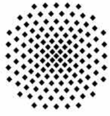
Institut für Architektur von Anwendungssystemen (IAAS)



Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart

Diplomarbeit Nr. 2874

# Verteilte Workflow-Engine:

# Implementierung einer Runtime Umgebung für BPEL Prozesse auf Basis von EWFNs

Stefan Varnhorn

| | |
|---|---|
| Studiengang: | Informatik |
| Prüfer: | Prof. Dr. Frank Leymann |
| Betreuer: | Dipl.-Inf. Daniel Wutke<br>Dipl.-Inf. Daniel Martin |
| Begonnen am: | 11.09.2008 |
| Beendet am: | 13.03.2009 |
| CR-Klassifikation: | C.2.4, H.4.1 |

# Inhaltsverzeichnis


**1 Einleitung** **5**

1.1 Motivation....6

1.2 Aufgabenstellung....7

1.3 Verwandte Arbeiten....8

1.4 Aufbau der Arbeit....9

**2 Grundlagen** **10**

2.1 Geschäftsprozessmanagement....10

2.1.1 Workflows und Workflow Management Systeme....11

2.2 Serviceorientierte Architekturen....13

2.3 Web Services....15

2.4 WS-BPEL....17

2.5 Tuple Spaces....18

2.6 Executable Workflow Networks (EWFN)....20

**3 Workflows mit Apache ODE** **23**

3.1 Architektur....24

3.2 BPEL-Compiler und ODE-Objektmodell....25

3.3 Ausführung von Prozessinstanzen....27

3.3.1 Beispiel für Ausführung eines Prozess-Kontrollflusses....29

3.3.2 Datenaustausch zwischen den Aktivitäten....33

**4 Verteile Ausführung von Workflows** **35**

4.1 Grundlegende Vorgehensweise....35

4.2 Verteilte Ausführung mit Tuple Spaces....36

4.3 Konfiguration der BPEL-Elemente....38

4.3.1 Konfigurationsinformationen....38

4.3.2 Deployment Descriptor....43

4.4 Tuplestruktur....49

4.4.1 Kontrollflusstoken....50

4.4.2 Datentoken....56

**5 Implementierung** **64**

5.1 Grundstruktur....64

5.2 Architektur....66

5.2.1 Laufzeitumgebung....67

5.3 Aktivitätsimplementierung....68

5.3.1 Aufbau einfacher Aktivitäten....69

5.3.2 Aufbau strukturierter Aktivitäten....71

5.4 Schnittstellenobjekte zum Zugriff auf Tuple Spaces....73

5.4.1 Kontrollflussschnittstelle....74

5.4.2 Datenaustauschschnittstelle....75

**6 Zusammenfassung und Ausblick** **77**

**Abkürzungsverzeichnis** **79**

**Abbildungsverzeichnis** **81**

**Verzeichnis der Listings** **83**

**Tabellenverzeichnis** **85**

**Literaturverzeichnis** **87**

# 1 Einleitung

Ein Geschäftsprozess ist eine zusammenhängende Folge von Aktivitäten, die in einem Unternehmen zur Erreichung eines oder mehrerer Ziele durchgeführt werden [Sta06]. Diese Ziele sind immer mit einer Wertschöpfung für das Unternehmen verbunden, sei es durch Fertigung eines Produktes oder durch Erbringen einer Dienstleistung. Da die dabei entstehenden Kosten und die benötigte Zeitdauer sehr stark von den Geschäftsprozessen abhängen, besitzen diese für Unternehmen eine enorme strategische Bedeutung.

Um die Dauer und Kosten für bestimmte Abläufe zu reduzieren, versucht man diese vollständig bzw. zu einem beträchtlichen Teil zu automatisieren. Dazu wird ihre Ausführung meist durch den Einsatz von Softwarelösungen unterstützt. Wurde dies früher häufig durch eine einzige große Anwendung realisiert, versucht man heutzutage davon ab zu kommen und kleinere Applikationen in einem Prozessablauf zu integrieren. Der Nachteil einer riesigen monolithischen Anwendung liegt hauptsächlich in der fehlenden Flexibilität. So dauert die Anpassung eines bestehenden Softwaresystems an neue Anforderungen oder geänderte Prozesse häufig sehr lange. Hinsichtlich der sich in der heutigen Zeit meist rasch ändernden Rahmenbedingungen und der ständigen Optimierung der Prozesse kann sich dies kaum ein Unternehmen leisten. Deshalb sieht ein neuerer Ansatz vor, Geschäftsprozesse durch die koordinierte Zusammenarbeit einzelner Dienste, welche bestimmte Aktivitäten umsetzen, zu realisieren. Dies führt zu dem Konzept der *serviceorientierten Architekturen* (SOA) [WCL+05], welche die „starre Verzahnung von IT-Systemen und Prozessabläufen [aufbrechen]“ [IDS08]. Ein Prozess wird zunächst nur auf einer fachlichen Ebene modelliert und in einem zweiten Schritt dann technisch ausführbar gemacht. Dazu werden den Aktivitäten des Prozessablaufes bestimmte Dienste zugeordnet, welche die gewünschte Funktionalität besitzen. Zur Ausführung eines solchen Prozesses ist dann eine Steuerungskomponente nötig, welche durch den modellierten Prozess steuert und zu einer bestimmten Aktivität den entsprechenden Dienst aufruft.

Dieses Konzept der serviceorientierten Architekturen wird in der heutigen Zeit hauptsächlich mittels *Web Services* und den dazugehörigen Technologien [W3C02]

umgesetzt. Dies sind Softwaredienste, die eine eindeutig beschriebene Schnittstelle besitzen und auf welche plattformunabhängig mittels Internettechnologien zugegriffen werden kann. Um Web Services nun zu einem Geschäftsprozess zusammenzufassen („*Orchestrierung*“) hat sich in der Praxis die *Web Service Business Process Execution Language* (WS-BPEL oder kurz BPEL) [OAS07] durchgesetzt. Sie ermöglicht die Modellierung von Geschäftsprozessen, bei denen die Aktivitäten durch Web Services realisiert sind. Für die Ausführung eines BPEL-Prozessmodells ist dann eine entsprechende Workflow Laufzeitumgebung nötig, welche den WS-BPEL-Standard implementiert. Diese übernimmt die Aufgabe der Navigation durch laufende Prozessinstanzen sowie die Kommunikation mit den externen Web Services.

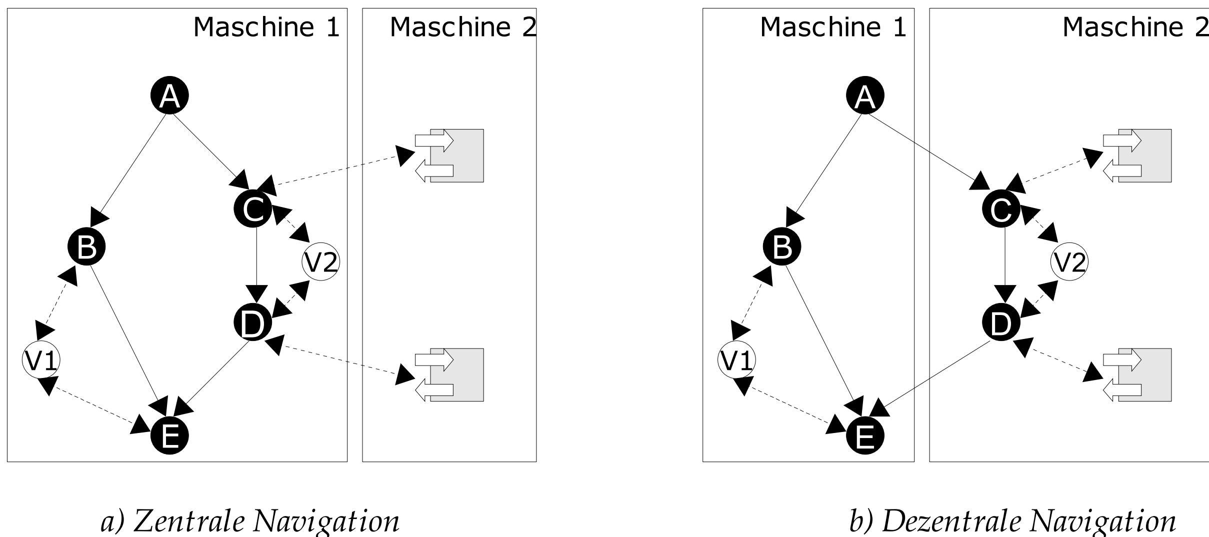
Mittlerweile sind eine Reihe solcher BPEL Workflow Engines auf dem Markt erhältlich, welche sich in Leistung oder Umfang der BPEL-Unterstützung unterscheiden (vgl. [LSR06]). Eine solche BPEL-Engine steht auch im Mittelpunkt dieser Arbeit, die die Entwicklung einer neuartigen verteilten Engine beschreibt.

## 1.1 Motivation

Um einen Geschäftsprozess ausführen zu können, muss ein *Workflow Management System* (WfMS) [LR00], wie zuvor erwähnt, durch das entsprechende Prozessmodell navigieren können. Bei bestehenden Workflow Engines wird diese Steuerung durch eine einzelne Komponente realisiert. Auch wenn das gesamte WfMS verteilt sein kann, ist dieser „*Navigator*“ stets eine einzige logische Komponente jeder Engine. Der Kontrollfluss eines Prozesses wird also immer zentral auf einer einzelnen Maschine bewertet und gesteuert, nämlich dort wo die Navigationskomponente läuft.

In Abbildung 1.1 (a) (vgl. [MWL08b]) ist ein einfacher Geschäftsprozess bei der Ausführung mit einer traditionellen BPEL-Engine dargestellt. Auf der Maschine 1 befindet sich die dabei Navigationskomponente. Hier wird der Prozessablauf, bestehend aus den (BPEL-) Aktivitäten A bis E, sowie den Variablen V1 und V2 ausgeführt, das heißt, der „Navigator“ steuert durch den Kontrollfluss des Prozesses und speichert die Instanzdaten der Variablen. In dem abgebildeten Beispiel sind die Aktivitäten C und D nun `invoke` Aktivitäten, die Web Services aufrufen, welche auf der Maschine 2 laufen. Zudem verwenden diese zwei Aktivitäten - und nur diese beiden - die Variable V2, welche als Teil des BPEL-Prozesses auf Maschine 1 liegt. Somit müssen mehrere Kommunikationsschritte, also Austauschoperationen von Nachrichten, zwischen den Maschinen 1 und 2 stattfinden, wobei der Zustand der Variablen V2 jeweils mit übertragen wird. Diese Interaktionen können nun „teuer“ sein, beispielsweise aufgrund einer geringen Bandbreite des Kommunikationskanals. Auch könnte mit der Variable V2 eine sehr große Datenmenge unnötigerweise mehrfach übertragen wer-

den, obwohl sie eigentlich sowieso nur auf Maschine 2 benötigt wird. Dieses kleine Beispiel zeigt schon recht deutlich, dass der traditionellen Ansatz der zentralen Ausführung von BPEL-Prozessen seine Schwächen besitzt und nicht unbedingt optimal ist.




*a) Zentrale Navigation* *b) Dezentrale Navigation*

*Abbildung 1.1: Zentrale und dezentrale Navigation*

Um diese Nachteile zu beheben, wird das Prozessmodell in der Abbildung 1.1 (b) auf beide teilnehmenden Maschinen verteilt. Die Navigation durch den Prozess findet nun verteilt auf beiden Maschinen statt. Nachdem die Aktivität A durchgeführt wurde, wird der Kontrollfluss teilweise an die Maschine 2 weitergegeben. Dort werden die entsprechenden Aktivitäten C und D ausgeführt. Da die Variable V2 nur von diesen benötigt wird, muss sie nicht mehrfach übertragen werden, sondern kann direkt auf Maschine 2 gespeichert werden. Nach Beendigung der Aktivität D wird der Kontrollfluss wieder an den Partner zurückgegeben.

In dem Beispiel wurde gezeigt, dass es effektiver sein kann, die Auswertung der Kontrollflusslogik näher an die tatsächlichen Funktionen zu verschieben. Diese Idee der Verschiebung von zentraler zur dezentralen Ausführung wird in dieser Arbeit weiterverfolgt und eine entsprechende prototypische BPEL Workflow Engine implementiert, welche eine verteilte Ausführung von BPEL-Prozessen erlaubt.

## 1.2 Aufgabenstellung

Grundlage der zu entwickelnden verteilten Workflow Engine ist das Modell der *Executable Workflow Networks* (EWFN) [MWL08a]. Diese Variante der Petrinetze basiert unter anderem auf dem Konzept der Tuple Spaces [Gel85]. In zwei vorangegangen

Arbeiten wurde bereits eine theoretische Abbildung von WS-BPEL 2.0 auf EWFN vorgestellt [Pop08], sowie eine Tuple Space Middleware implementiert [Wu08].

In dieser Arbeit sollen nun die beiden Vorgängerarbeiten zu einer verteilten Workflow Engine zusammengefügt werden, welche auf eine zentrale Vorgehensweise verzichtet und eine dezentrale Ausführung ermöglicht.

Dazu gliedert sich die Aufgabenstellung in drei Teile: Im ersten Teil soll die bestehende Open Source BPEL Engine *Apache ODE* [ODE] hinsichtlich deren Navigation und Datenaustausch zwischen Aktivitäten analysiert werden. Im zweiten, konzeptionellen Teil soll auf Basis der existierenden Vorarbeiten eine Tokenstruktur sowie entsprechende Templates entwickelt werden, die eine Ausführung der EWFNs auf der bestehenden Tuple Space Infrastruktur ermöglicht. Diese sollen im dritten, praktischen Teil prototypisch mit Hilfe der existierenden BPEL-Aktivitätsimplementierungen von Apache ODE umgesetzt werden.

## 1.3 Verwandte Arbeiten

Die Abbildung von Workflows auf Petrinetze wurde schon in einigen Arbeiten behandelt. So liefert [Aal98] einen grundsätzlichen Überblick über die Verwendung von Petrinetzen im Bereich Workflow Management. Dabei wird vor allem auf die Möglichkeit der Modellierung von Prozessen durch Petrinetze und deren Analyse eingegangen. In [Sta05] wird nun eine Pattern-basierte Abbildung von BPEL auf Petrinetze vorgestellt. Allerdings ist auch hier der Fokus auf der Verifikation von Prozessen, also dem statischen Überprüfen auf Korrektheit oder Erreichbarkeit von Elementen. Im Gegensatz dazu zielt der in dieser Arbeit verwendete Ansatz der EWFN [MWL08a] auf die verteilte Ausführung von BPEL-Prozessen.

Ebenfalls eine Reihe an verschiedenen Arbeiten behandeln das Thema der verteilten Workflow Management Systeme. So zeigt [JSH+01] Gründe, die für eine verteilte Ausführung von Workflows sprechen, wie Organisationsstrukturen und technische Infrastrukturen. Eine Architektur für verteilte Workflow Systeme wird in [MWW+98] eingeführt. Hierbei wird ein Prozessmodell in mehrere Teilprozesse aufgeteilt, welche bei verschiedenen Partnern ausgeführt werden. Die Synchronisation der einzelnen Teilprozesse erfolgt hierbei über eine Manager-Komponente (TP-Monitor). In [KL06] wird ein Ansatz für die verteilte Ausführung von BPEL-Prozessen wird vorgestellt. Dabei wird ebenfalls der BPEL-Prozess in verschiedene Fragmente aufgeteilt. Dies sind jedoch wiederum kleinere BPEL-Prozesse, die miteinander kommunizieren. Zur Ausführung kann jedes Fragment auf ein separates Workflow Management System eingebracht werden (Deployment). Auf eine zentrale Synchronisationskomponente

kann hierbei zum Teil verzichtet werden, da die Interaktionen zwischen den Fragmenten direkt über BPEL-Elemente, wie `invoke` oder `receive` realisiert werden. Für manche BPEL-Aspekte wie Schleifen oder aufgeteilte Scopes sind aber doch Koordinationsprotokolle nötig.

## 1.4 Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich in fünf Kapitel. Nach diesem Einleitungskapitel wird in Kapitel 2 zuerst ein einheitliches Verständnis für das Geschäftsprozessmanagement und Workflows mit WS-BPEL geschaffen. Zudem werden noch die Excetuable Workflow Networks erläutert, sowie das Konzept der Tuple Spaces und die verwendete Implementierung vorgestellt, da diese eine bedeutende Grundlage für diese Arbeit bilden.

Eine existierende Laufzeitumgebung für BPEL-Prozesse ist Apache ODE. Da diese für die Implementierung der Verteilten Wokflow Engine als Basis dient, stellt die Analyse dieser Open Source Engine einen Teil der Aufgabenstellung dar. In Kapitel 3 wird die Architektur von ODE, sowie das darin verwendete Objektmodell zur Repräsentation von BPEL-Prozessen vorgestellt. Zudem wird die Navigation durch eine Prozessinstanz und die Bereitstellung der benötigten Variablen und Daten erläutert.

Das Kapitel 4 stellt den konzeptionellen Teil dieser Arbeit dar. Dabei wird der prinzipielle Ablauf bei der verteilten Ausführung eines BPEL-Prozesses beschrieben. Des Weiteren werden die benötigten Grundlagen für die Realisierung gelegt. Dazu werden Aktivitäten so konfiguriert, dass sie auf Basis einer Tuple Space Implementierung miteinander kommunizieren können. Dies beinhaltet des Austausch von speziellen *Nachrichten-Token*, welche ebenfalls in diesem Kapitel erläutert werden.

Ein bedeutender Teil dieser Arbeit ist auch die prototypische Implementierung einer verteilten Laufzeitumgebung auf Basis der zuvor erstellten Konzepte. Diese Umsetzung wird in Kapitel 5 erläutert. Dabei wird die Architektur der erstellten Lösung beschrieben und die verteilte Navigation beispielhaft erläutert.

# 2 Grundlagen

## 2.1 Geschäftsprozessmanagement

Unter einem *Geschäftsprozess* versteht man eine Menge von Aktivitäten, die jeweils eine oder mehrere Eingaben benötigen können. Das Ergebnis der Ausführung dieser Aufgaben besitzt für den Kunden einen Wert [HC94]. Die einzelnen Aktivitäten und Funktionen stehen zueinander in einer zeitlichen und inhaltlich logischen Reihenfolge [SJ96], wobei sie auch über mehrere organisatorische Einheiten verteilt sein können [Öst95]. Häufig ist ein Geschäftsprozess aus verschiedenen Teilprozessen aufgebaut (vgl. Abb. 2.1) bzw. lässt sich je nach geforderter Betrachtungsweise detaillieren oder vereinfachen. So besteht beispielsweise der Geschäftsprozess „Bestellabwicklung" für eine Managementsicht nur aus den Aktivitäten „Auftragsannahme", „Bearbeitung" und „Auslieferung". Für die tatsächliche Ausführung des Prozesses stellt die Aufgabe „Bearbeitung" aber einen eigenen Teilprozess dar, welcher aus den Funktionen „Kundendaten kontrollieren", „Warenbestand überprüfen" und "Rechnung erstellen" aufgebaut ist.

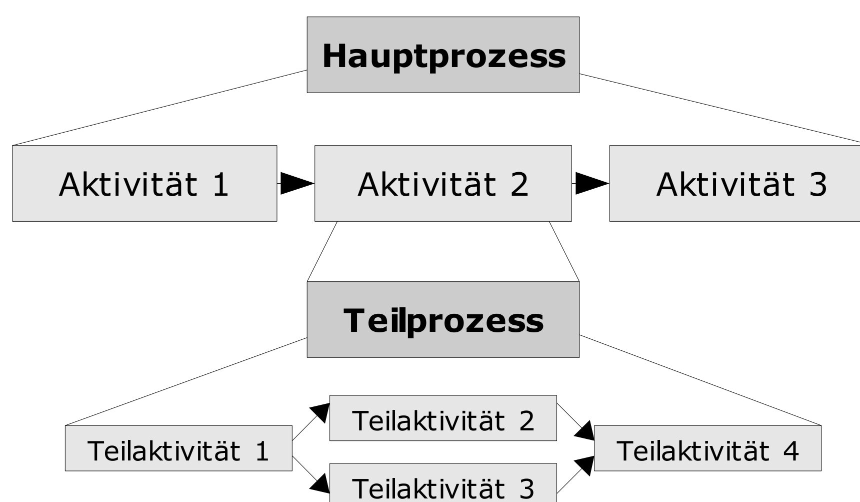



*Abbildung 2.1: Aufbau eines Geschäftsprozesses*

Da die Geschäftsprozesse den zeitlichen Aufwand und die Kosten der betrieblichen Leistungserstellung maßgeblich beeinflussen, besitzen sie einen wesentlichen Anteil am Erfolg eines Unternehmens. Deshalb wird das so genannte *Geschäftsprozessmanagement* heutzutage von den meisten Unternehmen als eine notwendige und selbstverständliche Aufgabe betrachtet [Gad05]. Darunter fallen unter anderem Aufgaben wie das Erkennen der im Unternehmen vorhandenen Prozesse, die Analyse dieser Abläufe und deren ständige Optimierung.

Grundsätzlich lassen sich dabei zwei Ebenen unterscheiden: die konzeptionelle Ebene und die operative Ebene des Geschäftsprozessmanagements [Gad05]. Auf der fachlichen, konzeptionellen Ebene geht es um das Identifizieren von Prozessen, die für das Unternehmen von Bedeutung sind, sowie deren Modellierung. Diese dient der Dokumentation der regelmäßigen Arbeitsabläufe und verfolgt das Ziel der Standardisierung dieser Abläufe innerhalb des Unternehmens Für die Modellierung der Prozesse auf dieser Ebene gibt es viele verschiedene, meist grafische, Möglichkeiten. Bekannte Beispiel für solche Modellierungsmethoden sind die *Business Process Modelling Notation (BPMN)* [OMG09] oder die Methode der *Ereignisgesteuerten Prozessketten (EPK)* [KMS92]. Ein weiterer Aspekt der fachlichen Ebene ist die Kontrolle der eingeführten Prozesse. Darunter versteht man zum Beispiel die Überprüfung, ob ein veränderter, vermeintlich verbesserter, Prozessablauf tatsächlich die erwarteten Gewinne hinsichtlich bestimmter Messgrößen erzielt hat. Auf der operativen Ebene steht die Ausführung der Prozesse im Mittelpunkt und deren Überwachung (*Monitoring*). Diese liefert Daten über den Prozess, wie beispielsweise die reale durchschnittliche Zeitdauer eines Ablaufes oder die Wartezeiten zwischen der Ausführung einzelner Aktivitäten. Im Zusammenhang mit der computergestützten, automatisierten Ausführung von Geschäftsprozessen wird die operative Ebene häufig auch als *Workflow Management* bezeichnet.

### 2.1.1 Workflows und Workflow Management Systeme

Geschäftsprozesse werden bei der Ausführung häufig von Softwareanwendungen unterstützt, wozu sie in den entsprechenden Systemen abgebildet sein müssen (vgl. [Öst95]). Eine Möglichkeit dazu bieten komplexe Anwendungen, welche speziell zur Umsetzung eines bestimmten Prozesses entwickelt werden. Deren Nachteil ist unter anderen aber die fehlende Flexibilität: Bei einem geänderten Prozessablauf muss die Software angepasst und entsprechend geändert werden, was häufig mit erheblichem Aufwand verbunden ist. Eine flexiblere Alternative dazu sind die so genannten *Workflow Management Systeme (WfMS)*. Diese realisieren die Steuerung durch das Modell eines Workflows.

Unter einem *Workflow* versteht man dabei einen ganz oder teilweise automatisierten Arbeitsablauf. Ein *Workflow-Modell* ist die Beschreibung eines Geschäftsprozesses (Prozessmodell), welche so weit technisch verfeinert wurde, dass sie von einem WfMS zur Steuerung der automatisierten Ausführung verwendet werden kann. Der Workflow selbst ist die Instanz eines Workflow-Modells und kann somit als ein Geschäftsprozess, der sich in Ausführung befindet, aufgefasst werden. In Abbildung 2.2 wird noch einmal die Unterscheidung zwischen einen (Geschäfts-)Prozess und einem Workflow verdeutlicht. In der Praxis wird diese Trennung allerdings nicht so streng umgesetzt und beide Begriffe teilweise synonym verwendet.

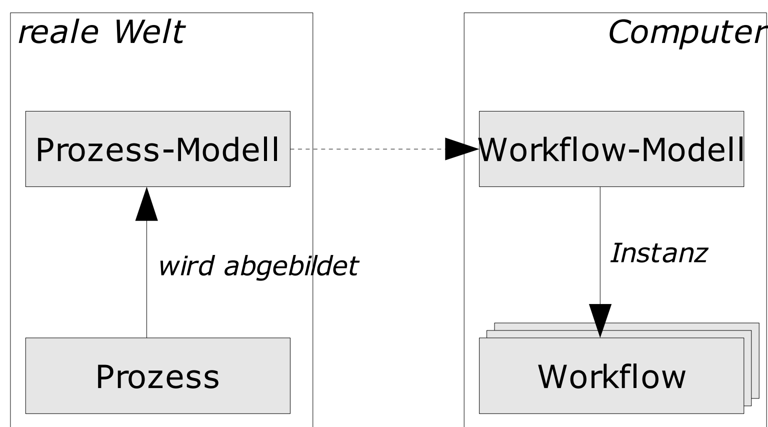



*Abbildung 2.2: Prozesse und Workflows*

Ein Workflow besteht aus drei Dimensionen. Die *Was?*-Dimension gibt an, welche Aktivitäten, wann gemacht werden müssen. Sie spiegelt also den Kontrollfluss des Prozesses wieder. Die *Wer?*-Dimension definiert, wer eine bestimmte Tätigkeit ausführen soll. Damit können einzelne Personen oder Organisationseinheiten gemeint sein, sowie bestimmte Maschinen oder Rechner, welche die Aufgabe automatisch erledigen. Mit der *Womit?*-Dimension wird angegeben, welche Werkzeuge oder Anwendungen dazu benötigt werden.

Ein Workflow Management System ist nun eine Softwareanwendung, die Workflows definiert, verwaltet und ausführt. Sie unterstützt also sowohl die Erzeugung eines Workflow-Modells (*Build Time*) als auch dessen Ausführung (*Run Time*) [LR00]. Ein WfMS kann Workflow-Modelle, die in einer formalisierten Form (z.B. WS-BPEL) vorliegen, interpretieren (ausführen) und mit den Prozessbeteiligten (Wer?-Dimension) interagieren. Darunter fällt auch die Möglichkeit, andere Anwendungen, welche eine Aktivität des Workflows unterstützen, mit den erforderlichen Eingabedaten aufzurufen [WMC05]. In Abbildung ist das vereinfachte Referenzmodell eines WfMS der *Workflow Management Coalition (WfMC)* mit den wichtigsten Funktionen und deren Zusammenspiel dargestellt. Eine zentrale Komponente ist dabei der *Workflow Enact-*

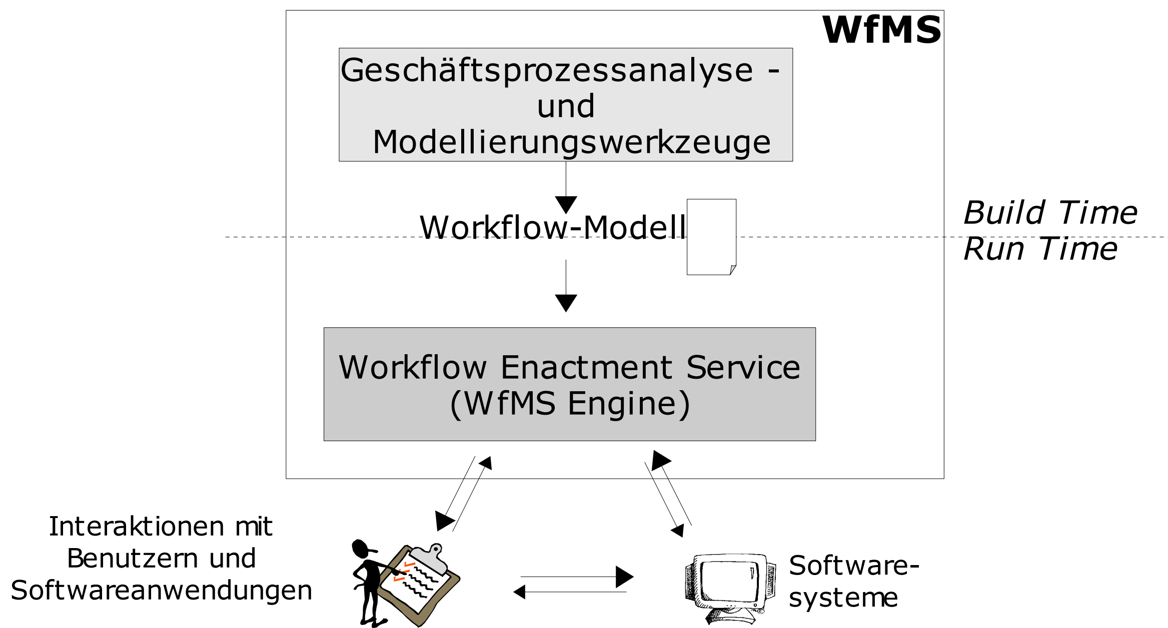



*Abbildung 2.3: Hauptbestandteile eines WfMS*

*ment Service*. Dieser steuert durch den Kontrollfluss des Workflow-Modells und leistet die Interaktion mit den Prozessteilnehmern. Zusätzlich zu den dargestellten Bestandteilen liefert ein WfMS auch noch Funktionen für die Verwaltung und Überwachung der Workflows

## 2.2 Serviceorientierte Architekturen

Serviceorientierte Architekturen, kurz SOA, sind ein Konzept, wie Softwareanwendungen durch das Zusammenspiel verschiedener Dienste aufgebaut werden können. Ein Dienst ist dabei eine Softwarekomponente, welche eine abgeschlossene Geschäftsfunktion kapselt [WCL+05]. Ein wichtiges Merkmal serviceorientierter Architekturen ist das Prinzip der losen Kopplung (loose coupling) [DJM+05]. Damit ist gemeint, dass die Partner, also beispielsweise Dienstanbieter und Dienstverwender, so wenig Annahmen übereinander machen müssen wie nötig [HW03]. Dies bedeutet, dass beispielsweise die konkrete Implementierung oder Programmiersprache einer Komponente für einen Verwender des Services unbedeutend sind. Ein Dienst wird nach außen hin nur durch eine Schnittstelle definiert, welche die Nutzung des Dienstes beschreibt.

Zur Verwendung eines Service innerhalb einer anderen Anwendung könnte der Dienst statisch darin eingebunden werden. Dies widerspricht aber dem Prinzip der losen Kopplung, da dann die Adresse eines bestimmten Dienstes zur Entwicklungszeit der Anwendung bekannt sein muss. Deshalb gibt es auch die Möglichkeit des dy-

namischen Bindens eines Services. Dazu muss ein Dienstbenutzer aber in der Lage sein, Dienste nach bestimmten Kriterien zu suchen und zu finden. Um dies zu gewährleisten, bieten sich Verzeichnisdienste an. Diese ermöglichen es, dass Dienste dort registriert werden und stellen eine Suchmöglichkeit zur Verfügung.

Zusammengefasst lässt sich also sagen, dass an einer SOA sind verschieden Rollen beteiligt sind: ein Dienstanbieter, ein Dienstverwender und ein Dienstverzeichnis. In Abbildung 2.4 ist das Zusammenspiel der Beteiligten noch einmal mit dem bekannten „Dreieck der Serviceorientierten Architekturen“ veranschaulicht.

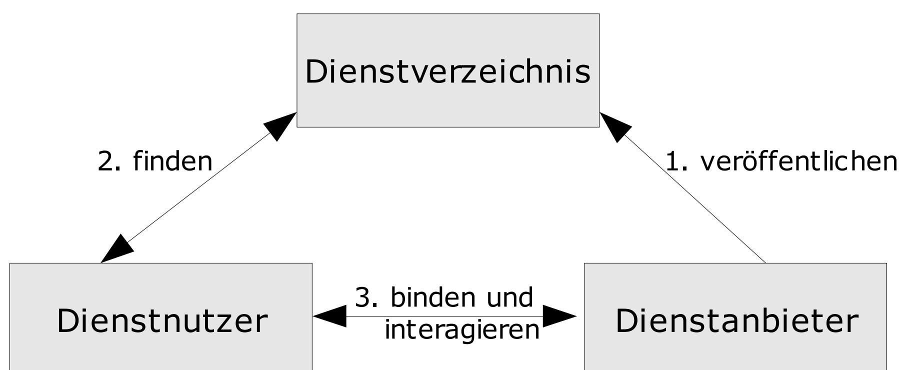



*Abbildung 2.4: SOA-Dreieck*

Diese Eigenschaften einer SOA ermöglichen eine prozessorientierte Gestaltung von Softwareanwendungen [DJM+05]. Ein Unternehmen kann zunächst seine Abläufe modellieren und dann entsprechende Dienste für einzelne Aktivitäten verwenden. Dies entspricht auch dem *„Two-Level-Programming“ Paradigma* [Ley03], welches hinter den heutigen Workflow Beschreibungssprachen steht [MWL08a]. Dabei werden Anwendungen durch eine Trennung von Kontrollfluss und Geschäftsfunktionen entwickelt. Den Aufgaben, welche innerhalb des Kontrollprogrammes vorkommen, werden entsprechende Dienste zugewiesen (vgl. Abb. 2.5). Diesem Prinzip folgen auch die zuvor beschrieben Workflow Systeme (Abschnitt 2.1).

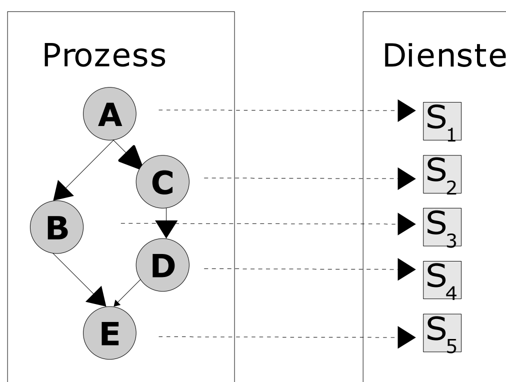



*Abbildung 2.5: Two-Level-Programming-Paradigma*

Eine technische Realisierung der serviceorientierten Architekturen und eine Workflowsprache zur Orchestrierung von Diensten folgen in den nächsten zwei Abschnitten

## 2.3 Web Services

*Web Services* sind eine Möglichkeit, um „die Interoperabilität zwischen verschiedenen Softwareanwendungen, die auf einer Vielzahl von Plattformen und/oder Frameworks laufen“ [W3C04a] zu realisieren. Das *World Wide Web Consortium (W3C)* definiert einen Web Service, als ein Softwaresystem, welches über einen *Uniform Resource Identifier (URI)* [BFM05] identifiziert werden kann und dessen Schnittstelle mittels XML beschrieben ist. Diese Beschreibung kann von anderen Softwareanwendungen gefunden werden, welche mit einem Web Service durch den Austausch von XML-Nachrichten über Internetprotokolle interagieren können [W3C04b].

Mit Hilfe der Web Service Technologien kann eine beliebige Funktion als eine *„virtuelle Komponente“* [Ley03] hinter einer Schnittstelle gekapselt werden. Für den Verwender dieser Komponente ist es nicht von Bedeutung und auch nicht zu erkennen, wie und in welcher Programmiersprache diese Funktion tatsächlich implementiert ist (vgl. Abb. 2.6). Diese Komponenten können nun zu komplexeren Anwendungen integriert werden. Damit liefern sie eine technische Umsetzung des Konzeptes der serviceorientierten Architekturen, wobei sie den zentralen Aspekt eines Dienstes einnehmen.

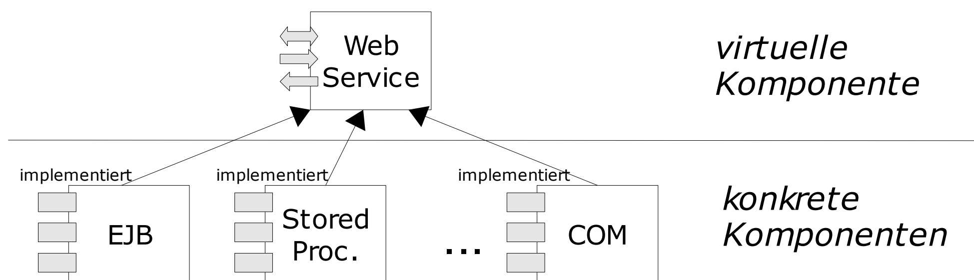



*Abbildung 2.6: Web Service als virtuelle Komponente (nach [Ley03])*

Die Web Service Architektur [W3C04a] sieht eine Reihe an verschiedenen Technologien und Spezifikationen vor, welche unterschiedliche Anforderungen der Web Services abdecken [DJM+05]. Diese werden häufig in einem Schichtenmodell, dem sog. Web Service Stack, welcher in Abbildung 2.7 vereinfacht dargestellt ist, veranschaulicht.

Auf der untersten Ebene befindet sich die Transportschicht. Web Services sind dabei an keine bestimmtes Kommunikationsprotokoll gebunden, sondern können mit ver-

schiedenen Protokollen verwendet werden, über welche Nachrichten ausgetauscht werden. Auf der darüberliegenden Schicht werden die Nachrichten genauer spezifiziert. Diese sind im XML-Format, welches über das *SOAP* Nachrichtenprotokoll ausgetauscht werden. Die Beschreibungsschicht liefert mit der Sprache *WSDL* eine Möglichkeit um die Schnittstellen eines Web Services, also beispielsweise welche Operationen angeboten werden und welche Nachrichten ausgetauscht werden müssen. Die oberste Schicht befasst sich mit dem Einsatz von Web Services im Sinne des „Two-Level-Programming“ Paradigmas (vgl. Abschnitt 2.2 und Abbildung 2.5). Dies bedeutet die koordinierte Ausführung von Diensten um eine komplexeren Prozess zu realisieren. Die wichtigste Technologie ist dabei WS-BPEL, welche in Abschnitt 2.4 noch ausführlich behandelt wird. Zusätzlich zu den in Abbildung angegebenen Technologien gibt es noch zahlreiche weiter so genannte *WS**-Technologien, die hier nicht ausführlich betrachtet werden.

*Abbildung 2.7: Web Service Stack (nach [W3C04a])*

Wie zuvor beschrieben sind das Anbieten, Suchen und Finden von Diensten zentrale Aspekte einer SOA [DJM+05]. Dazu werden die zuvor erwähnten Technologien verwendet. Nicht zu den eigentlichen Web Service Spezifikationen gehört der Verzeichnisdienst *UDDI*. Der Grund dafür ist, dass Web Services auch ohne entsprechendes Verzeichnis verfügbar gemacht und verwendet werden können. Im Zusammenhang mit dem Suchen und dynamischen Binden eines Dienstes ist ein Verzeichnis aber unerläßlich. Die Abbildung 2.8 zeigt nochmals das SOA-Dreieck, erweitert um die entsprechenden Technologien. Ein Dienstanbieter veröffentlicht zunächst seinen Web Service bei einem Verzeichnis. Ein Verwender sucht darin nach einem Dienst und erhält einen Verweis auf den Anbieter. Dort fragt die Schnittstellenbeschreibung (WSDL) ab und interagiert mit dem Dienst. Die Kommunikationsschritte erfolgen jeweils durch den Austausch von SOAP-Nachrichten.

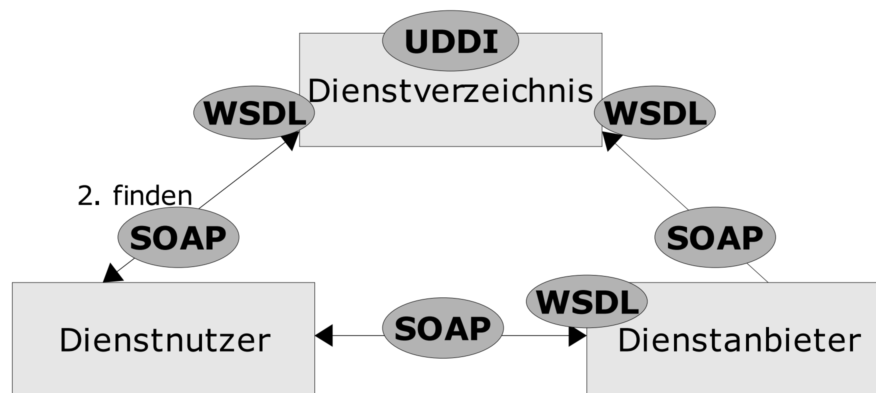



*Abbildung 2.8: Web Service Dreieck*

## 2.4 WS-BPEL

Im vorherigen Abschnitt wurden die grundlegenden Technologien im Zusammenhang mit Web Services kurz vorgestellt. Der Web Service Stack (Abb. 2.8) beschäftigt sich auf der obersten Ebene mit der Integration von Web Services. Im Sinne des „Two-Level-Programming“-Paradigmas [Ley03] (vgl. Abb. 2.5) ist dies die Ebene der Prozessbeschreibung. Die Ebene der einzelnen Komponenten ist dabei durch die zuvor beschriebenen Web Services umgesetzt. Die Integration dieser Dienste bedeutet nun die koordinierte Ausführung von verschiedenen Web Services um bestimmte Abläufe zu unterstützten – häufig auch als *Orchestrierung* bezeichnet. Dazu hat sich in der Praxis die XML-basierte *Web Service Business Process Execution Language (WS-BPEL)*, kurz BPEL, durchgesetzt.

Mit BPEL können zwei Aspekte der Prozessmodellierung abgedeckt werden. Zum einen gibt es die so genannten *abstrakten* BPEL-Prozesse. Diese können beispielsweise genutzt werden, um den Nachrichtenaustausch mit beteiligten Partnern (=Web Services) zu beschreiben, ohne dabei die internen Verarbeitungsschritte innerhalb des Prozesses öffentlich zu machen [OAS07]. Die zweite Form sind die *ausführbaren* BPEL-Prozesse, welche als Eingabe für ein WfMS bzw. dessen Workflow Engine (vgl. Abb. 2.3) dienen. Vereinfacht gesagt, beschreibt ein abstrakter Prozess das Verhalten nach außen, während ein ausführbaren Prozess auch die interne Realisierung des Prozess angibt [ACK+04].

Der logische Ablauf eines Workflows wird in einem BPEL-Prozess durch Aktivitäten *(activities)* definiert. Diese lassen sich in elementare Aktivitäten *(basic activities)* und zusammengesetzte Aktivitäten *(structured activities)* unterscheiden. Die elementaren Aktivitäten spiegeln einzelne „Funktionen“ wieder. Dies sind beispielsweise der Aufruf eines Web Services (`invoke`), das Empfangen bzw. Senden einer Nachricht von

oder an einen Partner (`receive` / `reply`) oder das Signalisieren eines Fehlers (`throw`). Die zusammengesetzten Aktivitäten beinhalten andere Aktivitäten (sowohl strukturierte als auch einfache) und legen damit einen Kontrollfluss und Reihenfolge der Aktivitäten festgelegt. Dazu gehören zum Beispiel die Aktivitäten `sequence` (sequentielle Ausführung der beinhalteten Aktivitäten) oder `flow` (parallele Ausführung).

Der aktuelle Zustand eines BPEL-Prozesses, dazu gehören zum Beispiel Daten aus empfangenen Nachrichten, wird in Variablen gespeichert. Diese können durch durch die Aktivität `assign` manipuliert werden oder als Parameter für die Interaktion mit den Partnern genutzt werden.

Bestimmte Teile eines BPEL-Prozesses lassen sich in einem Block (`scope`) zusammenfassen. Diese Blöcke können geschachtelt sein, d. h., ein Block selbst kann verschiedene Blöcke beinhalten. Dieses Konzept dient im Wesentlichen der Fehlerbehandlung. Dazu werden für einen Block bestimmte Fehlerbehandlungsmaßnahmen (FaultHandler oder CompensationHandler) definiert. Tritt bei einer Aktivität innerhalb eines Blockes ein Fehler auf, so wird der Fault Handler dieses Blockes gestartet und auf der Fehler entsprechend verarbeitet.)

Ein BPEL-Prozess wird nach außen durch eine WSDL-Definition als Web Service beschrieben. Dies ermöglicht das rekursive Einbinden von BPEL-Prozessen innerhalb anderer BPEL-Prozesse.

Die Kommunikation mit Web Services erfolgt über das Konzept der Partner Links. Ein *Partner Link* ist dabei eine Verbindung zwischen dem Prozess und einem Web Service, welche einen bestimmten *Partner Link Type* besitzt. Dieser spezifiziert den Nachrichtenaustausch zwischen zwei Web Services, indem den beteiligten Diensten Rollen zugeordnet werden. Eine Rolle referenziert dabei jeweils einen Port Type in der WSDL-Definition des Web Services. Kommuniziert ein Prozess nun mit einem Web Service, so wird immer der entsprechende Partner Link und die zugehörigen Rollen angegeben. Somit wird eindeutig festgelegt welche PortTypes bei der Interaktion verwendet werden.

## 2.5 Tuple Spaces

Das Konzept der *Tuple Spaces* wurde im Rahmen der *Linda Coordination Language* [Gel85], welche ein Modell zur Koordination und Kommunikation bei verteilten Systemen darstellt, entwickelt. Ein Tuple Space ist hierbei zunächst nur eine Art verteilter Speicher für geordnete Listen (Tupel). Die Interaktion zwischen einem Benutzer und einem Tuple Space erfolgt durch eine sehr einfache Schnittstelle, welche die drei Ope-

rationen `out`, `in` und `read` zur Verfügung stellt. Über diese drei Kommunikationsprimitive werden Tuple in dem Tuple Space gespeichert (`out`) oder gelesen (`in`, `read`). Für das Lesen eines Tupels gibt es zwei Varianten: mit der Operation `read` wird „nur" gelesen, wo hingegen mit `in` das Tupel aus dem Tuple Space genommen wird (destruktives Lesen). Das Lesen bzw. Entnehmen eines bestimmten Tupels aus dem Space erfordert, dass es auch identifiziert werden kann. Dies erfolgt über ein *Template Matching* Verfahren. Ein *Template* ist dabei ein spezielles Tupel, welches für einige Felder Werte spezifiziert und andere Felder offen lässt bzw. einen Platzhalter dafür verwendet. Ein Lesen gibt zu einem solchen Template ein Tupel zurück, welches die Werte der spezifizierten Felder des Templates besitzt. Dies entspricht dem Prinzip des *Query-by-Example* [Zlo75]. Sollten mehrere Tupel in dem Tuple Space die Anfrage des Templates erfüllen, so ist es nichtdeterministisch, welches Tupel gelesen wird.

Die Kommunikation zweier Komponenten über ein Tuple Space durch das Schreiben und Lesen eines Tupels ist in Abbildung dargestellt (*one-to-one*). Weitere Kommunikationsmuster, wie *one-to-many* oder *many-to-many* lassen sich über Linda-Primitive realisieren [Pop08].

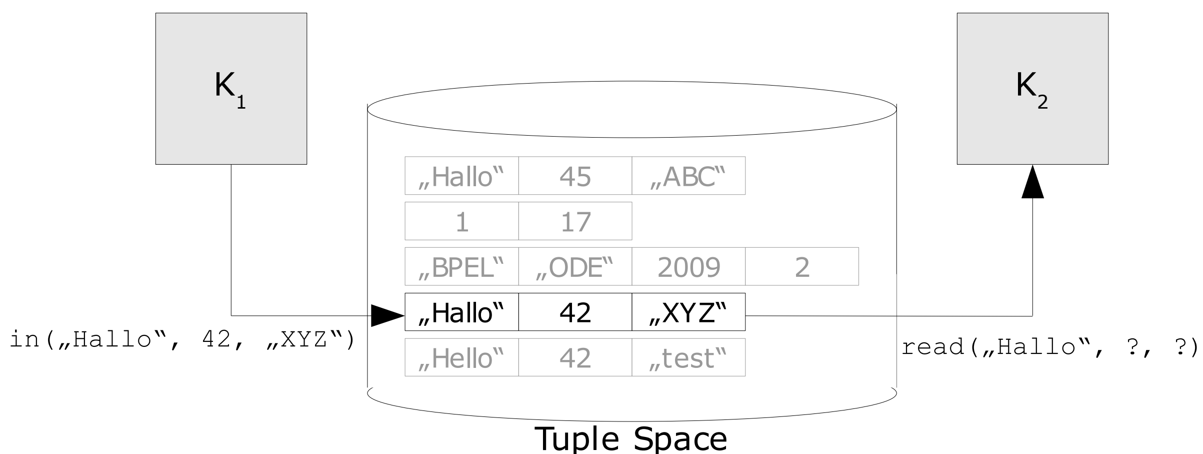



*Abbildung 2.9: Kommunikation über Tuple Spaces (nach [Pop08])*

Die Verwendung von Tuple Spaces als „Kommunikationskanal" ist im Sinne einer losen Kopplung von Anwendungen ähnlich zu messageorientierter Middleware [MWL08b]. Sie ermöglichen die Kommunikation zwischen mehreren Partnern, wobei diese wenig Annahmen übereinander machen müssen. Das Prinzip der losen Kopplung zeichnet sich durch verschiedene Autonomie-Ebenen aus [HW03]. Bei dem Informationsaustausch über Tuple Spaces sind die beteiligten Kommunikationspartner nicht aneinander gekoppelt hinsichtlich Zeit, Ort und Referenz [KRJ05]. Den Aspekt der *„Referenz Autonomie"* erfüllen sie dadurch, dass ein Sender nicht weiß, wer die Nachricht erhält. Für ihn genügt es, die Nachricht auf den Tuple Space zu stellen, wer sie letztendlich liest und verarbeitet ist für den Erzeuger der Nachricht unbedeutend.

Dasselbe gilt für die *„zeitliche Unabhängigkeit". D*er Senderprozess muss beispielsweise gar nicht mehr existieren, wenn ein Empfänger die Nachricht verarbeitet. Auch eine *Plattform Autonomie* wird gewährleistet, da die Kommunikationspartner nur über ein simples Interface auf den Tuple Space zugreifen. Dabei ist es egal in welcher Sprache die einzelnen Partner geschrieben sind. Einen Vergleich zwischen dem Konzept der Spaces und Messaging Middleware findet sich in [FKL+07].

Mittlerweile sind einige Implementierungen des Linda bzw. Tuple Space Konzeptes erhätlich, wie z. B. TSpaces [TSpace] von IBM oder Sun's JavaSpaces. Für diese Arbeit wird eine Tuple Space Middleware verwendet, welche im Rahmen einer Vorgängerarbeit entwickelt [Wu08] wurde. Das dabei entwickelte Tuple Space System leistet neben den Grundoperationen zum Schreiben (`out`) und Lesen (`in`, `read`) noch weitere Funktionalitäten zum Zugriff auf einen Tuple Space. Diese betreffen besonders die Synchronisation und Transaktionsunterstützung. So kann es zum Beispiel vorkommen, dass für einen Client bestimmte Operationen zu einer Transaktion zusammengehören. Damit der Client die Transaktions-Kontrolle nicht selbst realisieren muss, bietet die Tuple Space Implementierung eine entsprechende Unterstützung schon an. Ebenso wird von der Tuple Space Implementierung schon der gleichzeitige Zugriff auf bestimmte Tupel innerhalb verschiedener Transaktionen synchronisiert.

Weitere zusätzliche Funktionen, welche die verwendete Tuple Space Implementierung besitzt, sind beispielsweise die Operationen `update` oder `scan`. Die `update`-Operation ermöglicht das Verändern von Tupeln direkt auf dem Tuple Space. Sie fasst also das Lesen eines Tupel, dessen Veränderung durch einen Client und das erneute Einbringen auf den Tuple Space in einer Operation um. Mit `scan` werden wird ein Template übergeben und die Funktion gibt zurück, wieviele Tupel sich in dem Tuple Space befinden, die dieses erfüllen.

## 2.6 Executable Workflow Networks (EWFN)

Petrinetze [Pet62] sind ein Modell zur Beschreibung von nebenläufigen Abläufen. Sie stellen einen gerichteten Graphen dar, der aus zwei verschiedenen Arten von Knoten besteht: Stellen (auch Plätze genannt) und Transitionen. Eine Stelle ist dabei eine Art der Speichermöglichkeit für Daten (Token), eine Transition beschreibt einen Verarbeitungsschritt. Dieser kann ausgeführt werden, wenn sich an den Eingangsstellen ein bestimmte Anzahl von Token befindet.

Ein wichtiges Merkmal der Petrinetze ist es, dass keine zentrale Stelle zur Steuerung des Kontrollflusses vorhanden ist. Jede Aktion wird lokal aktiviert und ausgeführt. Die Kommunikation zwischen den einzelnen Aktivitäten erfolgt dabei asynchron.

Diese Eigenschaften sollen nun für die verteilte Ausführung von BPEL-Wokflows genutzt werden und damit die zuvor beschriebenen Nachteile einer zentralen Navigationskomponente aufheben. Die Steuerung durch einen BPEL-Prozess soll dabei durch den Austausch von Token zwischen den einzelnen Aktivitäten des Prozesses erfolgen.

Um dies zu ermöglichen, muss ein BPEL-Geschäftsprozess zunächst mal in ein Petrinetz übersetzt werden. Eine Art der Überführung wird in [Sta05] beschrieben. Allerdings war der Zweck dieser Arbeit, die Verifikation von BPEL-Prozessen und nicht die Ausführung. Deshalb wird in [MWL08a] das Modell der *Executable Workflow Networks* (EWFN) definiert. Das Ziel war dabei eine Darstellung für BPEL-Prozesse, welche mit einem (erweiterten) Tuple Space System ausgeführt werden können. Die Grundlage für EWFN bilden gefärbte Petrinetze [Jen92] und Boolesche Netzwerke [LSW97].

Ein EWFN ist ein gerichteter, bipartiter Graph, welcher durch das Tupel (Σ, P, T, F, X, X, $L_w$) definiert wird.

- Σ ist die Menge aller Token (Tupel). Man unterscheidet dabei zwei Kategorien: Kontrollfluss- und Datentoken. Zusätzlich gibt es ein empty Tuple ε, welches aussagt, dass kein Token erzeugt wurde.
- P ist eine endliche Menge an Plätzen (Stellen).
- T ist eine endliche Menge an Übergängen (Transitionen).
- $F \subseteq (T \times P) \cup (P \times T \times R)$, mit R = {*read, take*} ist die Menge der Kanten. Eine Kante verbindet dabei niemals zwei Plätze oder Stellen miteinander. Der jeweilige Kantentyp entspricht den Linda-Operationen. So beschreibt eine Kante von einer Transition zu einem Platz (also aus $(T \times P)$ eine write-Operation. Kanten von einem Platz zu einer Transition beschreiben eine Lese-Operation. Dabei wird mit dem Wert für R angegeben, ob es sich um ein destruktives (take) oder erhaltendes (read) Lesen handelt.
- X ist die Menge aller Templates. Dies sind spezielle Tupel, die für die einzelnen Felder entweder konkrete Werte oder ein Platzhaltersymbol (*) besitzen.
- $A: (P \times T \times R) \rightarrow X$ ist eine Abbildung, welche eingehenden Kanten einer Transition Templates zuordnet, so dass gilt: $\forall (p,t,r) \in F \cap (P \times T \times R) : A(p,t,r) \in X$.
- $L_w$: $(T \times P) \rightarrow \Sigma$ beschreibt die Linda write-Funktion.

Die Funktionsweise der EWFN ist ähnlich wie die der Petrinetze. Eine Transition ist aktiviert, wenn sich an allen eingehenden Stellen bestimmte Token befinden. Das Lesen dieser Token erfolgt über ein Template-Matching, welches durch eine der Linda read-Operationen realisiert wird. Wenn eine aktivierte Transition feuert, kann sie auf den ausgehenden Plätzen neu Token erzeugen. Für eine ausführliche Definition der EWFN sei an dieser Stelle auf [MWL08a] verwiesen.

Für die Transformation eines BPEL-Prozesses in eine äquivalentes Executable Workflow Network wird in [Pop08] ein Verfahren beschrieben. Darin werden Pattern vorgestellt, wie bestimmte BPEL-Aktivitäten durch ein EWFN dargestellt werden können. Diese zeichnen sich durch eine Schnittstellenstruktur aus, wodurch die Aktivitäten miteinander verbunden werden können. So bietet beispielsweise jedes Pattern für eine BPEL-Aktivitäten einen start- und einen ended-Platz. Auf den ended-Platz wird nun von einer Transition, die zu einer Aktivität gehört, ein Kontrollflusstoken erzeugt. Dieser ended-Platz dient nun für eine nächste Aktivität wieder als Eingangsstelle, also als start-Platz.

# 3 Workflows mit Apache ODE

Apache ODE [ODE] („*Orchestration Director Engine*“) ist eine Open Source Laufzeitumgebung (*BPEL Workflow Engine*) für BPEL-Geschäftsprozesse. Die aktuellste verfügbare Version ist Apache ODE 1.2, welche im Rahmen dieser Arbeit verwendet wird. Sie ermöglicht die Ausführung von BPEL-Prozessen und leistet die Interaktion mit Web Services. Dabei werden sowohl Prozesse der Version WS-BPEL 2.0 [OAS07] als auch der Vorgängerversion BPEL4WS 1.1 [ACD+03] unterstützt.

*Abbildung 3.1: Apache ODE*

Apache ODE ist in zwei Varianten verfügbar: als *Web Application Archive* (.war-Datei) und als *Java Business Integration (JBI) Service Assembly*. Zur Ausführung wird ein entsprechender Servlet-Container bzw. ein JBI-Container benötigt, in dem ODE deployed wird. Die beiden Varianten unterscheiden sich darin, wie die Kommunikation mit externen Web Services realisiert wird. In der Form der .war-Datei wird *Apache Axis2* [AXIS2] als Kommunikationsumgebung und SOAP-Engine verwendet; bei der JBI-Variante die Java Business Integration [JBI]. Im Rahmen dieser Arbeit wird ODE stets in der Web Application Archive Variante genutzt und als Servlet-Container *Apache Tomcat* [TOMCAT] eingesetzt.

Ein Teil der Aufgabenstellung dieser Arbeit ist die Analyse von Apache ODE bezüglich der Realisierung der Navigation und des Datenaustausches zwischen Aktivitäten. Zudem dient die BPEL Engine als Grundlage für die Implementierung der verteilten Workflow Laufzeitumgebung, weshalb Apache ODE in diesem Kapitel ausführlich beschrieben wird. Dabei wird zunächst die Architektur der Anwendung dargestellt und danach bestimmte Komponenten erläutert, wobei der Schwerpunkt der Betrachtung auf der Navigation und dem Datenaustausch liegt.

## 3.1 Architektur

In Abbildung 3.2 ist die Architektur von Apache ODE durch die Hauptkomponenten und deren Zusammenspiel dargestellt. Die wichtigsten Bestandteile sind dabei der *ODE BPEL Compiler*, die *ODE BPEL Runtime Engine*, die *ODE Data Access Objects* (DAOs) und der *ODE Integration Layer*.

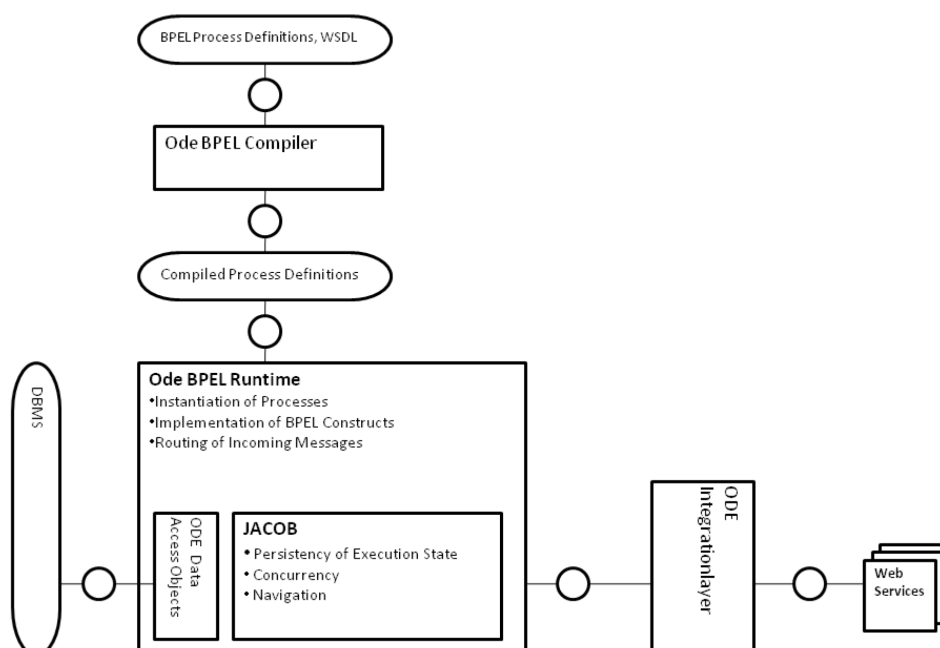



*Abbildung 3.2: Architektur von Apache ODE (übernommen von [ODEb])*

Vereinfacht gesagt, wird ein BPEL-Prozess zunächst mittels des Compilers in eine interne Repräsentation umgeformt. Die Laufzeitumgebung (*BPEL Runtime Engine*) nutzt diese Darstellung des Prozesses bei der Ausführung und speichert die benötigten Instanzdaten in einem persistenten Speicher, auf den über die DAOs zugegriffen wird. Die Kommunikation zwischen Apache ODE und im BPEL-Workflow beteiligten Web Services erfolgt über den Integration Layer [ODEb]. Die einzelnen Komponenten werden nachfolgend kurz vorgestellt, dabei wobei besonders auf den Compiler bzw. das von ihm erzeugte Objektmodell sowie die ODE Runtime eingegangen wird.

**ODE Integration Layer**

Eine BPEL-Workflow Engine benötigt eine Schnittstelle nach außen, über die mit Web Services kommuniziert werden kann. Die Kommunikation mit den Partnern erfolgt

dabei über den Austausch von SOAP-Nachrichten (vgl. Abschnitt 2.3). Diese Aufgabe übernimmt in Apache ODE der Integration Layer. Er stellt dazu Kommunikationskanäle für die Engine zur Verfügung [ODEb] und verbirgt deren Implementierungsdetails [SUP07]. Als tatsächlichen Mechanismus für den Nachrichtenaustausch kann Apache Axis2 oder JBI verwendet werden.

**ODE Data Access Objects**

Um die benötigten Daten persistent zu speichern, verwendet Apache ODE einen Datenspeicher, auf welchen über so genannte *Data Access Objects* (DAOs) zugegriffen wird. Typischerweise ist dieser Speicherplatz (*ODE Data Store*) eine relationale Datenbank [ODEb]. Die DAOs bilden nun eine Schnittstelle zu diesem Speicher, welche dessen konkreten Typ verbergen. Da standardmäßig eine relationale Datenbank verwendet wird, bietet ODE schon eine JDBC-Implementierung der DAO-Schnittstelle an. Es ist grundsätzlich auch möglich den Data Store auszutauschen und andere Mechanismen zur persistenten Sicherung zu verwenden, wie beispielsweise XML-Datenbanken oder Dateisysteme. In diesem Fall muss die DAO-Schnittstelle aber selbst implementiert werden.

Die Daten, welche persistent gehalten werden müssen, betreffen die Ausführung von Prozessinstanzen und das Management der Prozesse. Zur Ausführung gehört dabei der Zustand einer Prozessinstanz durch Kontrollfluss und benötigten Instanzdaten, wie Variablen oder Correlation Sets. Diese werden im Abschnitt 3.3 ausführlich dargestellt. Unter den Management-Daten versteht man Informationen, die das Monitoring der Prozesse erlauben. Dies sind zum Beispiel Informationen, die angeben, wann eine Prozessinstanz welches Prozessmodells gestartet wurde und wann sie wieder beendet wurde. Auf diese Daten und deren Verwendung wird hierbei nicht weiter eingegangen.

## 3.2 BPEL-Compiler und ODE-Objektmodell

Um in Apache ODE ein Prozessmodell verfügbar zu machen, muss es sich zusammen mit allen zugehörigen Bestandteilen in einem Verzeichnis (*Deployment Unit*) befinden. Eine solche „Deployment-Einheit“ besteht beispielsweise aus BPEL-Prozessdefinition, benötigten XML Schema Dokumenten und WSDL-Beschreibungen des Prozesses und seiner Partner. Das Deployment eines Prozessmodells erfolgt durch Kopieren dieser „Deployment-Einheit“ in ein bestimmtes Verzeichnis von ODE („Deployment-Verzeichnis“).

Beim Starten von Apache ODE wird ein Objekt der Klasse `DeploymentPoller` erzeugt, welches regelmäßig im Hintergrund das „Deployment-Verzeichnis“ auf neue „Deployment-Einheiten“ überprüft. Sollte eine neue Einheit gefunden werden, wird der BPEL-Prozess validiert und in eine Baumdarstellung, dem sog. *BPEL Object Model (BOM)* überführt. Dies ist ähnlich zum Parsen eines XML-Dokumentes und dessen Repräsentation als *Document Object Model (DOM)*. Der *ODE BPEL Compiler* übernimmt nun diese BOM-Darstellung und erzeugt daraus das *ODE Objektmodell*. Dieses wird nach der Kompilierung als `.cbp`-Datei abgespeichert und kann bei einem Neustart von ODE daraus wieder erzeugt werden.

Das ODE-Objektmodell ist die einzige Darstellung des Prozesses, welches für die Ausführung genutzt wird [ODEb], d.h., die ursprüngliche BPEL-Prozessdefinition wird nach erfolgreicher Kompilierung nicht mehr benötigt. Im ODE-Objektmodell wird jedes kompilierte BPEL Element durch einem Objekt repräsentiert, welches die benötigten Daten aus der Prozessdefinition beinhaltet. Diese Objekte sind Instanzen von Klassen, deren Name jeweils mit einem `O` beginnen und die von der Klasse `OBase` erben. In Abbildung 3.3 ist ein Ausschnitt aus dem Klassendiagramm des ODE-Objektmodells angeben.

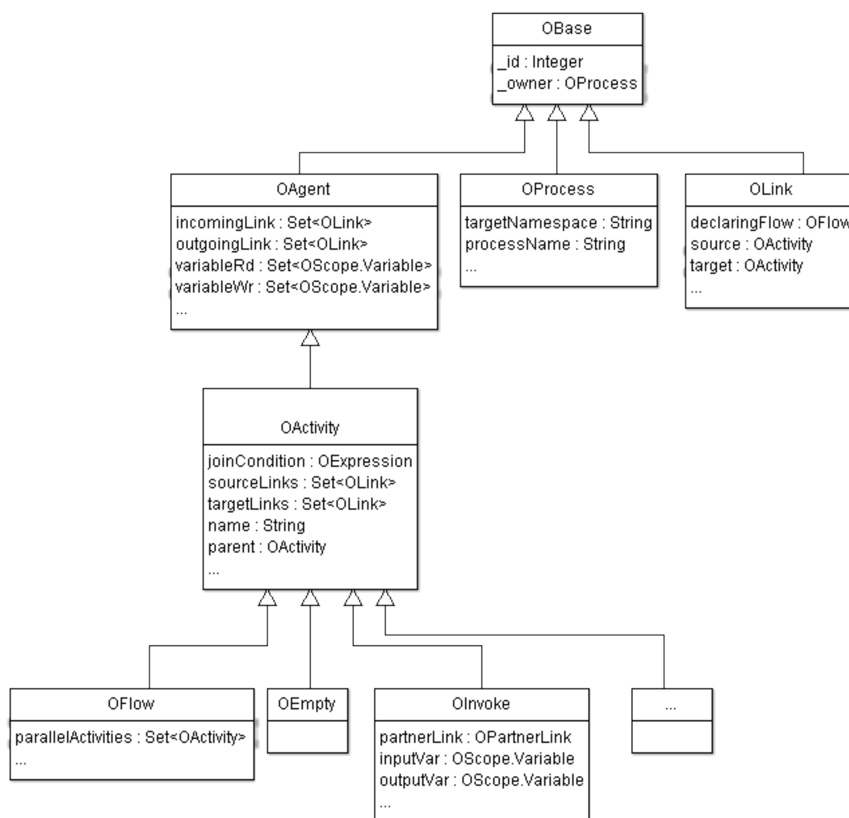



*Abbildung 3.3: Ausschnitt aus Klassendiagramms des ODE-Objektmodells*

Die Basisklasse `OBase` besitzt die Attribute `_id` und `_owner`, womit die Objekte identifiziert werden können. Mit `_id` wird ein Objekt (= BPEL-Element) eindeutig bezeichnet und mit `_owner` ein Verweis auf das Prozessmodell gegeben, zu dem es gehört. Das Prozesselement selbst wird durch ein Objekt der Klasse `OProcess` repräsentiert, welches eine Spezialisierung von `OBase` ist. Es speichert Informationen, wie beispielsweise Prozessname oder einen XML-Zielnamensraum des Prozesses. Sein `_owner` ist dabei eine Null-Referenz. Die innerhalb eines Prozesses vorkommenden Links werden in Objekte der Klasse `OLink` kompiliert. Alle BPEL-Aktivitäten werden als spezialisierte Klassen von `OActivity` repräsentiert. Diese gibt unter anderen die eingehenden und ausgehenden Links an der Aktivität an. Außerdem besitzt jede Aktivität noch einen Verweis auf eine Elternaktivität. Die Objektdarstellungen der einzelnen BPEL-Aktivitäten erben diese Informationen und speichern zusätzlich noch Daten, welche nur für sie von Bedeutung sind. So besitzen beispielsweise Objekte, die eine *invoke*-Aktivität repräsentierten, unterschiedliche Attribute als Objekte, die einen *Flow* beschreiben. Ein *Invoke* benötigt einen Partner Link und Variablen als Parameter für die Interaktion, während für den *Flow* die Aktivitäten von Bedeutung sind, die in ihm enthalten sind.

Dieses von Apache ODE verwendete Objektmodell unterscheidet sich allerdings in einigen Punkten von dem eigentlichen BPEL-Elementmodell. So verwendet ODE zum Beispiel kein eigenes Objekt um die BPEL-Aktivität *receive* zu repräsentieren. Diese wird stattdessen in ein semantisch gleichwertiges Konstrukt umgeformt, welches aus einer *pick*-Aktivität besteht, die nur einen *onMessage*-Zweig besitzt, der eine *empty*-Aktivität realisiert.

## 3.3 Ausführung von Prozessinstanzen

Die Navigation durch Prozessinstanzen baut in Apache ODE auf dem *JACOB Framework* (*Java Concurrent Objects*) [ODEa] auf. Dieses stellt Nebenläufigkeit der Ausführung von Aktivitäten und die persistente Speicherung der Prozesszustände zur Verfügung. Damit gewährleistet es, dass Prozessinstanzen nach einer Unterbrechung weitergeführt werden können. Das theoretische Modell hinter dem JACOB Framework ist das *ACTOR-Modell* [Agh86] für nebenläufige Systeme.

Alle BPEL-Aktivitäten stellen auf Instanzebene eine Unterklasse der Klasse `BPELJacobRunnable` dar, welche wiederum von `JacobObject` erbt. Ein `JacobObject` besitzt Funktionen, die zur Ausführung von Prozessinstanzen innerhalb von JACOB benötigt werden [SUP07]. Die direkte Unterklasse davon ist `JacobRunnable`, welche prinzipiell nur eine abstrakte Methode besitzt, nämlich `run()`. Der Einstiegspunkt für alle BPEL-bezogenen Objekte ist die Klasse `BpelJacobRunnable`. Sie bietet zum Beispiel

Methoden um auf die BPEL-Datenbank zuzugreifen oder um Fehler zu signalisieren. Von ihr erbt die Klasse `ACTIVITY`, welche die Basisklasse aller BPEL-Aktivitäten darstellt. Diese verweist auf ein `ActivityInfo`-Objekt, welche die Beschreibung einer kompilierten Aktivität enthält, also das zugehörige Objekt aus dem Ode-Objektmodell. Die einzelnen Aktivitäten erben von `ACTIVITY` und implementieren die `run()`-Methode aus `JacobRunnable`. In dieser wird die tatsächliche Funktionslogik der BPEL-Aktivität realisiert [SUP07]. In Abbildung 3.4 sind die beschriebenen Klassen und deren Zusammenhänge nochmal in einem Klassendiagramm abgebildet. Die angegebenen Methoden und Attribute sind dabei nicht vollständig.

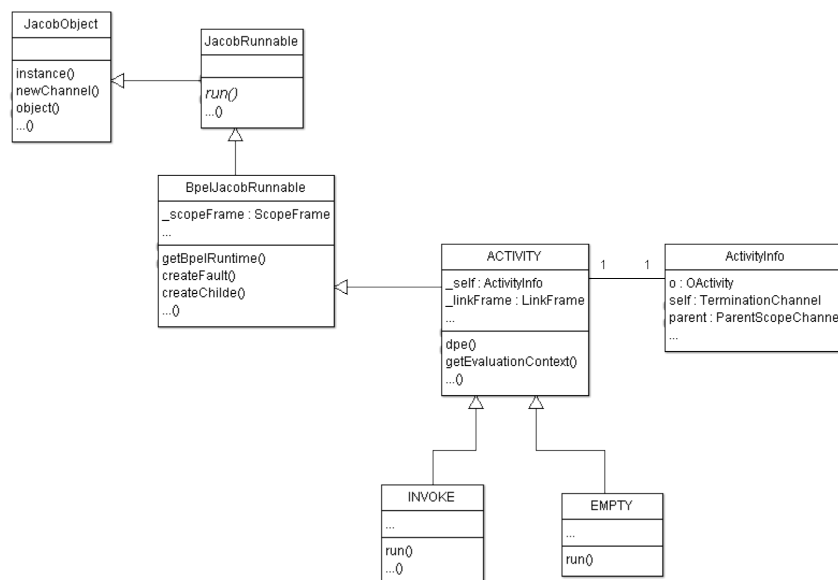



*Abbildung 3.4: Ausschnitt aus Klassendiagramm für JACOB-Objekte*

Neben diesen JACOB-Objekten stellen *Kommunikationskanäle* ein wichtiges Konzept des Frameworks dar. Sie dienen der Synchronisation zwischen den Aktivitäten und ermöglichen die Abarbeitung des Kontrollflusses. Eine Aktivität kommuniziert mit anderen Aktivitäten durch das Senden von Signalen über solche Kanäle. Für die Aktivitäten werden sofort bei deren Instanziierung einige grundlegende Kanäle zur Verfügen gestellt [ODEa], wie zum Beispiel ein `TerminationChannel` oder ein `ParentScopeChannel` (vgl. auch Abb. 3.4, Attribute von `ActivityInfo`). Über diese kann sie Signale an ihre Umgebung senden. Am anderen Ende eines solchen Kanals existieren so genannte *Listener*, die beim Empfang eines Signals eine bestimmte Funktion übernehmen.

Die tatsächliche Ausführung eines Prozesses findet nun in der *JACOB VPU (Virtual Processing Unit)* statt. Bei der Erzeugung einer neuen Prozessinstanz wird ein neues

Objekt der Klasse `BpelRuntimeContextImpl` generiert. Dieses stellt BPEL-Funktionalität zur Verfügung, die nicht direkt in JACOB vorhanden ist [Ste08]. Bei der Instanziierung eines Objektes der Klasse `BpelRuntimeContextImpl` wird auch ein Objekt der Klasse `JacobVPU` erzeugt. Es gibt also für jede Prozessinstanz eine eigene Virtual Processing Unit, was das parallele Ablaufen von mehreren Prozessinstanzen gleichzeitig unterstützt.

Für die Ausführung einer Prozessinstanz durch die VPU, verwendet diese eine so genannte *Execution Queue*. Dies ist ein Behälter für alle JACOB-Objekte, also Kanäle und deren Listener, sowie Aktivitäten [SUP07] einer Prozessinstanz. Eine Execution Queue spiegelt dadurch zu jedem Zeitpunkt den aktuellen Zustand der Ausführung wieder. Bei einer Unterbrechung kann sie serialisiert und gespeichert werden, und somit den Ausführungszustand sichern. Die Hauptaufgabe der VPU bei der Realisierung des Kontrollflusses besteht darin, das jeweils nächste Aktivitätsobjekt aus der Execution Queue zu nehmen und auszuführen, indem es dessen `run()`-Methode aufruft.

### 3.3.1 Beispiel für Ausführung eines Prozess-Kontrollflusses

Die bisher abstrakt beschriebene Vorgehensweise der Workflow Ausführung durch das JACOB Framework mit Hilfe der JACOB Objekte, Kanälen, Listenern und der JACOB VPU soll anhand eines Beispieles verdeutlicht werden. Dies ist allerdings etwas vereinfacht und dient lediglich der Veranschaulichung des Prinzips ohne dabei auf alle technische Besonderheiten von Apache ODE detailliert einzugehen. So wird beispielsweise auf die Verwendung von `ACTIVITYGUARD`s verzichtet, welche die Funktion der Überprüfung von eventuell vorhandenen eingehenden und ausgehenden Links einer Aktivität übernimmt. Auch wird die genaue Form, wie eine Aktivität in der Execution Queue repräsentiert ist, nämlich in Form einer sog. *Abstraction*, nicht angegeben. Vereinfachend wird dafür angenommen, dass es sich lediglich um das entsprechende JACOB-Objekt der Aktivität handelt, welches einen Verweis auf seine Channel besitzt.

Die Ausführung folgenden Ausschnittes eines BPEL-Prozesses aus (Listing 3.1) soll simuliert werden. Dieser stellt lediglich die sequenzielle Ausführung zweier Aktivitäten dar. Dabei gilt die Annahme, dass sich das SEQUENCE Objekt, welches ein JACOB-Objekt dieser *Sequence*-Aktivität ist, an erster Stelle der Aktivitäten in der Execution Queue befindet und dort auch danach keine weiteren Aktivitäten vorhanden sind. Die Zustände der Execution Queue während der Ausführung sind in der Abbildung 3.5 darstellt.

```
...
   <sequence>
      <assign> ... </assign>
      </empty>
   </sequence>
...
```

*Listing 3.1: Ausschnitt aus einem Beispiel-BPEL-Prozess*

Die Aufgabe der JACOB VPU ist es, jeweils das erste Element aus der Aktivitäten-Warteschlange zu entfernen und dessen `run()`-Methode aufzurufen. Laut Annahme ist dies nun ein `SEQUENCE` Objekt (vgl. Abb. 3.5, Zustand 1). In diesem Fall wird also `SEQUENCE.run()` aufgerufen. Die vereinfachte Implementierung einer *Sequence*-Aktivität als `JacobObject`, also die Klasse `SEQUENCE`, ist in Listing 3.2 angegeben. Dessen `run()`-Methode erzeugt zunächst einen neuen Kommunikationskanal des Typs `ParentScopeChannel`. Dann ruft sie die Methoden `newChannel()` und `instance()` auf. Beide Methoden werden in der Klasse `JacobObject` implementiert (vgl. Abb. 3.4) und fügen die entsprechenden Elemente (Kanal bzw. Aktivität) in die Execution Queue ein. Die Funktion `instance()` setzt dabei die als Parameter übergebene Aktivität an die letzte Stelle der Aktivitäten-Warteschlange. Dies ist in diesem Fall ein Aktivitätsobjekt, welches durch die Funktion `createChild()` erzeugt wurde. Im Sinne der BPEL-Aktivität *Sequence* liefert diese das nächste Element der Sequenz, also ein `ASSIGN`-Objekt, zusammen mit einem Verweis auf den zugehörigen Channel. Danach

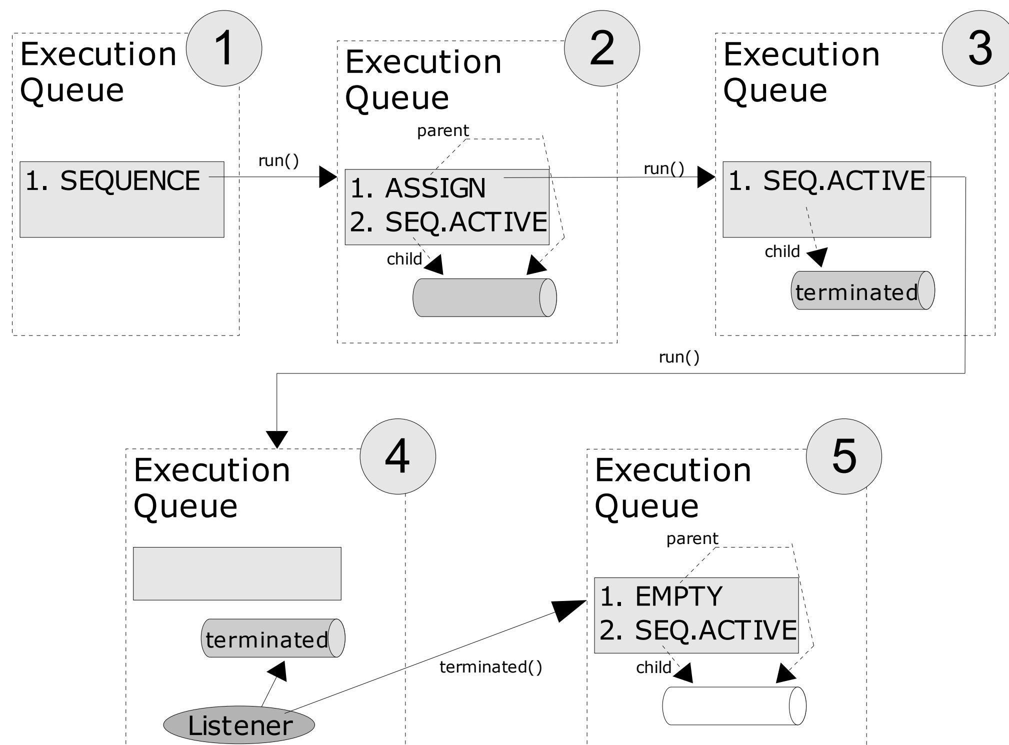



*Abbildung 3.5: Beispiel der Ausführung eines BPEL-Prozesses in Apache ODE*

wird in der `run()`-Methode durch `instance()` noch ein Objekt der Klasse `SEQUENCE.ACTIVE` zur Warteschlange hinzugefügt. Damit ist die `run()`-Methode des `SEQUENCE`-Objektes beendet und die Execution Queue beinhaltet die entsprechenden Daten, wie in Abbildung 3.5 (Zustand 2) dargestellt.

```
class SEQUENCE extends ACTIVITY {
   ...
   public void run() {
      ParentScopeChannel childChnl = new ParentScopeChannel();
      newChannel (childChnl);
      instance (createChild (childChnl)));
      instance (new ACTIVE (childChnl)))
   }

   private class ACTIVE extends BpelJacobRunnable {
      ...
      public void run() {
         object(new ParentScopeListener() {
            public void completed() {
               ParentScopeChannel childChnl = new
                                            ParentScopeChannel();
               newChannel (childChnl);
               Activity child = createChild (childChnl)
               if (child != null) {
                  instance (createChild (childChnl)));
                  instance( new ACTIVE() )
               }
               else {
                  // Benachrichtige eigene Elternaktivität
               }
            }
         };
      }
   }
}
```

*Listing 3.2: Vereinfachte Beispiel-Implementierung eines SEQUENCE Objektes in JACOB*

In der Warteschlange befindet sich nun also ein `ASSIGN`- vor einem `SEQUENCE.ACTIVE`-Objekt. Die VPU nimmt nun wiederum das erste Element von der Aktivitäten-Warteschlange, also das `ASSIGN`, und ruft dessen `run()`-Methode (Listing 3.3) auf. Diese erfüllt nun zunächst die funktionellen Aufgaben der *Assign*-Aktivität, also die Wertzuweisung von Variablen. Nachdem diese Aufgaben erledigt sind, sendet sie ein `completed`-Signal an den Kanal seiner Elternaktivität. Dies ist derselbe Kanal, der zuvor bei Ausführung des JACOB-Objektes `SEQUENCE` erzeugt und an das `ASSIGN` übergeben wurde (`childChnl` für `SEQUENCE` = `parentChnl` für `ASSIGN`). Damit ist das `ASSIGN` beendet. Der aktuelle Prozesszustand, wie er sich jetzt in der Execution Queue befindet, wird in Abbildung 3.5 (Zustand 3) veranschaulicht.

```
class ASSIGN extends ACTIVITY {
   ...
   public void run() {
      // Funktionen zum Erfüllen der assign-Semantik
      ....
      // nach deren Beendigung Elternaktivität benachrichtigen
      parentChnl.completed();
   }



class EMPTY extends ACTIVITY {
   ...
   public void run() {
      parentChnl.completed();
   }
```

*Listing 3.3: Vereinfachte Beispiel-Implementierungen von ASSIGN und EMPTY*

Als nächsten Schritt nimmt die VPU das erste Element von der Warteschlange, diesmal also das `SEQUENCE.ACTIVE`-Objekt. Mit diesem Konstrukt kann die Sequenz wieder die Steuerung für den Kontrollfluss übernehmen. Die `run()`-Methode dieses Objektes registriert durch die `object()`-Methode einen ChannelListener bei der Execution Queue, welche von einem `JacobObject` implementiert ist (vgl. Abb. 3.5, Schritt 4). Der Listener reagiert auf Signale auf dem Kanal `childChnl`, welcher noch derselbe ist wie zuvor. Auf diesem Channel wird jetzt ein `completed` Signal empfangen, welches durch das `ASSIGN` zuvor erzeugt wurde. Deshalb wird die entsprechende Methode des Listeners aufgerufen. Diese ist ähnlich zu der `run()`-Methode von `SEQUENCE` und erzeugt wieder die entsprechenden Objekte in der Execution Queue, nämlich `EMPTY` und `SEQUENCE.ACTIVE`, sowie einen entsprechenden Channel (Abb. 3.5, Zustand 5).

Die weiteren Bearbeitungsschritte laufen nun wieder genauso ab, wie zuvor. Die VPU nimmt das erste Element der Execution Queue, also `EMPTY` und ruft dessen `run()`-Methode (siehe Listing 3.3) auf. Diese sendet nach Beendigung, bei einer leeren Aktivität also sofort, ein Signal an seinen Eltern-Kommunikationskanal zurück, wodurch sie mitteilt, dass sie erfolgreich ausgeführt wurde. Die JACOB VPU nimmt das `SEQUENCE.ACTIVE` Objekt und ruft dessen `run()`-Methode auf, welches versucht das nächste Element der Sequenz zu erzeugen (`createChild()`). Da die *Assign*-Aktivität aber die letzte Aktivität der *Sequence* war, ist auch diese hiermit beendet. Diese sendet dann eine entsprechende Nachricht an seine Elternaktivität (nicht mehr im Beispiel abgebildet).

Wie beschrieben, veranschaulicht die Abbildung 3.5 den Ablauf der beschriebenen Prozessausführung anhand der jeweiligen Inhalte der Execution Queue. Anhand die-

ser Abbildung wird noch mal deutlich, dass die Execution Queue immer alle notwendigen Informationen (Aktivitäten in einer Warteschlange geordnet, Kanäle und deren Listener) enthält, die benötigt werden, um den aktuellen Prozesszustand hinsichtlich des Kontrollfluss zu beschreiben.

### 3.3.2 Datenaustausch zwischen den Aktivitäten

Neben dem Kontrollfluss ist auch der Zugriff auf Instanzdaten ein wesentlicher Bestandteil der Ausführung eines BPEL-Prozesses. Wenn eine Aktivität nun bestimmte Daten, wie Variablen benötigt, liest sie diese über die Data Access Objects vom persistenten Speicher, also dem ODE Data Store. Dabei muss sie wissen, auf welche Daten sie zugreifen darf bzw. welche Daten zur selben Prozessinstanz gehören wie die Aktivität selbst.

Jede Aktivität besitzt durch die Repräsentation im ODE-Objektmodell Verweise auf die benötigten Datenelemente (vgl. Abschnitt 3.2). Dadurch kennt sie den eindeutigen Bezeichner der Daten, auf welche sie zugreifen soll. Allerdings genügt dieser Bezeichner noch nicht, da sie auch ein tatsächliche Instanz des Datenobjektes zugreifen muss, welche zur selben Prozessinstanz gehört, wie die Aktivität. Dazu besitzt jede Aktivitätsimplementierung, also das Jacob-Objekte einer Aktivität, einen Verweis auf ein Objekt der Klasse `ScopeFrame`. Dieses spiegelt zur Laufzeit den Zustand der Scope-Instanz wieder, zu welchem die Aktivität gehört, und bietet die Möglichkeit der Auflösung von Daten-Objektbezeichner zu den Daten der entsprechenden Instanz.

Das `ScopeFrame`-Objekt wird durch die *Scope*-Aktivität `SCOPEACT` erzeugt und beinhaltet unter anderem eine Instanz-Id des Scopes und einen Verweis auf den `ScopeFrame` des Eltern-Scopes. Dieser ergibt sich daraus, dass Scopes immer geschachtelt sind. Lediglich der Scope auf oberster Ebene (Prozess-Scope) besitzt keinen Eltern-Scope und somit eine Null-Referenz. Eine Aktivität gibt beim Aufruf eines Kindes dieses `ScopeFrame`-Objekt immer mit, wodurch gewährleistet ist, dass jede Aktivität den Zustand des aktuellen Scopes kennt, in dem sie sich befindet. Sollte das aufgerufene Kind ein Scope sein, erzeugt dessen Aktivitätsimplementierung `SCOPEACT` dementsprechend einen neuen `ScopeFrame`, welcher an die eingeschlossenen Aktivitäten weitergegeben wird.

Greift eine Aktivität, also die `run()`-Methode eines Aktivitäts-JACOB-Objekts, nun bei der Ausführung einer Prozessinstanz zum Beispiel auf eine Variable zu, dann gibt sie den Bezeichner dieses BPEL-Variablen-Elementes an seinem `ScopeFrame`, welcher de Element-Bezeichner zu der tatsächlichen Instanz der Variablen auflöst. Dieser überprüft dabei, ob die Variable innerhalb seines Scopes definiert wurde und gibt einen entsprechenden Verweis bestehend aus Variablen-Bezeichner und Instanz zu-

rück. Wird dabei die Variable nicht innerhalb eines Scopes gefunden, so wird rekursiv über den `ScopeFrame` der Eltern-Scopes weitergesucht. Als Ergebnis dieser Auflösung kennt ein Aktivitätsobjekt nun neben dem Bezeichner der benötigten Variablen auch noch die zugehörige Instanz und kann somit auf die korrekten Variablendaten zugreifen. Dies geschieht durch den Aufruf einer entsprechenden Funktion der BPEL-Runtime, welche über die Abstraktionsschicht der DAOs die Daten aus einem persistenten Speicher holt und der Aktivität zur Verfügung stellt.

# 4 Verteile Ausführung von Workflows

In diesem Kapitel, welches den Schwerpunkt der vorliegenden Arbeit darstellt, wird die konzeptuelle Idee der verteilten Ausführung von Workflows dargestellt. Dazu wird zunächst der grundsätzliche Ablauf beschrieben, wie ein Workflow in eine entsprechende ausführbare Form überführt wird und das Konzept der verteilten Ausführung durch Tokenaustausch über Tuple Spaces erläutert. Zudem wird in diesem Kapitel untersucht, welche zusätzlichen Informationen ein BPEL-Prozess benötigt um auf die gewünschte Weise in verteilter Umgebung ausgeführt werden zu können. Dies betrifft vor allem die Konfiguration der BPEL-Aktivitätsimplementierungen. Des Weiteren werden die verschiedenen Token definiert, welche zwischen den einzelnen Aktivitäten ausgetauscht um einen verteilten Kontrollfluss und Datenaustausch zu ermöglichen.

## 4.1 Grundlegende Vorgehensweise

Die Methodik hinter dem verwendeten Ansatz zur verteilten Ausführung von Workflows wird in [MWL08a] beschrieben. Sie sieht die Umformung eines BPEL-Workflows in ein spezielles ausführbares Modell vor, welches schließlich auf die verschiedenen Teilnehmer verteilt wird und dort ausgeführt werden kann. Dabei umfasst sie die Schritte der *Prozess-Modellierung*, *Prozess-Segmentierung*, *Prozess-Transformation* und der *verteilten Prozessausführung* (vgl. Abbildung 4.1).

Zunächst einmal muss der Workflow definiert und modelliert werden. Dazu wird für diese Arbeit WS-BPEL als Workflowsprache verwendet. Die Basis für die verteilte Ausführung ist somit eine BPEL-Prozessdefinition.

Ein solcher BPEL-Prozess muss nun auf die teilnehmenden Partner aufgeteilt werden. Dies bedeutet, dass der Prozess auf verschiedene Maschinen, die jeweils einen bestimmten Teilaspekt ausführen, verteilt wird (vgl. einführende Beispiel zur Motivation dieser Arbeit in Abschnitt 1.1 und Abb. 1.1). Dieser Schritt wird Prozess-Segmen-

tierung genannt und erzeugt einen *Deployment Descripzor*, welcher die Abbildung von BPEL-Elementen auf die tatsächliche Infrastruktur beschreibt. Die Segmentierung kann manuell oder automatisch durchgeführt werden. Die automatische Variante bewertet das Prozessmodell nach bestimmten Gesichtspunkten und berechnet daraus eine optimale Verteilung auf die Partner. Dies kann nach mehreren Kriterien optimiert sein, also zum Beispiel nach der Anzahl der entfernten Dienstaufrufe von einem Prozessabschnitt oder nach den Kosten der Aktivitätsausführung [MWL08a]. Auf diese automatische Berechnung der Aufteilung eines Prozesses wird hier nicht weiter eingegangen.

Als nächstes muss der BPEL-Prozess nun in ein ausführbares Modell umgeformt werden. Dabei wird das Modell der EWFN verwendet. Die Transformation von BPEL nach EWFN ist in [Pop08] ausführlich beschrieben. Als letzter Schritt muss das Prozessmodell, welches in EWFN vorliegt, auf einer entsprechenden Infrastruktur umgesetzt werden, welche schon im Deployment Descriptor beschrieben ist. Dies bedeutet, dass einzelne Prozesssegmente auf die entsprechenden Maschinen verteilt werden. Das Modell der EWFN sieht zur Realisierung des Kontrollflusses einen Token-Austausch-Mechanismus vor. Dieser wird über Tuple Spaces realisiert, wie nachfolgend beschrieben. Nach dem Deployment der Prozessteile können Prozessinstanzen ausgeführt werden, indem sie die benötigten Token über die Tuple Space Infrastruktur austauschen.

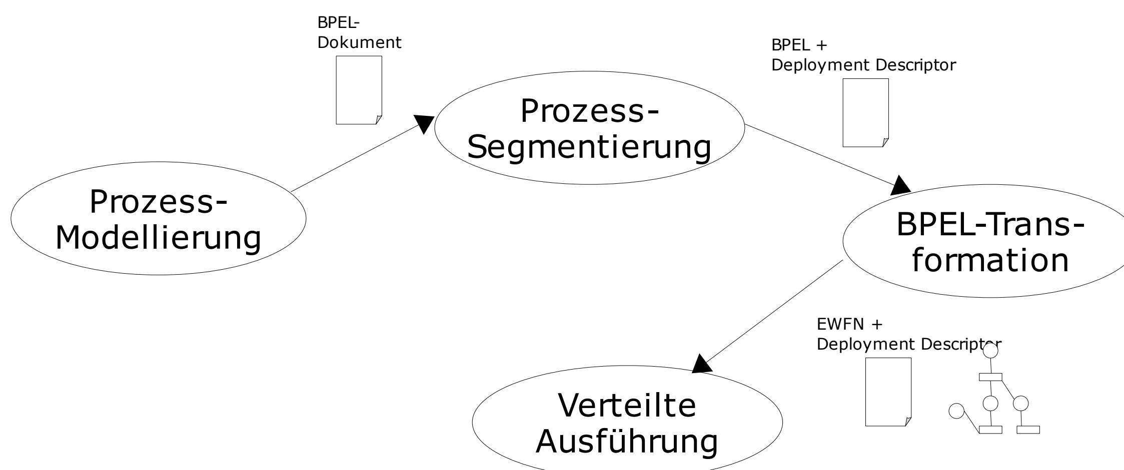



*Abbildung 4.1: Vom Modell zur verteilten Ausführung eines Prozesses*

## 4.2 Verteilte Ausführung mit Tuple Spaces

Bisher wurde das Konzept zur Ausführung über die EWFNs beschrieben. Dies ist ein theoretisches Modell, welches eine Art der Petrinetze beschreibt. Dabei interagieren verschiedene Verarbeitungsschritte (Transitionen) durch den Austausch von Token

über so genannte Stellen. Die Anwendung dieses abstrakten Konzeptes für die Ausführung von BPEL-Prozessen soll nun etwas veranschaulicht werden.

In Abbildung 4.2 ist der grundlegende Ablauf der Steuerung des Kontrollflusses durch den Tokenaustausch dargestellt. Die Aktivitäten kommunizieren miteinander indem sie Token auf Tuple Spaces schreiben bzw. davon lesen. Dabei sind Aktivitäten als die WfMS-Implementierung einer BPEL-Aktivität zu verstehen, also beispielsweise eine *invoke*-Aktivität und nicht der tatsächliche Web Service, der davon aufgerufen wird. Für den Kontrollfluss werden so genannte *Control Flow Token* (in Abbildung 4.2 als CF bezeichnet) verwendet. Eine Aktivität schreibt ein solches Token auf einen Tuple Space um seinen Nachfolgern zu signalisieren, dass diese gestartet werden können. Aktivitäten, die momentan nicht in einer Prozessinstanz ausgeführt werden, warten bis sie durch entsprechende Token gestartet werden. Das heißt, sie lesen solange auf Tuple Spaces bis sich dort alle Token befinden, die sie starten lassen (Startbedingung). Eine Aktivität kann mehrere Kontrollflusstoken benötigen um gestartet zu werden (vgl. mehrere eingehende Links). Ebenso kann sie beliebig viele Token erzeugen, beispielsweise um mehrere Kinderaktivitäten (vgl. Flow) zu aktivieren oder um mehrere ausgehende Links umzusetzen.

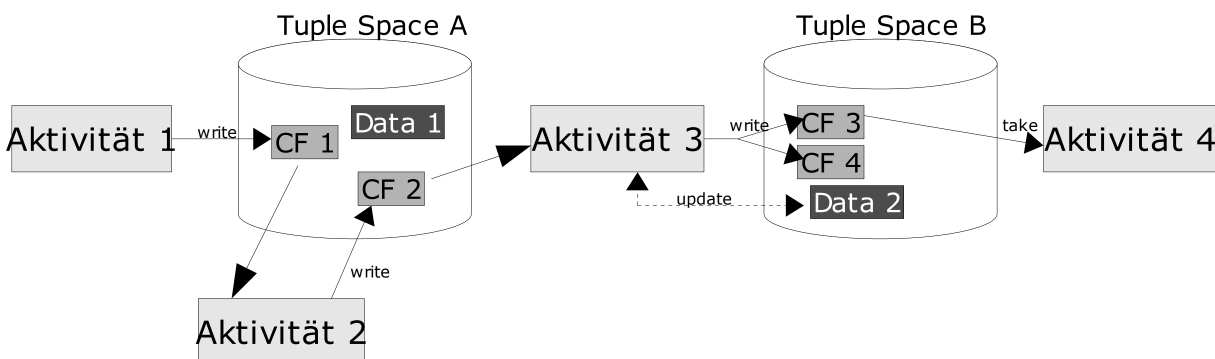



*Abbildung 4.2: Kontrollflusses durch Tokenaustausch über Tuple Spaces (in Anlehnung an [MWL08b])*

Auf ähnliche Weise erfolgt der Austausch von Daten zwischen den Aktivitäten, wie zum Beispiel Variablen. Diese werden ebenfalls durch Token realisiert, welche in einem Tuple Space gespeichert sind. Diese Daten-Token können von den Aktivitäten aus de Spaces gelesen und verändert werden. Allerdings ist das Lesen hier nicht destruktiv, da die Informationen, nachdem sie von einer Aktivität gelesen wurde, auch noch anderen Aktivitäten zur Verfügung stehen sollten.

Eine Aktivität, im Sinne der Implementierung einer BPEL-Aktivität in einer BPEL-Engine, ist also ein Client bei einem Tuple Space Server. Dieser hat hinsichtlich des Kontrollflusses die Aufgabe durch blockierendes Lesen auf Token zu warten, die seine

Startbedingung erfüllen. Sind diese vorhanden realisiert er die Logik der BPEL-Semantik, wobei er die eventuell benötigten Daten durch Zugriff auf Tuple Spaces erhält. Nach erfolgreicher Ausführung erzeugt die Aktivität dann Token um den Kontrollfluss an seine Nachfolger weiterzugeben. Danach wartet sie wieder auf die benötigten eingehenden Token.

## 4.3 Konfiguration der BPEL-Elemente

Um ein BPEL-Prozessmodell in der verteilten Umgebung unter Verwendung von Tuple Spaces ausführbar zu machen, muss das Modell mit Zusatzinformationen erweitert werden. Diese beschreiben, wie BPEL-Prozesselemente, beispielsweise Aktivitäten oder Variablen, auf die Infrastruktur abgebildet werden. Damit wird das Zusammenspiel verschiedener Aktivitäten, die auf unterschiedlichen Maschinen laufen können, durch einen Austausch der Daten und Kontrollflussinformationen über Tuple Spaces ermöglicht (vgl. Abb. 4.2). Dazu müssen die Aktivitäten aber entsprechend konfiguriert werden, damit sie beispielsweise wissen, auf welchen Tuple Space sich die Token befinden, auf die sie zugreifen.

In diesem Abschnitt wird nun beschrieben, welche Zusatz-Informationen bestimmte BPEL-Elemente benötigen, damit eine verteilte Ausführung des Prozesses möglich ist. Zudem wird ein XML Format definiert, mit welchem diese benötigten Informationen in einer strukturierten Form dargestellt werden können. Ein solches XML-Dokument kann als *Deployment Descriptor* eingesetzt werden, um den Prozess auf verschiedene Teilnehmer zu verteilen.

### *4.3.1 Konfigurationsinformationen*

Die Elemente eines BPEL-Prozesses, welche auf eine darunter liegende Tuple Space Infrastruktur abgebildet werden, lassen sich in Aktivitäten und Daten unterscheiden. Diese beiden Arten haben andere Anforderungen darin, wie sie abgebildet bzw. konfiguriert werden müssen. Das heißt, dass für Daten und Aktivitäten unterschiedliche Zusatzinformationen benötigt werden, welche nachfolgend erläutert sind.

**Datenelemente**

Die BPEL-Datenelemente definieren die zur Prozessausführung benötigten Daten. In BPEL sind dies Partner Links, Variablen und Correlation Sets. Bei dem entwickelten Modell der verteilten Ausführung (vgl. Abschnitt 4.2) werden deren konkreten Ausprägungen, also die Daten, welche innerhalb der Prozessinstanzen benötigen werden, als Token repräsentiert. Diese sind in einem Tuple Space gespeichert, auf den Aktivi-

täten zugreifen können um die benötigten Instanzdaten zu erhalten. Die Konfigurations-Information um ein Datenelement auf eine Tuple Space Infrastruktur abzubilden, besteht daher lediglich aus dem Speicherort (Tuple Space), an dem sich Instanz-Token des Elementes während der Ausführung befinden.

Für diese Arbeit wird angenommen, dass sich die Token eines Datenelementes, also deren Ausprägungen für bestimmte Instanzen, immer nur auf einem bestimmten Tuple Space befinden. Eine eventuelle Optimierungsmöglichkeit wäre es, Daten-Token auch auf verschiedene Spaces zu verteilen. Dies bedeutet, dass zum Beispiel die Instanz einer Variablen sowohl als Token auf einem Tuple Space A, als auch auf einem Tuple Space B liegt. Eine Aktivität 1 liest nun die Variable besser von Space A, während die Aktivität 2 dagegen effizienter auf Space B zugreift. Grund dafür kann beispielsweise eine schnellere Netzwerkanbindung zu den entsprechenden Remote Tuple Spaces sein. Der offensichtliche Nachteil bzw. Mehraufwand dieser verteilten Lösung liegt aber darin, dass die Token immer synchronisiert werden müssen damit sie sich jeweils auf dem gleichen Stand befinden. Diese Synchronisation könnte erfolgen, indem Aktivitäten, welche nur lesend auf die Daten zugreifen, nur einen bestimmten Tuple Space kennen. Dies wäre zum Beispiel der Space, auf welchen sie am effizientesten zugreifen können. Manipulierende Aktivitäten kennen dagegen alle Speicherorte eines Datenelementes und ändern bei einem Update der Daten entsprechend alle Token auf jedem Tuple Space, wo sich das Datenelement befinden kann. Dies erfordert aauch noch zusätzliche Konfigurationsinformationen für die Aktivitäten wie zum Beispiel die Unterscheidung ob lesend oder manipulierend und die Angabe der entsprechend benötigten Tuple Spaces. Eine detaillierte Betrachtung, ob bzw. wann ein einziger Speicherort für ein Datentoken oder eine Verteilung mit entsprechenden Synchronisationsmechanismen besser ist, erfolgt an dieser Stelle nicht.

**Aktivitäten**

Um die Aktivitäten für eine verteilte Ausführung konfigurieren zu können, benötigen diese zwei Arten an Informationen. Zum einen betreffen sie den Kontrollfluss, zum anderen die Verwendung von Instanzdaten. Diese Daten sind, wie gerade erläutert, in einem Tuple Space gespeichert. Eine Aktivität muss nun so konfiguriert werden, dass sie die entsprechenden Speicherorte der verwendeten Daten kennt. Der Kontrollfluss wird durch den Austausch von Token zwischen zwei Aktivitäten wiedergegeben.

Für eine Aktivität besteht der Kontrollfluss dabei aus drei Aspekten. Zum Ersten die Startbedingung für diese Aktivität, also der Token, die gegeben sein müssen, um sie zu starten. Der zweite Aspekt betrifft nur die strukturierten Aktivitäten, nämlich der interne Kontrollfluss. Dieser beinhaltet das Erzeugen von Token für die eingeschlossenen Aktivitäten und das Lesen von Token, welche von den Kind-Aktivitäten gene-

riert werden. Als drittens ist der Kontrollfluss nach Beendigung einer Aktivität von Bedeutung, d.h., die Benachrichtigung von möglichen Nachfolge-Aktivitäten bzw. der Elternaktivität. Um diese zu realisieren, muss eine Aktivität so konfiguriert werden, dass sie die entsprechenden Tuple Spaces kennt, über welche die verschiedenen Token ausgetauscht werden.

Für den eingehenden Kontrollfluss benötigt eine Aktivität nun Informationen über den Speicherort aller eingehenden Token. Dies sind erstens die Token, welche von der Elternaktivität erzeugt werden um eine eingeschlossene Aktivität zu aktivieren und zweitens solche Token, welche von Aktivitäten generiert werden, die die Quellaktivität von eingehenden Links darstellen.

Für den internen Kontrollfluss benötigt eine strukturierte Aktivität die Tuple Spaces, über welche sie mit den eingeschlossenen Aktivitäten kommuniziert. Diese Kommunikation bedeutet, dass eine Aktivität Kontrollflusstoken erzeugt, die als Startbedingung der Kinder dienen. Über denselben „Kanal“ (Tuple Space) benachrichtigt ein Kind seinen Vater über seine Beendigung. Dieser kann damit seine Kind-Aktivitäten, entsprechend der BPEL-Logik der Aktivität synchronisieren.

Nach Beendigung der Aktivität gibt sie den Kontrollfluss in Form von erzeugten Token weiter. Diese richten sich zu Einen an die Eltern-Aktivität um zu signalisieren, dass sie wieder die Kontrolle übernehmen kann und an zum Anderen an Aktivitäten, welche durch einen ausgehenden Link in Abhängigkeit zu der Aktivität stehen.

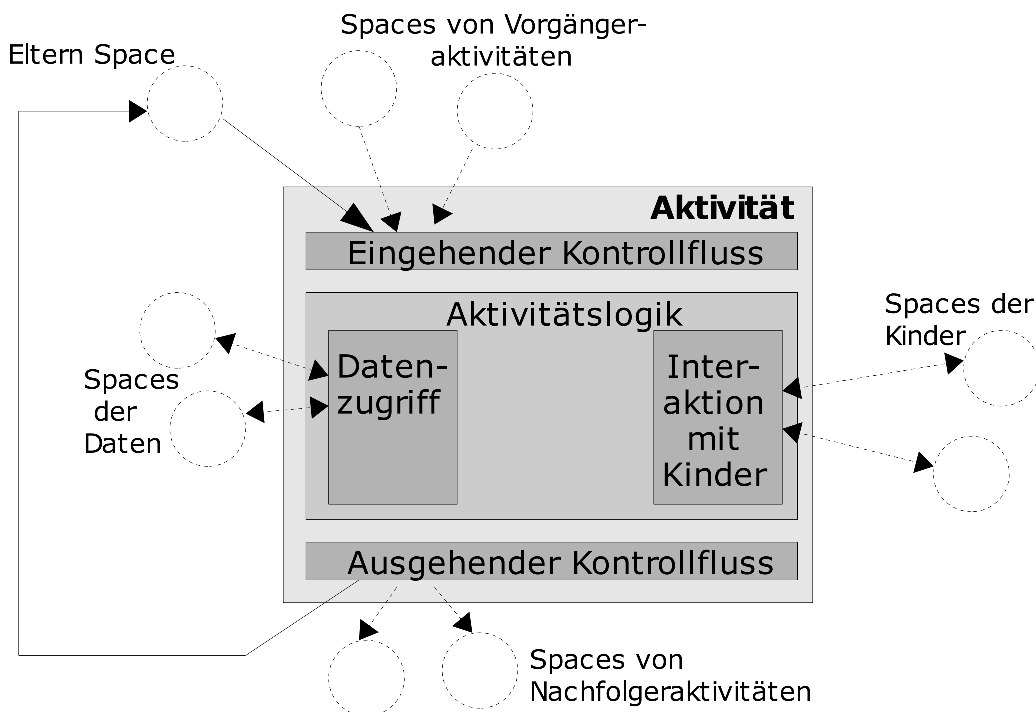



*Abbildung 4.3: Schnittstellen von Aktivitäten zu Tuple Spaces*

Die Abbildung 4.3 veranschaulicht noch einmal die benötigten Konfigurationsinformationen einer Aktivität. Dabei bedeutet Information in diesem Fall immer Kenntnis darüber, auf welchem Tuple Space sich welche Token befinden. Es gilt zu beachten, dass nicht für jedes Token ein eigener Tuple Space existieren muss. So kann die Kommunikation mit zwei verschiedenen Aktivitäten auch über den selben Tuple Space erfolgen. Dies bedeutet beispielsweise, dass sich zwei Token, die die Startbedingung einer Aktivität bilden, in ein und denselben Tuple Space befinden können.

Zuvor wurde gesagt, dass die Eingangsbedingung einer Aktivität aus mehreren Token bestehen kann. Solche von einer Elternaktivität und solche aufgrund eingehender Links. Allerdings ist zu beachten, dass dies prinzipiell zwei unterschiedliche Arten sind, um eine Aktivität zu starten. Dies kommt daher, dass BPEL sowohl über eine Art der Graph-basierten Prozessmodellierung (über Links) als auch über eine Operations-basierten Modellierung (durch Schachtelung von Aktivitäten) verfügt.

Die BPEL-Aktivitäten sind immer ineinander geschachtelt, wobei jede Aktivität direkt von ihrer Elternaktivität aktiviert wird. Die Aktivität auf oberster Ebene eines BPEL-Prozesses wird in diesem Sinne vom Prozess selbst (BPEL-Element `process`) gestartet. Zusätzlich dazu gibt es noch die Möglichkeit über Links den Kontrollfluss zu synchronisieren. Diese Links besitzen bei der Ausführung allerdings noch einen bestimmten Status, nämlich den Wert der so genannten *Transition Condition*.

Damit besitzt also jede Aktivität als Startbedingung auf jeden Fall einen Token, welcher von der Elternaktivität erzeugt wird. Optional dazu kann die Eingangsbedingung aus weiteren Token bestehen, die aus den Links resultieren. Wie erwähnt unterscheiden sich diese beiden Token prinzipiell dadurch, dass mit den Links zusätzlich noch ein Status übertragen wird. Dies muss bei der Umsetzung des Kontrollfluss durch den Austausch von Token beachtet werden.

Eine Möglichkeit wäre es, den Kontrollfluss nur über die Eltern - Kind-Beziehungen zwischen Aktivitäten zu realisieren. Dies bedeutet, dass eine Aktivität zunächst nur durch ein Token der Elternaktivität aktiviert wird. Als Eingangsbedingung für eine Aktivität dient damit nur ein einziges Token, womit eine Aktivität lediglich wissen müsste, wo sich dieses befindet bzw. wo sie es lesen kann. Die Kontrollflussaspekte der Links werden dagegen als Daten-Token realisiert, welche den Status der Übergangsbedingung eines Links speichern. Eine aktivierte Aktivität wartet dann solange bis die Transition Conditions aller eingehenden Links einen Wert besitzen und evaluiert diese entsprechend seiner Join-Bedingung. Konkret bedeutet das, dass eine Aktivität (= Client auf einem Tuple Space) solange blockierend auf einem Tuple Space liest bis ein entsprechendes Token der Elternaktivität vorhanden ist. Danach wird sie in den Zustand „aktiviert“ gesetzt, worin sie auf die Werte der Transition Condition al-

ler eingehenden Links wartet. Dies ist erneut ein blockierendes Lesen bis alle Token der Links einen Wert (*true* oder *false*) für ihre Übergangsbedingung besitzen, also der Status des Links nicht mehr *unset* ist. Erst danach wird entsprechend der BPEL-Semantik die Join-Condition evaluiert und die Logik der Aktivität ausgeführt oder die Aktivität ausgelassen.

Auf eine solche Weise wird der Kontrollfluss in Apache ODE realisiert (vgl. 3.3.1), wo die Implementierung einer strukturierten Aktivität immer seine Kinder aufruft. Dies erfolgt dort indirekt durch das Einfügen entsprechender Aktivitäten in eine Warteschlange. Die Aktivitäten benachrichtigen nach ihrer Ausführung wieder die Eltern-Aktivität durch ein entsprechendes Signal auf dem Kanal, der zur Kommunikation zwischen diesen beiden Aktivitäten dient, dem sog. „ParentScopeChannel".

Dieser Ansatz hat im Zusammenhang der verteilten Ausführung durch Tokenaustauch den Vorteil, dass alle Aktivitäten zum Starten jeweils genau ein einziges Token benötigen, unabhängig davon, wie viele Links auf sie „zeigen". Dies macht die Konfiguration entsprechend einfach, alle Aktivitäten müssen nur wissen, über welchen Tuple Space, sie mit ihren Eltern kommunizieren. Zudem spiegelt es die Tatsache wieder, dass jede BPEL-Aktivität zwar eine Elternaktivität besitzt, aber nicht unbedingt eingehende Links haben muss. Nachteil ist allerdings, dass eine Aktivität, welche Links besitzt, im Zustand „aktiviert" blockiert ist bis alle Links vorhanden sind. Dies wird am Beispiel in Abbildung 4.4 verdeutlicht. Die *Flow*-Aktivität wird für eine Prozessinstanz 1 aufgerufen und ihre Aufgabe ist es an alle eingeschlossenen Aktivitäten ein entsprechendes Token für die Instanz 1 (vgl. Aufbau der Kontrollfluss-Token in Abschnitt 4.4) zu senden. Damit ist die *Flow*-Aktivität zunächst beendet und übernimmt für diese Instanz erst wieder die Kontrolle, wenn alle Kinder eine entsprechende Nachricht gesendet haben. Sofort danach wird der *Flow* nochmals von einer neuen Prozessinstanz 2 gestartet und wieder werden die Token für alle Aktivitäten erzeugt. Die Aktivitätsimplementierungen sind nun so aufgebaut, dass sie aus einem „Template-Matching-Abschnitt" bestehen, welcher auf Token wartet, die die Startbedingung erfüllen. Sind diese vorhanden, wird für jede Instanz ein eigener Thread erzeugt, der die Aktivitätslogik realisiert (vgl. Kapitel 5). Dies bedeutet, dass die Aktivitäten A, B und C bzw. deren jeweiliger Template-Matching-Abschnitt die eingehenden Token verarbeitet und sofort entsprechende Threads erzeugt. In diesem Fall sind also sechs parallele Threads aktiv, wobei die Aufgabe der beiden Threads für die Aktivitätsinstanzen von C zunächst nur darin besteht blockierend zu lesen, bis die Datentoken der eingehenden Links vorhanden sind. Beinhaltet ein Flow nun beispielsweise viele Aktivitäten, die allerdings durch Links synchronisiert werden, bedeutet dies, dass unter Umständen sehr viele Threads existieren, die sich im Zustand „aktiviert" befinden und nichts tun außer auf Token zu warten.

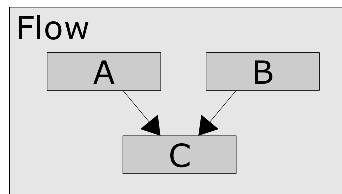



*Abbildung 4.4: Beispiel einer einfachen Flow-Aktivität*

Um dies zu vermeiden, werden Links nun sowohl über Kontrollflusstoken, als auch über Datentoken (vgl. Abschnitt 4.2 über den Aufbau der Token) realisiert. Eine Aktivität besitzt somit als Startbedingung nicht nur ein Token durch die Elternaktivität, sondern auch noch Kontrollflusstoken, die von den Quellaktivitäten der eingehenden Links erzeugt werden. Diese beinhalten aber zunächst keinen Status des Links, sondern geben nur an, dass die Vorgängeraktivität beendet wurde und die entsprechenden Links gesetzt hat, also die Datentoken der Links erzeugt bzw. deren Status festgelegt hat. Liegen für eine Instanz alle Eingangstoken vor, bedeutet dies für eine Aktivität, dass sie die eingehenden Links überprüfen und seine Join-Condition evaluieren kann. Dies realisiert sie, in dem sie auf die Datentoken zugreift, welche jeweils die Instanz eines Link, also dessen Status, beinhaltet. Nach Evaluierung der Eingangsbedingung kann eine Aktivität ausgeführt werden oder ausgelassen werden, im Sinne der von BPEL definierten Semantik.

Die Aufteilung der Token, welche einen Link repräsentieren in zwei Token, nämlich Kontrollfluss- und Daten-Token könnte auch recht einfach umgangen werden, indem die Kontrollfluss-Token direkt den Wert des Links beinhalten. Für einen Kontrollfluss, welcher nicht einen Link wiedergibt, also der Aufruf einer Aktivität durch ihre Eltern-Aktivität, müsste dazu im Token der Wert des Links immer auf true gesetzt werden. Dies wird allerdings nicht verwendet, da häufig der Kontrollfluss nicht durch einen Link gegeben wird und somit das zusätzliche Feld meist unnötig wäre. Zudem stellt dies eine Vermischung zweier BPEL-Konzepte dar, nämlich der Flow-basierten Modellierung (mittels Links) und der Operator-basierten Modellierung eines Prozesses, indem jeder Kontrollfluss dann konzeptionell über eine Art Link erfolgen würde.

### 4.3.2 Deployment Descriptor

Mit Hilfe eines *Deployment Descriptor* werden die zuvor genannten Zusatzinformationen zur Konfiguration der BPEL-Elemente beschrieben. Zudem soll er angeben, zu welchen Prozess-Segmenten bestimmte Aktivitäten gehören. Damit dient er als Eingabeformat für die Verteilung der Prozesssegmente auf die teilnehmenden verteilten BPEL-Engines.

### Grundstruktur

Die Grundstruktur des Deployment Descriptors ist in Listing 4.1 dargestellt.

```
<process processId="ProcessIdentifier">

    <partnerLinks>?
    </partnerLinks>

    <variables>?
    </variables>

    <correlationSets>?
    </correlationSets>

    <links>?
    </links>

    <activities>
    </activity>

</process>
```

*Listing 4.1: Grundstruktur des Deployment Descriptors*

Das Wurzelelement `process` besitzt das Attribut `processId`, welches dazu dient die Abbildung des Prozessmodells auf eine Infrastruktur zu identifizieren. Dieser Bezeichner wird bei der Ausführung verwendet um Token eindeutig den Aktivitäten desselben Prozessmodells zueinander zuordnen zu können.

### Datenelemente

Die Datenelemente (Partner Links, Variablen, Correlation Sets und Links) benötigen wie zuvor beschrieben wenig Zusatzinformationen. Sie werden vom Deployment Descriptor als erstes, vor den Aktivitäten, beschrieben. Der Grund dafür ist, dass sie die Grundlage der Aktivitäten sind bzw. von diesen verwendet werden. In Listing 4.2 sind die XML-Elemente zur Beschreibung einer Variablen im Deployment Descriptor beschrieben.

```
<variables>?
   <variable elementId="anyId"
             linkedElement="XPath-Expression"
             spacename="anyName" />+
</variables>
```

*Listing 4.2: Variablenelemente im Deployment Descriptor*

Datenelement besitzen, wie auch Aktivitäten, einen innerhalb des Prozessmodells eindeutigen Bezeichner. Dieser wird durch den Deployment Descriptor gegeben (`elementId`) und kann durch ein *Pre-Processing* der BPEL-Datei automatisch erstellt werden. Dieser Identifier dient den Aktivitäten auch als Bezugspunkt und ermöglicht einen effizienteren Zugriff auf die Daten-Token zur Laufzeit (vgl. Abschnitt 4.4.1).

Ein Element innerhalb des Deployment Descriptors verweist über einen XPath-Ausdruck im Attribut `linkedElement` auf ein bestimmtes BPEL-Element des zugehörigen BPEL-Prozessmodells. Die Referenz auf ein bestimmtes Element des Prozesses wäre auch über dessen Namen möglich. Allerdings müssen diese Namen innerhalb eines BPEL-Prozesses nicht eindeutig sein und können zum Beispiel in geschachtelten Scopes überschrieben werden. Als Vorgabe könnte auch gegeben werden, dass die Namen innerhalb eines BPEL-Prozesses so vergeben werden müssen, dass sie eindeutig sind. Dies wird konzeptionell an dieser Stelle nicht gemacht, da es eine Einschränkung für die modellierten BPEL-Prozesses wäre. Da aber möglichst alle BPEL-Prozesse, die der Spezifikation entsprechen, unterstützt werden sollen, wird diese Vorgabe nicht gemacht. Allerdings wird für die Implementierung eine solche Einschränkung zur Vereinfachung aber genutzt.

Das Attribut `spacename` enthält nun die eigentliche Information, die zur Abbildung auf die Tuple Space Infrastruktur benötigt wird. Dies ist der Speicherort, an dem sich die Instanzen dieses Elementes befinden.

Analog zu Listing 4.2 werden die weiteren Datenelemente des BPEL-Prozesses durch den Deployment Descriptor konfiguriert. Dies sind neben Variablen noch die Elemente Partner Links, Correlation Sets und Links (siehe Listing 4.3).

```
<partnerLinks>?
   <partnerLink elementId="anyId"
                linkedElement="XPath-Expression"
                spacename="anyName" />+
</partnerLinks>



<correlationSets>?
   <correlationSet elementId="anyId"
                   linkedElement="XPath-Expression"
                   spacename="anyName" />+
</correlationSets>
```

```
<links>?
   <link elementId="anyId"
         linkedElement="XPath-Expression"
         spacename="anyName" />+
</links>
```

Listing 4.3: Weitere Datenelemente im Deployment Descriptor

**Aktivitäten**

Die Aktivitäten eines BPEL-Prozesses benötigen andere Konfigurationsdaten zur verteilten Ausführung als die Datenelemente. Sie müssen, wie zuvor erläutert, vor allem wissen, wo sich die Kontrollflusstoken befinden, die eine bestimmte Aktivität betreffen. Dies sind sowohl die eingehenden als auch die ausgehenden Kontrollfluss-Nachrichten, wobei nach dem zuvor beschriebenen Konzept mehrere Token und somit auch mögliche Tuple Spaces vorhanden sein können. In Listing 4.4 ist die Darstellung der benötigten Konfigurationsdaten für eine Aktivität in der vorgeschlagenen XML-Form des Deployment Descriptors dargestellt.

Um die Prozesssegmentierung, also die Aufteilung eines BPEL-Prozesses auf verschiedene Prozess-Teile zu ermöglichen, dient das Attribut `segment`. Mit diesem wird angegeben zu welchem Prozessteil eine bestimmte Aktivität zugeordnet ist.

```
<activity elementId="anyId" segment="segment">

   <controlflow>
      <parent tuplespace="Spacename" />
      <source activityId="anyId" tuplespace="Spacename"/>?
      <target activityId="anyId" tuplespace="Spacename"/>?

      <children>?
         <child activityId="" tuplespace="Spacename" />+
      </children>

      <fault tuplespace="Spacename" handlerID="anyId">
   </controlflow>

   <data>?
      <partnerLink    id="elementId" />*
      <variable       id="elementId" />*
      <correlationSet id="elementId" />*
      <sourceLinks>?
          <link linkId="elementId" />+
      </sourceLinks>
      <targetLinks>?
         <link linkId="elementId" />+
      </targetLinks>
```

```
      </data>

   </activity>
```

*Listing 4.4: Aktivitäten im Deployment Descriptor*

Die Angaben über die Tuple Spaces, welche zur Realisierung des Kontrollflusses benötigt werden, befinden sich innerhalb des Elementes `controlflow`. Dieses enthält mindestens das Element `parent`, welches den Tuple Space angibt, über den die Kommunikation zwischen einem Element und seinem Elternknoten stattfindet. Da eine Aktivität auf jeden Fall zumindest durch eine Eltern-Aktivität aufgerufen wird, wird der entsprechende Tuple Space immer benötigt. Wie erläutert, können Aktivitäten durch die Links noch weitere Eingangs-Kontrollflusstoken besitzen. Der Speicherort eines solchen Tokens wird durch das Element `source` beschrieben. Analog dazu dienen die `target`-Elemente zur Angabe der Tuple Spaces, auf welche eine Aktivität seine erzeugten Token schreiben soll. Um die Token den korrekten Aktivitäten zuordnen zu können, benötigen diese Elemente noch den Bezeichner der Aktivität, von der das Eingangstoken stammt bzw. welche durch ein erzeugtes Token aufgerufen werden soll. Näheres dazu wird in Abschnitt 4.4.1 bei der Vorstellung der Kontrollflusstoken beschrieben. Neben dem eingehenden und ausgehenden Kontrollfluss einer Aktivität, gehört auch noch die Aktivierung der eingeschlossenen Aktivitäten dazu. Die dafür verwendeten Tuple Spaces werden innerhalb des optionalen Elementes `childrens` angegeben, wobei auch hier der Bezeichner der zugehörigen Aktivität mitgegeben werden muss. Zusätzlich gehört zum Kontrollfluss auch noch die Möglichkeit Fehlermeldungen an einen Fault Handler zu übergeben. Dies geschieht ebenfalls durch den Austausch von Token, wobei in dem Element `fault` angegeben wird, auf welchen Tuple Space ein Handler auf entsprechende Fehlermeldungen wartet.

Zu beachten ist natürlich noch, dass zwischen zwei kommunizierenden Aktivitäten die entsprechenden Tuple Spaces im Deployment Descriptor korrekt angegeben sind. Dies bedeutet, dass zum Beispiel die `source` und `target`-Elemente zweier Aktivitäten übereinstimmen müssen. In Listing 4.5 erzeugt die Aktivität 37 Token, welche eine Link-Beziehung zu Aktivität 42 darstellen. Diese müssen natürlich auf denselben Tuple Space geschrieben werden, auf welchen Aktivität 42 liest. Dazu muss bei der Aktivität 37 das Attribut `tuplespace` des `target`-Elementes, welches auf die Aktivität 42 verweist, denselben Wert besitzen, wie das zugehörige `source`-Element der Aktivität 42. Diese Bedingung sollte immer erfüllt sein und lässt sich statisch durch Parsen des Deployment Descriptors überprüfen. Ähnliches gilt nicht nur für Aktivitäten, die über einen Link verbunden sind gelten, sondern auch für die Eltern-Kind-Beziehung zweier Aktivitäten.

```
<activity elementId="37"..>

   <controlflow>
      ...
      <target activityId="42" tuplespace="Space_ABC"/>
   </controlflow>
      ...
</activity>


<activity elementId="42"..>

   <controlflow>
      ...
      <source activityId="37" tuplespace="Space_ABC"/>
   </controlflow>
      ...
</activity>
```

*Listing 4.5: Beispiel zur Verwendung von Quell- und Ziel-Tuple Spaces*

Neben Informationen zum Kontrollfluss benötigen Aktivitäten noch das Wissen, wo sie auf welche Daten zugreifen können. Diese Informationen werden innerhalb des Elementes `data` gegeben. Dabei verweisen sie auf die zuvor definierten Bezeichner der Datenelemente. Es genügt also die Speicherorte der Datenelemente in einem separaten Abschnitt, wie zuvor beschrieben, anzugeben, da eine Aktivität nur noch darauf verweist.

**Erzeugung und Verwendung des Deployment Descriptor**

Wie schon erwähnt lassen sich Teile dieses Deployment Descriptor Dokumentes in einem ersten Schritt automatisch aus einer BPEL-Datei erzeugen. Dies ist zum einen das Generieren der eindeutigen Bezeichner für alle beschriebenen Datenelemente und Aktivitäten (`elementId`). Des Weiteren lassen sich aus der BPEL-Datei sämtliche Verweise generieren, also beispielsweise das komplette `data`-Element innerhalb einer Aktivität mit samt seiner Kind-Elemente. Ebenso ist es es möglich, für alle Aktivitäten die entsprechenden `source`- und `target`-, sowie `children`-Elemente zu generieren und zugehörige `activityId`s darin zu erzeugen. Allerdings noch ohne Angabe eines Tuple Spaces.

Nicht automatisch aus der BPEL-Datei erzeugen, lassen sich die Daten, welche die eigentliche Abbildung auf die tatsächliche Infrastruktur beschreiben. Dies sind, vereinfacht gesagt, alle verwendeten Tuple Spaces. Die Angabe der Tuple Space Namen, als Attribute innerhalb der Elemente des Deployment muss also in einem zweiten Schritt erfolgen. Zudem verlangen die Aktivitäten noch die Angabe eines Segmentes durch

das Attribut `segment`, mit dem die Aufteilung des Prozesses in verschiedene Teile übernommen werden kann. Dies kann ebenso nicht automatisch aus der BPEL-Prozessdefinition generiert werden, sondern stammt aus dem Prozess-Segmentierungs-Schritt.

Beim Deployment eines Prozesses auf die teilnehmenden BPEL-Engines wird ein BPEL-Prozess und der zugehörigen Deployment Descriptor benötigt. Die Workflow Engine fragt beim Deployment dieser beiden Dokumente nach, welches Segment auf ihr ausgeführt werden soll. Damit werden nur die Aktivitäten dieses Prozessteiles auf der entsprechenden Maschine zur Verfügung gestellt. Durch die Konfigurationsdaten des Deployment Descriptor besitzen sie allen notwendigen Informationen um mit anderen Aktivitäten interagieren zu können.

## 4.4 Tuplestruktur

Nachdem jetzt geklärt wurde, wie die Aktivitäten konfiguriert sein müssen, um den Kontrollfluss durch einen Token-Austausch-Mechanismus zu realisieren, werden nun die entsprechenden Token definiert. Diese Token werden als Tupel umgesetzt, die in einem Tuple Space gespeichert sind und dort abgerufen werden können.

Grundsätzlich lassen sich zwei Arten von Tokentypen unterscheiden: *Kontrollfluss-* und *Datentoken*. Bei der Ausführung einer Prozessinstanz werden Kontrollflusstoken für die Navigation durch das entsprechende Modell genutzt. Datentoken enthalten zusätzliche Informationen wie die Instanzen von Variablen oder Partner Links, die zur Ausführung benötigt werden.

Ein Tupel ist im Sinne der Linda Tuple Spaces (vgl. Abschnitt 2.5) zunächst nur eine Liste von typisierten Feldern. Um die vorgeschlagene Tuplestruktur generisch zu halten, werden die benötigten Felder konzeptuell in Header- und Body aufgeteilt. Im Header befinden sich dabei solche Felder, die alle Tokentypen benötigen, im Body dagegen die spezifischen für eine bestimmte Art von Token. In Tabelle 4.1 sind die Header-Felder dargestellt.

| Feldname | Datentyp | Beschreibung |
|---|---|---|
| Tokentyp | {CF, DATA} | Gibt die Grobunterteilung zwischen den zwei Tokentypen an |
| Prozess-Id | Integer | Bezeichner des Prozessmodells |
| Instanz-Id | Integer | Bezeichner einer Prozessinstanz |

| | | |
|---|---|---|
| Constant | Boolean | Gibt an, ob das entsprechende Token veränderbar ist oder nicht |

*Tabelle 4.1: Header-Felder der Token*

Die Token dienen der Kommunikation und dem Datenaustausch zwischen den Aktivitäten. Die Felder Prozess-Id und Instanz-Id ermöglichen es den Aktivitäten auf solche Token zuzugreifen, die zu ihrem eigenen Prozessmodell gehören. Dabei kennt eine Aktivität immer das eigene Prozessmodell durch dessen Bezeichner aus den Konfigurationsinformationen. Die aktuelle Instanz einer Prozessausführung ist für eine Aktivität erst zur Laufzeit eines Workflows bekannt. Bei der Realisierung des Kontrollflusses ist die Instanz aber auch noch nicht von Bedeutung, dieser wird erst bei dem Zugriff auf Daten-Token von Aktivitäten benötigt. Einzelheiten dazu sind in den folgenden Unterabschnitten beschrieben.

Das Feld Constant legt ein Token als unveränderbar fest. Dies betrifft zum Einen die Kontrollflusstoken, aber auch bestimmte Datentoken. Eine Kontrollflusstoken wird von einer Aktivität erzeugt und von einer nachfolgenden Aktivität aus dem Tuple Space entfernt (destruktives Lesen). Dazwischen darf es nicht mehr verändert werden, allerdings gibt es dafür auch keinen Grund. Der eigentliche Zweck des Attributes betrifft vielmehr die Datentoken, insbesondere im Zusammenhang mit Correlation Sets. Ist ein Correlation Set einmal initialisiert, darf es nicht mehr verändert werden.

### 4.4.1 Kontrollflusstoken

Die Kontrollflusstoken dienen der Navigation durch eine Prozessinstanz. Sie werden benötigt um Aktivitäten zu aktivieren und um den Ablauf innerhalb strukturierter Aktivitäten zu synchronisieren. Zudem werden sie zur Dead Path Elimination verwendet.

Für das Modell der Executable Workflow Networks wurden in [MWL08a] entsprechende Kontrollflusstoken als Tupel („CF“ $\times$ $\mathbb{B}$ $\times$ $\mathbb{N}$ $\times$ $\mathbb{N}$) definiert. Hierbei ist $\mathbb{B}$ = {true, false} und gibt an, ob es sich um einen positiven oder negativen (dead Path) Kontrollfluss handelt. Die beiden Felder mit den natürlichen Zahlen stellen die Bezeichner eines Prozessmodells und einer Prozessinstanz dar. In [Pop08] wurde dieses Tupel um ein History-Feld erweitert, sowie in $\mathbb{B}$ der dritte Wert „failed“ aufgenommen.

Aufbauend auf diesen Überlegungen wurde für die Implementierung das Kontrollflusstoken als ein Tupel der Form („CF“ $\times$ Prozess-Id $\times$ Instanz-Id $\times$ KF-Typ $\times$ Zie-

laktivität-Id × Quellaktivität-Id × History) definiert. Die ersten Felder entsprechen dabei den zuvor erläuterten Header-Feldern. Die weiteren Felder sind in Tabelle 4.2 genauer definiert. Ihre Bedeutung wird nachfolgend noch erläutert.

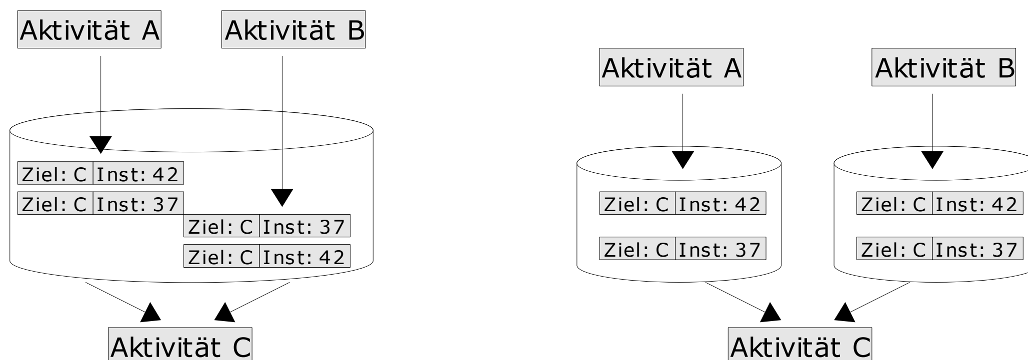
| Feldname | Datentyp | Beschreibung |
|---|---|---|
| KF-Typ | {true, false, failed} | Gibt an, ob es sich um einen positiven Kontrollfluss oder einen Dead Path bzw. Fehlerfall handelt |
| Zielaktivität-Id | Integer | Bezeichner der Aktivität, die das Token verarbeiten soll |
| Quellaktivität-Id | Integer | Bezeichner der Aktivität, die das Token erzeugt hat |
| History | (Integer*) | Tupel, welches die Scope-Instanzen angibt, die den Kontext des Tokens beschreiben |

*Tabelle 4.2: Felder eines Kontrollflusstokens*

Das Feld KF-Typ entspricht dem oben genannten $\mathbb{B}$-Feld. Es gibt also den Typ des Kontrollflusses an. Der Wert „true“ besagt, dass der Ablauf bisher korrekt war und eine Aktivität, welches dieses Token verarbeitet, normal gestartet werden kann. Der Wert „false“ bedeutet, dass die Aktivität sich auf einem toten Pfad befindet und „failed“ heißt, dass bei einer vorherigen Aktivität ein Fehler aufgetreten ist. Die Verarbeitung eines „failed“- bzw. „false“-Token durch eine Aktivität ist ähnlich. Bei einem Token mit dem Wert „failed“ wird die Aktivität nicht ausgeführt, da sich bei irgendeiner Vorgängeraktivität ein Fehler ereignet hat und deshalb alle nachfolgenden Aktivitäten nicht mehr ausgeführt werden. Deshalb gibt diese Aktivität ebenso ein „failed“-Token an seine Nachfolger weiter. Der Wert „false“ gibt an, dass die Vorgängeraktivität sich schon auf einem nicht ausgeführten Pfad (*dead path*) befindet und dient der Dead Path Elimination. Eine Aktivität die an ihrem Eingang einen solchen Token empfangen hat, verarbeitet diesen folgendermaßen. Wenn die Join-Condition für die Aktivität erfüllt ist, der Links des toten Pfades ist false, wird die Aktivität normal ausgeführt und erzeugt sie für ihre Nachfolger normale, positive, Kontrollflusstoken. Ist diese Bedingung nicht erfüllt, dann wird sie nicht ausgeführt und erzeugt dementsprechend DP-Token („false“). Eine Aktivität, die ein „failed“-Token empfangen hat, wird nicht ausgeführt und gibt dieses Token immer automatisch weiter. Der Unterschied zwischen „failed“ und „false“ liegt in der Verarbeitung durch eine Aktivität also darin, dass eine Aktivität, die ein Token mit dem Wert „failed“ empfangen hat, nie ausgeführt wird, während sie bei einem „false“-Token trotzdem gestartet und durchgeführt werden kann.

Der Bezeichner der Zielaktivität wird benötigt, um durch das Token eine bestimmte Aktivität „aufzurufen". Eine Aktivität besitzt einen, innerhalb eines Prozessmodells eindeutigen Identifier (vgl. Konfigurationsdaten der Aktivitäten). Damit eine Aktivität gestartet werden kann, d.h., deren Logik soll innerhalb einer Prozessinstanz ausgeführt werden, muss ihre Eingangsbedingung erfüllt sein. Dies Bedingung sind, wie zuvor erläutert ein oder mehre entsprechende Eingangs Token. Da eine Aktivitätsimplementierung zunächst ihre Instanz noch nicht kennt, muss sie allein durch die Felder Prozessmodell-Id und Zielaktivität-Id die Token lesen, welche an sie adressiert werden. Tatsächlich ist das aber nur in dem Fall, dass eine Aktivität lediglich ein eingehendes Token als Startbedingung besitzt, erfolgreich. Hierbei könnte eine Aktivität ein Template verwenden, welches nur das Prozessmodell und die Aktivität beinhaltet, um entsprechende Start-Token zu lesen.

Etwas komplizierter ist dies bei mehreren Eingangstoken. In diesem Fall reicht die Angabe des Prozessmodells und der Ziel-Aktivität nicht mehr aus. Dies wird anhand der Abbildung 4.5 verdeutlicht. Dabei besitzt die Aktivität C als Eingangsbedingung zwei Token, die von den Aktivitäten A und B erzeugt werden. Ein Lesen durch das vorgeschlagene Template, also durch Verwendung der Zielaktivität, liefert hierbei nicht unbedingt das korrekte Ergebnis. Schreiben beide Erzeuger die Token auf denselben Tuple Space (Abb. 4.5 a), so kann es vorkommen, dass die Aktivität C zweimal Token liest, welche von Aktivität A erzeugt wurden und dies als Erfüllung ihrer Startbedingung ansieht. Bei der Nutzung zweier verschiedener Tuple Space (Abb. 4.5 b) könnte dies so nicht vorkommen. Allerdings tritt zusätzlich in beiden Fällen noch ein weiteres Problem auf. Bei dem Lesen der Token durch Aktivität C kennt diese noch nicht die tatsächliche Prozessinstanz, die ausgeführt werden kann. So wäre es auch noch möglich, dass die zwei gelesenen Token zwar von den beiden Aktivitäten A und B stammen, aber beispielsweise das Token A zu Instanz 42 gehört, während Token B aus Instanz 37 stammt.




*a) Kontrollflusstoken über einen Tuple Space* *b) Kontrollflusstoken über zwei Tuple Spaces*

*Abbildung 4.5: Probleme bei Kontrollflusstoken ohne Absender-Informationen*

Um das erste Problem zu lösen, dient das Absender-Feld in den vorgeschlagenen Kontrollflusstoken (Tabelle 4.2). Wenn eine Aktivität nun ein Token lesen möchte, so verwendet sie in einem Template zusätzlich noch den Erzeuger dieses Token. Die entsprechenden Informationen, also von wem sie auf welchem Tuple Space ein Token erhält, besitzt sie durch die Konfigurationsdaten. Dies ermöglicht es der Aktivität, am Beispiel in Abbildung 4.6, durch zwei verschiedene Template, einmal mit Absender A und einmal Absender B, zwei Token zu lesen, welche von den beiden Vorgängeraktivitäten stammen und somit die Startbedingung bilden.

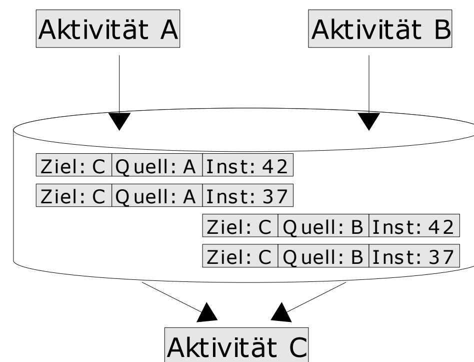



*Abbildung 4.6: Kontrollflusstoken mit Absender-Informationen*

Allerdings ist damit immer noch nicht das Problem gelöst, dass die gelesenen Token aus zwei unterschiedlichen Instanzen stammen können. Dieses muss auf Aktivitätsebene durch ein Synchronisieren des Lesens der zwei Token gelöst werden. Beispielsweise kann sie warten bis ein Token vorhanden ist und mittels dessen angebener Prozessinstanz dann das Template für das zweite Token bilden und darauf warten. Befinden sich alle eingehenden Token, welche die Startbedingung einer Aktivität bilden, innerhalb desselben Tuple Spaces, so liefert die verwendete Tuple Space Implementierung [Wu08] schon eine entsprechende synchronisierte Leseoperation. Diese kann durch eine Menge von Templates für die unterschiedlichen Quellaktivitäten, bei denen jeweils dieselbe Zielaktivität und Prozessmodell verwendet werden, aufgerufen werden. Zusätzlich wird für das Feld der Prozessinstanz ein Platzhalter angegeben. Diese synchronisierte Leseoperation wartet nun so lange bis für jedes Template ein Token vorhanden ist, wobei die Prozessinstanz bei allen Token identisch ist.

Das Feld Quellaktivität-Id besitzt nicht nur bei mehreren Eingangstoken, sondern auch noch im Zusammenhang mit strukturierten Aktivitäten eine Bedeutung. Eine strukturierte Aktivität ruft die eingeschlossenen Aktivitäten auf, indem es ein Token erzeugt, welche an ihre Kinder gerichtet sind. Die entsprechenden Bezeichner der Kindaktivitäten kennt die Aktivität aus den Konfigurationsdaten. Ist diese Kind-Aktivität beendet, sollte die Elternaktivität wieder die Kontrolle übernehmen, wozu der Austausch eines Tokens zwischen Kind- und Vateraktivität erfolgt. Die eingeschlosse-

ne Aktivität erzeugt dieses „Antwort"-Token, indem sie bei ihrem Eingangstoken die Felder der Quell- und Zielaktivität. Eine Kindaktivität muss daher keine Informationen über seinen Vater besitzen, abgesehen vom Tuple Space, über den die Kommunikation stattfindet. Dieser ist laut Annahme derselbe, wie der, über den der Vater sein Kindelement aktiviert hat. Die Vateraktivität liest Kontrollflusstoken, die an sie gerichtet sind, wie zuvor beschrieben durch den eigenen Bezeichner und die Prozessmodell-Id und der erwarteten Absender. Eine strukturierte Aktivität muss nun so realisiert werden, dass sie zusätzlich zu den zuvor beschriebenen Token für die Startbedingungen auch noch eingehende Token, die von den Kindern erzeugt werden, verarbeiten kann. Dazu besitzt sie zwei Arten von „Eingangsbedingungen" bzw. besteht aus zwei Teilen, die unterschiedliche Bedingungen besitzen. Zunächst einmal die externe, wie bisher beschrieben und dazu noch eine interne Eingangsbedingung für Token der Kind-Elemente. Während die externe Bedingung eine neue Aktivitätsinstanz startet, dient die interne dazu, die Kontrolle über die eingeschlossenen Aktivitäten wieder zu bekommen. Dies bedeutet, dass eine strukturierte Aktivität beim Empfangen eines Tokens, welches von einer eingeschlossenen Aktivität erzeugt wurde, die entsprechende Synchronisationsmaßnahmen ergreift.

Die Abbildung 4.10 zeigt ein Beispiel dieser Vorgehensweise anhand einer Sequence S, die aus den Aktivitäten A und B besteht. Zunächst wird die Sequence-Aktivität S aufgerufen (1.). Dies geschieht durch ein Token (Ziel: S, Quelle: X), welches die externe Schnittstelle betrifft. Die Sequence-Aktivität startet also für eine neue Instanz, dies bedeutet, dass die erste Aktivität der Sequenz aufgerufen wird. Deswegen erzeugt S einen Token (Ziel: A, Quelle S), welcher A aufruft (2.). Die Aktivität A nimmt dieses Token (3.) und erzeugt nach erfolgreicher Ausführung ein neues Token, bei dem Absender und Empfänger umgedreht sind (4.). Die Aktivität S wartet in der Zwischenzeit auf neue Token, welches an sie adressiert ist. Das nächste so gelesene Token (5.) ist jenes, welches von A erzeugt wurde. Dann erkennt die Sequence S durch den Absender des soeben gelesenen Tokens, dass dieses von einer eingeschlossenen Aktivität, nämlich A, erzeugt wurde und übernimmt damit die Aufgabe der Synchronisation. Das heißt, sie stellt fest, welches die Folgeaktivität von A ist und erzeugt ein entsprechendes Token (Ziel: B, Quelle: S), welches an die nachfolgende Operation B gerichtet ist (6.). Die Aktivität B verarbeitet dieses, wird ausgeführt und sendet danach ein Token zurück (8.), welches von S „empfangen" wird (9.). Die Aktivität S erkennt nun, dass B die letzte Aktivität der Sequenz und benachrichtigt somit seinen eigenen Vater X, dass sie beendet ist, was nicht mehr abgebildet ist.

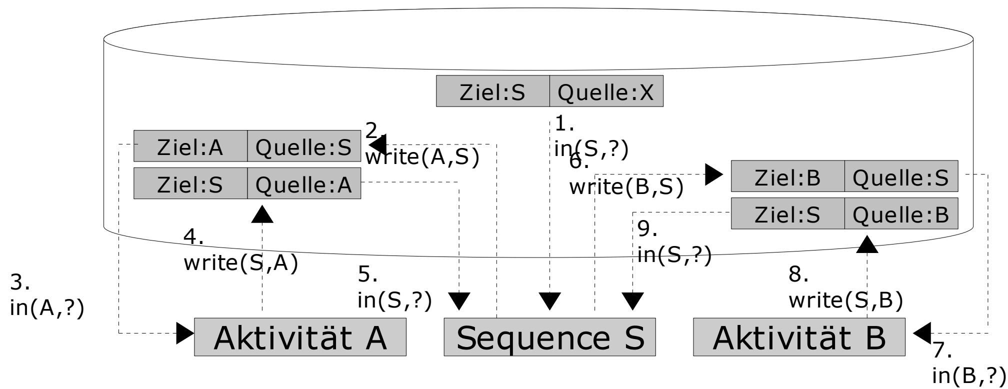



*Abbildung 4.7: Beispiel für die Verwendung von Absender-Informationen bei strukturierten Aktivitäten*

Damit sind bis auf History-Feld alle Teile eines Kontrollflusstokens (Tabelle 4.2) erläutert. Dieses Feld stellt für eine Aktivität den Prozesszustand, bestehend aus einer Menge aktiver Scopes, dar [Pop08]. Es handelt sich dabei um ein Tupel, welches die Bezeichner der Scope-Instanzen beinhaltet, innerhalb denen das Token erzeugt wurde. Beim Starten einer neuen Scope-Instanz, fügt diese ihre Laufzeit-Id an das bisherige History-Feld an. Wird diese wieder beendet, entfernt sie diesen Bezeichner aus dem Feld. Damit ist das History-Feld eine Liste, welche die Reihenfolge der Scope-Instanzen wiedergibt.

In Abbildung 4.8 ist an einem Beispiel die Entwicklung des History-Feldes dargestellt. Dabei besteht ein Prozess aus einer While-Aktivität, welche zweimal durchlaufen wird. Diese enthält einen Scope A, welcher wiederum eine Aktivität X beinhaltet. Auf oberster Ebene gilt der Prozess selbst als ein Scope. Dieser besitzt bei der Ausführung die Instanz-Id „P1“. Beim ersten Durchlauf der While-Schleife ruft diese den Scope A auf, wobei die History innerhalb des zum Aufruf verwendeten Tokens bisher nur aus P1 besteht. Die Scope Aktivität wird dadurch ausgeführt und ruft die Aktivität X auf, wobei sie an das History-Feld seine eigene Instanz „A1“ hinzufügt. Somit wird dieses nun das Tupel (P1, A1) dargestellt. Nach Beendigung der inneren Aktivität ist auch diese Scope-Instanz, also „A1“ beendet, weshalb die Scope-Aktivität wieder ihre Elternaktivität, also das While aufruft und dabei die Instanz A1 aus dem History-Feld des entsprechenden Tokens entfernt. Bei der zweiten Ausführung der Schleife wird dann wieder eine neue Scope-Instanz erzeugt, welche beim Aufruf der Aktivität X das History-Feld um Instanz A2 erweitert. Nach Beendigung wird der Kontrollfluss wieder zurückgeben. Da die While-Aktivität nur zweimal ausgeführt wird, geben die Aktivitäten dieses immer an ihre Elternaktivität zurück bis die Prozess-„Aktivität“ erreicht wird und somit die Ausführung dieser Prozessinstanz beendet ist

Während dieses Ablaufes wurde die Aktivität X nun zweimal aufgerufen, allerdings innerhalb verschiedener Instanzen des Scopes, in dem sie sich befindet. Die Aktivität besitzt durch das jeweils verwendete History-Feld Informationen über den Prozesszustand, also der Reihenfolge der Scope-Instanzen, in welchen sie sich momentan befindet. Diese Daten kann eine Aktivität zum Beispiel nutzen um auf eine Variablen-Instanz zuzugreifen, welche innerhalb des Scopes A definiert und zu ihrer Prozessinstanz gehört.

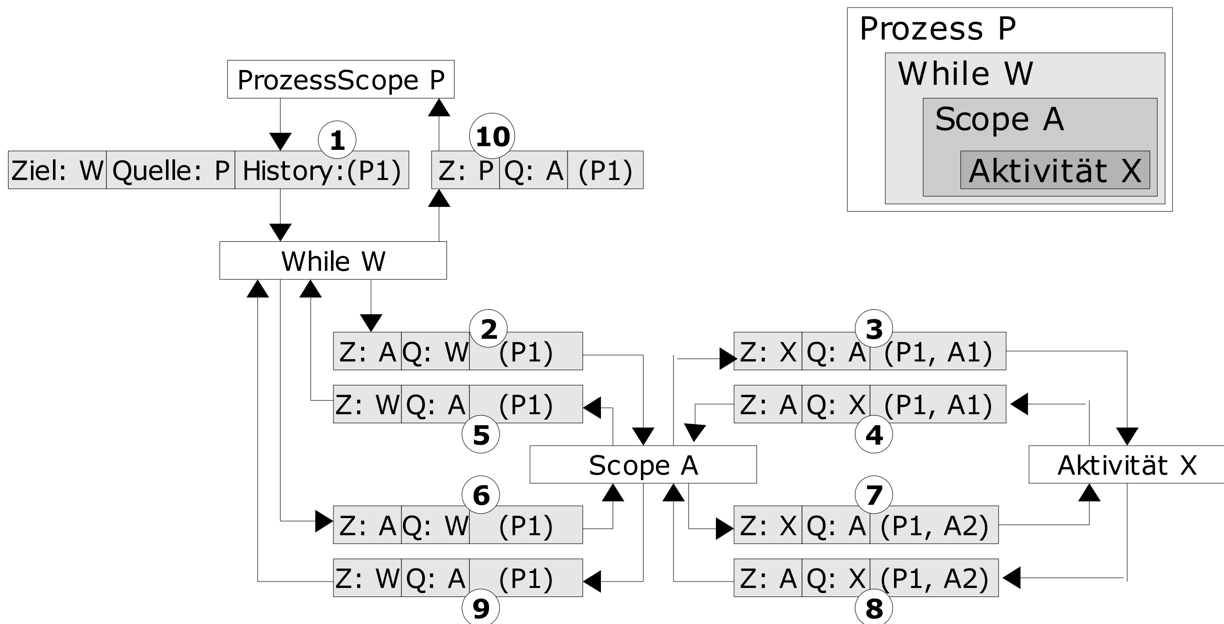



*Abbildung 4.8: Beispiel für Entwicklung des History-Feldes der Kontrollflusstoken*

### 4.4.2 Datentoken

Mit Hilfe der Datentoken werden verschiedene Aspekte einer BPEL-Prozessinstanz gespeichert und den Aktivitäten zur Verfügung gestellt. Dies sind Variablen, Correlation Sets und Partner Links, aber auch Nachrichten und Fehlermeldungen. Zudem werden Links als Daten modelliert. Im Folgenden werden die verschiedenen Arten der Datentoken für die Prozesselemente erläutert, wobei natürlich noch die oben definierten Header-Felder (z.B. Prozess-Id, Instanz-Id) hinzukommen.

#### Variablen

Datentoken, welche die Instanz einer Variablen darstellen, müssen alle Informationen bereitstellen, die den Zustand einer Variablen enthält und zusätzlich noch den Aktivitäten einen Zugriff darauf ermöglichen. In Tabelle 4.3 sind die entsprechenden Felder angegeben.

| Feldname | Datentyp | Beschreibung |
|---|---|---|
| Variable-Id | Integer | Bezeichner der Variable im Prozessmodell |
| Scope-Instanz-Id | Integer | „Laufzeit-ID“ des Scopes, der das Token der Variablen erzeugt hat. |
| Wert | String | Gibt den tatsächlichen Inhalt einer Variablen wieder. |

*Tabelle 4.3: Felder des Tokens einer Variablen*

Ein Variablentoken speichert den Wert einer Variablen bei Ausführung einer Prozessinstanz. Aktivitäten greifen auf diese Daten lesend oder manipulierend zu, wozu sie die benötigten Variablentoken korrekt identifizieren müssen. Eine Aktivität und eine Variable, auf die sie zugreift, gehören natürlich immer zum selben Prozessmodell und zur selben Prozesssinstanz. Damit werden diese beiden Informationen auf jeden Fall zu Identifizierung einer Dateninstanz benötigt. Dies genügt aber noch, da sich in einem Prozessmodell in der Regel mehrere Variablen befinden und eine Aktivität genau eine ansprechen möchte.

Um eine Variable zu bestimmen wird das Feld Variable-Id verwendet. Dieser Bezeichner entspricht der ID, die der Variablen im Deployment Descriptor vergeben wurde (vgl. Abschnitt 4.3). Er identifiziert also ein Variablen-Element innerhalb eines Prozessmodells. Dieser soll einen effizienten Zugriff auf die Variable ermöglichen.

In Abbildung 4.9 ist der Verwendungszweck dieser Variablen-Id an einem Beispiel dargestellt. In mehreren verschachtelten Scopes wird eine Variable x definiert. Nun verwendet die invoke-Aktivität eine Variable x. Befinden sich die Token für beide Variablen auf demselben Tuple Space, welcher entsprechend derjenige ist, der für die Aktivität zum Lesen des Token konfiguriert wurde, kann ein Lesen nur über den Namen nicht erfolgen. Dieses würde unter Umständen das falsche Variablen Token für die Aktivität lesen. Um dies zu Umgehen wurde jeder Variable ein eindeutiger Be-

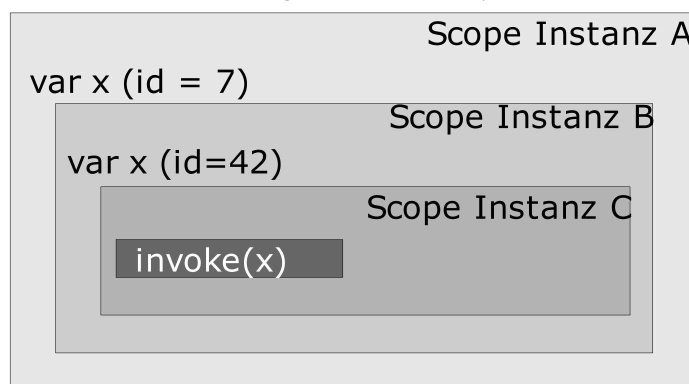



*Abbildung 4.9: Beispiel zur Verwendung eines Identifiers bei Datentoken*

zeichner zugewiesen. Diesen kann eine Aktivität nun verwenden, um auf die korrekte Variable zuzugreifen. In dem obigen Beispiel verwendet die invoke-Aktivität zum Lesen des Variablentokens ein Template, welches seine Prozessinstanz, das Prozessmodell und den Bezeichner der Variablen festlegt.

Allerdings stellt sich heraus, dass dieser Ansatz ebenfalls nicht zum gewünschten Erfolg führt. In Abbildung 4.10 ist der Fall, dargestellt, dass ein Scope S wiederholt ausgeführt wird. Beim jedem Durchlauf der Schleife erzeugt der Scope S eine Variablentoken mit den Feldern (Prozessinstanz_Id =1, Variablen_Id=15). Da der Zustand einer Variablen auch nach Beendigung des Scopes noch gespeichert wird (für das Compensation Handling) existieren für die Variable innerhalb der Prozessinstanz also mehrere Token, welche jeweils in einer anderen Instanz des Scopes erzeugt wurden.

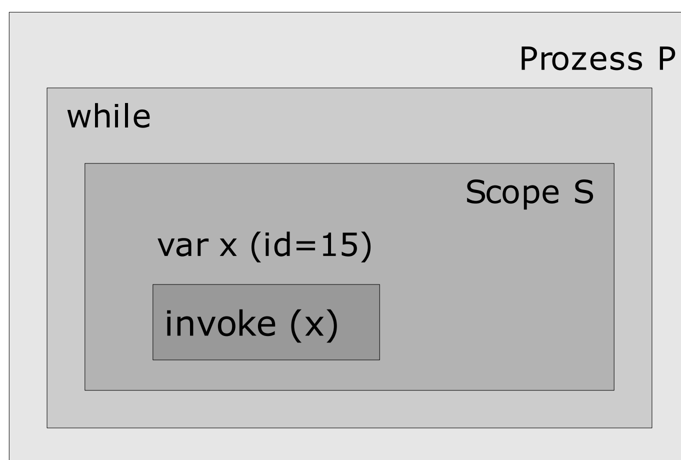



*Abbildung 4.10: Beispiel für die wiederholte Ausführung von Scopes*

Aus diesem Grund beinhaltet ein Variablentoken noch das Feld Scope-Id, welches angibt, in welcher Scope-Instanz es erzeugt wurde. Im obigen Beispiel würde so beim ersten Durchlauf der Schleife ein Token mit Scope-Instanz 15 erzeugt, während das Token beim zweiten Durchlauf die Instanz 19 besitzt. Die invoke-Aktivität innerhalb des zweiten Durchlaufs könnte also über ein Template, welches nur den Bezeichner der Variablen und die aktuelle Prozessinstanz beinhaltet, sowohl auf das Token der Scope-Instanz 15, als auch das der Instanz 19 zugreifen, obwohl für sie die Instanz 19 von Bedeutung ist. Um nun das richtige Token zu lesen, verwendet die Aktivität das History-Feld, mit dem sie aufgerufen wurden. Dieses kann sie sequentiell von hinten durchgehen und jeweils die entsprechende Scope-Instanz dem Template hinzufügen um die Variable zu finden. Im hier erläuterten Beispiel würde dies sofort bei der ersten Anfrage das korrekte Token liefern, da die Variable innerhalb des direkt umschließenden Scopes der Aktivität definiert ist. In dem Beispiel aus Abbildung 4.9 müsste wird das invoke mit einem History (A, B, C) aufgerufen. Eine erste Anfrage mit Sco-

pe=C bliebe dabei ohne Erfolg, erst der zweite Versuch mit Scope=B bringt das gewünschte Ergebnis.

Die kann bei tief verschachtelten Scopes unter Umständen zu sehr vielen sequentiellen Lesezugriffen auf einen Tuple Space führen, die ohne Erfolg bleiben. Ein Ansatz das Lesen einer Variable zu optimieren und auf das sequentielle Durchlaufen des History-Feldes zu verzichten, war es, ein weiteres Feld (active) zum Token hinzuzufügen. Der Scope, der die Variable initialisiert, setzt dieses auf true und bei Beendigung auf false. Eine Aktivität, die auf die Variable zugreift, fügt dann beim entsprechenden Lese-Template noch die Bedingung active=true an. In dem oben genannten Beispiel aus 4.10 wäre damit das Problem gelöst. Bei der Ausführung der invoke-Aktivität im zweiten Durchlauf (Scope-Instanz 19), ist nur für dieses Token das Feld active auf true gesetzt. Das Token welches zum ersten Durchlauf der Schleife, also der Scope-Instanz 15, gehört ist nach Beendigung dieses Scopes auf false gesetzt.

Allerdings funktioniert auch dieser Ansatz nicht wie gewünscht. Enthält ein ForEach-Konstrukt Scopes, so können mehrere Instanzen dieser Scopes dieser gleichzeitig aktiv sein. Damit reicht auch die Kombination (instance, variable, active) nicht aus. Deswegen wird auf einer Optimierung des Variablen-Zugriffes verzichtet und eine Aktivität verwendet zum Lesen eines entsprechenden Variablen-Token das History-Feld und den Bezeichner der Variablen.

Im Feld Wert befindet sich nun der tatsächliche Zustand der Variablen. Der Datentyp ist dabei von dem definierten Datentyp der BPEL-Variable abhängig. Im Token wird der Zustand, also der Wert der Variablen, als String repräsentiert. Dieser kann dabei je nach definiertem Datentyp beispielsweise die String-Darstellung eines Integers sein oder auch ein XML-String, der einen Message-Type repräsentiert.

Ähnlich zu den Variablen sind auch die Token, welche die Instanzen der weiteren Datenelement, also Partner Links, Correlation Sets und Links repräsentieren aus. Deren Besprechung erfolgt daher nicht so ausführlich, wie bei der Tokenstruktur der Variablen.

**Correlation Sets**

Mit Hilfe von Correlation Sets können Prozesses, mit denen mehrere Nachrichten ausgetauscht werden, eindeutig identifiziert werden. Die Felder, welche die entsprechenden Token besitzen, sind in Tabelle 4.4 angegeben.

| Feldname | Datentyp | Beschreibung |
|---|---|---|
| CorrelationSet-Id | Integer | Bezeichner des Corelation Sets im Prozessmodell |
| Scope-Instanz-Id | Integer | „Laufzeit-ID“ des Scopes, der das Token der Variablen erzeugt hat. |
| initiated | boolean | Gibt an, ob die Instanz des Correlation Set schon initialisiert ist |
| (PropertyNameSpace, Property-Name, Property-Wert)+ | (String, String, String)+ | Die Werte, die die Properties des Correlation Sets an |

*Tabelle 4.4: Felder des Tokens eines Correlation Sets*

Der Zugriff auf die Instanz eines Correlation Sets durch eine Aktivität erfolgt analog zur Verwendung von Variablentoken. Die Adressierung eines bestimmten Tokens erfolgt durch dessen Bezeichner aus dem Prozessmodell und der Laufzeit-Id des Scopes, in welchem die Instanz des Correlation Sets erzeugt wurde. Dabei wird der Zugriff auf ein bestimmtes Token durch eine Aktivität wiederum durch das sequentielle Durchsuchen des History-Feldes realisiert.

Ein Correlation Set wird in BPEL immer innerhalb eines Scopes definiert. Bei der Initialisierung des Scopes erzeugt die Aktivität, die den Scope implementiert, dann ein Token, welches die Instanz des Correlation Sets repräsentiert. Dazu setzt sie entsprechend die Instanz-Id des Scopes. Weiterhin setzt sie das Feld `initiated` auf false, da ein erzeugtes Correlation Set beim Start des umgebenden Scopes immer als nicht initialisiert gilt [OAS07]. Es kann durch Aktivitäten initialisiert, bei denen das BPEL-Attribut `initiate` auf „true“ bzw. „false“ gesetzt ist.

Die Werte des Correlation Sets wurden durch eine solche initialisierende Aktivität gesetzt und dürfen danach nicht mehr verändert werden. Diese Aktivitäten kennen die Property-Definitionen des BPEL-Prozess bzw. der zugehörigen WSDL-Beschreibungen dadurch, dass laut Annahme jede Aktivität Zugriff auf den kompletten Prozess besitzt [Pop08]. Da sich die Property-Definitionen nicht innerhalb des Prozesses selbst befinden, sondern in einer WSDL-Beschreibung, werden diese nicht innerhalb des Deployment Descriptors angegeben. Die konkreten Instanzdaten werden aber in den Token gespeichert, indem eine Liste aus Tupeln der Form (Name Space, Name, Wert) gespeichert wird. Mit Name Space und Name wird dabei eine bestimmte Property Definition über ihren qualifizierten Namen angesprochen. Mit Wert wird dann die konkrete Ausprägung dieser Property, also deren Wert zur Laufzeit der Instanz, ange-

geben. Dies ist abhängig von Datentyp, den die Property beschreibt. Die Token beinhalten, analog zu den Variablen, immer eine Repräsentation des Wertes als String.

**Partner Links**

Analog zu Variablen und Correlation Sets sind auch die Datentoken, welche Partner Links repräsentieren, aufgebaut.

| Feldname | Datentyp | Beschreibung |
|---|---|---|
| PartnerLink-Id | Integer | Bezeichner des Links im Prozessmodell |
| Scope-Id | Integer | „Laufzeit-Id“ des Scopes, der den Partner Link erzeugt hat |
| Wert | String | Konkreter Wert des Partner Links zur Ausführungszeit |

*Tabelle 4.5: Felder des Tokens eines Partner Links*

Dies Felder besitzen dabei, dieselben Bedeutungen, wie zuvor bei Variablen und Corrlation Sets erläutert.

**Links**

Links dienen zur Flow-basierten Modellierung des Kontrolflusses. Ihre Bedeutung im Zusammenhang mit der Realisierung des Kontrollfluss durch den Tokenaustausch wurde zuvor schon erläutert. Der Aufbau eines Tokens, welcher die Instanz eines Links widerspiegelt, ist in Tabelle 4.4 dargestellt.

| Feldname | Datentyp | Beschreibung |
|---|---|---|
| Link-Id | Integer | Bezeichner des Links im Prozessmodell |
| LinkStatus | {unset, true, false} | Der Wert, den die Transition Condition des Links besitzt |
| Scope-Id | Integer | „Laufzeit-ID“ des Scopes, indem sich der Flow befindet, welcher diesen Link erzeugt. |

*Tabelle 4.6: Felder des Tokens eines Links*

Das Feld Link-Id wird wie bei allen Datentoken zum Zugriff auf ein spezielles Token verwendet. Aktivitäten, welche das Token eines Links lesen wollen, kennen dabei den Bezeichner des Links sowie die entsprechende Prozessinstanz in der sie sich befinden.

Diese Informationen genügen auch bei den Links nicht, da hier derselbe Fall auftreten kann, wie bei den Variablen beschrieben, nämlich dass durch eine `ForEach`-Aktivität mehrere Token für dieselbe Prozessinstanz eines Links existieren. Bei Variablen, Correlation Sets und Partner Links, lässt sich dies lösen, indem das entsprechende Token, die Instanz des Scopes enthält, in welchem die Daten definiert wurden. Eine Aktivität, welche kann zum Zugriff aud die Daten sein eigenes History-Feld mit diesen Scope-Instanzen vergleichen und somit das korrekten Token ansprechen.

Die vorkommenden Links werden allerdings nicht durch einen Scope definiert, sondern durch eine Flow-Aktivität. Da Flows zur Laufzeit keine Instanz-Id besitzen, bzw. diese nicht an Kind-Elemente übergeben wird, müssen die Links anders eindeutig bestimmt werden. Dazu wird wieder derselbe Ansatz der Scope-Instanz übernommen. Eine Flow-Aktivität generiert die entsprechenden Token der Links und setzt als Scope-Id, den letzten Wert des History-Feldes aus dem Token, mit welchem die Flow-Aktivität selbst aufgerufen wurde. Da alle Aktivitäten, welche auf die Links benutzen, sich entweder innerhalb derselben Schachtelungsebene des Flows befinden oder tiefer, beinhaltet deren History-Feld auf jeden Fall den Wert des Scopes, welcher die Links definiert hat.

**Fehlertoken**

Aktivitäten können Fehler werfen, welche von entsprechenden Fault Handlern aufgefangen werden. Diese müssen ebenfalls als Token repräsentiert werden um zwischen einer Aktivität und einem Handler ausgetauscht werden zu können.

| Feldname | Datentyp | Beschreibung |
|---|---|---|
| Fehlername | String | Vollständig qualifizierter Name des Fehlers |
| Scope-Id | Integer | „Laufzeit-ID“ des Scopes, indem der Fehler aufgetreten ist. |
| Fehlerdaten | String | Inhalt der zugehörigen Fehlervariablen |

*Tabelle 4.7: Felder des Tokens einer Fehlermeldungen*

Ein Fehler muss immer über einen qualifizierten Namen identifizierbar sein. Da dieser nicht unbedingt innerhalb des BPEL-Prozesses definiert sein muss, kann kein vergebener Bezeichner verwendet werden, sondern die Token müssen diesen Namen beinhalten. Mit dem Feld Scope-Id wird angegeben innerhalb welches Scopes der Fehler aufgetreten ist. Dazu muss die Aktivität, welche den Fehler erzeugt, das letzte Element seiner History verwenden. Dieses gibt den Scope an in welchen sich die Aktivität befindet. Das Feld wird dazu benötigt, dass ein entsprechender Fault Handler,

nämlich, der dieser Scope-Instanz, die Fehlermeldung verarbeiten kann. Zusätzlich kann zu einem Fehler noch eine Variable zugeordnet werden, welche ihn näher beschreibt bzw. zusätzliche Daten dazu beinhaltet. Der Inhalt dieser Variablen wird im Feld Fehlerdaten als String-Repräsentation bereitgestellt.

#### Nachrichtentoken

Unter Nachrichtentoken versteht man die Token, welche die empfangenen Nachrichten eines Prozesses wiedergeben. Der Aufbau eine solchen Tokens ist in Tabelle 4.8 angegeben.

| Feldname | Datentyp | Beschreibung |
|---|---|---|
| PartnerLink-Id | Integer | Bezeichner eines Partner Links |
| Operation | String | Name einer WSDL-Operation |
| Nachrichtendaten | String | Darstellung des Inhalts (Body) der Nachricht |

*Tabelle 4.8: Felder des Tokens einer empfangenen Nachricht*

Nachrichtentoken dienen der Verarbeitung von empfangenen Nachrichten durch receive Aktivitäten. Ein receive wird dabei in BPEL durch den Partner Link und die Operation beschrieben, über welche die Nachricht empfangen wurde. Wartet eine receive-Aktivität auf eine Nachricht, so liest sie auf dem Tuple Space an dem sie die Nachricht erwartet, nach entsprechenden Nachrichtentoken, durch Angabe des Partner Links und der Operation. Das Feld Nachrichtendaten enthält die eigentliche Information der Nachricht. Da der Austausch durch SOAP-Nachrichten erfolgt, ist dies der Body der empfangenen SOAP-Message.

# 5 Implementierung

Die Implementierung realisiert die im vorherigen Kapitel beschriebenen Konzepte zur verteilten Ausführung. Dabei baut sie auf Apache ODE auf und verwendet einige Aspekte davon weiter, wie beispielsweise die Kommunikation mit externen Web Services. Neu umgesetzt wird allerdings die Navigation und der Datenaustausch zwischen den Aktivitäten, welcher durch den Tokenaustausch-Mechanismus abläuft. Dazu müssen die Aktivitäten von Apache ODE umgeschrieben werden, damit sie ihre Daten aus einem Tuple Space erhalten können und der Kontrollfluss nicht mehr durch das JACOB Framework realisiert wird. Dagegen werden die BPEL-Logiken der Aktivitäten, wenn möglich, beibehalten und nicht neu implementiert.

In diesem Kapitel wird die Implementierung beschrieben. Dabei wird die Architektur der entwickelten Lösung im Zusammenhang mit der Architektur von ODE erläutert und speziell auf die Struktur der Aktivitäten, sowie deren Schnittstellen zu den Tuple Spaces eingegangen.

## 5.1 Grundstruktur

Wie erläutert, sind die beschriebenen Aktivitäten prinzipiell nur Clients auf bestimmten Tuple Spaces. Sie müssen durch Lesen mittels bestimmter Templates zunächst auf Token warten, welche ihre Startbedingung erfüllen. Sind diese vorhanden, kann die BPEL-Logik der Aktivität durchgeführt werden, wobei sie die benötigten Instanzdaten ebenfalls noch von Tuple Spaces liest. Nach der Ausführung erzeugt eine Aktivität wiederum die beschriebenen Token um den Kontrollfluss an die nachfolgenden Aktivitäten weiterzugeben.

Grundsätzlich ergeben sich daraus zwei Möglichkeiten, wie die Aktivitätsimplementierungen umgesetzt werden können, um damit BPEL-Prozesse zu unterstützen. Zum Einen wäre es möglich, alle BPEL-Aktivitäten jeweils als alleinstehende, für den Einsatz innerhalb eines Prozessmodells konfigurierbare Anwendung zu realisieren. Zum Anderen könnten sie in einer Engine zusammengefasst werden, welche bestimmte

Funktionalitäten bündelt und die das Deployment von BPEL-Prozessen bzw. Teilprozessen ermöglicht.

Die Möglichkeit Aktivitäten als alleinstehende Anwendungen zu realisieren, bedingt, dass es für jede BPEL-Aktivität eine eigene Implementierung gibt, welche zunächst einmal völlig unabhängig von anderen Aktivitäten ist. Diese müssen so konfigurierbar sein, dass sie für den Einsatz innerhalb eines BPEL-Prozessmodelles verwendet werden können. Das heißt, sie müssen über die Konfigurationsdaten (vgl. Abschnitt 4.3) so eingerichtet werden, dass sie als Clients auf bestimmten Tuple Spaces agieren und dabei über genügend Informationen besitzen um über Templates diese Token zu lesen, welche für sie von Bedeutung sind. Dieser Ansatz hat zur Folge, dass ein BPEL-Prozess durch mehrere einzelne Anwendungen umgesetzt wird, nämlich denen, die für diesen Prozess entsprechend konfiguriert sind. Dabei muss für jede innerhalb der BPEL-Prozessdefinition vorkommende Aktivität eine eigene Anwendung existieren. Diese können auf verschiedenen Maschinen laufen und der Ansatz erfüllt damit die Anforderungen und Grundzüge des im Rahmen dieser Arbeit vorgestellten Konzeptes der verteilten Ausführung. Der Nachteil ist allerdings, dass ein BPEL-Prozess in der Regel aus recht vielen Aktivitäten aufgebaut ist. Die Anwendungen, welche diese realisieren, müssten nun alle einzeln konfiguriert werden. Zudem wird der Prozess meist nicht so weit vollständig verteilt, dass alle Aktivitäten auf verschiedenen Maschinen laufen, obwohl dies prinzipiell natürlich möglich ist. Auf einem Server läuft häufig ein Teilprozess, bestehen aus mehreren Aktivitäten. Wenn diese jeweils als eigenständige Anwendungen realisiert sind, bedeutet dies, dass bei einem Neustart des Servers auch sämtliche Aktivitäts-Anwendungen einzeln neu gestartet werden müssen.

Aus diesen Gründen wird diese Variante nicht verwendet, sondern die Aktivitätsimplementierungen in einer Anwendungen gebündelt. Dabei handelt es sich um die spezielle Art einer BPEL-Laufzeitumgebung (BPEL-Engine). Allerdings führt diese, anders zu den herkömmlichen WfMS nicht den gesamten Prozess aus, sondern nur Teile davon, nämlich die Aktivitäten, welche darauf laufen sollen. Es wird also prinzipiell nicht der gesamte Prozess in dieser Umgebung ausgeführt, sondern lediglich bestimmte Aktivitäten davon. Das hat zur Folge, dass die Navigationskomponente dieser BPEL-Engine entfällt und der Kontrollfluss wie beschrieben durch Tokenaustausch mit anderen Aktivitäten erfolgt. Diese können wiederum innerhalb der selben Engine laufen oder aber in anderen Instanzen der Laufzeitumgebung, also zum Beispiel auf anderen Maschinen. Die Steuerung des Kontrollflusses zwischen Aktivitäten, welche innerhalb derselben Engine laufen und dadurch direkt miteinander verbunden sind, könnte dabei auch die traditionelle Vorgehensweise durchgeführt werden. Damit bei der Implementierung der Aktivitäten aber nicht zwischen internen

und externen Kontrollfluss unterschieden werden muss, was einen weiteren Zusatzaufwand mit sich bringen würde, wird auch der interne Kontrollfluss über den definierten Tokenaustausch realisiert. Also auch Aktivitäten innerhalb derselben Laufzeitumgebung sind miteinander nur über Tuple Spaces miteinander verbunden und es gibt keine zentrale Navigationskomponente auf dieser Engine. Dieser Ansatz der Bündelung von Aktivitäten hat gegenüber eigenständigen Anwendungen den Vorteil, dass nicht alle Aktivitäten einzeln konfiguriert werden müssen, sondern dies über einen Deployment Descriptor erfolgen kann, der einen Prozess auf verschiedene Instanzen der Engine aufteilt und die benötigten Austauschmechanismen, also Tuple Spaces als Verbindung zwischen Aktivitäten, festlegt. Dieser wird der BPEL-Engine zusammen mit der BPEL-Prozessdefinition übergeben, wodurch die entsprechenden Aktivitäten darauf eingebracht werden und ausgeführt werden können.

## 5.2 Architektur

Die entwickelte Laufzeit-Umgebung basiert auf Apache ODE und übernimmt davon noch einige Aspekte. So wird beispielsweise der Integration Layer weiterhin genutzt um mit externen Web Services zu interagieren. Ebenso wird der Compiler und das von ihm erzeugte Objektmodell beibehalten bzw. erweitert. Die grobe Architektur ist in Anlehnung zu der vorgestellten Architektur von Apache Ode (vgl. Abb. 3.2) in der Abbildung 5.1 dargestellt.

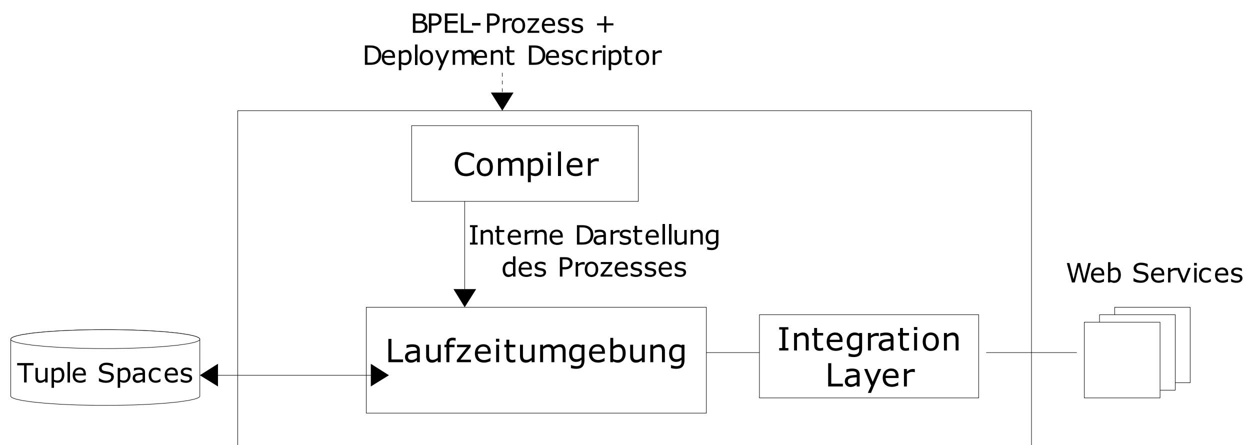



*Abbildung 5.1: Architektur der Lösung im Vergleich zu Apache ODE*

Der Compiler überführt den BPEL-Prozess in eine intern verwendete Darstellung. Zusätzlich kommen noch die Angaben des Deployments Descriptors, die unter anderem wiedergeben, welche Aktivitäten des Prozesses durch die Laufzeitumgebung unterstützt werden sollen. Zu beachten ist, dass die Aktivitäten, welche auf dieser Engine laufen, einen Zugriff auf diese interne Repräsentation des kompletten Prozesses

besitzen und nicht nur auf die Teilaktivitäten, welche durch das Deployment hier eingebracht sind. Dies ist im Einklang mit den EWFNs, die annehmen, dass alle Aktivitäten Zugriff haben auf die XML-Darstellung des BPEL-Prozesses [Pop08]. Allerdings steht diese nicht als Token zur Verfügung, sondern eben durch das Objektmodell, welche eine Darstellung der XML-Form des Prozesses liefert.

### 5.2.1 Laufzeitumgebung

Die Laufzeitumgebung realisiert die Ausführung der Aktivitäten eines Prozessmodells, die durch das Deployment auf dieser Engine zur Verfügung gestellt sind. Zudem übernimmt sie von Apache ODE die Möglichkeiten zum Empfangen von Nachrichten über den Integration Layer.

Die zentrale Aufgabe der Laufzeitumgebung ist es die Ausführung der Aktivitäten zu gewährleisten, wozu sie im Gegensatz zu herkömmlichen WfMS auf eine zentrale Steuerungskomponente verzichten sollte. Stattdessen bestand das Ziel bei der Entwicklung daraus, diesen zu den Aktivitäten zu verschieben.

Da der Kontrollfluss über Token realisiert wird, muss eine Funktionalität vorhanden sein, durch welche die einzelnen Aktivitäten gestartet werden können, wenn entsprechende Token vorhanden sind. In einem ersten Ansatz wurden alle Aktivitäten, die auf der Engine laufen sollen, in einer zyklischen Liste gespeichert. Die Aufgabe der Laufzeitumgebung besteht bei dieser Vorgehensweise darin, ständig diese Liste zu durchlaufen und nacheinander für jede Aktivität überprüfen, ob entsprechende Token vorhanden sind, die deren Startbedingung erfüllen. Dazu stellt sie jeweils eine Leseanfrage an die entsprechenden Tuple-Spaces durch Templates, welche die benötigten Token der Aktivität beschreiben. Gibt diese kein Ergebnis zurück, wird eine neue Anfrage gestartet, diesmal für die Token, die die nächste Aktivität der Liste benötigt. Werden für eine Aktivitäten alle Token deren Startbedingung gelesen, wird daraufhin ein neuer Thread gestartet, welcher die Implementierung dieser Aktivität ist und dessen Logik für eine konkrete Prozessinstanz durchführt. Dies hat den Vorteil, dass zu jedem Zeitpunkt nur Threads von solchen Aktivitäten aktiv sind, welche tatsächlich für eine Prozessinstanz gestartet sind und momentan ausgeführt werden. Der Nachteil ist natürlich das zyklische Abfragen nach dem Vorhandensein von Token, welche die Eingangsbedingungen für die Aktivitäten sind.

Aus diesen Grund wurde dieser Ansatz wieder verworfen und jede Aktivität durch einen eigenen Thread realisiert. Ein solcher Aktivitätsthread liest zunächst einmal blockierend auf den Tuple Spaces, an denen sich die Token für seine Startbedingung befinden können. Befinden sich alle erwarteten Token dort, geht die Aktivität dazu über die entsprechende Logik durchzuführen und nach deren Beendigung Kontroll-

flusstoken für die Nachfolger zu erzeugen. Danach wartet der Thread wieder auf die Token, welche seine Startbedingung bilden. Einzelheiten dazu folgen im nächsten Abschnitt. Die Aufgabe der Laufzeit besteht nun nicht mehr darin ständig nach eingehenden Startbedingungen für die verschiedenen Aktivitäten zu warten. Stattdessen muss sie lediglich für alle Aktivitäten entsprechende Threads erzeugen und verwalten.

Der Vorteil im Gegensatz zum Ansatz mit der verketteten Liste liegt darin, dass eine Aktivitätsinstanz sofort dann gestartet wird, wenn die Token, welche die Startbedingung darstellen, vorliegen. Das blockierende Lesen der Aktivitäts-Threads ist nämlich genau dann beendet, wenn die erwarteten Token in den Tuple Spaces von den Vorgängern erzeugt sind. Durch den jeweils eigenen Thread warten die Aktivitäten alle gleichzeitig auf ihre Eingangstoken und können somit sofort gestartet werden. Im Falle der verketteten Liste lesen die Aktivitäten nacheinander – und nicht parallel – ob Token vorhanden sind. Dies hat zur Folge das eine Aktivitätsinstanz erst dann gestartet werden kann, wenn diese Aktivität beim zyklischen Überprüfen auf Eingangstoken an der Reihe ist. Befinden sich viele Aktivitäten in der Liste, kann dies unter Umständen länger dauern bis für eine Aktivität die Überprüfung der Startbedingung erfolgt. Allerdings sind hierbei nur „aktive“ Threads für die Aktivitäten vorhanden, also solche, die eine konkrete Instanz realisieren und keine, welche durch das Lesen blockiert sind.

Eine weitere Aufgabe der Laufzeit ist das Empfangen von Nachrichten und deren Verarbeitung. Dies wird mit Hilfe der von Apache ODE zur Verfügung gestellten Funktionalität realisiert. Wird eine Nachricht empfangen (über ODE-Funktionalität des Integration Layers), dann wird diese in ein Nachrichtentoken überführt. Das Nachrichtentoken wird von einem zuständigen receive im Rahmen der Prozessausführung verarbeitet.

## 5.3 Aktivitätsimplementierung

Mehrfach wurde eine Aktivität schon als Tuple Space Client definiert. Bei der Beschreibung der Daten, welche nötig sind, um eine Aktivität auf eine Tuple Space Infrastruktur abzubilden, wurde schon die verschiedenen Aspekte betrachtet, welche durch eine Aktivitätsimplementierung realisiert werden müssen (vgl. Abb. 4.3). Diese unterscheiden sich in Kontrollfluss und Aktivitätslogik. Wobei der Kontrollfluss einer Aktivität wiederum in Aspekte des eingehenden und des ausgehenden Kontrollflusses unterteilt werden kann. Für die Aktivitätslogik wird zudem noch ein Zugriff auf Daten benötigt. Diese Punkte stellen jeweils Lese- oder Schreibeoperationen auf Tuple Spaces dar.

Zuvor wurde schon vorgestellt, dass jede Aktivität die in der Laufzeitumgebung ausgeführt wird, prinzipiell einen eigenen Thread darstellt. Dieser besteht jeweils aus einem Teil, der auf eingehende Token wartet und einem Teil, der nach dem Empfang der Token die eigentliche Logik der Aktivität durchführt. Dabei ist allerdings zwischen einfachen Aktivitäten und strukturierten Aktivitäten zu unterscheiden, da die strukturierten Aktivitäten nicht nur aus einem einzigen Thread bestehen, sondern noch aus einem zweiten. Der Grund dafür wird nachfolgend beschrieben.

### 5.3.1 Aufbau einfacher Aktivitäten

Für die einfachen Aktivitäten genügt es, den eingehenden und ausführenden Kontrollfluss sowie die BPEL-Logik dieser Aktivität zu betrachten. Das heißt, die Aktivität wird durch eingehende Token gestartet, führt seine Logik im Sinne der BPEL-Semantik dieser Aktivität aus und erzeugt danach Token, welche für die folgenden Aktivitäten bestimmt sind. Zudem können sie noch Instanz-Daten benötigten, welche durch Token repräsentiert ist und entsprechend gelesen werden müssen. Der schematische Aufbau der Implementierung eines Threads, welcher eine einfache Basis-Aktivität realisiert, ist in Abbildung 5.2 dargestellt.

*Abbildung 5.2: Schema einer Aktivitäts-Implmentierung*

Der „Template Matching“-Abschnitt realisiert das Warten auf eingehende Kontrollflusstoken. Dazu verwendet er eine Menge von Templates, die aus den Konfigurationsdaten erzeugt werden. Für jedes eingehende Token existiert dabei jeweils ein Template, welches durch die Zielaktivität, den Absender sowie das Prozessmodell ein bestimmtes Token bestimmt. Besteht die Startbedingung einer Aktivität aus mehreren Token, so müssen diese Templates zusammengefasst werden, um sicherzustellen, dass die gelesenen Token auch wirklich zusammengehören, also alle aus derselben Prozessinstanz stammen. Dazu kann, wie in Abschnitt 4.4.1 erläutert, die sync-Leseoperation des verwendeten Tuple Spaces genutzt werden. Allerdings funktioniert diese nur wenn sich die Token auf demselben Tuple Space befinden, sonst muss das Syn-

chronisieren des Lesens vom „Template Matching“-Abschnitt der Aktivität selbst realisiert werden.

Sind die Token gefunden, wird die Logik der Aktivität ausgeführt, wozu zunächst einmal die eingehenden Links überprüft werden. Die Ausführung der Logik erfolgt in der Regel innerhalb desselben Threads wie das Template-Matching. Dies bedeutet, dass sobald die Eingangsbedingung erfüllt ist, führt der Aktivitäts-Thread die BPEL-Logik für die Prozessinstanz aus, welche durch die eingegangenen Token bestimmt wurd. Dies ist gleichzusetzen mit einem Zustandsübergang des Aktivitäts-Threads von Warten, als dem blockierenden Lesen, zur Ausführung (vgl. Abbildung 5.3). Nach Beendigung der Aktivitätslogik werden für die Prozessinstanz, zur welcher die Ausführung gehörte, entsprechende Token erzeugt, welche die nachfolgenden Aktivitäten starten können.

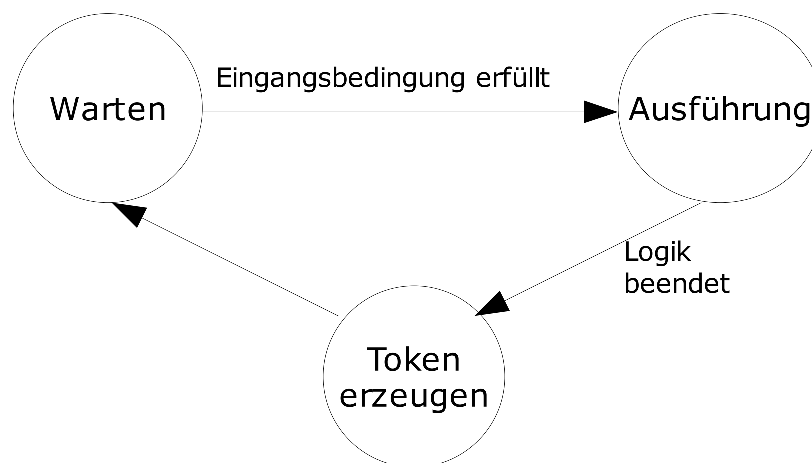



*Abbildung 5.3: Zuständsübergänge eines Aktivitäts-Threads*

Ein Spezialfall der Basis-Aktivitäten bildet das receive, bei dem Logik und das Template Matching in zwei verschiedene Threads getrennt. Der Grund dafür liegt, darin dass das receive laut BPEL-Semantik selbst eine blockierende Operation ist. Das heißt, die Aktivität wartet nachdem sie gestartet wurde auf eine eingehende Nachricht. Würde man dies innerhalb desselben Threads realisieren wie das Template Matching hätte dies zur Folge, dass der Thread der receive-Aktivität im Zustand „Ausführung“ blockiert ist bis eine Nachricht empfangen wird. Das bedeutet weiterhin, dass sie nicht für eine weitere Instanz durch das Template Matching gestartet werden kann. Dadurch kann keine weitere Prozessinstanz mehr vollständig durchgeführt werden bis für den receive-Thread, welcher sich im Zustand Ausführung befindet, eine Nachricht empfangen wurde. Aus diesem Grund wird die Aktivität receive aufgeteilt, in einen Thread der das Warten auf neue Eingangs-Token übernimmt (Template Matching) und einen der die Ausführung einer konkreten Instanz übernimmt und nach dessen Beendigung die Token für die Nachfolger erzeugt.

Diese Aufteilung eines Aktivitäts-Threads in zwei unterschiedliche Thread, also Warten und Ausführung, könnte ebenso für die weiteren Aktivitäten verwendet werden. Der Grund warum dies nicht umgesetzt wird, liegt alleine in der Annahme, dass die weiteren Aktivitäten prinzipiell so kurz sind, dass das Erzeugen eines Extra-Threads keine wesentlichen Vorteile hinsichtlich der Parallelität erzielt.

Das Schema einer Aktivität (Abb. 5.2) zeigt, dass ein Aktivitäts-Thread aus unterschiedlichen Teilen besteht, also aus Template-Matching, Logik und Erzeugen von Token. Dabei unterscheiden sich einzelne Aktivitäten nur durch die Logik, die sie realisieren. Das Template-Matching und das Erzeugen der Token sind für alle Aktivitäten prinzipiell gleich. Diese Aspekte werden deshalb in spezielle Objekte gekapselt, welche diese Aufgaben übernehmen und durch eine Aktivität entsprechend konfiguriert sind. Ebenso wird der Datenzugriff, welcher durch Lesen auf Tuple Spaces realisiert wird, in entsprechende wieder verwendbare Objekte gekapselt. Diese Objekte geben also jeweils die Schnittstellen zu den Tuple Spaces an, welche von den Aktivitäten verwendet werden können. Die entsprechenden Schnittstellen-Objekte werden in Abschnitt 5.4 genauer vorgestellt.

### 5.3.2 Aufbau strukturierter Aktivitäten

Der Aufbau von strukturierten Aktivitäten unterscheidet sich von den Basis-Aktivitäten, da sie zusätzlich noch den Kontrollfluss über die eingeschlossenen Aktivitäten koordinieren müssen. Ihre eigentliche Logik besteht dabei auch aus dem Erzeugen von Token, welche an die eingeschlossenen Aktivitäten gerichtet sind, und dem Lesen solcher Token, welche von diesen nach deren Beendigung erzeugt wurden.

Aus diesem Grund werden die strukturierten Aktivitäten in zwei Teile aufgeteilt. Diese Aufteilung ist ähnlich zu einer in Apache ODE vorgenommenen Aufteilung der strukturierten Aktivitäten, wo zum Beispiel eine Sequence-Aktivität durch Objekte der Klassen `SEQUENCE` und `SEQUENCE.ACTIVE` (vgl. Listing 3.2) realisiert wird. Diese zwei Teile sind im Fall der entwickelten Laufzeitumgebung nun zwei Threads, die jeweils beide ständig aktiv sind und auf bestimmte Token warten, also einen eigenen „Template Matching“-Abschnitt besitzen. Allerdings unterscheiden sich diese in den gelesenen Token, welche die Logik des Thread starten. Ein Teil übernimmt den eingehenden äußeren Kontrollfluss (Aktivieren der Aktivität selbst), der andere den inneren (Aufruf und Kontrolle der eingeschlossenen Aktivitäten). Auch ist dementsprechend die Logik unterschiedlich, welche die beiden Teile beim Erhalt ihrer jeweiligen Token realisieren.

Der erste Thread dient der Instanziierung einer strukturierten Aktivität. Er realisiert also den eingehenden Kontrollfluss und das Starten der Aktivität. Der schematische

Aufbau ist in Abbildung 5.4 gegeben. Die Templates, welche er für dafür benutzt, sind analog zu denen der einfachen Aktivitäten. Sie bestehen also aus der Beschreibung der Token, die von Vorgängeraktivitäten an die strukturierte Aktivität gesendet werden. Auch hier ist zu beachten, dass bei mehreren Token ein synchronisiertes Lesen vorgenommen werden muss, um zu gewährleisten, dass die Token zur selben Prozessinstanz gehören. Nach Empfangen der Menge an Token, welche die Startbedingung erfüllen, also wenn das Template Matching Token gelesen hat und nicht mehr den Thread blockiert, startet, wie bei den einfachen Aktivitäten, die eigentliche Ausführung der Logik. Diese beinhaltet das Aufrufen einer oder mehrerer eingeschlossenen Aktivitäten, gemäß der BPEL-Semantik der Aktivität. Im Falle einer Sequence wäre dies zum Beispiel zunächst das Starten der ersten Aktivität durch Erzeugen der entsprechenden Token, welche diese verarbeiten kann. Beim Flow hingegen werden Token an alle Kind-Aktivitäten erzeugt, was jedoch nicht bedeuten muss, dass diese auch direkt ausgeführt werden (vgl. Link-Konzept). Nach dem Erzeugen der entsprechenden Token an die Kind-Aktivitäten ist die Logik des ersten Threads abgeschlossen und er geht wieder in den Warten-Zustand auf neue Prozessinstanzen, welche diese Aktivität benutzen.

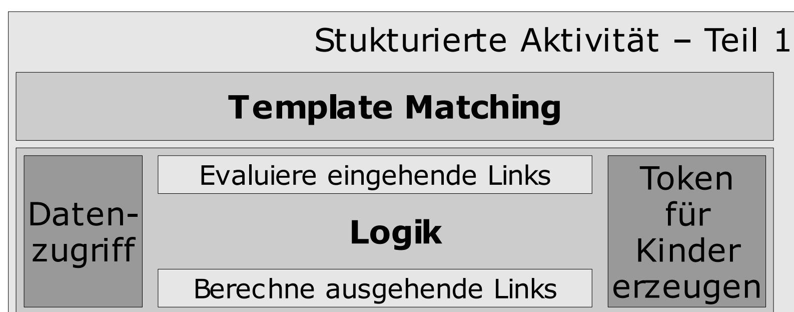



*Abbildung 5.4: Aufbau einer strukturierten Aktivität für den eingehenden Kontrollfluss*

Der Kontrollfluss wurde somit an die eingeschlossenen Aktivitäten übergeben, welche nach Beendigung wieder ein Token für ihre Eltern-Aktivität, also der strukturierten Aktivität selbst, erzeugen. Für die Verarbeitung dieser Token der Kind-Aktivitäten, dient der zweite Thread (Abb. 5.5) einer strukturierten Aktivität. Dieser besteht wiederum aus den Teilen „Template-Matching“, eigener Logik und dem Erzeugen von Ausgangstoken. Die Templates unterscheiden sich aber von den zuvor genannten des ersten Threads. Der zweite Teil-Thread wartet nämlich nicht auf Token, welche von den Vorgängeraktivitäten erzeugt wurden, sondern auf Token welche von den eingeschlossenen Aktivitäten stammen. Ist eines oder mehrere solcher Token vorhanden, übernimmt er die Koordination der eingeschlossenen Aktivitäten.

*Abbildung 5.5: Aufbau einer strukturierten Aktivität für die Koordination eingeschlossenen Kind-Aktivitäten*

Im Beispiel einer Sequence-Aktivität bedeutet dies, dass der zweite Teil immer genau auf ein Token wartet, welches von einer beliebigen Kindaktivität erzeugt werden kann. Dazu verfügt der „Template Matching"-Abschnitt über eine Menge von Templates, die jeweils ein Token beschreiben, welches von einer beliebigen Kindaktivität erzeugt wurde. Das Template benutzt also die Ziel- und Absender-Informationen des erwarteten Tokens, wobei die konkrete Instanz egal ist. Sobald dieser Sequence-Thread einen solchen Token gelesen hat, erfolgt die Ausführung seiner Logik. Dies bedeutet, dass der Absender des empfangenen Tokens betrachtet wird und danach ein Token erzeugt wird, welches an dessen Nachfolgeraktivität gerichtet ist. Für den Fall, dass der Absender die letzte Aktivität der Sequenz war, ist diese Instanz der Sequence-Aktivität beendet und Kontrollflusstoken an die nachfolgenden Aktivitäten werden erzeugt.

Im Beispiel eines Flow, werden im ersten Thread Kontrollflusstoken an alle eingeschlossenen Aktivitäten erzeugt. Das „Template Matching" des zweiten Threads wartet auf Token aller Kinder. Dieser kann also prinzipiell wie eine gewöhnliche Aktivität aufgefasst werden, welches eine Menge von Eingangstoken als Startbedingung besitzt. Das Lesen muss auch hier wieder synchronisiert werden, um zu gewährleisten, dass alle gelesenen Token zur selben Prozessinstanz gehören. Die Logik der Flow Aktivität weiß nun, dass alle eingeschlossenen Aktivitäten beendet sind. Damit ist auch sie beendet, d.h., sie erzeugt Token, welche für ihre Nachfolger bestimmt sind.

## 5.4 Schnittstellenobjekte zum Zugriff auf Tuple Spaces

In den vorherigen Abschnitt wurde der Aufbau der Aktivitäten ausführlich dargestellt. Jeder Aktivitäts-Thread besitzt dabei einen so genannten „Template Matching"-Abschnitt, welcher zum Lesen von Token, die als Startbedingung dienen, verwendet

wird. Zudem benötigen die meisten Aktivitäten einen Zugriff auf bestimmte Daten, welche ebenfalls durch Token repräsentiert sind. Dies sind somit jeweils Kommunikationsschritte mit einem oder mehreren Tuple Spaces. Um diese möglichst wieder verwendbar zu implementieren und nicht in jeder Aktivität neu entwickeln zu müssen, wurden diese Schnittstellen in spezielle „Tuple Space Zugriffsobjekte“ gekapselt. Diese werden von den Aktivitäten über deren jeweiligen Konfigurationen angepasst und zum Zugriff auf die Tuple Spaces genutzt. In Abbildung 5.6 ist ein Ausschnitt aus dem vereinfachten Klassendiagramm der Zugriffsklassen und deren Verwendung durch die Aktivitäten dargestellt. Die einzelnen Schnittstellen werden nachfolgend erläutert.

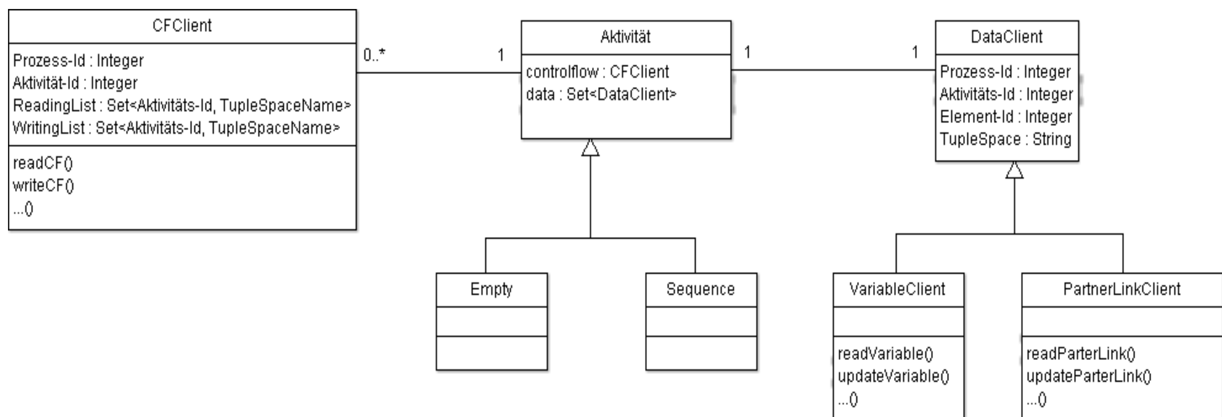



*Abbildung 5.6: Vereinfachtes Klassendiagramm der entwickelten Lösung*

### 5.4.1 Kontrollflussschnittstelle

Alle Aktivitätsimplementierungen besitzen wie zuvor beschrieben eine Schnittstelle zum Lesen von beliebig vielen Token, den so genannten „Template-Matching“-Abschnitt, sowie eine Schnittstelle zum Erzeugen von Token an nachfolgende Aktivitäten oder Kinder. Diese Schnittstelle wird durch Objekte der Klasse `CFClient` zur Verfügung gestellt. Ein solches Objekt beinhaltet einen Verweis auf die Aktivität, welche die Schnittstelle verwendet, durch desen Bezeichner und Prozessmodell. Zudem besitzt sie zwei Listen, welche jeweils Einträge enthalten, die aus einer Kombination einer Aktivitäts-Id und dem Namen eines Tuple Spaces bestehen. Diese stellen zum Einen alle Aktivitäten mitsamt den zugehörigen Tuple Spaces dar, welche die Startbedingung der Aktivität bilden und zum Anderen die Tuple Spaces, auf welche solche Token geschrieben werden sollen, die für nachfolgende Aktivitäten bestimmt sind.

Die Klasse stellt im Wesentlichen die beiden Funktionen `readCF` und `createCF` zur Verfügung. Mit der Funktion `readCF` wird das „Template Matching“ realisiert. Dabei benötigt sie keine weiteren Parameter. Wird sie aufgerufen, erzeugt sie aus den vor-

handenen Informationen der Eingangsliste die entsprechenden Templates zum Lesen aller Token. Die Funktion ist blockierend, d.h., sie gibt erst ein Ergebnis zurück, wenn alle Token vorhanden sind. Mit ihr wird auch das zuvor öfters erwähnte synchronisierende Lesen realisiert, also dem Sicherstellen, dass alle gelesenen Token, durch dieselbe Prozessinstanz zusammengehören.

Zum Erzeugen von Kontrollflusstoken an alle Nachfolger-Aktivitäten dient die Funktion `createCF`, welche als Parameter das History-Feld und die zugehörige Prozessinstanz der Aktivität nimmt, welche die Token erzeugen möchte. Zudem muss natürlich der Typ der Kontrollflusstoken, also ob es sich beispielsweise um einen regulären Kontrollfluss oder einen nicht erreichbaren Pfad (dead Path) handelt, angegeben werden. Die Funktion `createCF` erzeugt aus diesen Daten alle Token, die an die, in der Ausgangsliste vorkommenden Aktivitäten gerichtet sind. Diese Schreibeoperationen sind zu einer Transaktion zusammengefasst, also entweder werden alle Token erzeugt oder gar keine.

Die Implementierung einer einfachen Aktivität sieht dadurch folgendermaßen aus. Zunächst wird der Thread initialisiert, indem sie ein `CFClient`-Objekt erzeugt, welche für den Aktivitäts-Thread genutzt werden kann. Dazu benötigt der Konstruktor dieses Objektes die entsprechende Aktivitäts-Id und die Prozessmodell-Id. Dann fügt sie aus den Konfigurationsinformationen die Spaces und Aktivitäten, mit welchen sie kommunizieren möchte, in die entsprechenden beiden Listen ein. Damit ist der `CF-Client` dieser Aktivität schon eindeutig beschrieben und kann den Kontrollfluss durch Tokenaustausch realisieren. Als nächstes ruft der Aktivitätsthread die Funktion `readCF` des zugehörigen `CFClient`-Objektes auf und wartet damit bis die Ausführung der Logik gestartet werden kann. Sobald dieses die benötigten Eingangstoken gelesen hat, wird die Aktivitätslogik für eine Instanz, die aus den eingegangenen Token bekannt ist, ausgeführt. Nach Beendigung dieser Ausführung wird `createCF` mit den Daten aufgerufen, die sich aus dem Durchführen der Logik ergeben haben, womit die entsprechenden Token generiert werden.

### 5.4.2 Datenaustauschschnittstelle

Ähnlich dazu erfolgt das Lesen und Schreiben von Datentoken über spezialisierte Objekte der Klasse `DataClient`. Ein solches Objekt regelt dabei jeweils den Zugriff auf genau ein Datenelement, wie beispielsweise eine Variable oder ein Correlation Set. Eine Aktivität, die auf ein bestimmtes Datentoken zugreifen möchte, instanziiert dann zuerst für jedes benötigte Datenelement einen `DataClient`, also zum Beispiel einen `VariableClient`. Dazu verwendet sie ihre Aktivitäts- und Prozessmodell-Id, sowie den Bezeichner und Tuple Space des benötigten Datenelementes.

Während der Ausführung des Tuple Spaces benötigt eine Aktivität zum Lesen oder Verändern bestimmter Token dann lediglich den Aufruf einer Funktion, wie beispielsweise `readVariable` oder `updateVariable` auf dem, diesem Datenelement zugeordneten Client-Objekt. Eine solche Operation verlangt als Parameter den Prozesszustand aus Sicht der Aktivität durch deren History-Feld. Das Auflösen zu dem Token, welches zu dieser Instanz gehört, erfolgt durch die Funktion selbst.

Die Basisklasse `DataClient` bietet dazu eine Funktion an, welche das sequentielle Durchsuchen der Prozessinstanzen realisiert. Die spezialisierten Client-Klassen der einzelnen Datenelementtypen benutzen diese Methode um ein Token zu lesen und geben dann den entsprechenden Wert bzw. Zustand dieses Elements zurück.

# 6 Zusammenfassung und Ausblick

In dieser Arbeit wurde das Konzept der verteilten Ausführung von BPEL-Workflows noch weiter verfeinert und eine prototypische Realisierung dieses Konzeptes geliefert, welches die Umsetzbarkeit des Ansatzes zeigt. Dafür wurde die Struktur der ausgetauschten Token erläutert und eine Möglichkeit vorgestellt, wie einzelne Aktivitäten über einen Deployment Descriptor konfiguriert und Teile eines Prozesses auf verschiedene Teilnehmer verteilt werden können. Die entwickelte BPEL-Laufzeitumgebung dient als eine Art „proof-of-concept“, dass die verteile Ausführung von BPEL-Workflows dezentral durch den Austausch von Token gesteuert werden kann, wobei die Nachteile der herkömmlichen zentralen Ausführung umgangen werden können.

Gegenstand weiterer Arbeiten in diesem Bereich stellen sich unter anderem in der Prozess-Segmentierung, also der effizienten Aufteilung eines BPEL-Prozesses in verschiedene Teile aufgrund von verschiedenen Kriterien. Zudem ist die entwickelte BPEL-Laufzeit noch nicht vollständig und unterstützt noch nicht alle Aktivitäten.

# Abkürzungsverzeichnis

BPEL ........ Business Process Execution Language
BPEL4WS ........ Business Process Execution Language for Web Services
BPMN ........ Business Process Modelling Notation
CORBA ........ Common Object Request Broker Architecture
DOM ........ Document Object Model
EPK ........ Ereignisgesteuerte Prozessketten
EWFN ........ Executable Workflow Network
FTP ........ File Transport Protocol
HTTP ........ Hypertext Transfer Protocol
JBI ........ Java Business Integration
JMS ........ Java Messaging Service
OASIS ........ Organization for Advancement of Structured Information Standards
SMTP ........ Simple Mail Transfer Protocol
SOA ........ Servieorientierte Architektur
UDDI ........ Universal Description, Discovery and Integration
URI ........ Uniform Resource Identifier
W3C ........ World Wide Web Consortium
WfMS ........ Workflow Management System
WSDL ........ Web Service Description Language
XML ........ eXtensible Markup Language

# Abbildungsverzeichnis

Abbildung 1.1: Zentrale und dezentrale Navigation.........7
Abbildung 2.1: Aufbau eines Geschäftsprozesses.........10
Abbildung 2.2: Prozesse und Workflows.........12
Abbildung 2.3: Hauptbestandteile eines WfMS.........13
Abbildung 2.4: SOA-Dreieck.........14
Abbildung 2.5: Two-Level-Programming-Paradigma.........14
Abbildung 2.6: Web Service als virtuelle Komponente (nach [Ley03]).........15
Abbildung 2.7: Web Service Stack (nach [W3C04a]).........16
Abbildung 2.8: Web Service Dreieck.........17
Abbildung 2.9: Kommunikation über Tuple Spaces (nach [Pop08]).........19
Abbildung 3.1: Apache ODE.........23
Abbildung 3.2: Architektur von Apache ODE (übernommen von [ODEb]).........24
Abbildung 3.3: Ausschnitt aus Klassendiagramms des ODE-Objektmodells.........26
Abbildung 3.4: Ausschnitt aus Klassendiagramm für JACOB-Objekte.........28
Abbildung 3.5: Beispiel der Ausführung eines BPEL-Prozesses in Apache ODE.........30
Abbildung 4.1: Vom Modell zur verteilten Ausführung eines Prozesses.........36
Abbildung 4.2: Kontrollflusses durch Tokenaustausch über Tuple Spaces (in Anlehnung an [MWL08b]).........37
Abbildung 4.3: Schnittstellen von Aktivitäten zu Tuple Spaces.........40
Abbildung 4.4: Beispiel einer einfachen Flow-Aktivität.........43
Abbildung 4.5: Probleme bei Kontrollflusstoken ohne Absender-Informationen.........52
Abbildung 4.6: Kontrollflusstoken mit Absender-Informationen.........53
Abbildung 4.7: Beispiel für die Verwendung von Absender-Informationen bei strukturierten Aktivitäten.........55
Abbildung 4.8: Beispiel für Entwicklung des History-Feldes der Kontrollflusstoken. 56
Abbildung 4.9: Beispiel zur Verwendung eines Identifiers bei Datentoken.........57
Abbildung 4.10: Beispiel für die wiederholte Ausführung von Scopes.........58
Abbildung 5.1: Architektur der Lösung im Vergleich zu Apache ODE.........66
Abbildung 5.2: Schema einer Aktivitäts-Implmentierung.........69
Abbildung 5.3: Zuständsübergänge eines Aktivitäts-Threads.........70
Abbildung 5.4: Aufbau einer strukturierten Aktivität für den eingehenden Kontrollfluss.........72
Abbildung 5.5: Aufbau einer strukturierten Aktivität für die Koordination

eingeschlossenen Kind-Aktivitäten......................................................................................73
Abbildung 5.6: Vereinfachtes Klassendiagramm der entwickelten Lösung..................74

# Verzeichnis der Listings

Listing 3.1: Ausschnitt aus einem Beispiel-BPEL-Prozess....30
Listing 3.2: Vereinfachte Beispiel-Implementierung eines SEQUENCE Objektes in JACOB....31
Listing 3.3: Vereinfachte Beispiel-Implementierungen von ASSIGN und EMPTY....32
Listing 4.1: Grundstruktur des Deployment Descriptors....44
Listing 4.2: Variablenelemente im Deployment Descriptor....44
Listing 4.3: Weitere Datenelemente im Deployment Descriptor....46
Listing 4.4: Aktivitäten im Deployment Descriptor....47
Listing 4.5: Beispiel zur Verwendung von Quell- und Ziel-Tuple Spaces....48

# Tabellenverzeichnis

Tabelle 4.1: Header-Felder der Token.....50
Tabelle 4.2: Felder eines Kontrollflusstokens.....51
Tabelle 4.3: Felder des Tokens einer Variablen.....57
Tabelle 4.4: Felder des Tokens eines Correlation Sets.....60
Tabelle 4.5: Felder des Tokens eines Partner Links.....61
Tabelle 4.6: Felder des Tokens eines Links.....61
Tabelle 4.7: Felder des Tokens einer Fehlermeldungen.....62
Tabelle 4.8: Felder des Tokens einer empfangenen Nachricht.....63

# Literaturverzeichnis


[Aal98] W.M.P. van der Aalst: "*The Application of Petri Nets to Workflow Management*". Journal of Circuits, Systems and Computers 8 (1), S. 21-66, 1998

[ACD+03] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic und S. Weerawarana: "*Business Process Excecution Language for Web Services - Version 1.1*", 5. Mai 2003. Verfügbar unter: http://www.ibm.com/developerworks/library/specification/ws-bpel/

[ACK+04] G. Alonso, F. Casati, H. Kuno und V. Machiraju: "*Web Services. Concepts, Architectures and Applications*". Springer-Verlag Berlin Heidelberg New York, 2004

[Agh86] G. Agha: "*Actors: A Model of Concurrent Computation in Distributed Systems*". MIT Press, 1986

[AXIS2] Apache Axis2. http://ws.apache.org/axis2/

[BFM05] T. Berners-Lee, R. Fielding und L. Masinter: "*Uniform Resource Identifier (URI): Generic Syntax*". Request for Comments 3986, Januar 2005. Verfügbar unter: http://tools.ietf.org/html/rfc3986

[DJM+05] W. Dostal, M. Jeckle, I. Melzer und B. Zengler: "*Service-orientierte Architekturen mit Web Services. Konzepte - Standards - Praxis*". Spektrum Akademischer Verlag, 1. Auflage. 2005

[FKL+07] D. Fensel, E. Kühn, F. Leymann und R. Tolksdorf: "*Queues Are Spaces - Yet Still Both Are Not The Same?*". Technical Repot DERI, Univ. Innsbruck, 2007

[Gad05] A. Gadatsch: "*Grundkurs Geschäftsprozess-Management*". Vieweg, 4., erweiterte Auflage, 2005

[Gel85] D. Gelernter: "*Generative communication in Linda*". ACM Transactions on Programming Languages and Systems 7 (1), S. 80 -112, 1985

[HC94] M. Hammer und J. Champy: "*Business Reengineering. Die Radikalkur für das Unternehmen*". Campus-Verlag Frankfurt/Main, New York, 2. Auflage, 1994

[HW03] G. Hohpe und B. Woolf: "*Enterprise Integration Patterns. Designing, Building, and Deploying Messaging Solutions*". Addison-Wesley Longman, Amsterdam, 2003

[IDS08] IDS Scheer AG: "*ARIS Platform - Produktbroschüre*". 2008. Verfügbar unter: http://www.ids-scheer.de/set/6473/Produktbroschuere%202008-07.pdf

[JBI] Java Community Process: "*Java Business Integration 1.0* ". JSR 208, . Verfügbar unter: http://jcp.org/aboutJava/communityprocess/final/jsr208/index.html

[Jen92] K. Jensen: "*Coloured Petri Nets, Vol. 1: Basic Concepts*". EATCS Monographs on Theoretical Computer Science. Springer-Verlage Berlin, Heidelberg, New York, 1992

[JSH+01] S. Jablonski, R. Schamburger, C. Hahn, S. Horn, R. Lay, J. Neeb und M. Schlundt: "*A Comprehensive Investigation of Distribution in the Context of Workflow Management*". Proc. of ICPADS, 2001

[KL06] R. Khalaf und F. Leymann: "*Role-based Decomposition of Business Processes using BPEL*". International Conference of Web Services ICWS, 2006

[KMS92] G. Keller, M. Nüttgens und A.-W. Scheer: "*Semantische Prozeßmodellierung auf der Grundlage Ereignisgesteuerter Prozeßketten (EPK)*". Veröffentlichungen des Instituts für Wirtschaftsinformatik, Universität des Saarlandes, Heft 89, 1992

[KRJ05] E. Kühn, J. Riemer und G. Joskowicz: "*XVSM (eXtensible Virtual Shared Memory) Architecture and Application*". Technical Report, TU Wien, E185/1, Space Based Computing Group, 2005

[Ley03] F. Leymann: "*Web Services Distributed Applications Without Limits*". Proc. of BTW (Leipzig), 2003

[LR00] F. Leyman und D. Roller: "*Production Workflow: Concepts and Techniques*". Prentice Hall PTR Upper Saddle River, NJ, USA, 1999

[LSR06] C. Längerer, F. Schmitt und J. Rutschmann: "*Performance-Vergleich von BPEL-Engines*". Fachstudie Nr. 63, Universität Stuttgart, 2006

[LSW97] P. Langner, C. Schneider und J. Wehler: "*Prozessmodellierung mit ereignisgesteuerten Prozessketten (EPKs) und Petri-Netzen*". Wirtschaftsinformatik 39 (S. 479-489), 1997

[MWL08a] D. Martin, D. Wutke und F.Leyman: "*A Novel Approach to Decentralized Workflow Enactement*". 12th IEEE International EDOC Conference, 2008

[MWL08b] D. Martin, D. Wutke und F. Leymann: "*Tuplespace-based Infrastructure for Decentralized Enactment of BPEL Processes*". 9. Internationale Tagung Wirtschaftsinformatik (WI 2009): Business Services: Konzepte, Technologien, Anwendungen., 2008

[MWW+98] P. Muth, D. Wodtke, J. Weissenfels, A. Dittrich und G. Weikum: "*From Centralized Workflow Specification to Distributed Workflow Execution*". Journal of Intelligent Information Systems 10 (2), S. 159-184, 1998

[OAS07] OASIS Group: "*Web Service Business Process Execution Language 2.0*", 11. April 2007. Verfügbar unter: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html

[ODE] Apache ODE. http://ode.apache.org

[ODEa] ODE JACOB Framework. http://ode.apache.org/jacob.html

[ODEb] ODE-Architectural Overview. http://ode.apache.org/architectural-overview.html

[OMG09] Object Management Group: "*Business Process Modeling Notation (BPMN) Version 1.2*", Januar 2009. Verfügbar unter: http://www.omg.org/spec/BPMN/1.2/

[Öst95] H. Österle: "*Business Engineering: Prozess- und Systementwicklung (Band 1: Entwurfstechniken)*". Springer-Verlag Berlin Heidelberg u.a, 2. verbesserte Auflage, 1995

[Pet62] C. A. Petri: "*Kommunikation mit Automaten*". Schriften des Rheinisch-Westfälischen Institutes für instrumentelle Mathematik an der Universität Bonn, 1962

[Pop08] Petia Popova: " *Verteilte Workflow-Engine: Transformation von BPEL Prozessen in ein verteiltes Ausführungsmodell*". Diplomarbeit Nr. 2802, Universität Stuttgart, 2008

[SJ96] A.-W. Scheer und W. Jost: "*Geschäftsprozessmodellierung innerhalb einer*

*Unternehmensarchitektur*". in: G.Vossen und J.Becker (Hrsg.): Geschäftsprozessmodellierung und Workflow-Management: Modelle, Methode, Werkzeuge. Int. Thomson Publ., Bonn, 1996

[Sta05] C. Stahl: "*A Petri Net semantic for BPEL*". Technical Report, Humboldt Universität Berlin, 2005

[Sta06] J. Staud: "*Geschäftsprozessanalyse: Ereignisgesteuerte Prozessketten und objektorientierte Geschäftsprozessmodellierung für Betriebswirtschaftliche Standardsoftware*". 3. Auflage, Springer-Verlag; Berlin Heidelberg New York, 2006

[Ste08] T. Steinmetz: "*Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE*". Diplomarbeit Nr. 2729, Universität Stuttgart, 2008

[SUP07] Semantics Utilized for Process management within and between Enterprises (SUPER) Project: "*Execution Engine Design and Architecture*". Deliverable 6.1, Januar 2007. Verfügbar unter: http://www.ip-super.org/res/Deliverables/M12/D6.1.pdf

[TOMCAT] Apache Tomcat. http://tomcat.apache.org/

[TSpace] TSpaces: Intelligent Connectionware. http://www.almaden.ibm.com/cs/TSpaces/

[W3C02] W3C Web Services Activity. http://www.w3.org/2002/ws/Activity

[W3C04a] Web Service Architecture. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

[W3C04b] Web Services Architecture Requirements. http://www.w3.org/TR/2004/NOTE-wsa-reqs-20040211

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey und D. F. Ferguson: "*Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*". Prentice Hall PTR Upper Saddler River, NJ, USA, 2005

[WMC05] Workflow Management Coalition: "*The Workflow Reference Model*", 1995. Verfügbar unter: http://www.aiai.ed.ac.uk/project/wfmc/ARCHIVE/DOCS/refmodel/httoc.html

[Wu08] S. Wu: "*Verteilte Workflow-Engine: Ausführung verteilter Prozessmodelle auf*

*Basis von Tuplespaces.* ". Diplomarbeit Nr. 2831, Universität Stuttgart,

[Zlo75] M. Zloof: "*Query by Example*". Proc.of AFIPS National Computer Conference. 44, S. 431--438, 1975

Alle angegebenen Links wurden zuletzt am 11. März 2009 geprüft.

**Erklärung**

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen verwendet zu haben.

________________________________________

(Stefan Varnhorn)