*.

7.4 Web-Service-Binding für Tuplespaces

Zur Demonstration der Realisierbarkeit des in Kapitel 6 vorgestellten *Web-Service-Binding für Tuplespaces* auf unterschiedlichen Implementierungen des Tuplespace-Konzepts, wurde dieses sowohl unter Verwendung der *JavaSpaces*-Implementierung *Blitz*[1] als auch auf den im Rahmen des Projekts *TripCom*[2] entwickelten *Triple-Space* umgesetzt; die Implementierung des Web-Service-Binding für Triple-Space ist Teil des Triple-Space-Prototyps[3]. Zusätzliche Informationen zur *Triple-Space*-basierten Umsetzung sind Teil von [WCL+08, TNW+08, SCK+08]; weitere Details zur *JavaSpaces*-Umsetzung finden sich in [Sch07].

Grundlage der Realisierung beider Prototypen bildet *Apache Axis 2*[4] (in der Folge kurz als *Axis* bezeichnet). Axis implementiert unter anderen die Web-Service-Standards *SOAP* [GHM+07a], *WSDL* [CCMW01, CGM+04] und *WS-Addressing* [GHR06] und bietet damit eine Laufzeitumgebung für die Interaktion zwischen Web-Services. Als Grundlage für die Erläuterung der Realisierung werden zunächst die hierfür relevanten Aspekte der Axis-Architektur einführend vorgestellt.

Abbildung 7.15 zeigt die schematische Darstellung einer Interaktion eines Web-Service-Nutzers (*User App*) mit einem Web-Service unter Verwendung von Axis. Aus Sicht des Dienstnutzers interagiert dieser mit einer klientenseitigen

[1] http://www.dancres.org/blitz
[2] http://tripcom.org
[3] http://sourceforge.net/projects/tripcom
[4] http://ws.apache.org/axis2

Java-Schnittstelle (*Client API*), welche Dienstaufrufe des Dienstnutzers entgegennimmt, geeignet verarbeitet und diese an die dienstseitige Web-Service-Laufzeitumgebung übermittelt. Die Implementierung der Anwendungslogik des Web-Service (*WS App*) kann in Axis auf unterschiedliche Arten erfolgen. Axis unterstützt das Deployment (i) von *Plain Old Java Objects (POJO)* als Web-Service (so dass auf Seite des Service jegliche Java-Klasse als *WS App* verwendet werden kann) oder (ii) von speziell für Axis entwickelten Diensten. Diese Dienste interagieren mit Axis über die sogenannte *Service API* durch das Konsumieren und Produzieren von `OMElement`-Objekten, eine Repräsentation eines XML-Dokuments.

Kernelement der Axis Architektur sind die sogenannten *Handler*. Ein Handler implementiert jeweils einen bestimmten Teil der für die Verarbeitung einer SOAP-Nachricht notwendigen Gesamtfunktionalität (beispielsweise die Bestimmung der Web-Service-Implementierung, an die eine eingehende Nachricht übermittelt werden soll). Axis selbst definiert bereits unterschiedliche vorgefertigte Handler, welche – angelehnt an existierende Spezifikationen aus dem Bereich der Web-Services – u.a. die notwendige Funktionalität für die Adressierung von Endpunkten, Korrelation von Nachrichten und Realisierung der Standard-WSDL-MEP abbilden. Darüber hinaus erlaubt Axis Entwicklern die Definition und Implementierung eigener Handler. Sämtliche Axis-Handler haben dieselbe Signatur: Sie konsumieren ein sogenanntes `MessageContext`-Objekt, welches sowohl die aktuell verarbeitete SOAP-Nachricht (bestehend aus dem SOAP-Header- und dem SOAP-Body-Anteil) sowie eine Reihe für die Verarbeitung der Nachricht notwendiger Metainformationen enthält. Ein Handler kann den `MessageContext` einer Nachricht in beliebiger Form manipulieren; Resultat der Ausführung eines Handlers ist das verarbeitete `MessageContext`-Objekt. Um einen Handler in die eingehende oder ausgehende Verarbeitung einer Nachricht einzubinden, wird dieser in die sogenannte *In Pipe* bzw. die *Out Pipe* der *Axis Engine*, dem Kern des SOAP-Prozessors, eingehängt. Während der Verarbeitung einer eingehenden Nachricht passiert diese sämtliche in der *In Pipe* registrierten Handler; analog werden bei ausgehenden Nachrichten die Handler der *Out Pipe* ausgeführt.

Der überwiegende Anteil der Axis-Handler sind unabhängig von dem, für

die Kommunikation der SOAP-Nachricht zwischen Dienstnutzer und Dienstanbieter verwendeten, Transportprotokoll und Nachrichtenserialisierungsformat. Diese können also unabhängig vom konkreten Web-Service-Binding verwendet werden, dass für die Abwicklung der jeweiligen Interaktion zum Einsatz kommt. Die Realisierung eines Web-Service-Binding ist in Axis Teil von dessen, ebenfalls in Form von Handlern realisierten, sogenannten *Transport Frameworks* und besteht aus Implementierungen des `TransportSender`- bzw. des `TransportReceiver`-Interface. Eine Implementierung des `TransportSender`-Interface beinhaltet die transportspezifische Realisierung des Versendens einer Nachricht, die des `TransportReceiver`-Interface die Realisierung des Konsumierens eingehender Nachrichten. Die Implementierungen von `TransportSender` und `TransportReceiver` kommen dabei (zur Realisierung bidirektionalen Nachrichtenaustauschs) sowohl auf Seite der Dienstnutzer als auch auf Seite der Dienstanbieter zum Einsatz.

Als Teil der Standarddistribution von Axis werden `TransportSender` und `TransportReceiver` für die Protokolle HTTP, TCP, SMTP und JMS mitgeliefert; des Weiteren erlaubt Axis2 direkte Interaktion mit *Enterprise Java Beans (EJB)* [BM06].

Entsprechend der Axis-Architektur bildet die Erweiterung von Axis *Transport Framework* den Schwerpunkt der prototypischen Realisierung des *Web-Service-Bindings für Tuplespaces*. Abbildung 7.16 stellt die Erweiterung in Form eines vereinfachten UML-Klassendiagramms schematisch dar. Die Klassen `TuplespaceListener` und `TuplespaceSender` (im Fall der *Triple Space*-basierten Implementierung `TriplespaceListener` und `TriplespaceSender`) sind Implementierungen der `TransportListener` und `TransportSender`-Interfaces von Axis. Sie implementieren damit insbesondere die aus dem `Handler`-Interface ererbte `invoke`-Methode, welche von der *Axis Engine* während der Verarbeitung einer Nachricht mit dem `MessageContext`-Objekt der Nachricht als Parameter aufgerufen wird.

Die Funktionsweise des Binding-Prototyps lässt sich anhand des Beispiels einer dem *In-Out*-MEP folgenden Interaktion zwischen *User App* und *WS App* veranschaulichen. Um eine Interaktion mit dem Web-Service zu starten, interagiert *User App* mit der *Axis Engine* entweder direkt durch die *Client API* oder

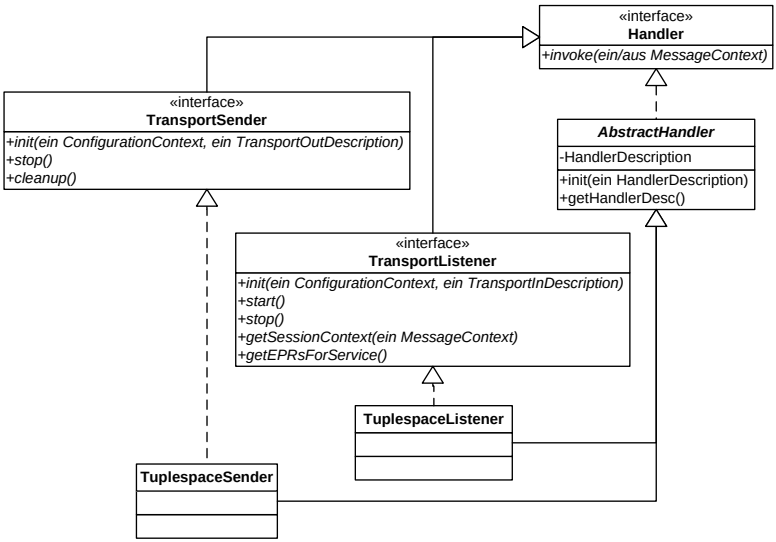


Abbildung 7.16: Erweiterung des *Transport Frameworks* von Axis zur Unterstützung des Web Service Bindings für Tuplespaces.

durch einen generierten Stub. Die *Axis Engine* nimmt die zu sendende Nachricht entgegen und übergibt sie der Reihe nach sämtlichen in der *Out Pipe* registrierten `Handler`-Objekten. Als letztem Handler der *Out Pipe* wird die Nachricht (in Form eines `MessageContext`-Objekts) an den `TuplespaceSender` übergeben. Dieser erzeugt (i) ein Tupel entsprechend der in Abschnitt 6.2 beschriebenen Tupelstruktur, weist (ii) den einzelnen Feldern des Tupels die der Nachricht entsprechenden Werte zu, baut (iii) eine Verbindung zu dem, vom Dienstanbieter überwachten, Tuplespace auf und schreibt (iv) die Nachricht in den Ziel-Tuplespace. Die weiteren vom `TuplespaceSender` ausgeführten Operationen sind abhängig vom MEP der Interaktion. Folgt die Interaktion beispielsweise einem *In-Only MEP*, so ist die Ausführung des Senders an dieser Stelle beendet. Folgt die Interaktion hingegen einem *In-Out MEP*, so wartet der `TuplespaceSender` nach dem Schreiben der Anfragenachricht blockierend auf das Eintreffen einer entsprechenden Antwortnachricht. Eine Korrelation

von Anfrage- und Antwortnachricht erfolgt durch die Definition eines Template, das eine Nachricht beschreibt, welche im Feld B-CorID den Wert des B-MsgID-Felds der Anfragenachricht trägt. Ermöglicht wird dies durch die Propagierung des WS-Addressing-*RelatesTo*-Headers in ein Feld des SOAP-Nachrichtentupels.

Auf Seite des Web-Service-Anbieters baut dessen `TuplespaceListener` eine Verbindung zu demjenigen Tuplespace auf, den der Service auf eingehende Anfragenachrichten hin überwacht. Dieser ist – den Erläuterungen in Abschnitt 6.4 folgend – durch eine URL beschrieben und ist Teil der WSDL-Beschreibung des Web-Services (Abschnitt 6.4). Das Template, das der Dienstanbieter für die Beschreibung zu empfangender Nachrichten verwendet, identifiziert Nachrichten, deren B-Destination-Eigenschaft der des Dienstanbieters entspricht. Schreibt ein Dienstnutzer ein dem Template entsprechendes Tupel in den jeweiligen Tuplespace, wird dieses durch die `TuplespaceListener`-Implementierung der Web-Service-Laufzeitumgebung des Dienstanbieters konsumiert. Ob das Konsumieren destruktiv (*take*) oder nicht-destruktiv (*read*) erfolgt, ist abhängig vom MEP der Interaktion. Nach dem Konsumieren des SOAP-Nachrichtentupels wird die in diesem enthaltene SOAP-Nachricht extrahiert, ein entsprechendes `MessageContext`-Objekt erzeugt und dieses der *Axis Engine* zur weiteren Bearbeitung in den Handlern der *In Pipe* übergeben. Nach erfolgreicher Bestimmung der auszuführenden Dienstimplementierung durch Axis (unter Verwendung existierender Handler), wird der SOAP-Body der Anfragenachricht der Anwendungslogik des Web-Service übergeben und diese ausgeführt. Folgt die Interaktion einem bidirektionalen MEP (d. h. einer Interaktion, in welcher der Web-Service die Verarbeitung einer Anfragenachricht entweder durch eine Antwort- oder Fehlernachricht bestätigt), so wird die von der Anwendungslogik erzeugte Ausgabe – der obigen Beschreibung des Dienstnutzers entsprechend – in Form eines neuen `MessageContext`- Objekts durch die Handler der *Out Pipe* an die `TuplespaceSender`-Implementierung auf Seite des Web-Services weitergeleitet. Dieser verhält sich hier analog der Dienstnutzerseite, mit der Ausnahme, dass vor dem Senden des Antwort-Tupels (der WS-Addressing Spezifikation [GHR06] folgend) das Feld B-CorID auf den Wert des Felds B-MsgID der Anfragenachricht und der Wert B-RType entsprechend gesetzt wird, um dem Sender der Anfragenachricht eine Template-basierte

Beschreibung der erwarteten Antwortnachricht zu ermöglichen.

7.5 Zusammenfassung

In diesem Kapitel wurde eine prototypische Implementierung der Konzepte vorgestellt, deren Erläuterung in den vorangehenden Kapiteln der Arbeit erfolgte.

Schwerpunkt dieser Beschreibung bildete die Präsentation der Kernaspekte der PS-Infrastruktur, d. h. die Realisierung von PS-Servern und PS-Klienten und des Partitionierungswerkzeugs. Spezieller Fokus der Beschreibung der Umsetzung der PS-Server war die Vorstellung der verwendeten Technologien und die Realisierung nicht-funktionaler Eigenschaften, wie z. B. der Wiederherstellung eines konsistenten Zustands nach einem Systemfehler. Die Beschreibung der PS-Klienten adressierte im Besonderen die Realisierung der PS-gestützten Koordination der PS-Klienten durch den Austausch von Kontrollflusstupeln und deren Kommunikation durch die Manipulation von Instanzdatentupeln.

Neben der Realisierung der PS-Infrastruktur wurde einer Übersicht der Umsetzung des entwickelten Verfahrens zur Partitionierung von BPEL-Prozessen gegeben und abschließend die prototypische Umsetzung des Web-Service-Binding für Tuplespaces unter Verwendung einer etablierten Web-Service-Laufzeitumgebung vorgestellt.

KAPITEL 8

ZUSAMMENFASSUNG UND AUSBLICK

Gegenstand dieser Dissertation war die Vorstellung eines Verfahrens und einer zugehörigen Laufzeitumgebung für die dezentrale Ausführung von, in WS-BPEL 2.0 formulierten, Produktionsprozessen. Als Teil der entwickelten Lösung waren eine Reihe unterschiedlicher Aspekte zu behandeln.

In Kapitel 1 wurde das Anwendungsfeld der Arbeit, die Ausführung von Geschäftsprozessen in Service-orientierten Architekturen, sowie die aus diesem resultierende Motivation für die dezentrale Prozessausführung erläutert. Ein Kernaspekt der Motivation der Arbeit stellt die Nicht-Invasivität des entwickelten Verfahrens dar, die es ermöglicht, auch solche BPEL-Prozesse, die nicht speziell im Hinblick auf eine dezentrale Ausführung entwickelt wurden, dezentral auszuführen, ohne dass dies Änderungen an deren Prozessmodell bedingt.

Anknüpfend an die Motivation der Arbeit erfolgte in Kapitel 2 die Vorstellung einiger wesentlicher Technologien aus dem Bereich der Beschreibung, Interaktion und Orchestrierung von Web-Services (WSDL, WS-Policy, UDDI, SOAP

und WS-BPEL), welche die Basis für die Umsetzung des entwickelten verteilten WfMS bilden. Im Anschluss hieran wurde eine Übersicht über die architekturellen Konzepte von WfMS anhand der Vorstellung des WfMC-Referenzmodells sowie unterschiedlicher Implementierungsbeispiele existierender WfMS zur Ausführung von BPEL erläutert. Neben diesen Produktbeispielen wurden einige relevante verwandte Arbeiten aus dem Forschungsbereich der verteilten Ausführung von Prozessen mit einer groben Übersicht ihrer Architektur sowie einer Vorstellung ihres Navigationsvorgangs vorgestellt. Als Grundlage für die Erläuterung des entwickelten automatischen Partitionierungsverfahrens für BPEL-Prozesse wurden weiterhin eine Reihe unterschiedlicher Verfahren zur Partitionierung von Prozessen unterschiedlicher Prozessmetamodelle präsentiert. Abschluss des Kapitels bildete eine Vorstellung der Konzepte der Linda-Tuplespaces, die die Grundlage für die in der Dissertation entwickelte Laufzeitumgebung – die sogenannte *Process-Space-Infrastruktur* – bilden. Auf diesen aufbauend wurden eine Reihe existierender Arbeiten, die Erweiterungen und Umsetzungen des Linda-Modells zum Gegenstand haben, kurz vorgestellt.

In Kapitel 3 wurden zunächst die Anforderungen an ein Prozessmetamodell zur Beschreibung von dezentral navigierbaren BPEL-Prozessen formuliert. Im Anschluss wurde das EWFN-Metamodell als eine Realisierung dieser Anforderungen einführend vorgestellt und die generelle Vorgehensweise erläutert, die der Überführung eines BPEL-Prozesses in seine EWFN-Repräsentation zugrunde liegt. Grundgedanke des EWFN-Metamodells ist die explizite Beschreibung der Koordination und Kommunikation, welche zwischen den einzelnen funktionalen Bausteinen eines Prozesses notwendig ist, um ein zur zentralen Ausführungssemantik von BPEL konformes Laufzeitverhalten zu erreichen. Sie stützen sich dabei auf den Formalismus der gefärbten Petri-Netze. Durch Zuweisung der Transitionen und Stellen eines EWFN-Modells eines Prozesses auf die Elemente der PS-Infrastruktur, PS-Klienten und PS-Server, wird ein umfangreiches Spektrum unterstützter Verteilungsszenarien desselben Prozesses realisierbar, ohne dass dies Änderungen dessen Prozessmodell verlangt. Hieran anknüpfend wurde die generelle Vorgehensweise zur Verarbeitung von Prozessen, die auf der PS-Infrastruktur ausgeführt werden sollen, über sämtliche Phasen des Prozesslebenszyklus hinweg vorgestellt. Aus Gründen der mög-

lichst weitgehenden Wiederverwendbarkeit existierender Werkzeuge wurde bei der Definition der PS-Methode besonderes Augenmerk auf die Anwendung existierender Technologien und Standards gelegt. Wesentliche Aspekte der PS-Methode sind die Erweiterung des Prozesslebenszyklus um die Phasen der Umgebungsdokumentation und der Evolution sowie die Erweiterung der Deployment-Phase um die Anwendung des entwickelten Partitionierungsverfahrens sowie die Bestimmung all jener Informationen, die für die Ausführung der Prozesspartitionierungsvorgangs notwendig sind.

Mit der Vorstellung des entwickelten Partitionierungsverfahrens von BPEL-Prozessen bildete Kapitel 4 einen inhaltlichen Schwerpunkt der vorliegenden Dissertation. Als Grundlage für die Beschreibung des Partitionierungsverfahrens wurden zunächst die Eigenschaften und Anforderungen vorgestellt, die aus dem dieser Arbeit zugrunde liegenden Einsatzszenario resultieren. Weiterhin wurden die Elemente eines Prozesses identifiziert, welchen im Rahmen des Partitionierungsverfahrens eine Partition in der Ausführungsumgebung des Prozesses zugewiesen werden kann bzw. muss. Der Partitionierungsvorgang eines Prozesses wird von verschiedenen Parametern beeinflusst, welche entweder anhand angefallener Ausführungsprotokolle von Prozessinstanzen bestimmt oder durch einen Partitionierer manuell vorgegeben werden können. Diese Parameter wurden im Einzelnen vorgestellt und es wurde erläutert, auf welche Weise der Parametrisierungsvorgang technisch umgesetzt wurde. Die Vorstellung des entwickelten mehrphasigen Partitionierungsverfahrens bildete den Abschluss dieses Kapitels. Das Verfahren selbst ist ein Hybrid unterschiedlicher Partitionierungsstrategien und kombiniert die Möglichkeit von statischen Partitionsvorgaben durch einen Nutzer mit einem automatischen Verfahren zur Prozesspartitionierung auf Grundlage des Optimierungskriteriums der Minimierung partitionsübergreifender Kommunikation. Weiterhin berücksichtigt das Partitionierungsverfahren unterschiedliche Einschränkungen, die aus dem Ziel einer Erhaltung der operationalen Semantik einer zentralen Prozessausführung resultieren.

Den zweiten inhaltlichen Schwerpunkt der Dissertation bildet die Erläuterung der Realisierung der entwickelten Laufzeitumgebung zur dezentralen Prozessnavigation in Kapitel 5. Als Grundlage für die Begründung getroffener

Design-Entscheidungen stützt sich diese auf eine Darstellung unterschiedlicher funktionaler und nicht-funktionaler Anforderungen. Hieran anknüpfend folgte die Erläuterung der Architektur der PS-Infrastruktur im Ganzen, des internen Aufbaus von PS-Klienten und PS-Servern sowie des Vorgangs der Prozessausführung im Detail. Diese deckt unterschiedliche Aspekte ab, wie beispielsweise die Realisierung transaktionalen Verhaltens während der Ausführung, die Verarbeitung auftretender Fehler sowie die Kompensation bereits ausgeführter Aktivitäten. Wesentlicher Unterschied zwischen der entwickelten PS-Infrastruktur und existierenden WfMS zur Ausführung von BPEL-Prozessen ist das Fehlen eines sogenannten Navigators. In traditionellen WfMS entscheidet dieser auf Grundlage des Prozessmodells und dem aktuellen Ausführungszustand einer Prozessinstanz, welche Aktivitäten des Modells zur Ausführung durch das WFMS bereit sind. In der PS-Infrastruktur verteilt sich die Funktion dieser Komponente auf die einzelnen, an der Ausführung eines Prozesses beteiligten, PS-Klienten. Weiterhin wurde erläutert, in welcher Weise der in [Mar10] vorgestellte *Coordination Kernel* für die Ausführung von BPEL-Prozessen zur Realisierung des Prozesskontrollflusses und der Kommunikation der Instanzdaten des Prozesses zwischen den Ausführungsteilnehmern verwendet werden kann. Neben diesen, direkt mit der Prozessausführung befassten, Teilen der Arbeit wurden auch angeschlossene Problemstellungen, wie das verteilte Deployment dezentral ausgeführter Prozesse und die Protokollierung der Prozessausführung, adressiert. Abschluss des Kapitels bildete eine Diskussion der Eigenschaften der dezentralen Prozessausführung unter Verwendung der PS-Infrastruktur.

Aufgrund der Nähe der entwickelten PS-Infrastruktur zu den Konzepten der Tuplespaces bietet diese – neben der Ausführung von Prozessen – auch eine geeignete Plattform für die Kommunikation zwischen den Teilnehmern einer Web-Service-Interaktion. In Kapitel 6 wurde in Form eines *Web-Service-Binding für Tuplespaces* – in Anlehnung an die existierenden Web-Service-Standards SOAP, WS-Addressing und WSDL – ein Verfahren vorgestellt, wie die Teilnehmer einer derartigen Interaktionen unter Verwendung der PS-Infrastruktur (oder anderen Implementierungen der Tuplespace-Konzepte, die die Operationen der Linda-Schnittstelle unterstützen) miteinander kommunizieren können.

Die Binding-Beschreibung umfasst dabei unter anderem eine Abbildung von SOAP-Nachrichten auf Tupel sowie eine Realisierung unterschiedlicher Web-Service-MEP mit den Operationen der Linda-Schnittstelle sowie deren formale Beschreibung unter Verwendung des π-Kalküls.

In Kapitel 7 wurde eine prototypische Umsetzung der in der Dissertation vorgestellten Konzepte aufgezeigt. Diese umfasst die Beschreibung der Realisierung der PS-Infrastruktur, d. h. der PS-Server und der PS-Klienten, die die Semantik der einzelnen BPEL-Aktivitäten implementieren, die Erläuterung der Realisierung des entwickelten Partitionierungswerkzeugs sowie des Tuplespace-Bindings als Erweiterung der Web-Service-Laufzeitumgebung *Apache Axis 2*.

8.1 Ausblick auf anknüpfende Arbeiten

Im Verlauf der Dissertation wurden einige Bereiche identifiziert, die thematisch an die vorgestellte Arbeit der dezentralen Navigation und verteilten Ausführung von BPEL-Prozessen anknüpfen und damit Ansatzpunkte für weitere Forschungsarbeiten bieten.

In Abschnitt 5.9 wurden einige Eigenschaften von BPEL diskutiert, die einen Zugriff unterschiedlicher Ausführungsteilnehmer auf dieselben Instanzdaten erforderlich machen, den sogenannten *Shared State*. Das Ziel der Arbeit – eine Vermeidung globalen Zustands (d. h. Zustand, der von allen Ausführungsteilnehmern eines Prozesses gemeinsam verwendet werden muss) – wurde zwar erreicht, dennoch erfordert die Einhaltung der operationalen Semantik von BPEL, beispielsweise hinsichtlich der Gewährleistung der Serialisierbarkeit von Variablenzugriffen während der Ausführung von *Isolated Scopes* oder der Realisierung der *Default Compensation Order* unterschiedliche Formen von *Shared State*. Unter Aufgabe dieser Eigenschaften wäre die Definition einer eingeschränkten Version von BPEL möglich, die speziell im Hinblick auf eine dezentrale Ausführung ausgerichtet ist und die Verwendung von Sprachmitteln verbietet, die sich negativ auf das Laufzeitverhalten bei dezentraler Prozessausführung auswirken Ein weiteres Feld möglicher anknüpfender Untersuchungen zur Verbesserung des Laufzeitverhaltens ist – neben der Einschränkung der Sprachmittel von BPEL – die Senkung von Laufzeitzusicherungen, welche einen

negativen Einfluss auf das Prozesslaufzeitverhalten haben. Ein Beispiel hierfür ist die Vorgabe eines Freshness-Werts von Instanzdaten (vgl. Abschnitt 5.6.2.6) und ein damit verbundenes Intervall, in welchem auch ein nicht aktueller Wert einer Instanzvariable verwendet werden kann.

In Abschnitt 5.9.3 wurden erste Erwägungen zu den Auswirkungen der Prozessausführung, unter Verwendung des EWFN-Metamodells und der PS-Infrastruktur, auf die Autonomie der Ausführungsteilnehmer des Prozesses angestellt. Thematisch anknüpfend ergibt sich hier ein weiteres zukünftiges Arbeitsfeld, nämlich die mögliche Untersuchung der entstehenden Freiheitsgrade von domäneninterner Adaption in Ausführung befindlicher Prozesse, ohne dass dies Auswirkungen auf domänenexterne Interaktionspartner hat.

Das in Kapitel 4 vorgestellte Partitionierungsverfahren stützt sich gegenwärtig auf eine Reihe unterschiedlicher Einflussgrößen, hinsichtlich welcher eine Parametrisierung der Prozesspartitionierung erfolgen kann. Die im Kontext dieser Arbeit identifizierten Einflussgrößen und Partitionierungsparameter (z. B. die erwartete Ausführungswahrscheinlichkeit von Aktivitäten oder die erwartete durchschnittliche Größe von Instanzdatenzugriffen) wurden vor dem Hintergrund der Anwendbarkeit auf beliebige Prozessmodelle gewählt. Eine Erweiterung dieser Parameter und eine entsprechende Berücksichtigung durch das Partitionierungsverfahren kann das anwendungsspezifische Partitionierungsverhalten, beispielsweise bei der Ausführung lokationsbezogener Prozesse, verbessern.

Ebenso wie das entwickelte Partitionierungsverfahren stützt sich die PS-Infrastruktur auf die Anforderungen einer generischen Ausführungsumgebung für Produktionsprozesse in BPEL; die speziellen Anforderungen einzelner Anwendungsfelder wurden nicht betrachtet. Die Erschließung derartiger Anwendungsfelder und die Anpassung der PS-Infrastruktur an deren Gegebenheiten und Anforderungen ist ein Bereich möglicher zukünftiger Arbeiten. Einer dieser Bereiche sind die sogenannten *Scientific Workflows*. Sie haben die prozessgestützte Abbildung von komplexen Simulationen zum Gegenstand, in welchen Manipulationen auf potentiell umfangreiche Datenmengen erforderlich sind. Als Resultat dieser Eigenschaft kann hier eine Verteilung der Orchestrierungslogik auf die Ausführungsteilnehmer, die die für die Simulation benötigten

Daten einbringen (das sogenannte *function shipping*), ein im Vergleich zur traditionellen zentralen Prozessnavigation (dem sogenannten *data shipping*) verbessertes Laufzeitverhalten erzielen [SGK+10].

LITERATURVERZEICHNIS

[AAD+07a] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. Konig, F. Leymann, R. Müller, G. Pfau, et al. Web Services Human Task (WS-HumanTask), Version 1.0, 2007.

[AAD+07b] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. Konig, F. Leymann, R. Muller, G. Pfau, et al. WS-BPEL Extension for People (BPEL4People), Version 1.0, 2007.

[AADH05] L. Aldred, W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede. On the Notion of Coupling in Communication Middleware. In *On the Move to Meaningful Internet Systems: CoopIS, DOA, and ODBASE*. 2005.

[ACKM04] G. Alonso, F. Casati, H. Kuno, V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer Verlag, 2004.

[AH04] W. M. P. van der Aalst, K. van Hee. *Workflow Management: Models, Methods and Systems (Cooperative Information Systems)*. MIT Press, 2004.

[AH05] W. M. P. van der Aalst, A. H. M. ter Hofstede. YAWL: Yet another Workflow Language. In *Information Systems*, Volume 30. Elsevier, 2005.

[AJ00] W. M. P. van der Aalst, S. Jablonski. Dealing with Workflow Change: Identification of Issues and Solutions. In *Computer Systems Science & Engineering*, Volume 15. CRL Publishing LTD, 2000.

[AKD99] D. Abramson, M. Krishnamoorthy, H. Dang. Simulated annealing Cooling Schedules for the School Timetabling Problem. In *Asia Pacific Journal of Operational Research*, Volume 16. 1999.

[Bau01] T. Bauer. *Effiziente Realisierung unternehmensweiter Workflow-management-systeme*. Tenea Verlag, 2001.

[BBC+07] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, J. Simeon. XML Path Language (XPath) Version 2.0. W3C Recommendation, http://www.w3.org/TR/xpath20/, 2007.

[BCF+07] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simeon. XQuery 1.0: An XML Query Language. W3C Recommendation, http://www.w3.org/TR/xquery/, 2007.

[BD00] T. Bauer, P. Dadam. Efficient Distributed Workflow Management Based on Variable Server Assignments. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*. 2000.

[Ber97] P. A. Bernstein. *Principles of Transaction Processing*. Morgan Kaufmann, 1997.

[BGZ97] N. Busi, R. Gorrieri, G. Zavattaro. On the Turing Equivalence of Linda Coordination Primitives, 1997.

[BHL95] B. Blakeley, H. Harris, R. Lewis. *Messaging and Queuing Using the MQI: Concepts & Analysis, Design & Development*. Mcgraw-Hill (Tx), 1995.

[BJW87] A. Birrell, M. Jones, E. Wobber. A Simple and Efficient Implementation of a Small Database. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*. 1987.

[BLFM05] T. Berners-Lee, R. Fielding, L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, 2005. URL http://www.ietf.org/rfc/rfc3986.txt.

[BLMM94] T. Berners-Lee, L. Masinter, M. McCahill. Uniform Resource Locators (URL). RFC 1738, 1994. URL http://www.ietf.org/rfc/rfc1738.txt.

[BM06] B. Burke, R. Monson-Haefel. *Enterprise JavaBeans 3.0*. O'Reilly Media, 2006.

[CCMN04] G. B. Chafle, S. Chandra, V. Mann, M. G. Nanda. Decentralized Orchestration of Composite Web Services. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters (WWW Alt.)*. 2004.

[CCMN05] G. Chafle, S. Chandra, V. Mann, M. G. Nanda. Orchestrating Composite Web Services under Data Flow Constraints. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*. 2005.

[CCMW01] E. Christensen, F. Crubera, G. Meredith, S. Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Note, http://www.w3.org/TR/wsdl, 2001.

[CD+99] J. Clark, S. DeRose, et al. XML Path Language (XPath) Version 1.0. W3C Recommendation, http://www.w3.org/TR/xpath, 1999.

[CDK05] G. F. Coulouris, J. Dollimore, T. Kindberg. *Distributed Systems*. Pearson Education, 2005.

[CGM+04] R. Chinnici, M. Gudgin, J. J. Moreau, J. Schlimmer, S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3C Recommendation, http://www.w3.org/TR/wsdl20/, 2004.

[Cha04] D. A. Chappell. *Enterprise Service Bus*. O'Reilly, 2004.

[CHL+07] R. Chinnici, H. Haas, A. A. Lewis, J. J. Moreau, D. Orchard, S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts. W3C Recommendation, http://www.w3.org/TR/wsdl20-adjuncts/, 2007.

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*. The MIT Press, 2009.

[CLZ00] G. Cabri, L. Leonardi, F. Zambonelli. MARS: A Programmable Coordination Architecture for Mobile Agents. In *IEEE Internet Computing*, Volume 4. 2000.

[Com08] D. E. Comer. *Computer Networks and Internets*. Prentice Hall, 2008.

[CR96] P. Ciancarini, D. Rossi. Jada: A Coordination Toolkit for Java, 1996.

[CR97] P. Ciancarini, D. Rossi. Jada-Coordination and Communication for Java Agents. In *Mobile Object Systems: Towards the Programmable Internet*. 1997.

[Cra10] S. Craß. *A Formal Model of the Extensible Virtual Shared Memory (XVSM) and its Implementation in Haskell - Design and Specification*. Diplomarbeit, TU Wien, Österreich, 2010. URL http://www.ub.tuwien.ac.at/dipl/2010/AC07806750.pdf.

[Dan08] O. Danylevych. *Stratifizierte Transaktionen*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2008. URL

http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2663.

[Dem08] C. Demler. *Definition eines Verfahrens und Algorithmus zur Segmentierung von BPEL-Prozessen*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Deutschland, 2008. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2829.

[DH04] H. Dehling, B. Haupt. *Einführung in die Wahrscheinlichkeitstheorie und Statistik*. Springer, 2004.

[DKL09] O. Danylevych, D. Karastoyanova, F. Leymann. Optimal Stratification of Transactions. In *Proceedings of the International Conference on Internet and Web Applications and Services (ICIW)*. 2009.

[EHF06] J. Ellis, L. Ho, M. Fisher. JDBC 3.0 Specification. Sun Microsystems, http://java.sun.com/products/jdbc/download.html#corespec30, 2006.

[Els97] U. Elsner. Graph Partitioning – A Survey, 1997.

[Erl05] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall International, 2005.

[Far88] C. Farhat. A Simple And Efficient Automatic FEM Domain Decomposer. In *Computers and Structures*, Volume 28. 1988.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, 1999. URL http://www.ietf.org/rfc/rfc2616.txt.

[FHA99] E. Freeman, S. Hupfer, K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Pearson Education, 1999.

[FKLT07] D. Fensel, e. Kühn, F. Leymann, R. Tolksdorf. Queues Are Spaces - Yet Still Both Are Not The Same? http://www.spacebasedcomputing.org/fileadmin/files/SBC-QueuesAreSpaces_20070511.pdf, 2007.

[FOW87] J. Ferrante, K. J. Ottenstein, J. D. Warren. The Program Dependence Graph and its Use in Optimization. In *ACM Transactions on Programming Languages and Systems*, Volume 9. 1987.

[Fow03] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2003.

[FW04] D. C. Fallside, P. Walmsley. XML Schema Part 0: Primer (Second Edition). W3C Recommendation, http://www.w3.org/TR/xmlschema-0/, 2004.

[GC92] D. Gelernter, N. Carriero. Coordination Languages and their Significance. In *Communications of the ACM*, Volume 35. 1992.

[Gel85] D. Gelernter. Generative Communication in Linda. In *ACM Transactions on Programming Languages and Systems*, Volume 7. 1985.

[Gel89] D. Gelernter. Multiple Tuple Spaces in Linda. In *Proceedings of the Parallel Architectures and Languages Europe (PARLE), Volume II: Parallel Languages*. 1989.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., 1995.

[GHM+07a] M. Gudgin, M. Hadley, N. Mendelsohn, J. J. Moreau, H. F. Nielsen, A. Karmarkar, Y. Lafon. SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation, http://www.w3.org/TR/soap12-part1/, 2007.

[GHM+07b] M. Gudgin, M. Hadley, N. Mendelsohn, J. J. Moreau, H. F. Nielsen, A. Karmarkar, Y. Lafon. SOAP Version 1.2 Part 2: Adjuncts. W3C Recommendation, http://www.w3.org/TR/soap12-part2/, 2007.

[GHR06] M. Gudgin, M. Hadley, T. Rogers. Web Services Addressing 1.0 – Core. W3C Recommendation, http://www.w3.org/TR/ws-addr-core/, 2006.

[GLZ06] R. Gorrieri, R. Lucchi, G. Zavattaro. Supporting Secure Coordination in SecSpaces. In *Fundamenta Informaticae*, Volume 73. 2006.

[Goo07] R. L. Goodstein. *Boolean Algebra*. Dover Publications, 2007.

[GR92] J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Series in Data Management Systems, 1992.

[Har87] D. Harel. Statecharts: A Visual Formalism For Complex Systems. In *Science of Computer Programming, Volume 8, Issue 3*. 1987.

[HBN+04] H. Haas, D. Booth, E. Newcomer, M. Champion, D. Orchard, C. Ferris, F. McCabe. Web Services Architecture. W3C Working Group Note, http://www.w3.org/TR/ws-arch/, 2004.

[HK97] M. Horstmann, M. Kirtland. DCOM Architecture. http://msdn.microsoft.com/en-us/library/ms809311.aspx, 1997.

[HR01] T. Härder, E. Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer, Berlin, 2001.

[HSB98] Y. Han, A. Sheth, C. Bussler. A Taxonomy of Adaptive Workflow Management. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW)*. 1998.

[HW04] G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2004.

[Int99] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), 1999.

[JAMS89] D. S. Johnson, C. R. Aragon, L. A. McGeoch, C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation. Part I, Graph Partitioning. In *Operations Research*, Volume 37. 1989.

[JB96] S. Jablonski, C. Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation*. International Thomson Computer Press, 1996.

[Jen96] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use: Volume 1*. Springer, 1996.

[JS93] M. Jerrum, G. Sorkin. Simulated Annealing for Graph Bisection. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*. 1993.

[Kar06] D. Karastoyanova. *Enhancing Flexibility and Reusability of Web Service Flows through Parameterization*. Shaker Verlag, 2006.

[Kay03] D. Kaye. *Loosely Coupled: The Missing Pieces of Web Services*. RDS Press, 2003.

[KBS04] D. Krafzig, K. Banke, D. Slama. *Enterprise SOA. Service Oriented Architecture Best Practices*. Prentice Hall International, 2004.

[KGV83] S. Kirkpatrick, C. Gelatt, M. Vecchi. Optimization by Simulated Annealing. In *Science*, Volume 220. 1983.

[Küh94] e. Kühn. Fault-Tolerance for Communicating Multidatabase Transactions. In *Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS)*. 1994.

[Küh03] e. Kühn. The Zero-Delay Data Warehouse: Mobilizing Heterogeneous Database. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*. 2003.

[Kha07] R. Khalaf. Note on Syntactic Details of Split BPEL-D Business Processes, 2007. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2007-02.

[Kha08] R. Khalaf. *Supporting Business Process Fragmentation While Maintaining Operational Semantics: A BPEL Perspective*. dissertation.de, 2008.

[KKL+05] M. Kloppmann, D. Koenig, F. Leymann, G. Pfau, A. Rickayzen, C. von Riegen, P. Schmidt, I. Trickovic. WS-BPEL Extension for Sub-processes BPEL-SPE. Joint White Paper, IBM and SAP, 2005.

[KKL08] O. Kopp, R. Khalaf, F. Leymann. Deriving Explicit Data Links in WS-BPEL Processes. In *Proceedings of the International Conference on Services Computing (SCC)*. 2008.

[KKS+06] D. Karastoyanova, R. Khalaf, R. Schroth, M. Paluszek, F. Leymann. BPEL Event Model, 2006. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2006-10.

[KL70] B. W. Kernighan, S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. In *The Bell System Technical Journal*, Volume 49. 1970.

[KL06] R. Khalaf, F. Leymann. Role-based Decomposition of Business Processes using BPEL. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*. 2006.

[Kle08] J. Klensin. Simple Mail Transfer Protocol. RFC 5321, 2008. URL http://www.ietf.org/rfc/rfc5321.txt.

[KLN+06] D. Karastoyanova, F. Leymann, J. Nitzsche, B. Wetzstein, D. Wutke. Parameterized BPEL Processes: Concepts and Implementation. In *Proceedings of the Business Process Management Conference (BPM)*. 2006.

[KMKS09] e. Kühn, R. Mordinyi, L. Keszthelyi, C. Schreiber. Introducing the Concept of Customizable Structured Spaces for Agent Coordination in the Production Automation Domain. In *Proceedings of 8th International Confbiberence on Autonomous Agents and Multiagent Systems (AAMAS)*. 2009.

[KML08] O. Kopp, R. Mietzner, F. Leymann. Abstract Syntax of WS-BPEL 2.0, 2008. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2008-06.

[KMS08] e. Kühn, R. Mordinyi, C. Schreiber. An Extensible Space-Based Coordination Approach for Modeling Complex Patterns in Large Systems. In *Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. 2008.

[KMWL08] O. Kopp, D. Martin, D. Wutke, F. Leymann. On the Choice Between Graph-Based and Block-Structured Business Process Modeling Languages. In *Modellierung betrieblicher Informationssysteme (MobIS)*. 2008.

[KMWL09] O. Kopp, D. Martin, D. Wutke, F. Leymann. The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages. In *Enterprise Modelling and Information Systems*, Volume 4. 2009.

[KN98] eva Kühn, G. Nozicka. Post-Client/Server Coordination Tools. In *Coordination Technology for Collaborative Applications - Organizations, Processes, and Agents [ASIAN 1996 Workshop]*. Springer-Verlag, 1998.

[KNS92] G. Keller, N. Nüttgens, A.-W. Scheer. Semantische Prozessmodellierung auf der Grundlage ereignisgesteuerter Prozessketten (EPK), 1992. Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi).

[Kri98] J. Krinke. Static Slicing of Threaded Programs. In *ACM SIGPLAN Notices*, Volume 33. 1998.

[KRJ05] e. Kühn, J. Riemer, G. Joskowicz. XVSM (eXtensible Virtual Shared Memory) - Architecture and Application, 2005.

[KUL06] O. Kopp, T. Unger, F. Leymann. Nautilus Event-driven Process Chains: Syntax, Semantics, and their mapping to BPEL. In *Proceedings of the 5th GI Workshop on Event-Driven Process Chains (EPK)*. 2006.

[Les08] T. van Lessen. Web Service Orchestrierung mit BPEL 2.0. W-JAX/SOACON 2008. München, Deutschland, http://www.slideshare.net/vanto/web-\services-orchestration-with-bpel-20-presentation, 2008.

[Ley97] F. Leymann. Transaktionsunterstützung für Workflows. In *Informatik-Forschung und Entwicklung*, Volume 12. 1997.

[Ley03] F. Leymann. Web Services: Distributed Applications without Limits. In *Business, Technology and Web*. 2003.

[Ley06] F. Leymann. Space-based Computing and Semantics: A Web Service Purist's Point-Of-View, 2006. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2006-05.

[LMS05] P. Leach, M. Mealling, R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, 2005. URL http://www.ietf.org/rfc/rfc4122.txt.

[LMW99] T. J. Lehman, S. W. McLaughry, P. Wyckoff. TSpaces: The Next Wave. In *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences (HICSS)*. 1999.

[LR97] F. Leymann, D. Roller. Workflow-Based Applications. In *IBM Systems Journal: Application Development*. 1997.

[LR99] F. Leymann, D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, 1999.

[LSW97] P. Langner, C. Schneider, J. Wehler. Prozessmodellierung mit ereignisgesteuerten Prozessketten (EPKs) und Petri-Netzen. In *Wirtschaftsinformatik*. 1997.

[Mar10] D. Martin. *A Tuplespace-Based Execution Model for Decentralized Workflow Enactment*. 2010.

[MBH+04] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. R. Payne. OWL-S: Semantic Markup for Web Services. W3C Member Submission, http://www.w3.org/Submission/OWL-S/, 2004.

[Mil92] D. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305, 1992. URL http://www.ietf.org/rfc/rfc1305.txt.

[Mil99] R. Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999.

[MJGN08] D. Mukherjee, P. Jalote, M. Gowri Nanda. Determining QoS of WS-BPEL Compositions. In *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC)*. 2008.

[MKDM03] V. Matena, S. Krishnan, L. DeMichiel, L. de Michiel. *Applying Enterprise JavaBeans: Component-based Development for the J2EE Platform*. Addison-Wesley Longman, 2003.

[MKL+09] G. Monakova, O. Kopp, F. Leymann, S. Moser, K. Schäfers. Verifying Business Rules Using an SMT Solver for BPEL Processes. In *Proceedings of the Business Process and Services Computing Conference (BPSC)*. 2009.

[MM05] F. Montagut, R. Molva. Enabling Pervasive Execution of Workflows. In *Proceedings of the International Conference on Collaborative Computing: Networking, Applications and Worksharing*. 2005.

[Moc87] P. V. Mockapetris. Domain Names - Implementation and Specification. RFC 1035, 1987. URL http://www.ietf.org/rfc/rfc1035.txt.

[Mon08] G. Monakova. *Ontology Based Partner Service Discovery Using a First-Order Logic Representation for BPEL Process Models*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Deutschland, 2008. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2741.

[MR00] M. zur Muehlen, M. Rosemann. Workflow-based Process Monitoring and Controlling – Technical and Organizational Issues. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences (HICSS)*. 2000.

[Mue01a] M. zur Muehlen. Process-driven Management Information Systems Combining Data Warehouses and Workflow Technology. In *Proceedings of the Fourth International Conference on Electronic Commerce Research (ICECR)*. 2001.

[Mue01b] M. zur Muehlen. Workflow-based Process Controlling – Or: What You Can Measure You Can Control. In *Workflow Handbook*. 2001.

[MWL08a] D. Martin, D. Wutke, F. Leymann. EWFN – A Petri Net Dialect for Tuplespace-Based Workflow Enactment, 2008.

[MWL08b] D. Martin, D. Wutke, F. Leymann. A Novel Approach to Decentralized Workflow Enactment. In *Proceedings of the 12th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*. 2008.

[MWL08c] D. Martin, D. Wutke, F. Leymann. Synchronizing Control Flow in a Tuplespace-based, Distributed Workflow Management System. In *Proceedings of the 10th International Conference on Electronic Commerce (ICEC)*. 2008.

[MWL08d] D. Martin, D. Wutke, F. Leymann. Using Tuplespaces to Enact Petri Net-Based Workflow Definitions. In *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services (iiWAS)*. 2008.

[MWL09] D. Martin, D. Wutke, F. Leymann. Tuplespace Middleware for Petri Net-Based Workflow Execution. In *International Journal on Web and Grid Services (IJWGS)*, Volume 5. 2009.

[MWSL07] D. Martin, D. Wutke, T. Scheibler, F. Leymann. An EAI Pattern-Based Comparison of Spaces and Messaging. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*. 2007.

[MWW+98] P. Muth, D. Wodtke, J. Weissenfels, A. Dittrich, G. Weikum. From Centralized Workflow Specification to Distributed Workflow Execution. In *Journal of Intelligent Information Systems*, Volume 10. 1998.

[NA98] Y. Nourani, B. Andresen. A Comparison of Simulated Annealing Cooling Strategies. In *Journal of Physics A: Mathematical and General*, Volume 31. 1998.

[Nar00] N. C. Narendra. Adaptive Workflow Management – An Integrated Approach and System Architecture. In *Proceedings of the 2000 ACM symposium on Applied computing – Volume 2*. 2000.

[NCS04] M. G. Nanda, S. Chandra, V. Sarkar. Decentralizing Execution of Composite Web Services. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. 2004.

[NLKL07] J. Nitzsche, T. van Lessen, D. Karastoyanova, F. Leymann. BPEL^{light}. In *Proceedings of the Business Process Management Conference*. 2007.

[oasa] Using WSDL in a UDDI Registry, Version 1.08 – OASIS Technical Note. http://www.oasis-open.org/committees/uddi-spec/doc/bp/uddi-spec-tc-bp-using-wsdl-v108-20021110.pdf.

[oasb] Using WSDL in a UDDI Registry, Version 2.0.2 – OASIS Technical Note. http://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-wsdl-v2.htm.

[oas02] UDDI Version 2.03 Data Structure Reference – OASIS Standard. http://uddi.org/pubs/DataStructure_v2.htm, 2002.

[oas04] UDDI, Version 3.0 – OASIS Published Specification. http://www.uddi.org/pubs/uddi_v3.htm, 2004.

[ODHA06] C. Ouyang, M. Dumas, A. H. M. ter Hofstede, W. M. P. van der Aalst. From BPMN Process Models to BPEL Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*. 2006.

[OHV+] D. Orchard, F. Hirsch, A. S. Vedamuthu, P. Yendluri, T. Boubez, Ü. Yalçinalp, M. Hondo. Web Services Policy 1.5 - Framework. W3C Recommendation, http://www.w3.org/TR/2007/REC-ws-policy-20070904.

[OMG04] Common Object Request Broker Architecture: Core Specification – Version 3.0.3, 2004. URL http://www.omg.org/docs/formal/04-03-12.pdf.

[OMG08] Business Process Modeling Notation, Version 1.1, 2008. URL http://www.omg.org/spec/BPMN/1.1/PDF.

[Org07] Web Services Business Process Execution Language Version 2.0 – OASIS Standard. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007.

[Org09] Web Services Coordination (WS-Coordination) 1.2 – OASIS Standard. http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec.html, 2009.

[Pap07] M. P. Papazoglou. *Web Services: Principles and Technology*. Longman, 2007.

[Pat99] N. W. Paton, Editor. *Active Rules in Database Systems*. Springer, New York, 1999.

[Pet80] C. A. Petri. Introduction to General Net Theory. In *Net Theory and Applications*. 1980.

[Pos81a] J. Postel. Internet Protocol. RFC 791, 1981. URL http://www.ietf.org/rfc/rfc791.txt.

[Pos81b] J. Postel. Transmission Control Protocol. RFC 793, 1981. URL http://www.ietf.org/rfc/rfc793.txt.

[PS98] C. H. Papadimitriou, K. Steiglitz. *Combinatorial Optimization*. Courier Dover Publications, 1998.

[Puh07] F. Puhlmann. *On the Application of A Theory for Mobile Systems to Business Process Management*. Dissertation. Universität Potsdam, Deutschland, 2007.

[Rei85] W. Reisig. *Petri Nets. An Introduction*. Springer, 1985.

[RHAM06] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, N. Mulyar. Workflow Control-Flow Patterns: A Revised View. In *BPM Center Report BPM-06-22, BPMcenter.org*. 2006.

[RM06] J. Recker, J. Mendling. On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. In *Proceeding of the 11th International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD)*. 2006.

[RN02] S. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2002.

[RW96] A. I. T. Rowstron, A. Wood. Solving the Linda Multiple rd Problem. In *Proceedings of the First International Conference on Coordination Languages and Models (COORDINATION)*. 1996.

[RW97] A. I. T. Rowstron, A. M. Wood. BONITA: A Set of Tuple Space Primitives for Distributed Coordination. In *Proceedings of the 30th HICSS: Software Technology and Architecture*. 1997.

[SA04] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 3920, 2004. URL http://www.ietf.org/rfc/rfc3920.txt.

[Sch05] C. Schuler. *Verteiltes Peer-to-Peer-Prozessmanagement - Die Realisierung einer Hyperdatenbank*. Ph.D. Thesis, ETH Zürich, 2005.

[Sch07] A. Schwind. *Space-Based Web Services: Konzepte und prototypische Implementierung mit Linda-Spaces*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Deutschland, 2007. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2692.

[SCK+08] O. Shafiq, D. Cerizza, J. Kopecky, D. Martin, M. Murth, B. Sapkota, G. T. del Valle, A. Turati, D. Wutke. TripCom Grounding for Semantic Web Services. http://tripcom.org/docs/del/D4.2.pdf, 2008.

[Seb07] R. W. Sebesta. *Concepts of Programming Languages*. Pearson Education (US), 8th revised international edition edition, 2007.

[SGK+10] M. Sonntag, K. Görlach, D. Karastoyanova, F. Leymann, M. Reiter. Process Space-Based Scientific Workflow Enactment. In *International Journal of Business Process Integration and Management (IJBPIM)*. 2010.

[SKI08] S. Stein, S. Kühne, K. Ivanov. Business to IT Transformations Revisited. In *Proceedings of the Workshop on Model-Driven Engineering for Business Process Management (MDE4BPM)*. 2008.

[SS83] D. Skeen, M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System. In *IEEE Transactions on Software Engineering*, Volume SE-9. 1983.

[sun03] JavaSpaces Service Specification. http://www.sun.com/software/jini/specs/js2_0.pdf, 2003.

[sun06a] Java Management Extensions (JMX) 1.4. http://java.sun.com/javase/6/docs/technotes/guides/jmx/JMX_1_4_specification.pdf, 2006.

[sun06b] Java Remote Method Invocation. http://java.sun.com/javase/6/docs/platform/rmi/spec/rmiTOC.html, 2006.

[sun08] Java Message Service Specification, Version 1.1. SUN Microsystems, 2008. http://java.sun.com/products/jms/docs.html.

[SW03] D. Sangiorgi, D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003.

[SWSS03] C. Schuler, R. Weber, H. Schuldt, H. Schek. Peer-to-Peer Process Execution with OSIRIS. In *Proceedings of the International Conference on Service-Oriented Computing (ICSOC)*. 2003.

[SWSS04] C. Schuler, R. Weber, H. Schuldt, H. J. Schek. Scalable Peer-to-Peer Process Management - The OSIRIS Approach. In *Proceedings of the International Conference on Web Services*. 2004.

[Tan01] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2001.

[Tan02] A. S. Tanenbaum. *Computer Networks*. Pearson Education, 2002.

[TG01] R. Tolksdorf, D. Glaubitz. Coordinating Web-Based Systems with Documents in XMLSpaces. In *Cooperative Information Systems*. 2001.

[TM04] R. Tolksdorf, R. Menezes. Using Swarm Intelligence in Linda Systems. In *Lecture Notes in Computer Science*. Springer, 2004.

[TNW+08] K. Teymourian, L. Nixon, D. Wutke, R. Krummenacher, H. Moritsch. Implementation of a Novel Semantic Web Middleware Approach Based on Triplespaces. In *Proceedings of the International Conference on Semantic Computing (ICSC)*. 2008.

[Var09] S. Varnhorn. *Verteilte Workflow Engine: Implementierung einer Runtime Umgebung für BPEL Prozesse auf Basis von EWFNs*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Deutschland, 2009. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2874.

[WCD95] J. Widom, S. Ceri, U. Dayal. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufman, 1995.

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, 2005.

[WCL+08] D. Wutke, D. Cerri, M. Lafite, D. Martin, J. Nitzsche, C. Schreiber. Methodology for Augmenting Semantic Web Services with WS-Standards. http://tripcom.org/docs/del/D4.4.pdf, 2008.

[WDGW08] M. Weidlich, G. Decker, A. Großkopf, M. Weske. BPEL to BPMN: The Myth of a Straight-Forward Mapping. In *Proceedings of the International Conference on Cooperative Information Systems (CoopIS)*. 2008.

[wfm95] Workflow Reference Model. http://www.wfmc.org/standards/docs/tc003v11.pdf, 1995.

[wfm98] Audit Data Specification. http://www.wfmc.org/View-document-details/WFMC-TC-1015-Ver-1.1-Audit-Data-Specification.html, 1998.

[WJT00] J. Waldo, T. Jini-Team. *The Jini(TM) Specifications*. Addison-Wesley Professional, 2000.

[WML08a] D. Wutke, D. Martin, F. Leymann. Facilitating Complex Web Service Interactions through a Tuplespace Binding. In *Proceedings of Distributed Applications and Interoperable Systems (DAIS)*. 2008.

[WML08b] D. Wutke, D. Martin, F. Leymann. Model and Infrastructure for Decentralized Workflow Enactment. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC)*. 2008.

[WML08c] D. Wutke, D. Martin, F. Leymann. Triple Space Binding for Web Services, 2008.

[WML09a] D. Wutke, D. Martin, F. Leymann. A Method for Partitioning BPEL Processes for Decentralized Execution. In *Proceedings of the 1st Central-European Workshop on Services and their Composition (ZEUS)*. 2009.

[WML09b] D. Wutke, D. Martin, F. Leymann. Tuplespace-based Infrastructure for Decentralized Enactment of BPEL Processes. In *Proceedings of 9. Internationale Tagung Wirtschaftsinformatik: Business Services, Konzepte, Technologien, Anwendungen (WI)*. 2009.

[WMLF98] P. Wycko, S. McLaughry, T. Lehman, D. Ford. TSpaces. In *IBM Systems Journal*, Volume 37. 1998.

[WPSW05] S. J. Woodman, D. J. Palmer, S. K. Shrivastava, S. M. Wheater. DECS: A System for Decentralised Coordination of Web Services. In *Proceedings of the Workshop on Middleware for Web Services (MWS)*. 2005.

[Wu08] S. Wu. *Verteilte Workflow-Engine: Ausführung verteilter Prozessmodelle auf Basis von Tuplespaces*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Deutschland, 2008. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2831.

[Wut06] D. Wutke. *Erweiterung einer Workflow-Engine zur Unterstützung von parametrisierten Web Service Flows*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Deutschland, 2006. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2401.

[WW97] D. Wodtke, G. Weikum. A Formal Foundation for Distributed Workflow Execution based on State Charts. In *Proceedings of the International Conference on Database Theory (ICDT)*. 1997.

[WWWa] JBoss JBPM – The Process Virtual Machine. http://docs.jboss.com/jbpm/pvm/article/.

[WWWb] ActiveVOS – Server Architecture. http://www.activevos.com/indepth/f_technicalNotes/a_

activeVOSArchitecture/ActiveBPELArchitecture.pdf.

[WWWD96] D. Wodtke, J. Weissenfels, G. Weikum, A. Dittrich. The Mentor Project: Steps Toward Enterprise-Wide Workflow Management. In *Proceedings of the Twelfth International Conference on Data Engineering (ICDE)*. 1996.

[YBV+07] P. Yendluri, T. Boubez, A. S. Vedamuthu, Ü. Yalçinalp, D. Orchard, F. Hirsch, M. Hondo. Web Services Policy 1.5 - Attachment. W3C Recommendation, http://www.w3.org/TR/2007/REC-ws-policy-attach-20070904, 2007.

[YY07] W. Yu, J. Yang. Continuation-Passing Enactment of Distributed Recoverable Workflows. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC)*. 2007.

[Zlo77] M. Zloof. Query-by-Example: A Data Base Language. In *IBM Systems Journal*, Volume 16. 1977.

[ZM05] J. Ziemann, J. Mendling. EPC-Based Modelling of BPEL Processes: a Pragmatic Transformation Approach. In *Proceedings of the 7th International Conference Modern Information Technology in the Innovation Processes of the Industrial Enterprises (MITIP)*. 2005.

Hyperlinks wurden zuletzt geprüft am 22.12.2009.

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

Algorithmenverzeichnis

SYMBOLVERZEICHNIS

Menge/Funktion	Beschreibung
$\pi_i(t)$	Projektion der i-ten Komponente des Tupels t
$\mathcal{A}$	Menge der Aktivitäten eines Prozesses
$\mathcal{E}$	Menge der Events eines Prozesses
$\mathcal{L}$	Menge der Synchronisationskanten eines Prozesses
$\mathcal{V}$	Menge der Variablen eines Prozesses
$CORSET$	Menge der Correlation-Sets eines Prozesses
PL	Menge der Partner-Links eines Prozesses
MEX	Menge der Message-Exchanges eines Prozesses
$FROM$	Menge der FROM-Elemente der ASSIGN-Aktivitäten eines Prozesses

TO	Menge der TO-Elemente der Assign-Aktivitäten eines Prozesses
WS	Menge der dokumentierten Web-Service-Implementierungen
PT	Menge der Port-Typen
B	Menge der *Labels* eines Prozesses, es gilt $\mathcal{B} = \mathcal{E} \cup \perp$
HR	Die *Hierachierelation* mit HR $\subseteq \mathcal{A} \times \mathcal{B} \times \mathcal{A}$
SR $\subseteq \mathcal{A} \times \mathcal{C} \times \mathcal{L}$	*Source*-Relation eines Prozesses
TR $\subseteq \mathcal{A} \times \mathcal{L}$	*Target*-Relation eines Prozesses
LR $\subseteq \mathcal{A} \times \mathcal{L} \times \mathcal{C} \times \mathcal{A}$	*Link*-Relation eines Prozesses, wobei $\mathcal{C}$ die Menge der *Transition-Conditions* eines Prozesses beschreibt
$\mathcal{L}_{in} : \mathcal{A} \rightarrow 2^{\mathcal{L}}$	Menge der eingehenden Synchronisationskanten einer Aktivität
$<^{s}_{seq}$	Ordungsoperator auf den Kind-Elementen einer Sequence-Aktivität $sin\mathcal{A}_{Sequence}$
$<^{i}_{if}$	Ordnungsoperator auf den Kind-Elementen einer If-Aktivität $i \in \mathcal{A}_{If}$
$\mathsf{type}_{\mathbb{L}} : \mathbb{L} \rightarrow$ TYPES	Typ eines Elements eines Prozesses
$\mathsf{children} : \mathcal{A} \rightarrow 2^{\mathcal{A}}$	Kind-Aktivitäten einer Aktivität eines Prozesses
$\mathsf{partnerLink}_{CO} : \mathcal{CO}_{message} \rightarrow PL$	Partner-Link einer Interaktionsaktivität
$\mathsf{portType}_{CO} : \mathcal{CO}_{message} \rightarrow PT$	Port-Type einer Interaktionsaktivität
O_{part}	Menge der Partitionierungsobjekte eines Prozesses

O_{param}	Menge der parametrisierbaren Objekte eines Prozesses
$\mathcal{PAR}$	Menge der Partitionierungsparameter
PAR	Menge der Typen der Partitionierungsparameter
$\mathsf{type}_{\mathcal{PAR}} : \mathcal{PAR} \rightarrow PAR$	Funktion zur Bestimmung des Typs eines Partitionierungsparameters
$\mathsf{portType}_{\mathcal{WS}} : \mathcal{WS} \rightarrow 2^{PT}$	Die Menge der WSDL-Port-Types, die von einem Web-Service implementiert werden
$\mathsf{param} : O_{param} \rightarrow 2^{\mathcal{PAR}} \cup \bot$	Funktion, die einem parametrisierbaren Objekt eine Parametermenge oder die leere Parametrisierung zuordnet
$\mathsf{compatible} : \mathcal{PAR}_{NFP} \times \mathcal{PAR}_{NFP} \rightarrow \mathbb{B}$	Kompatibilität nicht-funktionaler Parametrisierungen
$\mathcal{P}$	Menge der Partitionen
$\mathsf{partition} : O_{part} \cup \mathcal{PAR}_{FIX} \cup \mathcal{WS} \rightarrow \mathcal{P} \cup \bot$	Zuordnung einer Partition
$a \preceq_f ws$	Funktionale Kompatibiltät von Dienstnutzer $a \in \mathcal{A}_{Invoke}$ und $ws \in \mathcal{WS}$
$a \preceq_{nf} ws$	Nicht-funktionale Kompatibiltät von Dienstnutzer $a \in \mathcal{A}_{Invoke}$ und $ws \in \mathcal{WS}$
$\mathsf{parent} : \mathcal{A} \rightarrow \mathcal{A} \cup \mathcal{E} \cup \bot$	Eltern-Aktivität einer Aktivität
$\mathsf{parents} : \mathcal{A} \rightarrow 2^{\mathcal{A}}$	Eltern-Aktivitäten einer Aktivität
$\mathsf{source}_{\mathcal{L}} : \mathcal{L} \rightarrow \mathcal{A}$	Quelle einer Synchronisationskante
$\mathsf{target}_{\mathcal{L}} : \mathcal{L} \rightarrow \mathcal{A}$	Ziel einer Synchronisationskante
P	Menge der Stellen eines EWFN
T	Menge der Transitionen eines EWFN

S	Menge der Kanten eines EWFN
R	Menge der Kantentypen eines EWFN
$F : S \rightarrow (T \times P) \cup (P \times T \times R)$	Quelle und Ziel einer EWFN-Kante
X	Menge der Templates eines EWFN
$A : (P \times T \times R) \rightarrow X$	Zuordnung von Templates zu Kanten
$L_{read} : (X \times 2^{\Sigma^{MS}}) \rightarrow \Sigma$	Beschreibung der Leseoperation eines EWFN
$L_{write} : (T \times P) \rightarrow \Sigma$	Beschreibung der Schreiboperation eines EWFN
$\text{EWFN}_{\text{Part}}$	EWFN-Partitionierungsgraph
$\tilde{F}$	Quelle und Ziel einer Kante des EWFN-Partitionierungsgraphen
$\text{weight}_S : S \rightarrow \mathbb{R}$	Gewicht einer Kante des EWFN-Partitionierungsgraphen
$\text{isDataAccess}_S : S \rightarrow \mathbb{B}$	Funktion, die bestimmt, ob eine Kante eines EWFN-Graphen einen Instanzdatenzugriff (oder einen Zugriff auf eine Kontrollflussinformation) beschreibt
$\text{mapsToBPELElement} : P \cup S \cup T \rightarrow O_{part}$	Abbildung eines EWFN-Elements auf eine Partitionierungsobjekt eines Prozesses
$\text{mapsToParam} : P \cup S \cup T \rightarrow 2^{\mathcal{PAR}} \cup \perp$	Abbildung eines EWFN-Elements auf die Parametrisierungen seines zugehörigen Partitionierungsobjekts
$\text{noi} : O_{part} \rightarrow \mathbb{N} \cup \perp$	Anzahl der erwarteten Ausführungen eines Partitionierungsobjekts
$\text{ep} : O_{part} \rightarrow \mathbb{R} \cup \perp$	Ausführungswahrscheinlichkeit eines Partitionieungsobjekts

$\mathsf{join}(a)$	Berechnung der Join-Condition einer Aktivität $a \in \mathcal{A}$
$\mathsf{partition}_{PT} : P \cup T \to \mathcal{P}$	Partition eines EWFN-Elements
Ω	Lösungsraum eines Optimierungsproblems
ω	Kostenfunktion eines Optimierungsproblems
$\preceq$	Operator, der eine Ordnung auf den Elementen des Lösungsraums eines Optimierungsproblems definiert
$O_{global} \in \Omega$	Das globale Optimum eines Optimierungsproblems
$O_{lokal} \in \Omega$	Ein lokales Optimum eines Optimierungsproblems
$\varphi : \Omega \to 2^{\Omega}$	Nachbarschaft eines Elements des Lösungsraums eines Optimierungsproblems
$\mathsf{reduceTemperature} : \mathbb{R} \times \mathbb{N} \to \mathbb{R}$	Abkühlungsschritt in Simulated Annealing
$\mathsf{source}_S : S \to P \cup T$	Quelle eines EWFN-Kante
$\mathsf{target}_S : S \to P \cup T$	Ziel einer EWFN-Kante
$\mathsf{crossingArcs}_{PT} : P \cup T \to 2^S$	Partitionsübergreifende Kanten eines Knotens des EWFN-Graphen
$\mathsf{fixed}_{PT} : P \cup T \to \mathbb{B}$	Funktion, die definiert, ob die Partition eines Knotens des EWFN-Graphen verändert werden darf