BPEL^{light}:
Flexibilisierung von Prozessmodellen durch Entkopplung von Prozesslogik und Schnittstellenspezifikation

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der

Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Jörg Nitzsche

aus Stuttgart (Deutschland)

Hauptberichter: **Prof. Dr. rer. nat. Frank Leymann**

Mitberichter: **Jun.-Prof. Dr.-Ing. Dimka Karastoyanova**

Tag der mündlichen Prüfung: 26.07.2011

Institut für Architektur von Anwendungssystemen der Universität Stuttgart

2011

Zusammenfassung

Durch die digitale Revolution und der damit einhergehenden Globalisierung gewinnt die Integration heterogener Systeme immer mehr an Bedeutung. Geschäftsprozesse involvieren heutzutage nicht nur Systeme und Mitarbeiter eines Standortes. Bei global agierenden Organisationen bzw. Unternehmen können sie sich sowohl über Standorte in mehreren Ländern oder Kontinenten erstrecken als auch über mehrere Organisationseinheiten und sogar über Unternehmensgrenzen hinweg. Sie erfordern somit zwangsläufig die Integration unterschiedlichster heterogener Systeme. Eine der neuesten Entwicklungen der Anwendungsintegration ist die Web-Service-Technologie, die häufig auch als „Web-Services“ (WS) bezeichnet wird. Sie ist eine offene, XML-basierte Realisierung der Service-orientierten Architektur (SOA) und stellt einzelne Teile einer Anwendung als plattformunabhängige Dienste über ein Netzwerk zur Verfügung. Das ermöglicht die Entwicklung lose gekoppelter, verteilter Systeme in einer heterogenen Umgebung. Web-Services werden durch eine Menge sich ergänzender und teilweise aufeinander aufbauender Spezifikationen und Standards definiert. Diese decken dabei unterschiedliche Aspekte ab. Die Funktionalität bzw. Schnittstelle eines Web-Services in Form von zu Porttypen gebündelten Operationen wird mit der Web Service Description Language (WSDL) beschrieben. Zusätzlich zu dieser abstrakten Schnittstellenbeschreibung definiert WSDL auch die konkreten Aufrufinformationen wie Transportprotokoll und Nachrichtenformat, das so genannte Binding. Somit entkoppelt WSDL im Gegensatz zu anderen Schnittstellenbeschreibungssprachen die abstrakte Definition der Schnittstelle von den konkreten Aufrufinformationen und trägt damit zur angestrebten losen Kopplung bei. Zur Workflow-basierten Orchestrierung von Diensten, d.h. zur Modellierung und Ausführung Web-Service-basierter Geschäftsprozesse wird die Web Service Business Process Execution Language (WS-BPEL oder BPEL) verwendet. Die Sprache BPEL ist eine Kombination aus Graph- und Block-basierten Ansätzen zur Prozessmodellierung. Sie ermöglicht es, einen Fluss zwischen Aktivitäten zu spezifizieren, die von Web-Service-Operationen implementiert werden,

und ermöglicht somit die Prozess-orientierte Integration verteilter, heterogener Anwendungen.

Eine Begleiterscheinung der Globalisierung ist der ständig wachsende Preisdruck auf weltweit agierende Unternehmen. Um diesem Preisdruck standhalten zu können ist es notwendig, Prozesse zu optimieren. Eine Möglichkeit der Prozessoptimierung ist es, zur Implementierung von Aktivitäten eines Prozesses immer die zu gegebener Zeit kostengünstigsten Dienste auszuwählen. Dies wird von BPEL prinzipiell unterstützt: Aktivitäten verweisen lediglich auf die Operation, d.h. auf die abstrakte Schnittstellenbeschreibung des zu verwendenden Web-Services. Sie werden in der Prozessbeschreibung nicht gegen die tatsächliche Implementierung der Operation gebunden. Somit ist es möglich zur Laufzeit frei zwischen unterschiedlichen, die angegebene Operation implementierenden Diensten zu wählen. Allerdings bieten unterschiedliche Organisationen bzw. Unternehmen die gleiche Funktionalität in der Regel auf unterschiedliche Art und Weise an, d.h. unter Verwendung unterschiedlicher Porttypen und Operationen. Dass sich dieser Umstand kurz- oder mittelfristig ändert, dass es also standardisierte domänenspezifische Porttypen und Operationen gibt, wird von der Industrie als unrealistisch eingeschätzt. Folglich müssen BPEL-Prozessmodelle in der Regel jedes Mal geändert werden, wenn Dienste eines anderen, günstigeren Anbieters in Anspruch genommen werden sollen. Dies ist sehr aufwendig und schränkt die Agilität von Unternehmen in nicht unbedeutendem Maße ein.

Die Definition von Aktivitäten mittels eines Verweises auf eine WSDL-Operation hat nicht nur eine eingeschränkte Flexibilität von BPEL bezüglich der Partnerwahl zur Folge, sondern beschränkt die Aktivitäten zudem auf den Austausch von maximal zwei Nachrichten mit einem Geschäftspartner. Betrachtet man die Aktivitäten jedoch als Implementierung einer Geschäftsfunktion innerhalb eines Geschäftsprozesses, so ist der Austausch zweier Nachrichten mit einem Geschäftspartner im Allgemeinen nicht ausreichend, da komplexe Geschäftsfunktionen auch den Austausch mehrerer Nachrichten mit mehreren Geschäftspartnern erfordern können.

Diese Arbeit verfolgt die Idee von einem von WSDL unabhängigen BPEL, um die genannten Probleme mit herkömmlichem BPEL zu lösen. Sie definiert eine neue, auf BPEL basierende Workflowsprache namens BPEL^{light}. BPEL^{light} definiert ein von WSDL unabhängiges Interaktionsmodell zur Entkopplung der Prozesslogik von den Schnittstellenspezifikationen der Aktivitätsimplementierungen. Dieses Interaktionsmodell beschränkt sich nicht wie jenes von herkömmlichem BPEL auf die Definition bilateraler Interaktionen des Prozesses mit Geschäftspartnern, sondern ermöglicht die explizite Definition multilateraler Interaktionen. Darüber hinaus erlaubt es BPEL^{light}, die Semantik von Aktivitäten mittels so genannter Nachrichtenaustauschmuster zu spezifizieren anstatt einen fixen Satz an Aktivitäten vorzugeben, und trägt damit dem Umstand Rechnung, dass komplexe Geschäftsfunktionen einen komplexen Nachrichtenaustausch erfordern können. Die Sprache BPEL^{light} wird als Erweiterung zu BPEL definiert. Die operationale Semantik von BPEL bleibt also von den Erweiterungen unberührt. Herkömmliche BPEL-Engines mit entsprechenden Erweiterungen können somit verwendet werden, um BPEL^{light}-Prozesse auszuführen. Da Web-Services trotz der genannten Defizite zurzeit eine populäre Technologie zur Integration verteilter Anwendungen sind, zeigt diese Arbeit wie Web-Services verwendet werden können, um Aktivitäten von BPEL^{light}-Prozessen zu implementieren und wie der BPEL^{light}-Prozess selbst als Web-Service zur Verfügung gestellt werden kann. Ein weiterer Beitrag der Arbeit ist die Verwendung von den im Vergleich zu Web-Services weitaus flexibleren Semantischen Web-Services (SWS) als Aktivitätsimplementierungen von BPEL^{light}-Prozessen. Die im Rahmen dieser Arbeit vorgestellte, auf BPEL^{light} basierende Sprache zur Komposition von SWS trägt den Namen BPEL für Semantische Web-Services (BPEL4SWS). Sowohl für BPEL4SWS als auch für BPEL^{light} mit WS als Aktivitätsimplementierung wird gewährleistet, dass eine Migration von BPEL-Prozessmodellen zur jeweiligen Kombination von BPEL^{light}-Prozessmodellen mit entsprechenden Aktivitätsimplementierungen transparent für Nutzer des vom Prozess angebotenen Dienstes erfolgen kann. Für BPEL4SWS bedeutet das auch, dass jeder Prozess sowohl über eine semantische als auch über eine klassische WSDL-Schnittstelle aufgerufen werden kann. Die erarbeiteten Konzepte werden abschließend

durch eine Architektur und prototypische Implementierung einer Entwicklungsumgebung sowie zugehöriger Laufzeitumgebungen validiert.

Abstract

The Web Service Business Process Execution Language (BPEL) enables process oriented integration of heterogeneous applications that provide their functionality via Web Services (WSs). BPEL defines the control flow as well as the data flow between activities that are linked to Web Service operations. Since activities only refer to operations, i.e. to the abstract definition of the interface a service has to provide, BPEL supports late binding of activity implementations which enables selecting services from different providers during runtime. However, different organizations or enterprises provide their functionality usually via different interfaces, i.e. using different operations. Therefore, process models need to be adapted every time the process wants to make use of a service from a different service provider. This is expensive and restricts the flexibility of an enterprise tremendously.

Defining activities by referring to WSDL operations does not only result in limitied flexibility of BPEL regarding the selection of activity implementations. It also restricts BPEL to WSDL services for implementing activties, and limits an activity to exchange at most two messages with a business partner. However, activities are meant to represent business functions, and business functions can require exchanging multiple messages with several business partners. Hence, the ability to exchange at most two messages with one business partner with an activity is not adequate.

This thesis follows the idea of a WSDL-less BPEL to resolve the above mentioned issues of BPEL. It introduces a new BPEL-based workflow language called BPEL^{light}. BPEL^{light} defines an interaction model that is independent of WSDL and completely decouples process logic and interface specification of activity implementations. In contrast to the WSDL-based interaction model of BPEL that only enables the definition of bilateral interactions of processes with business partners, the interaction model of BPEL^{light} supports modelling multilateral interactions. Furthermore, BPEL^{light} allows defining the operational semantics of interaction activities via so called message exchange patterns instead of providing a fixed set of

interaction activities. This way, BPEL^{light} enables implementing complex business functions that require a complex message exchange via a single interaction activity. In addition to defining the language BPEL^{light}, the thesis shows how WS technology and approaches towards semantically describing WSs (semantic WSs, SWSs) can be used to find and bind implementations for BPEL^{light} activities. It also presents the architecture and a prototypical implementation of a BPEL^{light} infrastructure. The infrastructure facilitates modelling and execution of BPEL^{light} processes, and is capable of using WS technology and SWS technology to find and bind activitiy implementations.

Inhaltsverzeichnis

ABBILDUNGSVERZEICHNIS 15

LISTINGSVERZEICHNIS 19

TABELLENVERZEICHNIS 23

ABKÜRZUNGSVERZEICHNIS 25

KAPITEL 1 EINLEITUNG 29

1.1 Beiträge der Arbeit 33

1.2 Aufbau 36

KAPITEL 2 GRUNDLAGEN UND VERWANDTE ARBEITEN 41

2.1 Web-Services 41

2.1.1 Web Service Description Language (WSDL) 43

2.1.1.1 Interoperabilität 46

2.1.1.2 Beispiel: Onlineversandhandeldienst 47

2.1.2 Business Process Execution Language (BPEL) 48

2.1.2.1 Unterstützte Interaktionsszenarien von BPEL 54

2.1.2.2 BPEL-SPE (Extension for Sub-processes) – BPEL-Erweiterung zur Unterstützung von Subprozessen 60

2.1.2.3 Beispiel: BPEL-Prozess als Klient eines Onlineversandhandeldienstes 61

2.2 Semantische Web-Services 64

2.2.1 OWL-S 71

2.2.2 WSMO 76

2.2.3 SAWSDL 84

2.3 Prozessflexibilität 86

2.4 Zusammenfassung 90

KAPITEL 3 BPEL^{LIGHT} – EINE VON WSDL UNABHÄNGIGE PROZESSAUSFÜHRUNGSSPRACHE 93

3.1 BPEL^{light} zur Spezifikation ausführbarer Geschäftsprozesse 101

3.1.1 Das Rahmenwerk: Konversationen mit abstrakten Partnern..... 102

3.1.2 Die Definition des Nachrichtenaustauschs 103

3.1.3 Fault Handling in BPEL^{light} 110

3.1.4 Korrelieren von Nachrichten und Prozessinstanzen 112

3.1.5 Zwischenbilanz 114

3.1.6 Konversationen mit mehreren Instanzen des gleichen Partnertyps 116

3.2 BPEL^{light} zur Beschreibung von Nachrichtenaustausch-mustern 120

3.3 Nachrichtenaustauschmuster zur Beschreibung des Verhaltens von BPEL^{light} Interaktionsaktivitäten 131

3.3.1 Interaktionsaktivitäten mit komplexem Verhalten zur Definition höherwertiger Nachrichtenaustauschmuster..... 139

3.3.2 Interaktionsaktivitäten mit komplexem Verhalten in ausführbaren Prozessen..... 142

3.4 Nachrichtenaustauschmuster zur Kategorisierung von Prozessfragmenten 144

3.5 Beispiel: BPEL^{light}-Prozess als Klient eines Onlineversandhandeldienstes 147

3.6 Methodologie: Transformationen zwischen WS-BPEL und BPEL^{light} 149

3.6.1 Von WS-BPEL zu BPEL^{light} 149

3.6.2 Von BPEL^{light} zu WS-BPEL 156

3.7 Zusammenfassung 158

KAPITEL 4 BPEL^{LIGHT} MIT WEB-SERVICES 161

4.1 Web-Services als Aktivitätsimplementierungen.....162

4.2 Unterstützte Interaktionsszenarien des Gespanns BPEL^{light} mit Web-Services.....167

4.3 Beispiel: BPEL^{light}-Prozess als Klient eines mit WSDL beschriebenen Onlineversandhandeldienstes 177

4.4 Methodologie: Transformationen zwischen WS-BPEL und BPEL^{light} mit Web-Services als Aktivitätsimplementierungen...179

4.4.1 Von WS-BPEL nach BPEL^{light} mit Web-Services als Aktivitätsimplementierungen 179

4.4.2 Von BPEL^{light} mit Web-Services als Aktivitätsimplementierung zu WS-BPEL 181

4.5 Zusammenfassung 181

KAPITEL 5 BPEL^{LIGHT} MIT (SEMANTISCHEN) WEB-SERVICES – BPEL4SWS 183

5.1 Semantische Web-Services für lang laufende Prozesse 185

5.2 Semantische Web-Services als Aktivitätsimplementierungen.192

5.3 Unterstützte Interaktionsszenarien von BPEL4SWS 200

5.4 Beispiel: BPEL4SWS-Prozess als Klient eines semantisch beschriebenen Onlineversandhandeldienstes 213

5.5 Ontologiewissen über Daten in BPEL4SWS 215

5.5.1 Vereinfachung der Datenmanipulation auf Basis ontologischen Wissens 215

5.5.2 Entscheidungen im Kontrollfluss auf Basis ontologischen Wissens 223

5.6 Methodologie: Transformationen zwischen WS-BPEL und BPEL4SWS.....226

5.6.1 Von WS-BPEL zu BPEL4SWS..... 226

5.6.2 Von BPEL4SWS zu WS-BPEL..... 227

5.7 Zusammenfassung 228

KAPITEL 6 ARCHITEKTUR UND IMPLEMENTIERUNG 233

6.1 Ausführungsumgebung für BPEL^{light}-Prozesse.....235

6.1.1 Apache ODE 237

6.1.1.1 ODE Architektur 238

6.1.1.2 Erweiterbarkeit von ODE 242

6.1.2 Erweiterung von Apache ODE zu einer BPEL^{light}-Engine..... 243

6.1.2.1 Umsetzung der BPEL^{light}-Aktivitäten..... 244

6.1.2.2 Umsetzung der Erweiterungen `<bl:partner>` und `<bl:conversation>` 260

6.1.3 WSDL 1.1-Integrationsschicht – BPEL^{light}-Ausführung in Web-Service-Umgebungen 264

6.1.4 WSMO-Integrationsschicht – BPEL^{light}-Ausführung in Semantischen Web-Service-Umgebungen..... 266

6.2 Modellierungsumgebung für BPEL^{light}-Prozesse.....271

6.2.1 Einführung in den Eclipse BPEL-Designer..... 272

6.2.1.1 Architektur des BPEL-Editors 273

6.2.1.2 Grafischer Aufbau des BPEL-Editors 275

6.2.2 Erweiterung des Eclipse BPEL-Designers zu einem BPEL^{light}-Editor..... 277

6.2.2.1 Erstellen eines BPEL^{light} Projektes 277

6.2.2.2 Erstellen eines BPEL^{light}-Prozesses..... 278

6.2.2.3 Erweiterung des Datenmodells 278

6.2.2.4 Einlesen und Schreiben einer BPEL-Datei 280

6.2.2.5 Erweiterung der Implementierung um `<bl:partner>`, `<bl:conversation>` und `<bl:expression>`..... 281

6.2.2.6 Erweiterung der Implementierung um `<bl:interactionActivity>` und `<bl:pick>`..... 283

6.2.3 Editor zur Abbildung auf WSDL 1.1..... 287

6.2.4 BPEL4SWS-Editor – Abbildung von BPEL^{light} auf Semantische Web-Services 288

6.3 Zusammenfassung 289

KAPITEL 7 ZUSAMMENFASSUNG UND AUSBLICK 291

LITERATURVERZEICHNIS 295

ANHANG 309

DANKSAGUNGEN 323

Abbildungsverzeichnis

Abbildung 1: SOA-Dreieck 29
Abbildung 2: Evolution der Anwendungsentwicklung [LLN11] 30
Abbildung 3: Aufbau der Arbeit 37
Abbildung 4: Web-Services-Schichtenmodell [WCL+05] 42
Abbildung 5: Beziehungen der WSDL-Bausteine 44
Abbildung 6: Visualisierung der Operationstypen 45
Abbildung 7: Dimensionen eines Workflows (links) und eines Web-Service-Flows (rechts) 49
Abbildung 8: Die WSDL-Erweiterung `<partnerLinkType>` 51
Abbildung 9: Spezifikation einer `<invoke>`-Aktivität mittels WSDL 53
Abbildung 10: Aufruf eines kurzlebigen BPEL-Prozesses 55
Abbildung 11: Blockierender Aufruf eines Dienstes 56
Abbildung 12: Aufruf eines langlebigen BPEL-Prozesses 57
Abbildung 13: Nicht blockierender Aufruf eines Dienstes 59
Abbildung 14: BPEL-Prozesses als Klient eines Onlineversandhandel-dienstes 61
Abbildung 15: Vom World Wide Web über Web-Services und dem Semantischen Web hin zu Semantischen Web-Services [DOS03] 65
Abbildung 16: RDF Aussage „Harry Potter wurde geschrieben von Joanne K. Rowling“ 67
Abbildung 17: Die WSML-Sprachfamilie [Br08]; innen als Schichtenmodell; außen entlang der Dimensionen aufgespannt 69
Abbildung 18: Auszug der WSML-Syntax in BNF-Schreibweise 70
Abbildung 19: Definition des Konzepts „book“ in WSML 71
Abbildung 20: OWL-S Modell zur Beschreibung von Web-Services [MBH+04] 72
Abbildung 21: WSML-Syntax zur Definition von Zielen, Diensten und Mediatoren in BNF 78

Abbildung 22: Schematische Darstellung der Ausführungssemantik „achieveGoal“ 80
Abbildung 23: Entkopplung der Prozesslogik von der Schnittstellenbeschreibungssprache 99
Abbildung 24: Einzelner Aktivitätstyp zum Senden oder Versenden von Nachrichten 104
Abbildung 25: Auswahl der zu verwendenden Partnerinstanz 117
Abbildung 26: Beschreibung eines Xors mit BPEL^{light} 124
Abbildung 27: Nachrichtenaustauschmuster „Request with Referral“ 129
Abbildung 28: `<bl:interactionActivity>` deren Semantik durch ein Nachrichtenaustauschmuster bestimmt wird 132
Abbildung 29: Parametrisierbares Nachrichtenaustauschmuster 133
Abbildung 30: `<bl:interactionActivity>` deren Semantik durch das Nachrichtenaustauschmuster „Request with Referral“ bestimmt wird 135
Abbildung 31: Rekursive Definition eines „Request for Bid“s 140
Abbildung 32: Transformation einer One-way `<invoke>`-Aktivität zu BPEL^{light} 152
Abbildung 33: Transformation einer Request-Response `<invoke>`-Aktivität zu BPEL^{light} 153
Abbildung 34: Transformation einer `<receive>`-Aktivität, die eine One-way-Operation implementiert, zu BPEL^{light} 154
Abbildung 35: Transformation eines `<receive>`-`<reply>`-Paares zu BPEL^{light} 155
Abbildung 36: Abbildung von Konversationen, Partnern und Nachrichten auf entsprechende WSDL-Elemente 164
Abbildung 37: Aufruf eines kurzlebigen BPEL^{light}-Prozesses 168
Abbildung 38: Aufruf eines synchronen WSDL-Dienstes 170
Abbildung 39: Aufruf eines langlebigen BPEL^{light}-Prozesses 172
Abbildung 40: Aufruf eines asynchronen WSDL-Dienstes 175

Abbildung 41: BPEL^{light}-Prozess als Klient eines mit WSDL beschriebenen Onlineversandhandeldienstes 177
Abbildung 42: Transformation einer `<bpel:invoke>`-Aktivität zu einer `<bl:interactionActivity>`, die einen Web-Service über die gleiche Operation aufruft 180
Abbildung 43: „Dynamischer Partnerlinktyp“ durch WSMO-Goal und -Web-Service 188
Abbildung 44: Mit WSMO-Konzepten annotierter BPEL^{light}-Prozess 194
Abbildung 45: Aufruf eines kurzlebigen BPEL4SWS-Prozesses 201
Abbildung 46: Aufruf eines kurzlebigen Semantischen Web-Services 204
Abbildung 47: Aufruf eines langlebigen BPEL4SWS-Prozesses 207
Abbildung 48: Aufruf eines langlebigen Semantischen Web-Services 211
Abbildung 49: BPEL4SWS-Prozess als Klient eines mit WSMO beschriebenen Onlineversandhandeldienstes 214
Abbildung 50: Datenmanipulation in BPEL 216
Abbildung 51: Gesamtarchitektur 234
Abbildung 52: Allgemeine Architektur einer BPEL^{light}-Engine 235
Abbildung 53: Deploymenteinheit für ODE 237
Abbildung 54: Architektur von ODE 238
Abbildung 55: Deploymenteinheit für die BPEL^{light}-Engine 243
Abbildung 56: Klassenstruktur des ExtensionBundles „bpel-light“ [So08] 245
Abbildung 57: Parametrisierung eines MEP-Prozesses 251
Abbildung 58: Beispiel für eine entpackende `<bpel:assign>`-Aktivität 256
Abbildung 59: Beispiel für eine zusammenpackende `<bpel:assign>`-Aktivität 258
Abbildung 60: Klassendiagramm der implementierten BPEL-Erweiterungen im BOM 261
Abbildung 61: Klassendiagramm des erweiterten O-Models 262
Abbildung 62: Klassendiagramm der erweiterten Ausführungsumgebung 264
Abbildung 63: BPEL^{light}-Prozess als Klient in einer Web-Service-Ausführungsumgebung 266

Abbildung 64: BPEL^{light}-Prozess als Klient in einer Semantischen Web-Service-Ausführungsumgebung 270
Abbildung 65: Architektur des BPEL-Editors 274
Abbildung 66: Grafischer Aufbau des BPEL-Editors 276
Abbildung 67: Klassenmodell des BPEL^{light}-Editors [Ho08] 279
Abbildung 68: Ablauf der Deserialisierung einer `<bpel:extensionActivity>` [Hoeh08] 281
Abbildung 69: Grafische Darstellung der für den Prozess „buyProcess“ definierten BPEL^{light}-Konstrukte 282
Abbildung 70: Grafische Darstellung der `<bl:pick>`-Aktivität 284
Abbildung 71: Grafische Darstellung der `<bl:interactionActivity>` ... 285
Abbildung 72: Modellierungsumgebung zum Einsatz von BPEL^{light}-Prozessen in Web-Service-Umgebungen 287
Abbildung 73: Modellierungsumgebung zum Einsatz von BPEL^{light}-Prozessen in Semantischen Web-Service-Ausführungsumgebungen 288

Listingsverzeichnis

Listing 1: WSDL-Beschreibung des Onlineversandhandeldienstes 48
Listing 2: BPEL-Prozess als Nutzer des Onlineversandhandeldienstes 62
Listing 3: WSML-Beschreibung des Onlineversandhandeldienstes 83
Listing 4: SAWSDL-Beschreibung des Onlineversandhandeldienstes 86
Listing 5: Schema der mit BPEL^{light}-Elementen erweiterten <bpel:process>-Definition 103
Listing 6: Schema der `<bl:interactionActivity>` 105
Listing 7: Schema der `<bl:pick>`-Aktivität 107
Listing 8: Schema der mit einem `<bl:eventHandlers>`-Element erweiterten `<bpel:process>`-Definition 109
Listing 9: Fehlernachrichten senden bzw. empfangen: Schema der Aktivitäten `<bl:interactionActivity>` und `<bl:pick>` 111
Listing 10: Korrelieren von Nachrichten und Prozessinstanzen: Schema des `<bl:input>`-Elements der `<bl:interactionActivity>` 113
Listing 11: Szenario „Request with Referral" in BPEL^{light} 115
Listing 12: Schema des `partnerInstances`-Containers 117
Listing 13: Szenario „Request for Bid" in BPEL^{light} 119
Listing 14: Template für WSDL 2.0-Nachrichtenaustauschmuster [CHL+07] 120
Listing 15: Schema der `<bl:expression>` 124
Listing 16: Verwendung der `<bl:expression>` zur Definition eines Xor-Verhaltens 125
Listing 17: Schema des `<bl:onAlarm>`-Elements sowie der `<bl:wait>`-Aktivität 127
Listing 18: Nachrichtenaustauschmuster „Request with Referral" als abstrakter BPEL^{light}-Prozess 130

Listing 19: Schema der `<bl:interactionActivity>`: Elemente und Attribute zur Referenzierung parametrisierter Größen in Nachrichtenaustauschmustern 137
Listing 20: `<interactionActivity>` deren Semantik durch das Nachrichtenaustauschmuster „Request with Referral“ bestimmt wird..... 138
Listing 21: Schema des `<bl:timingExpression>`-Elements der `<bl:interactionActivity>` 143
Listing 22: `<bl:interactionActivity>` in einem konkreten Prozess deren Semantik durch das Nachrichtenaustauschmuster „Request with Referral“ bestimmt wird 144
Listing 23: Vollständiges Schema der `<bl:conversation>` 146
Listing 24: BPEL^{light}-Prozess als Klient eines Onlineversandhandel-dienstes 148
Listing 25: Schema des Deployment Deskriptors für BPEL^{light} mit Web-Services 166
Listing 26: Prozessbeschreibung und Deployment Deskriptor für den kurzlebigen, synchron aufrufbaren BPEL^{light}-Prozess *„performanceReviewHandlingProcess“* 169
Listing 27: Prozessbeschreibung und Deployment Deskriptor für den synchron aufrufenden BPEL^{light}-Prozess *„performanceReviewRequestProcess“* 171
Listing 28: Prozessbeschreibung und Deployment Deskriptor für den asynchron aufrufbaren BPEL^{light}-Prozess *„bookingHandlingProcess“*..... 173
Listing 29: Prozessbeschreibung und Deployment Deskriptor für den asynchron aufrufenden BPEL^{light}-Prozess *„bookingProcess“*.... 176
Listing 30: Deployment Deskriptor für einen BPEL^{light}-Prozess als Klient eines durch WSDL beschriebenen Onlineversandhandeldienstes..... 178

Listing 31: WSML-Beschreibung eines WSMO-Goals zur Bestellung bei einem Onlineversandhandeldienst 189
Listing 32: Erweitertes Schema des Deployment Deskriptors zur Annotation von Prozessen mit semantischen Beschreibungen 195
Listing 33: Erweitertes Schema des Deployment Deskriptors zur partiellen Abbildung von Prozessen auf Web-Service-Technologie 197
Listing 34: Erweitertes Schema des Deployment Deskriptors zur semantischen Abbildung auf Partnerinstanzen 199
Listing 35: Deployment Deskriptor und WSMO-Web-Service-Beschreibung für den kurzlebigen, synchron aufrufbaren BPEL4SWS-Prozess *„performanceReviewHandlingProcess“* 203
Listing 36: Deployment Deskriptor und WSMO-Goal für den synchron aufrufenden BPEL4SWS-Prozess *„performanceReviewRequestProcess“* 206
Listing 37: Deployment Deskriptor und WSMO-Web-Service für den asynchron aufrufbaren BPEL4SWS-Prozess *„bookingHandlingProcess“* 209
Listing 38: Deployment Deskriptor und WSMO-Goal für den asynchron aufrufenden BPEL4SWS-Prozess *„bookingProcess“* 212
Listing 39: Deployment Deskriptor für einen BPEL4SWS-Prozess, der Anforderungen an einen Onlineversandhandeldienst über ein WSMO-Goal stellt 213
Listing 40: Extrahierung der Bankdaten über herkömmliches Assignment 217
Listing 41: Schema der Erweiterung `<b4s:mediate>` 219
Listing 42: Die „onlineshop“-Ontologie 220
Listing 43: Die „banking“-Ontologie 220
Listing 44: Extrahierung der Bankdaten über Mediation 221
Listing 45: Mediation mit WSML 221

Listing 46: Syntax von WSML4BPEL in BNF 223
Listing 47: BPEL4SWS-Prozess mit in WSML formulierten logischen Ausdrücken zur Spezifikation von `transitionConditions` 225
Listing 48: XSD für den Request von einem Hauptprozess an einen MEP-Prozess 253
Listing 49: XSD für die Response von einem MEP-Prozess an einen Hauptprozess 254

Tabellenverzeichnis

Tabelle 1: Vergleich von WS-BPEL und BPEL^{light} 159
Tabelle 3: Vergleich von WS-BPEL und BPEL4SWS 229

Abkürzungsverzeichnis

API	Application Programming Interface
B2B	Business to Business (Integration)
BNF	Backus-Naur-Form
BOM	Business Process Execution Language Object Model
BPEL	Business Process Execution Language
BPEL4WS	Business Process Execution Language for Web Services
BPEL4SWS	Business Process Execution Language for Semantic Web Services
BPEL-SPE	Business Process Execution Language – Extension for Sub-processes
DAO	Data Access Objects
DBMS	Datenbankmanagementsystem
DD	Deployment Deskriptor
EAI	Enterprise Application Integration
EMF	Eclipse Modeling Framework
EPR	Endpoint Reference
ESB	Enterprise Service Bus
GEF	Graphical Editing Framework
GPL	Gnu General Public License
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol

IDL	Interface Definition Language
JACOB	Java Concurrent Objects
JBI	Java Business Integration
JMS	Java Message Service
JPA	Java Persistance API
JSP	Java Server Pages
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
RMI/IIOP	Remote Method Invocation over Internet Inter-Orb Protocol
MDD	Model Driven Development
MEP	Message Exchange Pattern
MVC	Model View Controller
REST	Representational State Transfer
SAWSDL	Semantic Annotations for WSDL and XML Schema Definition
SMTP	Simple Mail Transfer Protocol
SSB	Semantischer Service Bus
SOA	Service-orientierte Architektur
OASIS	Organization for the Advancement of Structured Information Standards
ODE	Orchestration Director Engine
OWL	Web Ontology Language
OWL-S	Web Ontology Language for Services

TCP/IP	Transmission Control Protocol/Internet Protocol
UDDI	Universal Description, Discovery and Integration
URI	Universal Resource Identifier
W3C	World Wide Web Consortium
WAR	Web Archive
WfMS	Workflow-Management-System
WS	Web-Services
WS-AT	Web Services Atomic Transaction
WS-BA	Web Services Business Activity
WS-BPEL	Web Services Business Process Execution Language
WSFL	Web Services Flow Language
WS-I	Web Services Interoperability
WS-MEX	Web Service Metadata Exchange
WSML	Web Service Modelling Language
WSMO	Web Service Modelling Ontology
WS-N	Web Services Notification
WS-RF	Web Services Resource Framework
WSDL	Web Service Description Language
XLANG	Web Services for Business Process Design
XML	Extensible Markup Language
XSD	XML Schema Definition

KAPITEL 1

EINLEITUNG

Geschäftsprozessmanagement [We07] im Allgemeinen und Workflowtechnologie [LeRo00] im Besonderen sind in den letzten Jahren verstärkt in den Fokus von Forschung und Industrie gerückt. Die Workflowtechnologie wurde bereits seit den achtziger Jahren entwickelt und Prozessorientierung als Paradigma schon über Jahre diskutiert. Erst jedoch mit dem Aufkommen der Web-Service-Technologie hat sich Workflowtechnologie in der Wirtschaft nachhaltig etabliert und findet auch in der Forschung immer mehr Beachtung.

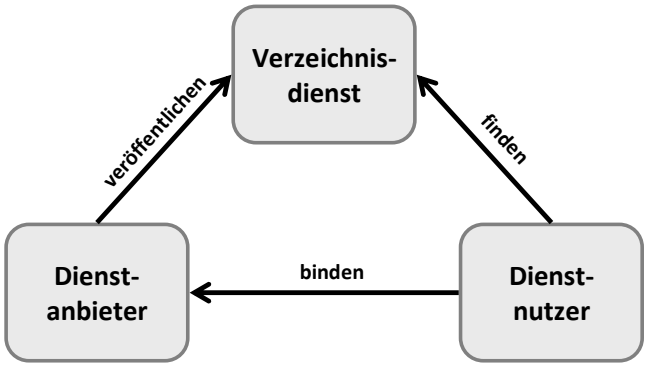


Abbildung 1: SOA-Dreieck

Die Web-Service-Technologie ist aufgrund der Notwendigkeit, heterogene Systeme zu integrieren, entwickelt worden. Sie ist die am Weitesten verbreitete Implementierung der Service-orientierten Architektur (SOA) [Bu00] und stellt einzelne Teile einer Anwendung als plattformunabhängigen Dienst über ein Netzwerk zur Verfügung. Der Aufbau und die Funktionsweise der SOA ist in Abbildung 1 dargestellt: Dienstanbieter *veröffentlichen* ihre Dienste durch Registrierung bei einem Verzeichnisdienst. Dienstnutzer *finden* über den Verzeichnisdienst geeignete Dienste auf und *binden* gegen diese. Auf diese Weise unterstützen

Web-Services die Entwicklung lose gekoppelter, verteilter Systeme in einer heterogenen Umgebung.

Anwendungen, welche die Geschäftstätigkeit eines Unternehmens unterstützen, nutzen über eine *Prozesslogik* verbundene *Geschäfts-funktionen* zur Verarbeitung von Geschäftsdaten [LeRo00]. In der Vergangenheit wurde die Prozesslogik oft in der gleichen Sprache wie die Geschäftsfunktionen realisiert (z.B. Java). Außerdem wurden beide Bestandteile miteinander verflochten. Workflowtechnologie ermöglicht es, die Prozesslogik von der Implementierung der Geschäftsfunktionen zu trennen, also eine *Flussunabhängigkeit* von Geschäftsfunktionen zu erreichen (Abbildung 2). Die Prozesslogik wird dazu in *Prozessmodellen* spezifiziert, deren Ausführung durch so genannte Workflowmanagement-systeme (WfMS) ermöglicht wird. Wird ein Prozessmodell ausgeführt, so spricht man von einer *Prozessinstanz*. Im Folgenden wird immer dann von einem Modell bzw. von einer Instanz gesprochen, wenn die Unterscheidung von Bedeutung ist. Ansonsten wird der allgemeine Begriff Prozess verwendet. Für die Spezifikation der Prozessmodelle existieren eigens entwickelte Workflowsprachen. Diese Workflowsprachen ermöglichen es, Kontroll- und Datenfluss zwischen einzelnen *Aktivitäten* zu definieren, welche die entsprechenden Geschäftsfunktionen aufrufen. Workflowtechnologie ermöglicht also Programmieren auf einer höheren, *Prozess-orientierten* Ebene. Sie wird nach der Einführung von Datenbankmanagementsystemen (DBMS) und der dadurch erreichten *Datenunabhängigkeit* von Anwendungen als ein weiterer evolutionärer Schritt in der Anwendungsentwicklung betrachtet [LeRo00].

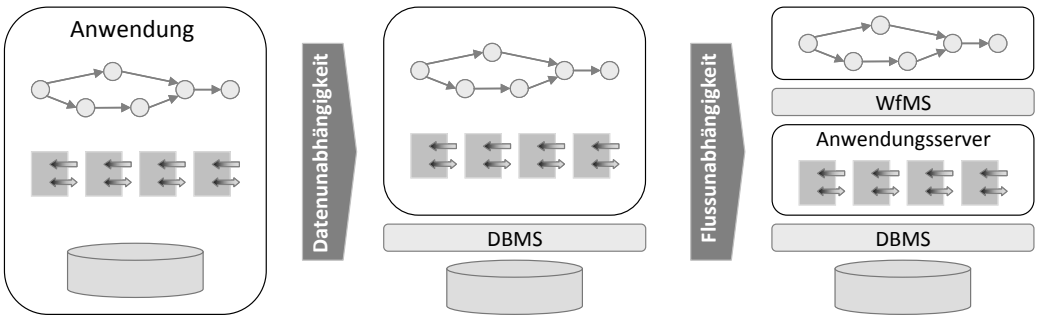


Abbildung 2: Evolution der Anwendungsentwicklung [LLN11]

Bis 2002 gab es keine von der Industrie allgemein anerkannten Standards zur Definition von Workflows. Das führte dazu, dass unterschiedliche Anbieter von Workflowprodukten jeweils unterschiedliche Repräsentationen von Workflows verwendeten. Im Jahr 2002 wurde von BEA Systems, IBM und Microsoft die Spezifikation der Business Process Execution Language (BPEL) [AAA+07] in der Version 1.0 veröffentlicht, jener Sprache, die sich in der Folgezeit zum einzigen allgemein in der Industrie anerkannten Standard im Bereich der Geschäftsprozessausführung entwickelt hat und bis zum heutigen Tage geblieben ist.

Die Sprache BPEL ermöglicht es, Prozesse als eine *Komposition* von Web-Services zu spezifizieren; sie verfolgt einen Workflow-basierten Ansatz zur Integration heterogener Systeme. BPEL verwendet die Web Service Description Language (WSDL 1.1) [CCMW01] zur Beschreibung der *Aktivitätsimplementierungen*. WSDL 1.1 definiert vier Operationstypen, um die Kommunikation zwischen einem Dienst und einem Dienstnutzer zu beschreiben. Diesen vier Operationstypen entsprechend bietet BPEL einen fixen Satz von *Interaktionsaktivitäten* zur Kommunikation mit Diensten, die Geschäftsfunktionen realisieren. In BPEL wird eine *lose Kopplung* der Prozesslogik von der Aktivitätsimplementierung erreicht, indem lediglich die Anforderungen einer Aktivität an deren Implementierung in Form von WSDL-Porttyp und –Operation spezifiziert werden, anstatt eine konkrete Implementierung eines Web-Services innerhalb der Prozessbeschreibung als Aktivitätsimplementierung zu binden. Dies ermöglicht es, prinzipiell zur Laufzeit eines Prozesses immer den am besten geeigneten Dienst auszuwählen.

Der Grad der Entkopplung von Prozesslogik und Aktivitätsimplementierung in BPEL ist jedoch nicht ausreichend: Dienste mit gleicher Funktionalität operieren im Allgemeinen auf den gleichen Daten, d.h. sie verarbeiten die gleichen Informationen. Das bedeutet allerdings nicht, dass die Daten zwangsläufig auch gleich strukturiert sind; standardisierte Nachrichtentypen sind noch nicht in allen Domänen vorhanden. Noch unwahrscheinlicher ist es, dass unterschiedliche Dienste gleicher Funktion identische Schnittstellen implementieren, da dies nicht nur die Standardisierung von Nachrichtentypen voraussetzt, sondern auch deren

Gruppierung zu Operationen und Porttypen. Wird eine Einigung auf Nachrichtentypen von der Industrie noch als möglich erachtet, so gilt die Standardisierung von Porttypen und Operationen als zu ambitioniert und nicht erreichbar. BPEL bindet also Aktivitäten zwar nicht an konkrete Implementierungen, schränkt aber durch die zwingende Angabe von Porttypen und Operationen die möglichen Implementierungen von Aktivitäten auf jene Dienste ein, die den entsprechenden Porttypen realisieren. Alle funktional passenden Dienste, die andere Porttypen implementieren, kommen nicht als Aktivitätsimplementierungen in Betracht. Die enge Kopplung von BPEL und WSDL-Schnittstellenspezifikationen schränkt daher die Flexibilität bezüglich der Wahl der zu verwendenden Dienste ein [KLNW06]. Eine weitere Beschränkung, die sich aus der Verwendung von WSDL-Operationen zur Definition der Interaktionsaktivitäten ergibt, ist, dass eine Aktivität maximal zwei Nachrichten mit einem Geschäftspartner austauschen kann. Zur Realisierung von Geschäftsfunktionen ist dies jedoch nicht ausreichend, da komplexe Geschäftsfunktionen den Austausch mehrerer Nachrichten mit mehreren Geschäftpartnern erfordern können. Auch der Grad der Wiederverwendbarkeit eines Prozesses oder eines Teils eines Prozesses sinkt durch die Kopplung von Aktivitäten und Porttypen: Die Definition des Flusses zwischen Aktivitäten kann nicht isoliert wieder verwendet werden, sondern nur in Kombination mit den Aktivitätsbeschreibungen in Form von Porttyp und Operation. Durch die Abhängigkeit von WSDL ist BPEL außerdem lediglich in Web-Service-Umgebungen einsetzbar. Andere Sprachen zur Beschreibung der Anforderungen von Aktivitäten an ihre Implementierung, wie beispielsweise Sprachen die zur Beschreibung Semantischer Web-Services (SWS) [CaSe06] verwendet werden, sind per Definition ausgeschlossen. Ein weiterer Nachteil der Kopplung von Prozesslogik und Schnittstellenbeschreibung in BPEL ist, dass fundiertes Wissen über Prozessmodellierung nicht ausreicht, um BPEL-Prozesse modellieren zu können, sondern auch detaillierte Kenntnisse über Web-Services notwendig sind.

Die bereits angesprochenen SWS sind eine weitere Realisierung der SOA und verfolgen einen anderen Ansatz um das Prinzip der losen Kopplung

umzusetzen als die Web-Service-Technologie: Bei Web-Services spezifizieren Dienstnutzer die Anforderungen an die zu verwendenden Dienste über den zu implementierenden Porttypen. Im Gegensatz dazu erfassen SWS die Bedeutung der zu kommunizierenden Daten mittels *Ontologien* [Fe01] und verwenden formale Sprachen, um die Vorbedingungen und die Auswirkungen eines Dienstaufrufs maschinen-verständlich zu beschreiben. Somit ermöglichen es SWS, dass Dienstnutzer nicht die Signatur eines zu verwendenden Dienstes in Form von Porttyp und Operation angeben müssen, sondern stattdessen lediglich ihre Anforderungen an einen Dienst mit Hilfe von SWS-Konzepten erfassen müssen. Diese können dann zum Auffinden funktional geeigneter Dienste verwendet werden.

1.1 Beiträge der Arbeit

Der Fokus dieser Arbeit liegt in der Flexibilisierung von BPEL-Prozessmodellen durch Separation von Prozesslogik und Schnittstellenspezifikation.

Beitrag 1: WSDL unabhängiges BPEL

Zunächst wird das Konzept einer von WSDL unabhängigen BPEL-basierten Workflowsprache eingeführt. Anschließend werden alternative Ansätze zur Definition einer solchen Sprache vorgestellt, analysiert und bewertet. Der Erfolg versprechendste Ansatz wird dann in dieser Arbeit im Detail ausgearbeitet ($BPEL^{light}$).

Beitrag 2: Spezifikation von $BPEL^{light}$

$BPEL^{light}$ entkoppelt die Prozesslogik vollständig von der Schnittstellendefinition der Implementierung der Aktivitäten. Darüber hinaus ist $BPEL^{light}$ unabhängig vom konkreten Realisierungsansatz der SOA (z. B. SWS, WS). $BPEL^{light}$ wird als Erweiterung zu BPEL realisiert und nutzt dabei die von BPEL definierten Erweiterungspunkte. In der vorliegenden Arbeit werden die Syntax und die operationale Semantik der entsprechenden BPEL-Erweiterung präsentiert. Die im BPEL-Standard

definierte operationale Semantik bleibt von den vorgestellten Erweiterungen unberührt.

Beitrag 3: Unterstützung multilateraler Konversationen in BPEL^{light}

BPEL ermöglicht es, bilaterale Interaktionen zu definieren. BPEL^{light} ermöglicht darüber hinaus, den logischen Zusammenhang mehrerer bilateraler Interaktionen zu erfassen. BPEL^{light} ist damit nicht wie BPEL beschränkt auf die Definition einer Ansammlung bilateraler Interaktionen, sondern ermöglicht die explizite Definition multilateraler Interaktionen.

Beitrag 4: Unterstützung beliebiger Nachrichtenaustauschmuster in BPEL^{light}

Darüber hinaus definiert BPEL^{light} nicht wie herkömmliches BPEL einen den WSDL 1.1-Operationstypen entsprechenden fixen Satz von *Interaktions-aktivitäten* [WCL+05], sondern erlaubt es, die Semantik von Interaktions-aktivitäten mittels so genannter Nachrichtenaustauschmuster (engl. *Message Exchange Pattern*, MEPs) zu spezifizieren, um dem Umstand gerecht zu werden, dass komplexe Geschäftsfunktionen auch einen komplexen Nachrichtenaustausch erfordern können. Die Definition des Profils für abstraktes BPEL^{light}, welches es ermöglicht Nachrichtenaustauschmuster und damit wieder verwendbare Aktivitätsbeschreibungen zu erstellen, ist ein weiterer Beitrag der Arbeit.

Beitrag 5: Anwendbarkeit von BPEL^{light} in Web-Service-Umgebungen

Da Web-Services zurzeit eine populäre Technologie zur Integration verteilter Anwendungen sind, zeigt diese Arbeit auch, wie Web-Services verwendet werden können, um Aktivitäten von BPEL^{light}-Prozessen zu implementieren und wie der BPEL^{light}-Prozess selbst als Web-Service zur Verfügung gestellt werden kann.

Beitrag 6: Möglichkeit zur transparenten Migration von BPEL zu BPEL^{light}

Hierbei wird sichergestellt, dass alle durch BPEL unterstützten Szenarien, d.h. sowohl synchrone als auch asynchrone Kommunikation, auch durch BPEL^{light} unterstützt werden. Weiterhin wird gewährleistet, dass eine Migration von BPEL-Prozessmodellen zu BPEL^{light}-Prozessmodellen

transparent für Nutzer des vom Prozess angebotenen Dienstes erfolgen kann.

Beitrag 7: Spezifikation von BPEL für Semantische Web-Services (BPEL4SWS)

Ein weiterer Beitrag der Arbeit ist die Verwendung von Semantischen Web-Services als Aktivitätsimplementierung von BPEL^{light}-Prozessen. Hierfür wird zunächst gezeigt, wie das SWS-Framework WSMO dahingehend erweitert werden kann, dass es in der Lage ist, für Geschäftsprozesse typische, lang laufende, asynchrone Kommunikation zu unterstützen. Anschließend werden Mechanismen definiert, die es ermöglichen, SWS zu verwenden, um sowohl die Funktionalität des Prozesses als auch die Anforderungen der Aktivitäten an entsprechende Implementierungen zu beschreiben. Die auf BPEL^{light} basierende Sprache zur Komposition von SWS trägt den Namen BPEL für Semantische Web-Services (BPEL4SWS).

Beitrag 8: Möglichkeit zur transparenten Migration von BPEL zu BPEL4SWS

Der modulare Aufbau der Sprache BPEL4SWS ermöglicht einen Prozess gleichzeitig als Web-Service und auch als SWS zu repräsentieren, so dass sowohl Web-Service-Technologien als auch SWS-Technologien dazu verwendet werden können, den Prozess aufzufinden und mit ihm zu interagieren. Das hat zur Folge, dass bei einer Umstellung von existierenden BPEL-Prozessen auf BPEL4SWS-Prozesse die ursprüngliche WSDL-Schnittstelle beibehalten werden kann. Für existierende Nutzer des vom Prozess angebotenen Dienstes ist die Umstellung von BPEL zu BPEL4SWS somit transparent.

Beitrag 9: Architektur und Prototypische Realisierung

Abschließend wird in dieser Arbeit die Architektur und prototypische Implementierung einer Entwicklungsumgebung sowie einer zugehörigen Laufzeitumgebung für BPEL^{light} vorgestellt. Die Laufzeitumgebung unterstützt die Ausführung von BPEL^{light}-Prozessen sowie beliebiger in BPEL^{light} spezifizierter Nachrichtenaustauschmuster und ermöglicht die Nutzung von Web-Services und SWS als BPEL^{light}-Aktivitätsimplemen-

tierungen. Die Entwicklungsumgebung unterstützt die Modellierung aller zur Ausführung notwendigen Artefakte.

1.2 Aufbau

Der Aufbau der Arbeit sowie die Abhängigkeiten der einzelnen Kapitel ist in Abbildung 3 dargestellt. Kapitel 2 gibt sowohl einen Überblick über die zum Verständnis der Arbeit nötigen Grundlagen als auch über verwandte Arbeiten im Gebiet der Prozessflexibilität. In den nachfolgenden Kapiteln werden die hier vermittelten Grundlagen gegebenenfalls an den entsprechenden Stellen um weitere Detailinformationen ergänzt. Das Kapitel beginnt mit einer Einführung in die Web-Service-Standards, im Besonderen die Sprache zur Beschreibung der Funktionalität von Web-Services (WSDL) und die Geschäftsprozessausführungssprache BPEL. Im Einführungskapitel zu BPEL werden unter anderem die von BPEL unterstützten Interaktionsszenarien vorgestellt, die als Maßstab für die neu entwickelte Sprache dienen. Es folgt eine Einführung in Semantische Web-Services.

Zunächst werden Ontologien als Formalismus zur Erfassung von domänenspezifischem Wissen vorgestellt, um dann darauf aufbauend drei zurzeit populäre Ansätze zur semantischen, d. h. Ontologie-basierten, Beschreibung von Web-Services vorzustellen. Dies sind (i) die Web Ontology Language for Services (OWL-S) [MBH+04], (ii) die Web Service Modelling Ontology (WSMO) [LPR05] und (iii) die Semantic Annotations for WSDL (SAWSDL) [FaLa07]. Neben der Vorstellung der unterschiedlichen Konzepte der Web-Services und Semantischen Web-Services wird das Beispiel eines Onlineversandhandeldienstes und eines durch einen Prozess realisierten Klienten dieses Dienstes eingeführt und gezeigt, wie sowohl Web-Service-Technologie als auch die unterschiedlichen SWS-Ansätze und -Technologien dazu verwendet werden können, dieses Beispiel zu implementieren bzw. die Implementierung von Teilaspekten des Beispiels zu unterstützen. Abschließend werden existierende Ansätze gezeigt, die zum Ziel haben, die Flexibilität bei der Wahl der Geschäftspartner zu erhöhen.

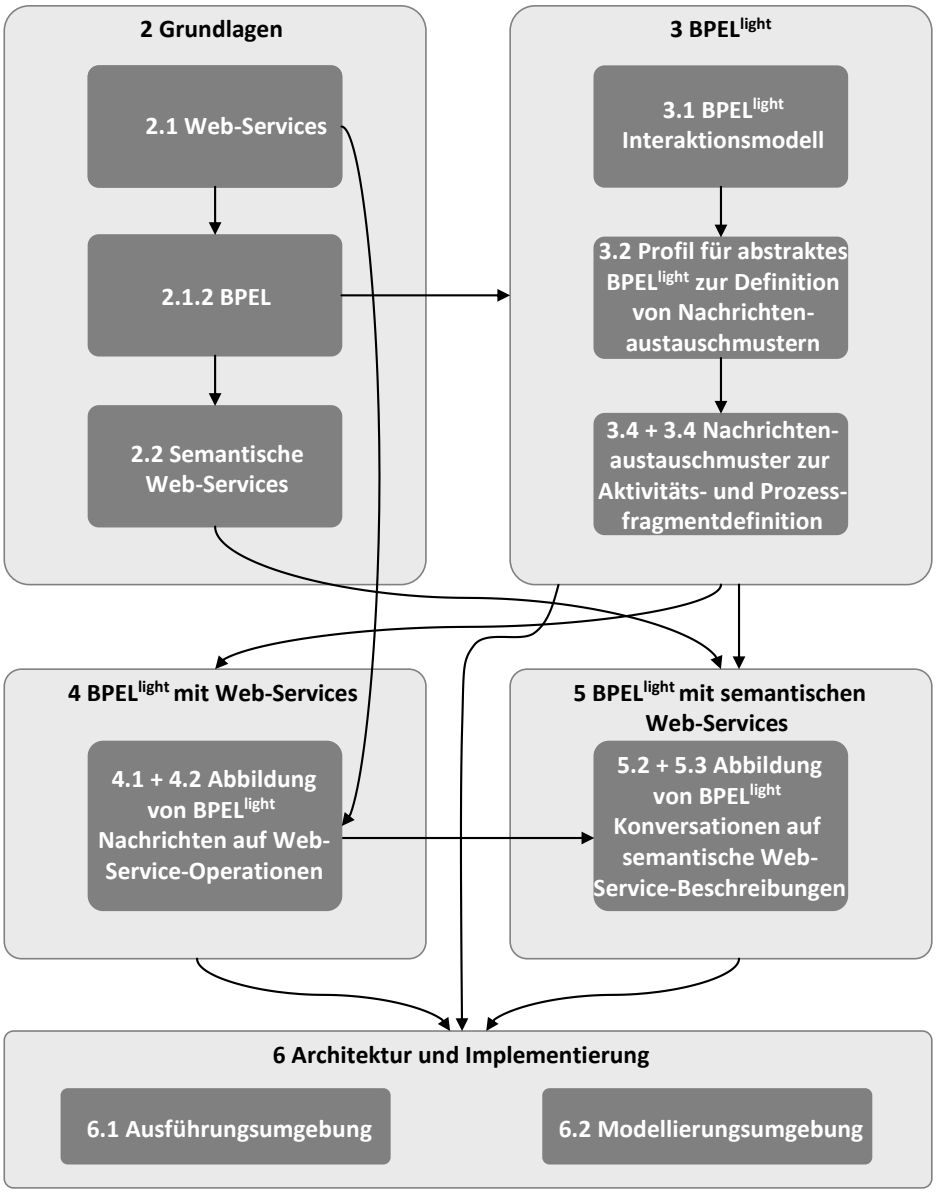


Abbildung 3: Aufbau der Arbeit

Kapitel 3 präsentiert eine Erweiterung von BPEL zu der von WSDL unabhängigen Sprache BPEL^{light}, welche die Prozesslogik von der Definition der potentiellen Aktivitätsimplementierungen in Form von Schnittstellenspezifikationen entkoppelt. Die Einführung von BPEL^{light} erfolgt sukzessive. Zunächst wird ein vollständig asynchrones Interaktionsmodell eingeführt, welches es ermöglicht, Interaktionen mit Geschäftspartnern unabhängig von WSDL zu beschreiben. Dieses Modell ist in der Lage, alle von BPEL unterstützten Interaktionszenarien abzudecken. Anschließend wird ein Profil für abstraktes BPEL^{light} zur Definition von

Nachrichtenaustauschmustern sowie weitere für die Beschreibung von Nachrichtenaustauschmustern nötige Erweiterungen zu BPEL vorgestellt. Diese abstrakten Nachrichtenaustauschmuster werden dann dazu verwendet das Verhalten einzelner BPEL^{light}-Aktivitäten bzw. Teilprozesse zu spezifizieren. Es folgt eine Beschreibung des in Kapitel 2 vorgestellten Klienten des Onlineversandhandeldienstes in BPEL^{light}. Danach wird die Transformation von WS-BPEL zu BPEL^{light} und anders herum diskutiert. Am Ende des Kapitels werden die wichtigsten Ergebnisse noch einmal zusammengefasst.

Kapitel 4 zeigt auf, welche Schritte nötig sind, um Aktivitäten von BPEL^{light}-Prozessen durch Web-Services zu implementieren, d.h. wie die Ein- und Ausgabe von Aktivitäten auf Nachrichten in WSDL 1.1-Operationen abgebildet werden können. Es wird gezeigt, wie alle in Kapitel 2 vorgestellten Interaktionsszenarien mit der Kombination von BPEL^{light} und WSDL 1.1 abgebildet werden können. Anschließend wird die Abbildung des in Kapitel 3 beschriebenen BPEL^{light}-Prozesses auf WSDL-Porttypen und –Operationen beschrieben. Der Diskussion der Transformation von WS-BPEL zu BPEL^{light} mit Web-Services als Aktivitätsimplementierungen und anders herum folgt die Zusammenfassung des Kapitels.

Kapitel 5 beginnt mit der Erweiterung des SWS-Frameworks WSMO, um für Geschäftsprozesse typische, lang laufende, asynchrone Kommunikation zwischen einem Service Nutzer und einem Service unter Verwendung von Web-Service-Technologie zu ermöglichen. Anschließend werden Mechanismen präsentiert, die – angewandt auf BPEL^{light} – in der Sprache BPEL4SWS resultieren: Das erweiterte WSMO-Framework wird verwendet, um sowohl die Anforderungen von Aktivitäten in BPEL^{light} als auch den Anforderungen entsprechenden Aktivitätsimplementierungen in Form von Services semantisch zu beschreiben. Außerdem wird die Funktionalität des Prozesses mit Konzepten des WSMO-Frameworks beschrieben. Anschließend wird der Ansatz anhand der zu unterstützenden Interaktionsszenarien evaluiert. Es folgt die Umsetzung des Klienten des Onlineversandhandeldienstes mit BPEL4SWS. Im Anschluss werden weitere Erweiterungen von BPEL vorgestellt, die davon Gebrauch machen, dass die Semantik der Daten – auf denen der Prozess agiert – durch

Ontologiekonzepte beschrieben ist: Zum einen eine Erweiterung zur Datenmanipulation auf Basis von semantischem Wissen über die Daten und zum anderen eine Erweiterung, die es ermöglicht, Entscheidungen im Kontrollfluss aufgrund semantischer Informationen zu treffen. Abschließend wird die Transformation von WS-BPEL zu BPEL4SWS-Prozessen und anders herum diskutiert und die wichtigsten Ergebnisse des Kapitels zusammengefasst.

Kapitel 5 stellt eine Architektur einer Modellierungs- und Ausführungsumgebung zur Umsetzung der erarbeiteten Konzepte vor und präsentiert eine Implementierung für ausgewählte Komponenten.

In Kapitel 7 wird die Arbeit zusammengefasst und ein Ausblick gegeben.

KAPITEL 2

GRUNDLAGEN UND VERWANDTE ARBEITEN

Dieses Kapitel beschreibt die zum Verständnis der Arbeit nötigen Grundlagen und gibt einen Einblick in verwandte Arbeiten. Zunächst werden die Web-Service-Technologie (Abschnitt 2.1) und im Besonderen die beiden für diese Arbeit relevanten Standards WSDL (Abschnitt 2.1.1) und BPEL (Abschnitt 2.1.2) vorgestellt: WSDL zur funktionalen Beschreibung von Diensten und BPEL zur Workflow-basierten Komposition von Diensten. Die Einführung von BPEL umfasst unter anderem einen Überblick über die von BPEL unterstützten Interaktionsszenarien. Es folgt ein Überblick über existierende Ansätze zur semantischen Beschreibung von Web-Services (Abschnitt 2.2): OWL-S (Abschnitt 2.2.1), WSMO (Abschnitt 2.2.2) und SAWSDL (Abschnitt 2.2.3). Neben der Vorstellung der unterschiedlichen Konzepte der Web-Services und Semantischen Web-Services wird das Beispiel eines Onlineversandhandeldienstes und eines durch einen Prozess realisierten Klienten dieses Dienstes eingeführt und gezeigt, wie sowohl Web-Service-Technologie als auch die unterschiedlichen SWS-Ansätze und -Technologien dazu verwendet werden können, dieses Beispiel zu implementieren bzw. die Implementierung von Teilaspekten des Beispiels zu unterstützen. Den Abschluss des Kapitels bildet ein Überblick über verwandte Arbeiten zum Thema Prozessflexibilität (Abschnitt 2.3) und eine Zusammenfassung des Kapitels (Abschnitt 2.4).

2.1 Web-Services

Web-Services (WSs) [WCL+05, Pa07] sind eine offene, XML-basierte Realisierung der Service-orientierten Architektur (SOA) [Bu00]. Die Web-Service-Technologie ist zur Vereinfachung der Enterprise Application Integration (EAI) [HoWo03] entwickelt worden und kann sowohl zur

Integration bereits existierender Systeme als auch zur Implementierung neuer Systeme herangezogen werden. Zur Beschreibung von Web-Services gibt es keinen alles umfassenden Standard. Stattdessen werden Web-Services durch eine Menge sich ergänzender und teilweise aufeinander aufbauender Spezifikationen und Standards definiert. Diese decken dabei unterschiedliche Aspekte der Web-Services ab. Abbildung 4 stellt zur Veranschaulichung die bekanntesten bzw. wichtigsten Standards und Spezifikationen in einem Schichtenmodell dar. Da in dieser Arbeit jedoch lediglich zwei Spezifikationen des Schichtenmodells relevant sind – die Web Service Description Language (WSDL) [CCMW01] zur funktionalen Beschreibung von Web-Services sowie die Business Process Execution Language (BPEL) [AAA+07] zur Orchestrierung von Web-Services zu einem Geschäftsprozess – fokussiert sich diese Arbeit bei der Beschreibung des Schichtenmodells auf eben diese beiden Spezifikationen. Eine detaillierte Beschreibung des kompletten Schichtenmodells findet sich in [WCL+05].

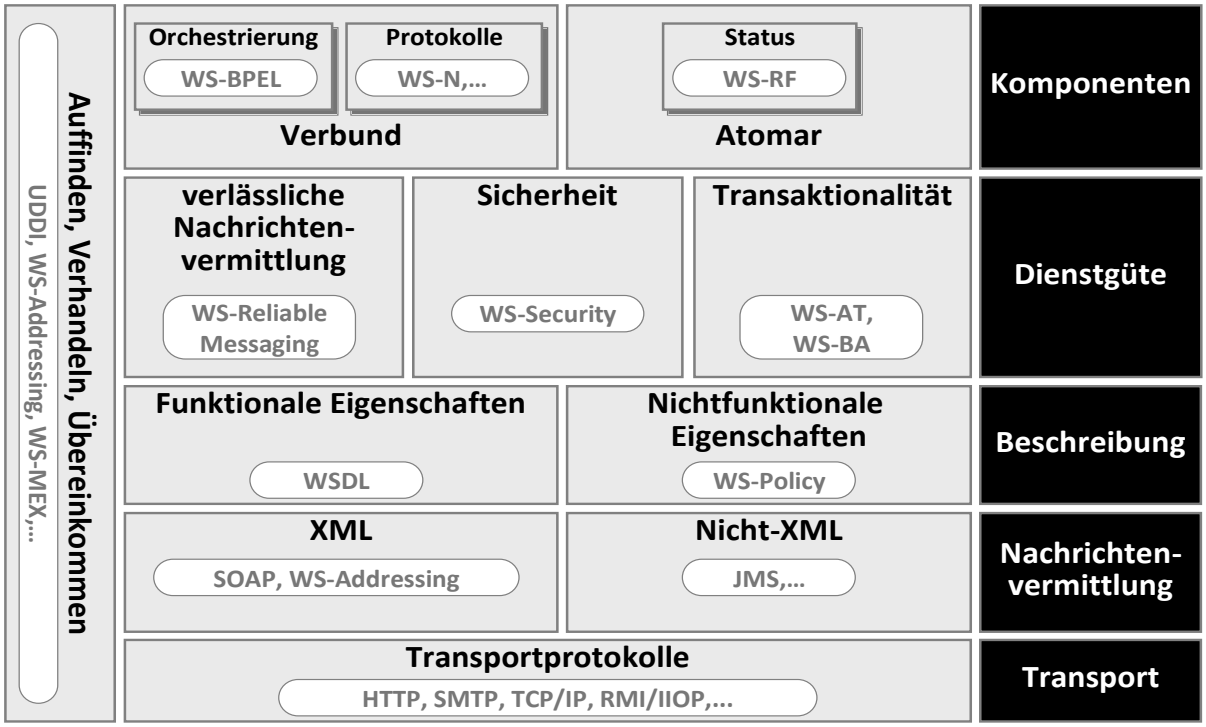


Abbildung 4: Web-Services-Schichtenmodell [WCL+05]

Dadurch, dass bei der Definition von Web-Services ausschließlich offene Standards verwendet werden, wird erreicht, dass Interoperabilität und Portabilität von verteilten Anwendungen unterschiedlicher Hersteller

möglich ist. Interoperabilität wird jedoch dadurch eingeschränkt, dass manche der Spezifikationen und Standards in ihren Aussagen nicht eineindeutig sind und deswegen von Herstellern, die diese Spezifikationen implementieren, nicht gleich interpretiert werden. Um einen höchstmöglichen Grad an Interoperabilität zu erreichen, wurde von der Industrie ein Konsortium gegründet, das WS-Interoperability (WS-I) Konsortium, welches sich damit beschäftigt, Mehrdeutigkeiten der wichtigsten Spezifikationen zu eliminieren und Best Practices für die Implementierung von Web-Services zu entwickeln. Dies wird in so genannten Profilen festgehalten. Da allerdings nicht immer Einigkeit über die Bedeutung einzelner Inhalte erzielt werden kann, kommt es vor, dass ein Profil vorschreibt, dass manche Teile einer Spezifikation nicht verwendet werden dürfen, um dem Profil zu entsprechen.

2.1.1 Web Service Description Language (WSDL)

Die Web Service Description Language (WSDL) dient dazu, funktionale Aspekte eines Dienstes, d.h. den Nachrichtenaustausch mit anderen Diensten, zu beschreiben. Dabei wird der Nachrichtenaustausch aus der Sicht des zu beschreibenden Dienstes betrachtet. Im Jahre 2001 wurde die WSDL-Spezifikation in der Version 1.1 [CCMW01] vom World Wide Web Consortium (W3C) als W3C *Note* herausgegeben. Im Jahr 2007 wurde WSDL 2.0 [CMRW07] dann eine W3C *Recommendation*, d.h. ein offizieller Standard. Da allerdings WSDL 2.0 bis zum heutigen Tag noch keine Akzeptanz in der Industrie gefunden hat und die Business Process Execution Language (BPEL) auf WSDL 1.1 aufbaut, wird hier nur auf diese Version von WSDL eingegangen. Wird im Folgenden von Web-Services gesprochen so bezieht sich das immer auf Dienste, die eine WSDL 1.1-Schnittstelle implementieren.

Die Spezifikation definiert, wie die auszutauschenden Nachrichten (*Message*s), deren Relation zueinander, das Binden (*Binding*) auf ein Transportprotokoll sowie der Endpunkt (*Port*) an dem der Dienst (*Service*) auf die Nachrichten wartet beschrieben werden können. Die Relation der Nachrichten untereinander wird dabei durch die Gruppierung zu Operationen (*Operation*s) definiert, die wiederum zu so genannten

Porttypen (*Port Types*) zusammengefasst werden. Die Beziehungen zwischen den genannten Elementen ist in Abbildung 5 dargestellt.

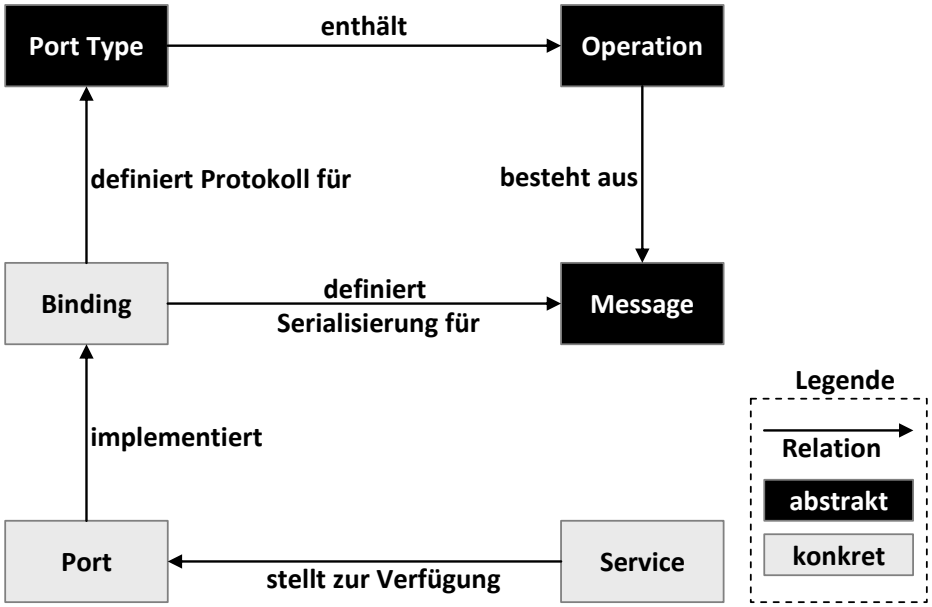


Abbildung 5: Beziehungen der WSDL-Bausteine

Die Bestandteile Message, Operation und Port Type dienen dazu, die Funktionalität eines Dienstes abstrakt zu beschreiben; Port und Binding hingegen definieren wo ein Dienst erreichbar ist und wie auf die Implementierung eines Dienstes konkret zugegriffen werden kann, d.h. unter anderem welches Protokoll verwendet wird und wie die Nachrichten serialisiert sein müssen. Im Folgenden sind lediglich die Elemente zur abstrakten Beschreibung eines Dienstes von Bedeutung. Deshalb wird auf jene Elemente, welche den Zugriff auf die konkrete Implementierung beschreiben, nicht weiter eingegangen.

In der Version 1.1 gibt die WSDL-Spezifikation vier feste Operationstypen vor:

- *Request-Response* – der Dienst empfängt zuerst eine Nachricht und sendet anschließend entweder eine Antwort oder eine Fehlernachricht,
- *One-way* – der Dienst empfängt lediglich eine Nachricht,
- *Solicit-Response* – der Dienst sendet eine Nachricht und empfängt anschließend entweder eine Antwort oder eine Fehlernachricht,
- *Notification* – der Dienst sendet eine Nachricht.

Die Operationstypen entsprechen den grundlegenden Szenarien der Integration von Geschäftsanwendungen (EAI): (i) das Empfangen einer Nachricht löst das Senden einer Antwortnachricht aus (WSDL-Operation des Typs Request-Response aus der Sicht des Dienstes und der duale Operationstyp Solicit-Response aus der Sicht eines den Dienst nutzenden Klienten) und (ii) das Empfangen einer Nachricht hat nicht zur Folge dass eine Antwort verschickt wird (WSDL-Operation des Typs One-way aus der Sicht des Dienstes und der duale Operationstyp Notification aus der Sicht eines den Dienst nutzenden Klienten).

Abbildung 6 zeigt die in dieser Arbeit verwendete Visualisierung der Operationstypen. Der Farbverlauf von hell nach dunkel zeigt die Richtung der (ersten) Nachricht an.

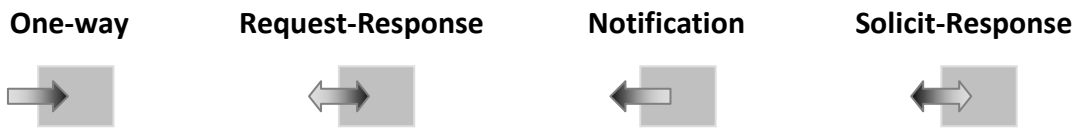


Abbildung 6: Visualisierung der Operationstypen

Bemerkenswert ist an dieser Stelle noch, dass WSDL keine Mechanismen zur Verfügung stellt, die es ermöglichen zu beschreiben, in welcher Reihenfolge die Operationen eines Dienstes von einem Klienten in Anspruch genommen werden können. Lediglich die Reihenfolge der Nachrichten innerhalb einer Operation ist definiert.

2.1.1.1 Interoperabilität

Die WSDL-Spezifikation ist bei der Beschreibung der verschiedenen Operationstypen nicht eindeutig. Das hatte zur Folge, dass die beiden Operationstypen Solicit-Response und Notification von verschiedenen Lagern der Industrie unterschiedlich interpretiert und implementiert wurden. Der Operationstyp Notification wurde beispielsweise von einer Gruppe von Herstellern als Unicast (Punkt-zu-Punkt Verbindung) implementiert, von einer anderen Gruppe hingegen als Multicast (Mehrpunktverbindung) [TaSt02].

Aus diesem Grund, und weil lediglich für die Operationstypen One-way und Request-Response in WSDL 1.1 Bindings definiert sind, schränkt die Web Service Interoperability Organisation im WS-I Basic Profile [BEF+06] die WSDL-Spezifikation der Version 1.1 dahingehend ein, dass – um Interoperabilität zu erreichen – von der Verwendung der Operationstypen Solicit-Response und Notification abzusehen ist.

Deshalb werden in der Praxis lediglich die zwei Operationstypen verwendet, bei denen entweder nur eine Nachricht empfangen wird oder das Empfangen einer Nachricht das Versenden einer Antwort auslöst. Operationen, die mit dem Senden einer Nachricht beginnen, finden keine Verwendung.

Soll Interoperabilität mit Web-Services erreicht werden, so wird es durch die gemachten Einschränkungen also unmöglich, den gesamten Nachrichtenaustausch eines Dienstes als Operationen des Dienstes zu beschreiben, falls der Dienst mehrere Nachrichten hintereinander verschicken will. Stattdessen müssen in diesem Fall zur vollständigen Beschreibung des Nachrichtenaustausches Operationen anderer – mit dem Dienst interagierender – Dienste herangezogen werden. Ein Umstand, der sich auf eines der Hauptprinzipien einer SOA – dem Prinzip der losen Kopplung – negativ auswirkt, da die Kopplung von Dienst und Dienstnutzer dadurch erhöht wird. Ein Beispiel für die Umsetzung dieser Kopplung zweier Parteien bietet die WSDL-Erweiterung partnerLinkType (siehe Kapitel 2.1.2).

2.1.1.2 Beispiel: Onlineversandhandeldienst

Zur Illustration der Fähigkeiten und Einschränkungen von WSDL unter Berücksichtigung des WS-I Basic Profiles, wird das folgende Beispiel herangezogen. Eine Anwendung verwendet einen Onlineversandhandeldienst wie beispielsweise Amazon[1], um Waren einzukaufen. Der Nachrichtenaustausch zwischen der Anwendung und dem Dienst ist wie folgt: Zunächst wird von Seiten der Anwendung eine Bestellung an den Onlineversandhandeldienst übermittelt, der daraufhin eine Rechnung zurückschickt. Anschließend werden dem Dienst die zur Bezahlung notwendigen Bankdaten wie z.B. Kreditkartennummer übermittelt. Dieser verschickt wiederum eine Bestätigungsnachricht. Sobald die Waren schließlich an den Lieferanten übergeben sind, sendet der Dienst eine Nachricht bezüglich der Lieferdetails.

Die Nachrichten können dabei folgendermaßen auf die WSDL-Schnittstelle des Dienstes abgebildet werden (siehe Listing 1): Die Entgegennahme der Bestellung und die anschließende Übermittlung der Rechnung erfolgt über die Operation „orderItem“ des Typs Request-Response. Der Austausch der beiden folgenden Nachrichten, d.h. die zur Bezahlung der Rechnung nötigen Daten und die Bestätigung wird über die Operation „payBill“ des Typs Request-Response abgewickelt. Beide Operationen sind über den Service „onlineShop“ verfügbar. Die letzte Nachricht, d.h. die Benachrichtigung bezüglich der Lieferdetails, die der Dienst seinem Klienten zukommen lässt, kann auf keinen gültigen Operationstyp abgebildet werden. Eine Operation des Typs Notification wäre zwar dazu geeignet, darf jedoch nicht verwendet werden. Folglich kann der Nachrichtenaustausch des Onlineversandhandeldienstes nicht durch die vom Dienst angebotene Operationen dargestellt werden. Hinzu kommt, dass die Reihenfolge, in der die beiden Operationen „orderItem“ und „payBill“ von einem Klienten genutzt werden können, nicht anhand der WSDL-Beschreibung ersichtlich ist.

[1] `http://www.amazon.de`

```
<wsdl:definitions name="onlineShop"
                  targetNamespace="http://example.org/"...>
  ...
  <wsdl:message name="order">
    <wsdl:part name="parameters" element="Order"/>
  </wsdl:message>
  ...
  <wsdl:portType name="sellerPT">
    <wsdl:operation name="orderItem">
      <wsdl:input message="order"/>
      <wsdl:output message="invoice"/>
    </wsdl:operation>
    <wsdl:operation name="payBill">
      <wsdl:input message="payment"/>
      <wsdl:output message="receipt"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:service name="onlineShop">
    ...
  </wsdl:service>
  ...
</wsdl:definitions>
```

Listing 1: WSDL-Beschreibung des Onlineversandhandeldienstes

2.1.2 Business Process Execution Language (BPEL)

Die Geschäftsprozessausführungssprache WS-BPEL (Web Services Business Process Execution Language), auch bekannt als BPEL oder BPEL4WS, ist ein Standard zur Workflow-basierten Komposition von Web-Services (Web-Service-Flow). BPEL ist eine Kombination und Weiterentwicklung der Sprachen WSFL (Web Service Flow Language) [Le01] und XLANG (Web Services for Business Process Design) [Th01]. Die erste Version der Spezifikation (BPEL 1.0 [CGK+02]) wurde 2002 von IBM, Microsoft und BEA veröffentlicht, eine weitere Version (BPEL 1.1 [CGKL03]) gemeinsam mit weiteren Industriepartnern im Mai 2003. Im Jahr 2007 wurde BPEL schließlich in der Version 2.0 [AAA+07] von OASIS[2] standardisiert.

Ein Workflow im Allgemeinen umfasst drei Dimensionen [LeRo00] (siehe Abbildung 7, links): Die Prozesslogik (‚*wie*' ist der Ablauf), Organisatorische Aspekte (‚*wer*' macht es) und die Infrastruktur (‚*womit*' wird es gemacht).

[2] http://www.oasis-open.org

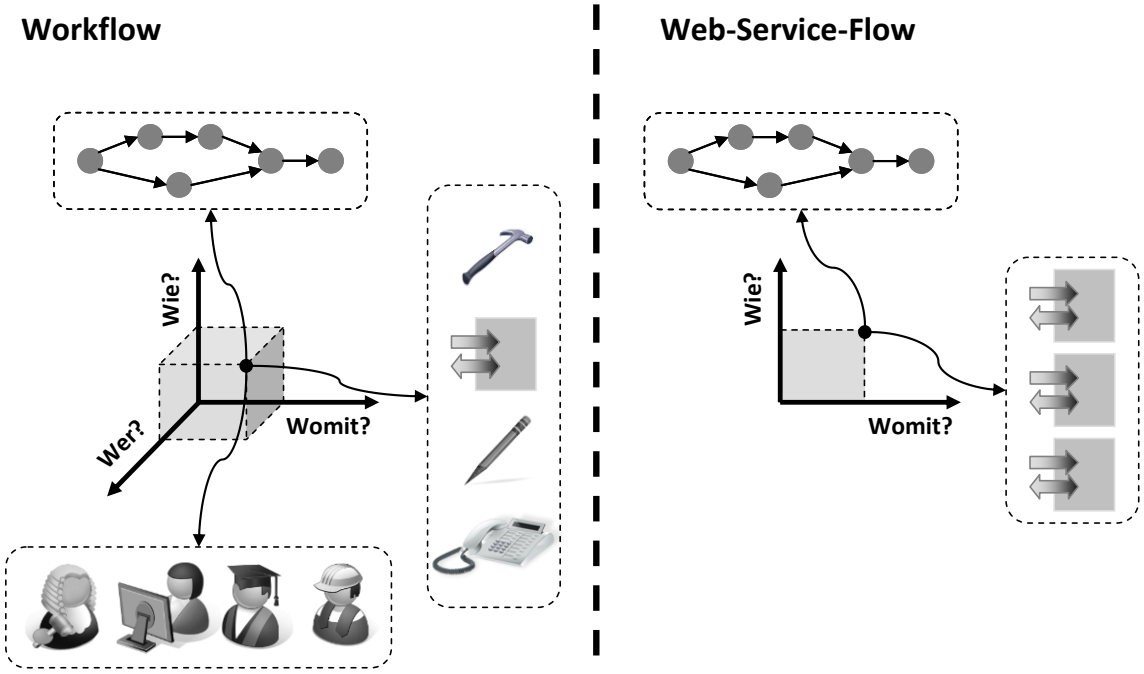


Abbildung 7: Dimensionen eines Workflows (links) und eines Web-Service-Flows (rechts)

Die Dimensionen eines Web-Service-Flows sind auf der rechten Seite von Abbildung 7 dargestellt. Wie bereits erläutert ist BPEL Teil des Web-Service-Schichtenmodells und baut demnach auf anderen Web-Service-Standards, im Besonderen auf WSDL auf. Das äußert sich darin, dass WSDL-Operationen dazu verwendet werden, die ‚womit'-Dimension zu spezifizieren. BPEL selbst definiert den Fluss zwischen diesen Operationen, also die ‚wie'-Dimension. Die ‚wer'-Dimension wird durch BPEL nicht direkt unterstützt, sondern durch die BPEL-Erweiterung BPEL4People [AAD+07a], die es ermöglicht, menschliche Interaktionen [AAD+07b] mit einem Workflow zu definieren.

Die Sprache BPEL kann nicht nur dazu verwendet werden, ausführbare Prozessmodelle zu spezifizieren, sondern sieht auch die Möglichkeit vor, abstrakte Prozessmodelle zu erstellen, bei denen Informationen ausgelassen werden können, die für die Definition eines ausführbaren Prozesses von Nöten sind. In der BPEL 2.0-Spezifikation sind bereits zwei so genannte *Profile* für abstraktes BPEL enthalten, von denen eines der Beschreibung des von außen sichtbaren Verhaltens („*observable behaviour*") eines Prozesses dient. Es kann also dazu verwendet werden,

die Reihenfolge zu beschreiben, in der WSDL-Operationen aufgerufen werden können. Wie bereits in Kapitel 2.1.1 erwähnt, ist dies mit WSDL allein nicht möglich.

BPEL ermöglicht es sowohl Web-Services zu komponieren als auch die Komposition selber wieder als Web-Service zur Verfügung zu stellen und bietet somit ein *rekursives Aggregationsmodell* für Web-Services. Die Komposition von Web-Services wird dabei als ein Fluss zwischen Web-Service-Operationen spezifiziert. Zu diesem Zweck stellt BPEL so genannte strukturierte Aktivitäten zur Verfügung die es ermöglichen, den Kontrollfluss zwischen Interaktionsaktivitäten zu beschreiben. BPEL spezifiziert keinen expliziten Datenfluss; stattdessen werden Daten in Variablen gehalten, auf die sowohl von Interaktionsaktivitäten als auch von Datenmanipulationsaktivitäten (d.h. `<assign>`) zugegriffen werden kann. Der Kontrollfluss zwischen Aktivitäten kann entweder *Block-basiert* durch Verschachteln strukturierter Aktivitäten wie z.B. `<sequence>` (für sequentiellen Kontrollfluss), `<flow>` (für parallelen Kontrollfluss) und `<if>` (für bedingte Verzweigungen im Kontrollfluss) oder *Graph-basiert* durch die Definition von `<link>`s (d.h. gerichteten Kanten) zwischen Aktivitäten in einer `<flow>`-Aktivität spezifiziert werden. Beide Stile können auch gemischt zum Einsatz kommen.

Da BPEL-Prozesse darauf ausgelegt sind, robuste Anwendungen zu realisieren, sind *Transaktionalität* und *Fehlerbehandlung* integrale Bestandteile von BPEL. Sie werden durch `<scope>`s, `<compensationHandler>` und `<faultHandler>` spezifiziert.

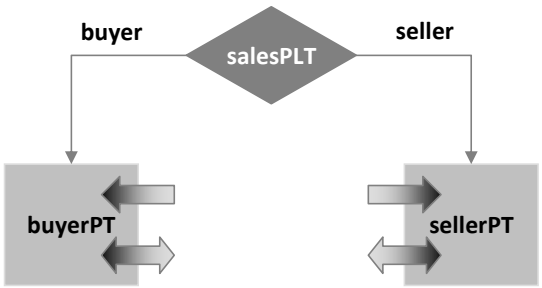
`<Scope>`s definieren transaktionale Geltungsbereiche mit einer auf Kompensation basierenden Wiederherstellungssemantik. `<FaultHandler>` und `<compensationHandler>` können einem `<scope>` zugeordnet werden. Dabei definieren `<faultHandler>`, wie im Fehlerfall zu verfahren ist und `<compensationHandler>`, wie bereits vollendete Aktivitäten kompensiert werden können.

Als Web-Service zur Verfügung gestellte Instanzen von BPEL-Prozessen sind typischerweise langlebig und statusbehaftet, d.h. es existieren gleichzeitig mehrere Instanzen eines Prozessmodells, die zu unterschied-

lichen Vorgängen gehören und daher auch unterschiedliche Status besitzen. Aus diesem Grund ist es wichtig, Nachrichten zu einer bestimmten Prozessinstanz senden zu können, d.h. einen bestimmten Vorgang adressieren zu können. Dies kann zum einen durch die Verwendung von WS-Addressing [GHR06] und zum anderen durch einen BPEL-Mechanismus, der *Correlation Sets* genannt wird, erreicht werden. Correlation Sets basieren auf der Idee, Nachrichten aufgrund ihres Inhaltes zu einzelnen Instanzen zuordnen zu können und ermöglichen es, in Nachrichten die entsprechenden Schlüsselfelder, nach denen die Zuordnung vorgenommen werden soll, zu identifizieren.

Um Kommunikation mit Web-Services zu ermöglichen, die konform zum Basic Profile der WS-Interoperability Organisation ist, d.h. ohne WSDL-Operationen des Typs Solicit-Response und Notification zu verwenden, führt BPEL das Konzept eines Partnerlinktypen (*Partner Link Type*) ein, der als Erweiterung zu WSDL definiert ist.

Ein Partnerlinktyp definiert zwei Rollen in Form von Porttypen – einen Porttypen, den der Prozess einem Partner zur Verfügung stellt, und einen Porttypen, den der Partner dem Prozess zur Verfügung stellen muss – und bindet die beiden zusammen. Die Operationen des Typs Notification und Solicit-Response einer Rolle werden dabei als Operationen des Typs One-way und Request-Response der jeweils anderen Rolle ausgedrückt.



(a) Szenario

```
<definitions
  targetNamespace="http://example.org"
  xmlns:plnk="http://docs.oasis-
      open.org/wsbpel/2.0/plnktype"...>
  ...
  <plnk:partnerLinkType
    name="salesPLT">
    <plnk:role name="buyer"
               portType="buyerPT"/>
    <plnk:role name="seller"
               portType="sellerPT"/>
  </plnk:partnerLinkType>
  ...
<definitions>
```

(b) Codefragment

Abbildung 8: Die WSDL-Erweiterung `<partnerLinkType>`

Abbildung 8 zeigt ein Beispiel eines Partnerlinktypen. Es definiert einen Kanal („salesPLT“) zwischen zwei abstrakten Geschäftspartnern (Rollen) namentlich „buyer“ und „seller“ durch den die Partner Nachrichten austauschen; diese Rollen sind durch Porttypen definiert, im Beispiel „buyerPT“ und „sellerPT“. Für den Fall dass nur ein Partner den anderen aufruft reicht es aus, nur eine Rolle zu definieren.

Um letztendlich einen Vertrag, d.h. eine Übereinkunft über den Kanal, über den Nachrichten auszutauschen sind, zu schließen, verweist ein BPEL-Partnerlink (*Partner Link*) auf einen Partnerlinktypen und spezifiziert, welche Rolle vom Prozess selber (*my Role*) und welche Rolle vom Partner (*partner Role*) eingenommen wird.

Die *Interaktionsaktivitäten* (`<receive>`, `<reply>`, `<invoke>` and `<pick>`) und `eventHandler` werden verwendet, um den eigentlichen Nachrichtenaustausch über einen Partnerlink zu definieren. Das umfasst die tatsächlichen Daten und die Art der Kommunikation (blockierende Interaktion vs. Nicht-blockierende Interaktion). Zu diesem Zweck referenzieren Interaktionsaktivitäten einen Partnerlink und eine WSDL-Operation. Empfangende Aktivitäten (d.h. `<receive>` und `<pick>`) sowie `eventHandler` implementieren eine Operation des Typs One-way des prozesseigenen Porttypen. Sie unterscheiden sich jedoch in ihrer operationalen Semantik. Eine `<receive>`-Aktivität wartet sobald sie aktiv ist bis die spezifizierte Operation aufgerufen wird und verarbeitet dann die einkommende Nachricht. Eventhandler können parallel zum definierten Fluss des Prozesses oder des Geltungsbereichs, für den sie definiert sind solange Nachrichten empfangen und weitere Aktionen auslösen wie der Prozess bzw. Gültigkeitsbereich aktiv ist, wohingegen die `<pick>`-Aktivität ein *„deferred choice“* [AWBH00] Verhalten implementiert und auch zur Definition des Flusses innerhalb eines Prozesses bzw. Geltungsbereichs verwendet werden kann. In Kombination mit einer im Kontrollfluss folgenden `<reply>`-Aktivität können die empfangenden Aktivitäten aber auch eine Operation des Typs Request-Response des prozesseigenen Porttypen implementieren. Die `<invoke>`-Aktivität verwendet Operationen des Typs One-way oder Request-Response, die vom Partner zur Verfügung gestellt werden und demnach für den Porttypen des Partners definiert ist.

In Abbildung 9 ist eine konkrete Ausprägung einer `<invoke>`-Aktivität dargestellt: Sie tauscht die zwei Nachrichten „order“ und „invoice“ mit einem Partnerdienst über den Partnerlink „salesPL“ aus. Der „salesPL“ initiiert den in Abbildung 8 dargestellten Partnerlinktyp „salesPLT“ und ordnet dem Prozess die Rolle „buyer“ und dem Partnerdienst die Rolle „seller“ zu. Der eigentliche Nachrichtenaustausch findet über die Operation „orderItem“ des Typs Request-Response statt, d.h. die Variablen „inputVariable“ und „outputVariable“ sind den Nachrichten der Operation entsprechend typisiert.

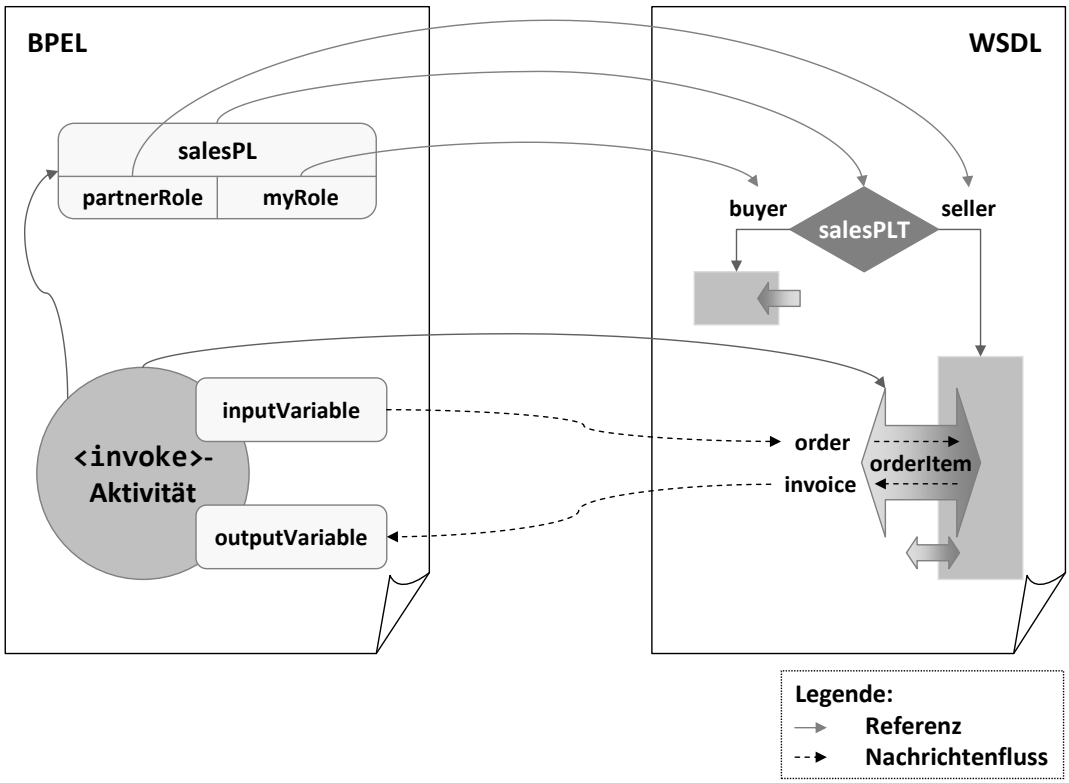


Abbildung 9: Spezifikation einer `<invoke>`-Aktivität mittels WSDL

BPEL verwendet WSDL Operationen und Porttypen um Aktivitäts-implementierungen zu definieren. Die eigentlichen Zugriffs- und Adressierungsinformationen, die für einen WSDL-Endpunkt (Port) definiert sind, sind jedoch nicht zwangsläufig Teil des BPEL-Prozessmodells. Diese Information wird entweder zum Zeitpunkt der Modellierung, zum Deployment oder sogar erst zur Laufzeit einer einzelnen Instanz angegeben [WCL+05]. Während der Modellierung können die Informationen über Endpunkt Referenzen (EPRs) in der `<from>`-

Spezifikation der in `<assign>`-Aktivitäten verwendeten `<copy>`-Elementen definiert werden und über eben diese `<assign>`-Aktivitäten statisch den entsprechenden Partnerlinks zugeordnet werden. Bei der Festlegung der EPRs zum Deployment werden so genannte Deployment Deskriptoren verwendet, um für alle Instanzen eines Prozessmodells die zu verwendenden Endpunkte festzulegen. Die flexibelste Lösung stellt allerdings der Einsatz eines Service Busses [Ch04] dar, der es ermöglicht, Endpunkt Referenzen dynamisch während der Ausführung zu ermitteln. Wird der Prozess von einem anderen Service aufgerufen, so kann dadurch auch in einem Partnerlink die Rolle des Kommunikationspartners initialisiert werden, die dann mit der EPR des Senders der Nachricht assoziiert wird.

Um während der Laufzeit EPRs für einen Partnerlink zu speichern, spezifiziert BPEL einen gesonderten „Container“, der von `<assign>`-Aktivitäten gelesen und geschrieben werden kann. Das ermöglicht es, eine Liste von EPRs in einer Variablen zu speichern, die EPRs nacheinander auszulesen, dem Partnerlink zuzuordnen und somit zu unterschiedlichen Zeitpunkten jeweils mit einer anderen konkreten Instanz zu kommunizieren. BPEL unterstützt also die Interaktion mit mehreren Instanzen des gleichen Partnertyps.

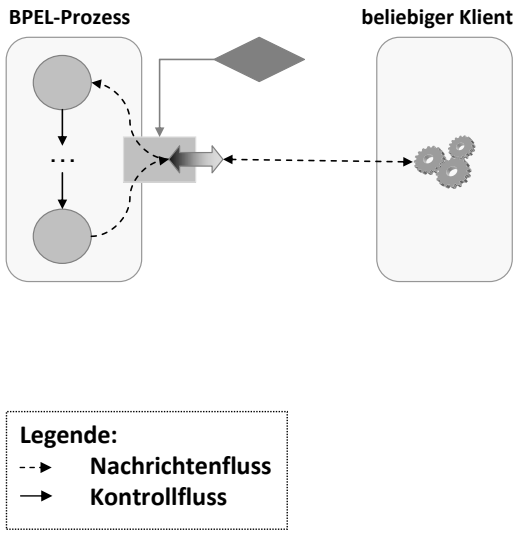
2.1.2.1 Unterstützte Interaktionsszenarien von BPEL

BPEL unterstützt unterschiedliche Kommunikationsstile, die anhand der folgenden Szenarien vorgestellt werden und die auch als Benchmark für die neu zu entwickelnden Sprachen ($BPEL^{light}$ und BPEL4SWS) verwendet werden.

(i) Aufruf eines kurzlebigen BPEL-Prozesses

Handelt es sich um einen kurzlebigen Prozess oder eines Teils davon, der zunächst eine Nachricht empfängt und anschließend eine Antwort verschickt, kann dies durch eine Request-Response-Operation abgebildet werden. Dieses Szenario wird in Abbildung 10 (a) dargestellt. Der vom BPEL-Prozess instantiierte Partnerlinktyp definiert in diesem Fall nur eine Rolle, die Rolle des Prozesses. Der Aufruf der Operation wird innerhalb des

Prozesses von einem `<receive>`-`<reply>`-Paar behandelt. Wird der Prozess aufgerufen, so wird die eingehende Nachricht durch die `<receive>`-Aktivität verarbeitet. Im späteren Verlauf des Prozesses wird die Antwort über die `<reply>`-Aktivität versendet. Tritt zwischen diesen Aktivitäten ein Fehler im Prozess auf, so kann anstatt der regulären Antwort über eine `<reply>`-Aktivität im `<faulthandler>` eine Fehlernachricht versendet werden.




```
<flow>
  <links>
    <link name="receiveTo..."/>
    <link name="...ToSend"/>...
  </links>...
  <receive name="receiveEmployeeDetails"
           partnerLink="hrmPL"
           operation="getPerformanceReview"
           variable="employeeDetails">
    <sources>
      <source
        linkName="receiveTo..."/>...
    </sources>...
  </receive>...
  <reply name="sendPerformanceReview"
         partnerLink="hrmPL"
         operation="getPerformanceReview"
         variable="performanceReview">
    <targets>
      <target
        linkName="...ToSend"/>...
    </targets>...
  </reply>
</flow>
```

(a) Szenario **(b) Codefragment**

Abbildung 10: Aufruf eines kurzlebigen BPEL-Prozesses

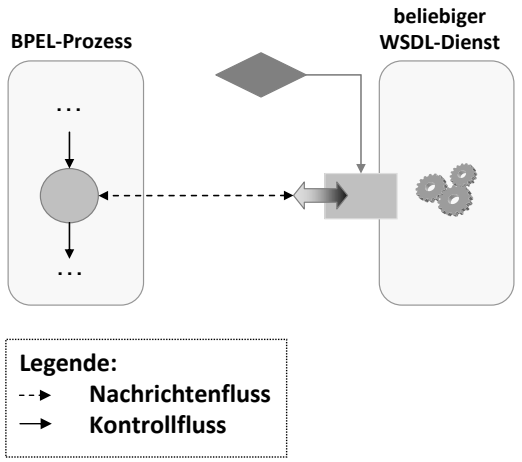
Abbildung 10 (b) zeigt eine konkrete Ausprägung des Szenarios. Der Partnerlink „hrmPL“ instantiiert den Partnerlinktypen „hrmPLT“, der im Gegensatz zum Partnerlinktypen „salesPLT“ in Abbildung 8 nur eine Rolle, die Rolle „humanResourceManagement“ (Personalwesen) definiert[3]. Diese Rolle wird vom Prozess eingenommen. Die Aktivität „receiveEmployeeDetails“ referenziert den Partnerlink „hrmPL“ und implementiert die Operation „getPerformanceReview“, d. h. sie nimmt die

[3] Die WSDL Beschreibung befindet sich in Anhang A.

über die Operation „getPerformanceReview“ eintreffende Nachricht entgegen und speichert sie in der Variablen „employeeDetails“ ab. Die darauf folgende `<reply>`-Aktivität „sendPerformanceReview“ referenziert den gleichen Partnerlink und implementiert den zweiten Teil der Operation, d.h. sie liest den Inhalt der Variablen „performanceReview“ aus und sendet ihn als Antwort zurück.

(ii) Blockierender Aufruf eines Dienstes

Der blockierende Aufruf eines Dienstes wird in Abbildung 11 (a) dargestellt. Auch in diesem Fall reicht es aus, einen Partnerlinktypen zu instantiieren, der nur eine Rolle spezifiziert, diesmal jedoch die Rolle des Partnerdienstes. Eine Operation eben dieses Partnerdienstes wird nun von einer `<invoke>`-Aktivität des Prozesses in Anspruch genommen, d.h. sowohl das Senden der Nachricht als auch das Empfangen der darauf folgenden Nachricht wird von der `<invoke>`-Aktivität realisiert. Alternativ zur erwarteten Antwortnachricht kann die `<invoke>`-Aktivität auch eine Fehlernachricht des Partnerdienstes verarbeiten.




```
<invoke
  name="getPerformanceReview"
  partnerLink="hrmPL"
  operation="getPerformanceReview"
  inputVariable="employeeDetails"
  outputVariable="performanceReview">
  ...
</invoke>
```

(a) Szenario **(b) Codefragment**

Abbildung 11: Blockierender Aufruf eines Dienstes

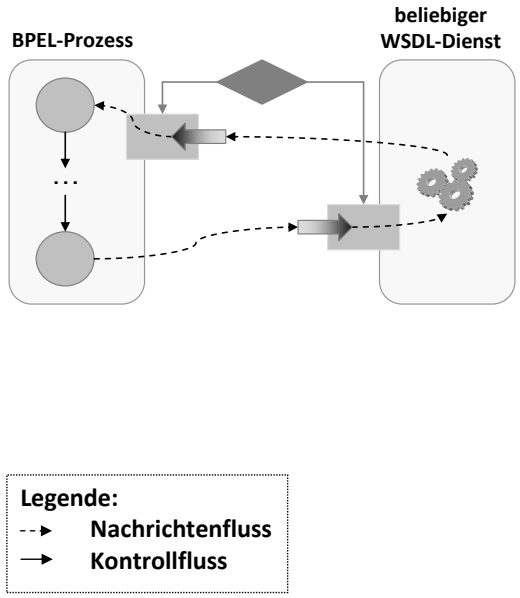
Eine konkrete Ausprägung dieses Szenarios ist in Abbildung 11 (b) dargestellt. Der Partnerlink „hrmPL“ instantiiert ebenso wie beim Aufruf eines kurzlebigen Prozesses den Partnerlinktypen „hrmPLT“, allerdings nimmt diesmal der Partnerdienst die Rolle „humanResourceManagement“

ein. Die Aktivität „getPerformanceReview“ referenziert den Partnerlink „hrmPL“ und ruft die vom Partnerdienst angebotene Operation „getPerformanceReview“ mit dem Inhalt der Variablen „employeeDetails“ auf. Anschließend speichert sie die über die gleiche Operation erhaltene Antwort in der Variablen „performanceReview“ ab.

Im Folgenden wird der in (i) und (ii) präsentierte Stil der Kommunikation auch als synchrone Kommunikation bezeichnet.

(iii) Aufruf eines langlebigen BPEL-Prozesses

Abbildung 12 (a) veranschaulicht die Verwendung einer `<receive>`- und einer `<invoke>`-Aktivität zum Aufruf eines langlebigen BPEL-Prozesses.



(a) Szenario

```
<flow>
  <links>
    <link name="receiveTo..."/>
    <link name="...ToSend"/>...
  </links>...
  <receive name="receiveTripDetails"
           partnerLink="travelAgentPL"
           operation="bookTrip"
           variable="tripDetails">
    <targets>
      <target
        linkName="receiveTo-..."/>...
    </targets>...
  </receive>...
  <invoke name="sendBooking"
          partnerLink="travelAgentPL"
          operation="receiveBooking"
          inputVariable="booking">
    <sources>
      <source
        linkName="...ToSend"/>...
    </sources>...
  </invoke>
</flow>
```

(b) Codefragment

Abbildung 12: Aufruf eines langlebigen BPEL-Prozesses

Der verwendete Partnerlink instantiiert einen Partnerlinktypen, der zwei Rollen definiert. Die `<receive>`-Aktivität verarbeitet die über eine vom Prozess angebotene One-way-Operation erhaltene Nachricht und die darauf folgende `<invoke>`-Aktivität verwendet eine vom Klienten

angebotene One-way-Operation als Call-back, um die Antwortnachricht auszuliefern.

Eine konkrete Ausprägung dieses Szenarios ist in Teil (b) von Abbildung 12 dargestellt. Der in diesem Beispiel verwendete Partnerlink „travelAgentPL“ instantiiert den Partnerlinktypen „travelAgentPLT“, der wie der in Abbildung 8 definierte Partnerlinktyp „salesPLT“ zwei Rollen spezifiziert: „travelAgent“ und „customer“[4]. Der Prozess nimmt die Rolle „travelAgent“ ein, der Klient die Rolle „customer“. Die Aktivität „receiveTripdetails“ referenziert den Partnerlink „travelAgentPL“ und implementiert die Operation „bookTrip“, d. h. sie nimmt die über die Operation „bookTrip“ empfangene Nachricht entgegen und speichert sie in der Variable „tripDetails“ ab. Die darauf folgende `<invoke>`-Aktivität „sendBooking“ referenziert ebenfalls den Partnerlink „travelAgentPL“ und sendet die Antwort unter Verwendung der vom Klienten angebotenen Operation „receiveBooking“ zurück.

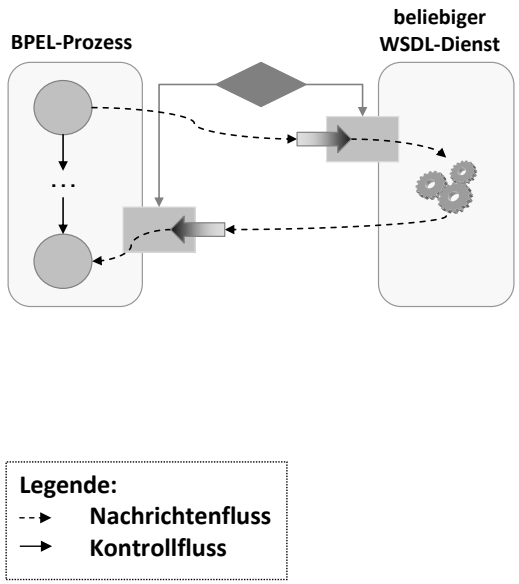
(iv) Nicht blockierender Aufruf eines Dienstes

Ein nicht blockierender Aufruf eines Dienstes mittels einer `<invoke>`- und einer `<receive>`-Aktivität wird in Abbildung 13 dargestellt. Der verwendete Partnerlink instantiiert einen Partnerlinktypen, der zwei Rollen definiert. Die `<invoke>`-Aktivität verwendet eine vom Dienst angebotene One-way-Operation um ihn aufzurufen. Die darauf folgende `<receive>`-Aktivität verarbeitet die Antwortnachricht, die dem Prozess vom Dienst über eine vom Prozess zur Verfügung gestellte One-way-Operation übermittelt wird.

In Abbildung 13 (b) ist eine konkrete Ausprägung dieses Szenarios dargestellt. Der Partnerlink „travelAgentPL“ instantiiert ebenso wie beim Aufruf eines langlebigen Prozesses den Partnerlinktypen „travelAgentPLT“, allerdings nimmt diesmal der Partnerdienst die Rolle „travelAgent“ ein und der Prozess die Rolle „customer“. Die Aktivität „sendTripDetails“ referenziert den Partnerlink „travelAgentPL“ und übermittelt die aus der Variablen „tripDetails“ ausgelesenen Daten an den Dienst über die

[4] Die WSDL Beschreibungen befinden sich in Anhang B und C.

Operation „bookTrip“. Die darauf folgende Aktivität „receiveBooking“ referenziert ebenfalls den Partnerlink „travelAgentPL“ und implementiert die Operation „receiveBooking“, d.h. sie nimmt die über die Operation „receiveBooking“ empfangene Nachricht entgegen und speichert sie in der Variablen „booking“ ab.



(a) Szenario

```
<flow>
  <links>
    <link name="sendTo..."/>
    <link name="...ToReceive"/>...
  </links>...
  <invoke name="sendTripDetails"
          partnerLink="travelAgentPL"
          operation="bookTrip"
          inputVariable="tripDetails">
    <sources>
      <source
        linkName="sendTo..."/>...
    </sources>...
  </invoke>...
  <receive name="receiveBooking"
           partnerLink="travelAgentPL"
           operation="receiveBooking"
           variable="booking">
    <targets>
      <target
        linkName="...ToReceive"/>...
    </targets>...
  </receive>
</flow>
```

(b) Codefragment

Abbildung 13: Nicht blockierender Aufruf eines Dienstes

Der in (iii) und (iv) präsentierte Stil der Kommunikation wird im Folgenden auch als asynchrone Kommunikation bezeichnet. Sie wird – für den Fall, dass Interoperabilität eine Anforderung ist – verwendet, wenn es sich um einen Aufruf langlebiger Prozesse bzw. Dienste handelt, da das WS-I Basic Profile auch vorschreibt, dass das in WSDL definierte HTTP-Binding verwendet werden muss. Eine Request-Response-Operation mit HTTP-Binding umgesetzt durch ein `<receive>`-`<reply>` Paar oder eine `<invoke>`-Aktivität würde schlicht und einfach in einem Time-out resultieren, falls die Antwort erst nach Minuten, Stunden oder gar Tagen zur Verfügung steht. Da One-way-Operationen nicht vorsehen, dass Fehlernachrichten

übermittelt werden, ermöglicht es BPEL bei dieser Art der Kommunikation nicht, Fehlernachrichten zu empfangen oder zu senden.

BPEL unterstützt nicht nur die genannten vier Szenarien, sondern ermöglicht durch die Orchestrierung von One-way- und Request-Response-Operationen des Prozesses und des Kommunikationspartners die Modellierung eines hinreichend komplexen Nachrichtenaustauschs. Durch die Modellierung mehrere Partnerlinks ist auch die Definition eines komplexen Nachrichtenaustauschs mit mehreren Partnern möglich.

2.1.2.2 BPEL-SPE (Extension for Sub-processes) – BPEL-Erweiterung zur Unterstützung von Subprozessen

Die BPEL-Erweiterung BPEL-SPE [KKL+05] beschreibt wie Geschäftslogik in einen Subprozess ausgelagert werden kann und wie dieser durch den so genannten Vater- bzw. Hauptprozess aufgerufen werden kann.

Ein Subprozess ist ein eigenständig ausführbarer BPEL-Prozess, der über eine instanziierende `<receive>`-Aktivität eine Nachricht vom aufrufenden Vaterprozess empfängt und als letzte Aktion entweder synchron, d.h. über eine `<reply>`-Aktivität (vergleiche 2.1.2.1 (i)), oder asynchron, d.h. über eine `<invoke>`-Aktivität (vergleiche 2.1.2.1 (iii)), antwortet. Über diesen einmaligen Aufruf hinaus darf es zwischen Vaterprozess und Subprozess keine weitere Interaktion geben. Der Aufruf aus dem Vaterprozess geschieht sowohl im synchronen als auch im asynchronen Fall über die Erweiterungsaktivität `<spe:call>`. Um dem Vaterprozess zu ermöglichen den Subprozess zu steuern, um also beispielsweise den Subprozess aus dem Vaterprozess heraus terminieren zu können, definiert BPEL-SPE ein Koordinationsprotokoll sowie zugehörige Zustandsübergangsdiagramme für die `<spe:call>`-Aktivität und den Subprozess.

Möchte man also ein Geschäftsziel, dessen Erfüllung einen komplexen Nachrichtenaustausch erfordert, mit einer einzelnen Aktivität umsetzen, so kann man eine `<spe:call>`-Aktivität verwenden, die einen Subprozess aufruft, der diesen Nachrichtenaustausch durchführt.

2.1.2.3 Beispiel: BPEL-Prozess als Klient eines Onlineversandhandeldienstes

Das bereits in Abschnitt 2.1.1.2 eingeführte Beispiel einer Anwendung, die einen Onlineversandhandeldienst verwendet, um Waren einzukaufen, wird hier weitergeführt. Die Anwendung wird im Folgenden durch einen BPEL-Prozess realisiert (siehe Abbildung 14). Der Austausch der Nachrichten Bestellung, Rechnung, Bezahlung und Bestätigung erfolgt, wie bereits beschrieben, über die Operationen „orderItem“ und „payBill“ des Onlineversandhandeldienstes. Da die Auslieferung der Lieferdetails nicht über eine Operation des Onlineversandhandeldienstes möglich ist, wird hierzu die vom Prozess implementierte Operation „receiveShippingDetails“ verwendet. Sowohl die Operationen des Onlineversandhandeldienstes als auch die Operation des Prozesses werden innerhalb eines eigenen Porttypen definiert. Diese beiden Porttypen werden durch die Definition des Partnerlinktypen „salesPLT“ aneinander gebunden. Die entsprechenden WSDL-Beschreibungen finden sich im Anhang (D und E).

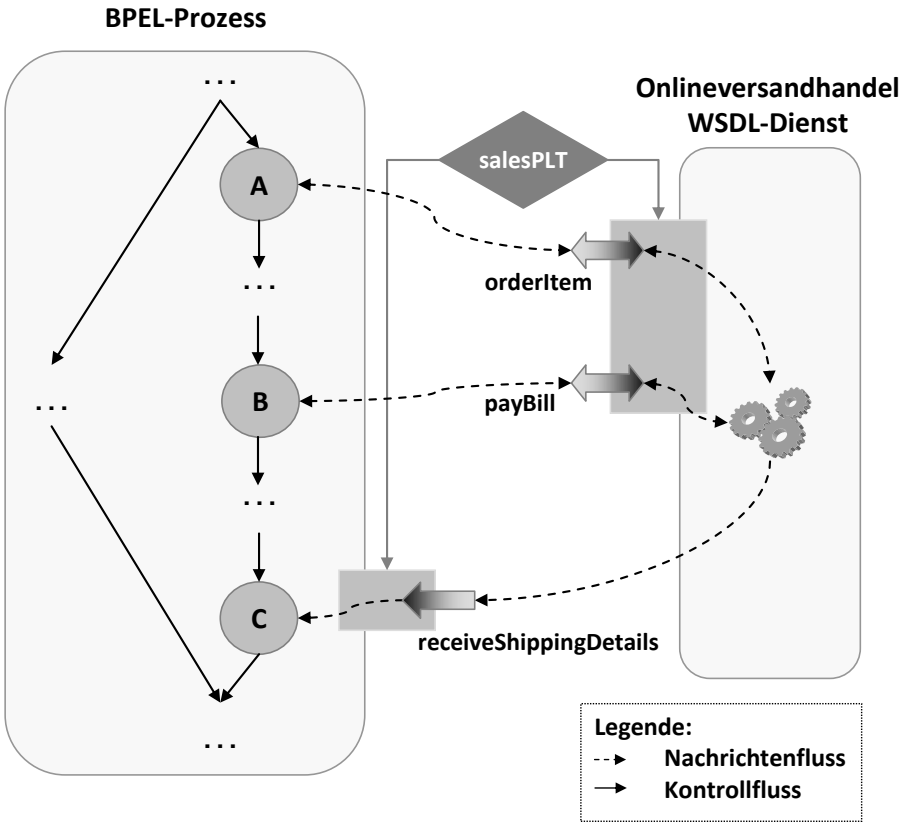


Abbildung 14: BPEL-Prozesses als Klient eines Onlineversandhandeldienstes

```
<process ...> ...
  <partnerLinks>
    <partnerLink name="salesPL"
                 partnerLinkType="salesPLT"
                 myRole="buyer"
                 partnerRole="seller"/>...
  </partnerLinks>...
  ...
    <flow>
      <links>
        <link name="...ToOrder" />
        <link name="orderTo..." />
        <link name="...ToPay" />
        <link name="payTo..." />
        <link name="...toReceiveShippingDetails" />
        <link name="receiveShippingDetailsTo..." />
      </links>
      <invoke name="order" partnerLink="salesPL" operation="orderItem" ...>...
        <sources>
          <source linkName="orderTo..." />...
        </sources>...
        <targets>
          <target linkName="...ToOrder" />...
        </targets>...
      </invoke>
      <invoke name="pay" partnerLink="salesPL" operation="payBill" ...>...
        <sources>
          <source linkName="payTo..." />...
        </sources>...
        <targets>
          <target linkName="...ToPay" />...
        </targets>...
      </invoke>
      <receive name="receiveShippingDetails" partnerLink="salesPL"
               operation="receiveShippingDetails" ...>...
        <sources>
          <source linkName="receiveShippingDetailsTo..." />...
        </sources>...
        <targets>
          <target linkName="...ToReceiveShippingDetails" />...
        </targets>...
      </receive>
    </flow>
  ...
</process>
```

Listing 2: BPEL-Prozess als Nutzer des Onlineversandhandeldienstes

Der BPEL-Prozess (siehe Listing 2) verwendet schließlich den „salesPLT“ zur Definition des Partnerlinks „salesPL“ und definiert dadurch einen Vertrag zwischen der vom Prozess eingenommenen Rolle „buyer“ und der vom Onlineversandhandeldienst eingenommenen Rolle „seller“. Durch die Prozesslogik wird definiert, in welcher Reihenfolge sich die Partner

gegenseitig über die von den Porttypen vorgegebenen Operationen aufrufen. Hierbei wird innerhalb des Prozesses die Definition des Kontrollflusses und die Schnittstellenbeschreibung des Prozesses sowie des zu verwendenden Partnerdienstes vermengt und aneinander gekoppelt.

Die `<invoke>`-Aktivität „order“ verweist auf den Partnerlink „salesPL“, und somit auf den Porttypen „sellerPT“ und auf die darin enthaltene Operation „orderItem“. Gleiches gilt für die `<invoke>`-Aktivität „pay“, welche sich ebenfalls auf den Partnerlink „salesPL“, also auch auf den Porttypen „sellerPT“, und auf die Operation „payBill“ des Onlineversand-handeldienstes bezieht. Die `<receive>`-Aktivität „receiveShippingDetails“ bezieht sich zwar auf eine prozesseigene Operation, referenziert aber auch den Partnerlink „salesPL“ und ist somit auch abhängig vom Porttypen „sellerPT“ des Partnerdienstes.

Soll der Prozess nun einen anderen Onlineversandhandeldienst verwenden weil dieser bessere Konditionen bietet, so ist dies durch die Trennung der abstrakten Beschreibung eines Dienstes durch Porttyp und Operation und der eigentlichen Implementierung prinzipiell möglich. Es kann erst zur Laufzeit die am besten geeignete Implementierung des während der Modellierung des Prozesses angegebenen Porttypen ausgewählt werden. Da es für Onlineversandhandeldienste jedoch keinesfalls einen standardisierten Porttypen gibt, implementiert in der Regel jeder Dienst einen eigens definierten Porttypen. In unserem Beispiel soll der bisher verwendete Dienst durch einen Dienst ausgetauscht werden, der den Porttypen „verkäuferPT“ implementiert. Um dies zu ermöglichen muss zunächst ein neuer Partnerlinktyp modelliert werden, der die Porttypen „verkäuferPT“ und „buyerPT“ aneinander bindet. Im nächsten Schritt muss ein neuer Partnerlink definiert werden, der einen Vertrag zwischen dem Prozess und dem neuen Dienst definiert. Schließlich muss auch noch die Definition der Aktivitäten des Prozesses angepasst werden: Die `<invoke>`-Aktivitäten „order“ und „pay“ müssen den neuen Partnerlink und entsprechende Operationen des Porttypen „verkäuferPT“ referenzieren und auch die `<receive>`-Aktivität muss angepasst werden, weil sie sich nun auf den neuen Partnerlink beziehen muss.

Eine minimale Änderung wie das Austauschen eines Geschäftpartners zieht also sehr viele Anpassungen nach sich, sowohl in der WSDL-Beschreibung als auch in der Prozessbeschreibung des BPEL-Prozesses selbst. Das ist ein großer Nachteil, da in einem agilen Unternehmen Geschäftspartner häufig ausgetauscht werden und dies in BPEL nur unzureichend unterstützt wird.

2.2 Semantische Web-Services

Web-Services ermöglichen die Entwicklung verteilter Systeme in heterogenen Umgebungen durch die Bereitstellung plattformunabhängiger Dienste. Sie haben, wie bereits in Abschnitt 2.1 gezeigt, allerdings auch Schwachpunkte. Implementiert bzw. modelliert man einen Klienten, der einen Dienst nutzt, so wird dieser gegen die Schnittstelle des Dienstes gebunden. Es muss also zunächst der richtige Dienst, genauer die Schnittstelle des richtigen Dienstes gefunden werden. Repositories für Web-Services stellen zu diesem Zweck Suchroutinen zur Verfügung. Sie verwenden Algorithmen, die auf rein syntaktischen Merkmalen basieren, um potentiell geeignete Dienste herauszufiltern und weniger geeignete Dienste auszublenden um dadurch die Übersichtlichkeit über die zur Verfügung stehenden Dienste zu erhöhen. Letztendlich muss aber der Modellierer bzw. Implementierer die Schnittstellenbeschreibungen interpretieren und den geeigneten Dienst für die zu lösende Aufgabe wählen. Zur Laufzeit kann dann automatisch eine Implementierung der ausgewählten Schnittstelle gebunden werden. Funktional äquivalente Dienste, die eine andere Schnittstelle implementieren, werden hierbei jedoch ignoriert. Somit stellt die Auffindbarkeit von Diensten ein Problem dar.

Das gleiche Problem lässt sich im Internet beobachten. Werden Inhalte im Netz gesucht, so muss der Benutzer sie letzten Endes selber finden. Suchmaschinen wie z.B. Google[5] können nur dabei helfen, potentiell relevante Seiten herauszufiltern und somit die Übersichtlichkeit der vorhandenen Informationen zu erhöhen. Sie ignorieren aber die Semantik der im Web präsentierten Inhalte. Es kann also nicht sichergestellt werden,

[5] `http://www.google.de`

dass die gefundenen Seiten auch die gewünschten Informationen beinhalten. Letztendlich muss also der Benutzer eine Seite lesen, sie interpretieren und entscheiden, ob eine Seite die gewünschten Informationen enthält oder nicht.

Um zusätzlich der Semantik der Inhalte des Webs Rechnung zu tragen, werden im semantischen Web Ontologien verwendet, die es ermöglichen, Inhalte semantisch zu beschreiben und in Beziehung zu setzen. Semantische Web-Services gehen noch einen Schritt weiter und nutzen Ontologien nicht nur um den statischen Inhalt des Webs über semantische Beschreibungen auffindbar zu machen, sondern auch dynamische Inhalte, d.h. Dienste.

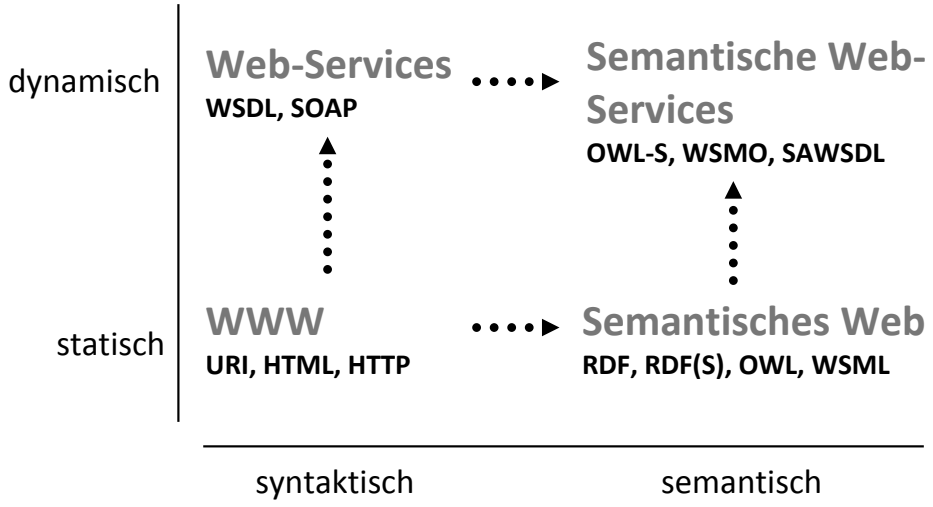


Abbildung 15: Vom World Wide Web über Web-Services und dem Semantischen Web hin zu Semantischen Web-Services [DOS03]

Die Beziehung der genannten vier Bereiche ist in Abbildung 15 dargestellt. Den Ausgangspunkt bildet links unten das statische syntaktische World Wide Web. Dieses wurde und wird in zwei Dimensionen weiterentwickelt. Zum einen in der horizontal dargestellten semantischen Dimension zu einem weiterhin statischen aber Semantischen Web und zum anderen in der vertikal dargestellten dynamischen Dimension zu syntaktisch beschriebenen über das Web erreichbaren Diensten, den so genannten Web-Services. Schließlich werden sowohl Konzepte der semantischen als

auch der dynamischen Erweiterung des Webs zu Semantischen Web-Services vereinigt.

Im Folgenden wird zunächst das Konzept der Ontologien eingeführt. Anschließend wird in den nächsten Abschnitten auf die unterschiedlichen Ansätze zur semantischen Beschreibung von Diensten eingegangen. Diese lassen sich gruppieren in zwei unterschiedliche Lager: Zu den Top-down Ansätzen gehören OWL-S (Abschnitt 2.2.1) und WSMO (Abschnitt 2.2.2). Sie spezifizieren beide jeweils ein konzeptionelles Model für Web-Services und bilden dieses an geeigneten Stellen auf WSDL ab. Beide Spezifikationen werden zurzeit zur Reference Ontology for Semantic SOA (RO4SSOA) [CCH+08] zusammengefasst. Neben den Top-down Ansätzen existieren auch Bottom-up Ansätze, z.B. SAWSDL (Abschnitt 2.2.3). SAWSDL definiert kein eigenes konzeptionelles Model für Web-Services sondern spezifiziert Mechanismen zur semantischen Annotation von WSDL-Beschreibungen. Im Folgenden werden sowohl Kenntnisse der das Web realisierenden Technologien als auch der bereits in Abschnitt 2.1 vorgestellten Technologien zur Realisierung von Web-Services vorausgesetzt.

Ontologien

Der Begriff der Ontologie stammt ursprünglich aus der Philosophie und bezeichnet dort die Lehre des Seins [Wa1726]. Erst in den 1990er Jahren wurde er als technischer Begriff in die Informatik eingeführt [StSt04] und bezeichnet eine „explizite formale Spezifikation einer Konzeptualisierung“ [Gr93] oder anders ausgedrückt ein formales Modell eines relevanten Ausschnitts einer bestimmten Anwendungsdomäne. Eine einfache Ontologie besteht aus Konzepten bzw. Klassen, aus Relationen und Instanzen. Darüber hinaus kann eine Ontologie über ein mächtiges Regelwerk verfügen, mit dem Zusammenhänge zwischen Objekten innerhalb einer Ontologie bzw. zu anderen Ontologien ausgedrückt werden können. Mit Hilfe dieses Regelwerks ist es möglich, wissensbasierte Aussagen abzuleiten bzw. Schlussfolgerungen aus der Ontologie zu ziehen. Das besondere an Ontologien ist, dass sie es ermöglichen, möglichst allgemein gültige Aussagen zu treffen, die nicht ausschließlich aus den Erfordernissen einer einzelnen Anwendung geprägt sind. Aus diesem

Grund eignen sie sich unter anderem dazu, die bereits in der Einleitung erwähnten, domänenspezifischen Standards für Nachrichten zu definieren. Darüber hinaus können sie zur Integration von Informationen heterogener Quellen verwendet werden. Als Ausgangs-, Ziel- oder Zwischenformat helfen sie bei der voll- oder teil-automatischen Nachrichtentransformation [St08].

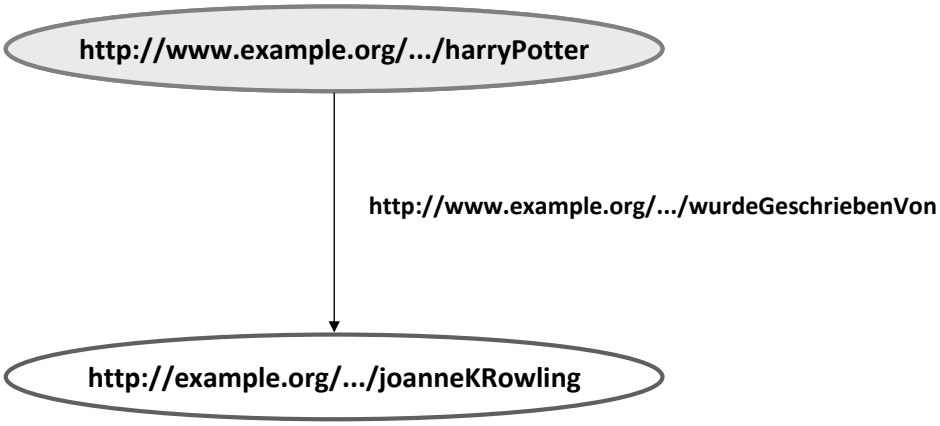


Abbildung 16: RDF Aussage „Harry Potter wurde geschrieben von Joanne K. Rowling“

Ontologien haben durch die Idee des Semantischen Webs von Tim Berners-Lee [BeFi99] einen enormen Auftrieb erhalten. Die zurzeit populärsten und am Weitesten verbreiteten Ontologiesprachen wurden vom W3C entwickelt und als W3C Recommendation standardisiert: Diese sind das Resource Description Framework (RDF) [KlCa04], RDF Schema (RDFS) [BrGu04] und die Web Ontology Language (OWL) [McHa04]. Die drei Sprachen werden alle in XML serialisiert, unterscheiden sich aber in ihrer Ausdrucksstärke.

RDF ermöglicht es Aussagen über Objekte, die durch eine URI [BFM98] identifiziert werden, in Tripleform zu definieren. Ein Tripel kann als Satzform mit Subjekt, Prädikat und Objekt verstanden werden. Aussagen können als gerichteter Graph visualisiert werden. In Abbildung 16 ist eine solche Aussage dargestellt. Sie sagt aus: „*Harry Potter*“ „wurde geschrieben von“ „*Joanne K. Rowling*“.

RDFS stellt darüber hinaus die Möglichkeit bereit, Objekte zu klassifizieren und semantisch zueinander in Beziehung zu setzen. Zum Beispiel kann mit

RDFS festgelegt werden, dass das Objekt „http://.../harryPotter“ eine Instanz der Klasse „http://.../book“ ist, dass das Objekt „http://.../joanneKRowling“ eine Instanz der Klasse „http://.../author“ ist und die Eigenschaft „http://.../wurdeGeschriebenVon“ dazu verwendet wird, um den Author eines Buchs festzulegen.

OWL geht bezüglich der semantischen Beschreibungen noch einen Schritt weiter. Die Sprache baut teilweise auf RDF und RDFS auf und bietet im Vergleich zu RDFS weitaus komplexere Funktionen zum Beschreiben der Beziehungen zwischen Objekten. Diese sind unter anderem die Definition der Äquivalenz oder Disjunktheit von Klassen sowie die Einschränkung der Kardinalität von Eigenschaften. Da je nach Anwendungsfall einer Ontologie ein unterschiedlicher Formalisierungsgrad angestrebt wird, existieren für OWL verschiedene Varianten unterschiedlicher Ausdrucksstärke: OWL Full, OWL DL und OWL Lite [McHa04].

OWL Full ist eine echte Erweiterung von RDF und RDFS, d.h. die Verwendung aller OWL sowie aller RDF und RDFS Modellierungsprimitiven sind erlaubt. Da OWL Full keinerlei Einschränkungen macht, ist die Sprache unentscheidbar [HPH03] und hat somit kaum praktische Relevanz, lediglich wenn die volle Kompatibilität zu RDF benötigt wird.

OWL DL entspricht einem entscheidbaren Fragment der Prädikatenlogik erster Ordnung und basiert auf der Beschreibungslogik $\mathcal{SHOIN}(\mathcal{D})$ [HPH03]. Sie schränkt die Verwendung von RDF, RDFS und OWL Modellierungsprimitiven entsprechend ein. OWL DL bleibt komplex (NEXPTIME), ist allerdings entscheidbar und eignet sich somit für die Definition ausdrucksstarker, aber noch entscheidbarer Ontologien.

OWL Lite entspricht ebenfalls einem entscheidbaren Fragment der Prädikatenlogik erster Ordnung und basiert auf der Beschreibungslogik $\mathcal{SHIF}(\mathcal{D})$ [HPH03], schränkt also die Verwendung von RDF, RDFS und OWL Modellierungsprimitiven weiter ein. OWL Lite ist relativ effizient (EXPTIME) und eignet sich für Taxonomien und „leichte“ Axiomatisierungen.

Eine mit OWL vergleichbare Ontologiesprache ist die Web Service Modelling Language (WSML). Sie baut nicht auf RDF und RDFS auf, verfügt aber wie OWL auch über unterschiedliche Varianten unterschiedlicher Ausdrucksstärken, die auf unterschiedlichen Formalismen basieren (siehe Abbildung 17) [Br08]:

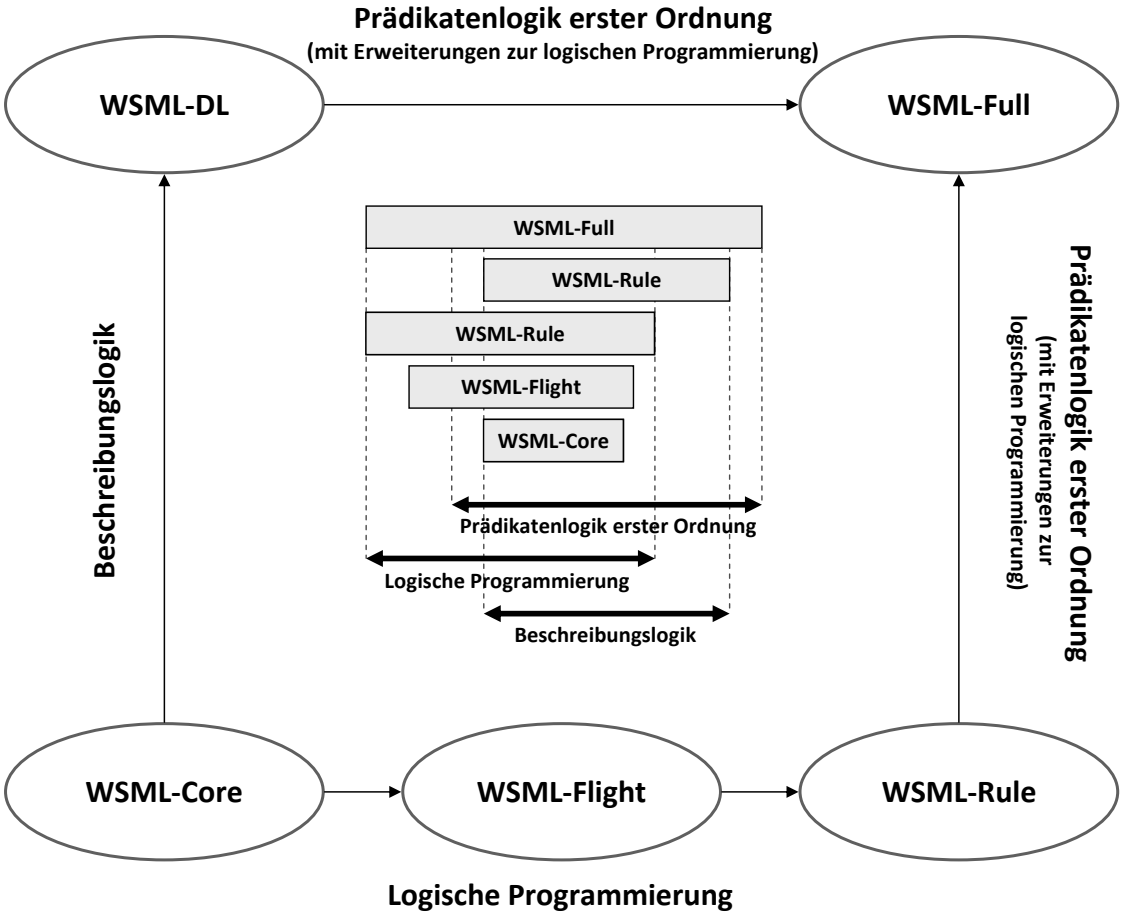


Abbildung 17: Die WSML-Sprachfamilie [Br08]; innen als Schichtenmodell; außen entlang der Dimensionen aufgespannt

WSML-Core unterstützt die Schnittmenge von Beschreibungslogik und Hornlogik, einer Variante der Logischen Programmierung [GHVD03]. Sie ist die am wenigsten ausdrucksstarke WSML-Variante und weist die besten Berechnungseigenschaften auf. WSML-Core unterstützt die Modellierung von Klassen, Attributen, binären Relationen und Instanzen sowie von Klassenhierarchien und Hierarchien von Relationen.

WSML-Flight erweitert WSML-Core um regelbasierte Sprachelemente. Es basiert auf F-Logik [KLW95], einer Variante der logischen Programmierung.

WSML-Rule ist eine Erweiterung von WSML-Flight um weitere regelbasierte Elemente zu einer vollständigen Logikprogrammiersprache.

WSML-DL ist eine Erweiterung von WSML-Core und basiert auf der Beschreibungslogik $\mathcal{SHIQ(D)}$, die OWL-DL zum größten Teil abdeckt.

WSML-Full vereinigt schließlich WSML-DL und WSML-Rule und basiert auf der Prädikatenlogik erster Ordnung. Es hat Erweiterungen, um die mit WSML-Rule mögliche logische Programmierung voll zu unterstützen.

```
wsml                = wsmlvariant? namespace? definition*
wsmlvariant         = ‘wsmlVariant’ full_iri
namespace           = 'namespace' prefixdefinitionlist
prefixdefinitionlist = full_iri |'{' prefixdefinition ( ',' prefixdefinition )* '}'
prefixdefinition    = name full_iri | full_iri
definition          = ontology | ...
ontology            = 'ontology' iri? header* ontologyElement*
header              = importsontology | annotations |...
importsontology     = 'importsOntology' irilist
ontologyElement     = concept | relation | instance | relationInstance |axiom
concept             = 'concept' id? superconcept? annotations* attribute*
superconcept        = 'subConceptOf' idlist
idlist              = id |'{' id ( ',' id )* '}' |'{' '}'
annotations         = 'annotations' annotationvalue* 'endAnnotations'
attribute           = id attributefeature* attType cardinality? idlist …
attType             = 'ofType' | 'impliesType'
cardinality         = '(' digit+ cardinalityNumber? ')'
cardinalityNumber   = digit+ | '*'
attributefeature    = 'transitive' | 'symmetric' | 'inverseOf' '(' id ')' |…
```

Abbildung 18: Auszug der WSML-Syntax in BNF

WSML kommt bei der Beschreibung von WSMO-Web-Services (siehe Abschnitt 2.2.2) zum Einsatz. Im Gegensatz zu RDF, RDFS und OWL besitzt WSML neben der XML Syntax auch eine „menschenlesbare“ und kompakte Syntax. OWL und WSML-Ontologien der entsprechenden Ausdrucksstärke können aufeinander abgebildet werden [Br08, StBr08]. Auszüge der Syntax von WSML in Backus-Naur-Form (BNF) sind in Abbildung 18 dargestellt.

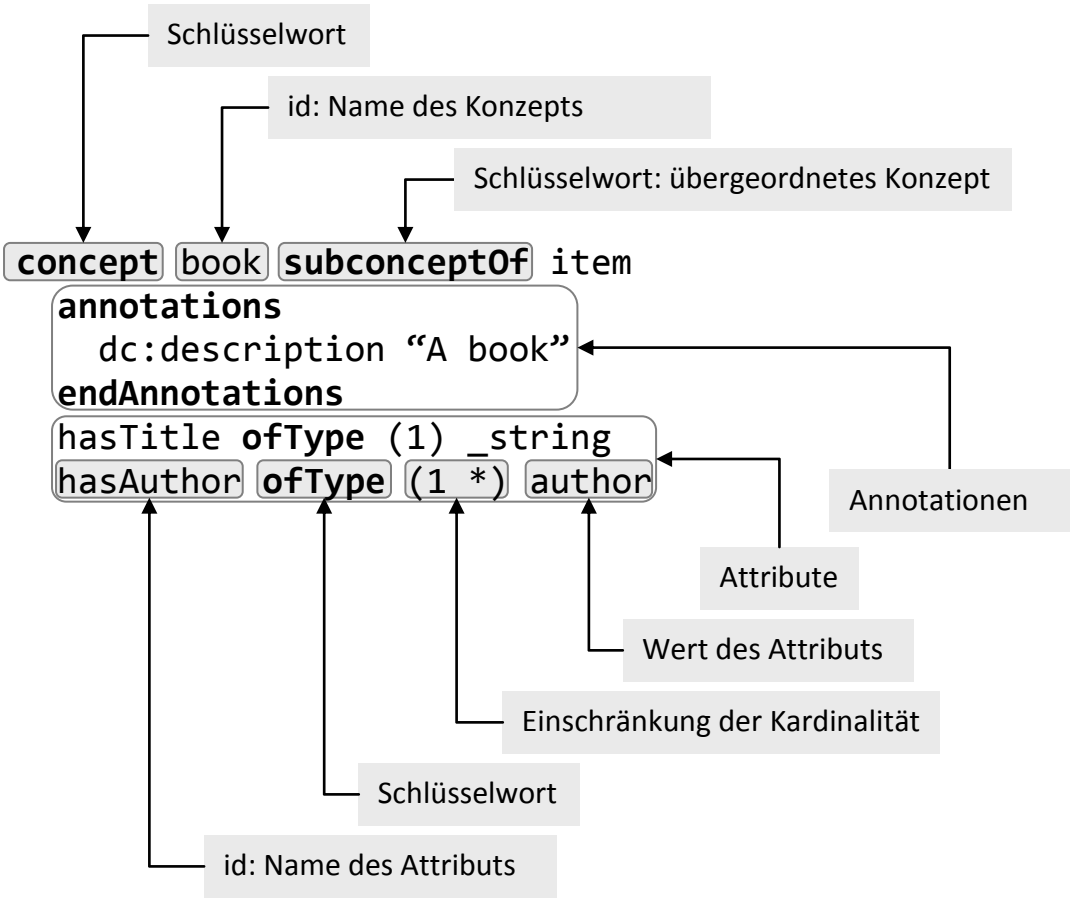


Abbildung 19: Definition des Konzepts „book“ in WSML

In Abbildung 19 ist die Definition des Konzepts „book“ in WSML dargestellt. Es ist Teil der sich in Anhang F befindlichen Ontologie „onlineshop“. Das Konzept „book“ (Buch) wird als Unterkonzept des Konzepts „item“ (Ware) definiert und besitzt zwei Attribute. Das Attribut „hasTitle“ ist vom Typ „string“. Eine Instanz des Attributs muss mit exakt einem Wert belegt werden, z.B. „Harry Potter“. Da jedes Buch mindestens einen Autor potentiell jedoch mehrere Autoren haben kann ist die Kardinalität des Attributs „hasAuthor“ entsprechend eingeschränkt. Der Wert einer Instanz

des Attributs muss eine Instanz des Typs „author“ sein, z.B. „joanneKRowling“. Zusätzlich ist das Konzept „book“ mit einer Instanz des Konzepts „description“ aus der Dublin Core Ontologie[6] annotiert, die häufig zur Beschreibung von Metadaten verwendet wird.

2.2.1 OWL-S

Die Web Ontology Language for Services (OWL-S) [MBH+04] war der erste Ansatz, Dienste semantisch, d.h. unter Zuhilfenahme von Ontologien zu beschreiben. Sie wurde 2004 beim W3C zur Standardisierung eingereicht, ist aber bis heute nicht über den Status einer „Submission“ herausgekommen. OWL-S verwendet OWL zur Definition einer Ontologie für Dienste (siehe Abbildung 20) mit dem Konzept *Service* als oberstes Element, welches über die drei Attribute (i) *presents*, (ii) *describedBy* und (iii) *supports* verfügt. Die Attribute sind vom Typ (i) *ServiceProfile*, (ii) *ServiceModel* und (iii) *ServiceGrounding* und widmen sich jeweils einer der folgenden drei essentiellen Fragen: (i) Was bietet ein Dienst einem potentiellen Klienten an?, (ii) wie kann der Dienst genutzt werden? Und (iii) wie können Klienten mit dem Dienst interagieren?

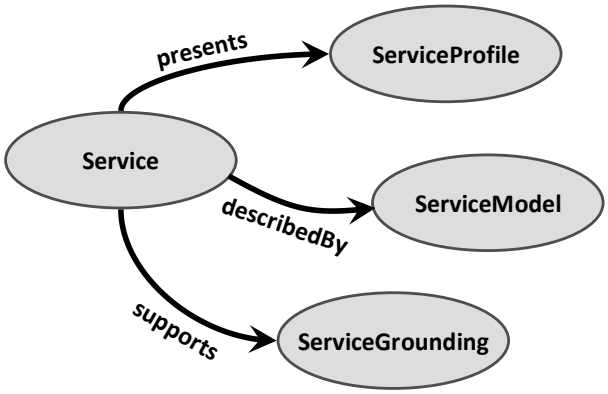


Abbildung 20: OWL-S Modell zur Beschreibung von Web-Services [MBH+04]

Das Konzept ServiceProfile beschreibt die Funktionalität des Dienstes durch *Inputs* (einkommende Nachrichten), *Outputs* (versendete Nachrichten), *Preconditions* (Vorbedingungen) und *Effects* (Nachbedingungen). Inputs und Outputs machen Aussagen über die Daten

[6] `http://dublincore.org/`

die der Dienst entgegennimmt bzw. zurückgibt. Preconditions und Effects hingegen machen Aussagen bzw. formulieren Bedingungen über den Zustand der Welt vor und nach dem Dienstaufruf.

Das Konzept ServiceModel beschreibt welche Nachrichtenreihenfolge eingehalten werden muss, um die Funktionalität des Dienstes abzurufen, d.h. in welcher Reihenfolge die Inputs und Outputs vom Dienst verarbeitet bzw. produziert werden können. Es setzt sich aus sogenannten *Atomic Processes* bzw. *Composite Processes* zusammen. Atomic Processes entsprechen in ihrem Verhalten den WSDL 1.1-Operationen. Composite Processes definieren Block-basiert einen Kontrollfluss zwischen Atomic Processes bzw. anderen Composite Processes mittels unterschiedlicher Kontrollkonstrukte. Diese sind: Sequence für sequentielle Ausführung, Split und Split-Join für parallele Ausführung, Any-Order für die sequentielle Ausführung in beliebiger Reihenfolge, Choice für die nichtdeterministische Wahl eines Pfades, If-then-else für die deterministische Wahl eines Pfades, Iterate, Repeat-While und Repeat-Until für die Definition einer Schleife.

Das Konzept ServiceGrounding beschreibt schließlich wie die Nachrichten technisch übermittelt werden. OWL-S baut hierbei auf WSDL 1.1 auf. Das Grounding beschreibt, wie die Ontologiedaten auf XML abgebildet werden und welche WSDL-Operation zur Übertragung der Nachricht verwendet wird. Die kleinste Einheit des ServiceModels, der Atomic Process, wird auf eine WSDL-Operation mit entsprechendem Nachrichtenaustauschmuster abgebildet. Außerdem kann für einen AtomicProcess die WSDL-`<service>`-Beschreibung sowie das zu verwendende `<port>`-Element, d.h. der konkrete Endpunkt, angegeben werden.

Diskussion

OWL-S beschreibt lediglich die Sicht des Dienstes; der Dienstnutzer ist nicht Teil des Models. Das hat zur Folge, dass die Interaktion während des gesamten Nachrichtenaustausches vom Dienstnutzer ausgehen muss, der Request-Response bzw. One-way-Operationen des Dienstes aufruft. Die Möglichkeit, dass der Dienst den Dienstnutzer kontaktiert ist bei OWL-S nicht gegeben. Dies reicht zwar nach Ansicht der Autoren von OWL-S aus, welche die Interaktion mit einem Web-Service lediglich als eine Reihe von

Remote Procedure Calls (RPC) beschreiben [MBH+04], ist jedoch nicht praktikabel, wenn es sich bei dem zu beschreibenden Dienst um einen lang laufenden Geschäftsprozess handelt. In diesem Fall ist asynchrone Kommunikation notwendig. Prinzipiell ist asynchrone Kommunikation bei Web-Services auch über ein asynchrones Binding, z.B. über ein JMS-Binding, möglich. Da das WS-I Basic Profil jedoch vorschreibt, dass lediglich das in der WSDL-Spezifikation definierte HTTP -Binding verwendet werden darf, muss die asynchrone Kommunikation auf Operationsebene stattfinden: Sowohl Dienst als auch Dienstnutzer müssen (One-way) Operationen anbieten, um sich gegenseitig aufzurufen, wie in Kapitel 2.1.2 bereits beschrieben. Dies ist aber in OWL-S nicht möglich. Im konzeptionellen Modell von OWL-S fehlt die Beschreibung des Klienten. Dadurch ist nicht modellierbar, dass der Klient eine Operation anbietet. Ein weiterer Nachteil der sich daraus ergibt ist, dass es nicht möglich ist zu modellieren, dass der Dienst nacheinander mehrere Nachrichten verschickt ohne zwischendurch Nachrichten zu empfangen. Das liegt wiederum daran, dass für einen Atomic Process, der eine einzelne Nachricht verschickt, kein Grounding definiert werden kann: Eine Notification Operation ist nicht zulässig und der Kommunikationspartner, der eine der Notification Operation des Dienstes entsprechende One-way-Operation anbieten kann, ist nicht Teil des konzeptionellen Modells von OWL-S. Von OWL-S unterstützte Szenarien sind somit lediglich RPCs, das Empfangen einer einzelnen Nachricht ohne Rückantwort, und jede beliebige Kombination dieser zwei Szenarien.

Da es das Konzept eines Klienten in OWL-S nicht gibt, kann ein Dienstnutzer nur beschreiben, wie der Dienst aussehen soll, den er nutzen möchte. Er kann nicht beschreiben, wie seine eigenen Inputs und Outputs aussehen, wie er sich verhält und wie der Nachrichtenaustausch auf eigene WSDL-Operationen abgebildet wird. Ein Klient, der einen Dienst sucht, kann dies lediglich anhand eines Templates des Dienstes tun.

Beispielszenario Onlineversandhandeldienst

Das in Abschnitt 2.1.1.2 und 2.1.2.3 vorgestellte Beispielszenario kann demnach – wie in der Folge beschrieben – nur teilweise durch OWL-S abgebildet werden.

Zur Beschreibung der auszutauschenden Nachrichten sowie der Vor- und Nachbedingungen wird das ServiceProfile folgendermaßen spezifiziert: Die ausgetauschten Nachrichten (Inputs und Outputs) sind in diesem Fall (i) die Bestellung, repräsentiert durch das Konzept „order“ (Input), (ii) die Rechnung, repräsentiert durch das Konzept „invoice“ (Output), (iii) die Kreditkartendetails repräsentiert durch das Konzept „creditCard“ (Input), (iv) die Bestätigungsnachricht repräsentiert durch das Konzept „receipt“ (Output) sowie (v) die Lieferdetails repräsentiert durch das Konzept „shippingDetails“ (Output). Eine Bedingung, die vor dem Aufruf des Onlineversandhandeldienstes gelten muss, ist, dass die Kreditkartennummer und die Artikelnummer der Artikel in der Bestellung existieren und dass die Kreditkarte gedeckt ist. Eine Nachbedingung des Dienstaufrufs ist, dass die Kreditkarte nach dem Dienstaufruf belastet ist.

Die Reihenfolge der auszutauschenden Nachrichten wird über das ServiceModel spezifiziert. Da ein Atomic Process entweder jeweils eine Nachricht empfangen und senden oder lediglich eine Nachricht senden oder empfangen kann, gibt es mehrere Möglichkeiten den Nachrichtenaustausch des Onlineversandhandeldienstes abzubilden. Er kann bespielweise durch eine Sequenz aus fünf Atomic Processes, die jeweils den Austausch einer Nachricht repräsentieren, modelliert werden. Denkbar ist aber auch eine Sequenz aus zwei Atomic Processes, die jeweils eine Nachricht empfangen und eine senden und einem Atomic Process, der lediglich eine Nachricht sendet sowie Sequenzen mit anderen Kombinationen aus Atomic Processes, die den Austausch einer Nachricht repräsentieren, und Atomic Processes, die jeweils eine Nachricht empfangen und senden.

Das ServiceGrounding kann nur für die ersten vier Nachrichten angegeben werden. Diese können jeweils paarweise auf Operationen des Typs Request-Response abgebildet werden: Das Konzept „order“ auf den Input

der Operation orderItem, das Konzept „invoice“ auf den Output derselben operation, das Konzept „CreditCardInformation“ auf den Input der Operation payBill und das Konzept „receipt“ schließlich auf den Output der Operation payBill. Ein Grounding für das Verschicken der Lieferdetails ist mit OWL-S nicht möglich. Zusätzlich kann die WSDL-`<service>`-Beschreibung „onlineShop“ als Grounding für die beiden Atomic Processes angegeben werden.

2.2.2 WSMO

Wie auch OWL-S befasst sich die 2005 dem W3C übermittelte Spezifikation der Web Service Modelling Ontology (WSMO) [LPR05] mit der Beschreibung von Diensten mittels Ontologien. WSMO geht nicht von WSDL als Basis aus sondern formalisiert das konzeptionelle Modell des Web Service Modelling Frameworks (WSMF) [FeBu02] und beschränkt sich daher nicht wie OWL-S allein auf die Beschreibung von Diensten sondern beschreibt auch die Ziele (Goals) des Service Nutzers.

Ein Dienst wird in WSMO über seine *Capability* (Fähigkeit) und sein *Interface* (Schnittstelle) beschrieben. Die Beschreibung der Capability erfolgt wie bei OWL-S über die Definition der Daten, die der Dienst empfangen, verarbeiten und versenden kann sowie den erforderlichen Zustand der realen Welt vor Dienstaufruf und die Auswirkungen des Dienstaufrufs auf den Zustand der realen Welt. Allerdings werden dafür unterschiedliche Begriffe verwendet. *Preconditions* und *Postconditions* entsprechen Inputs und Outputs in OWL-S und *Assumptions* entsprechen den Preconditions in OWL-S. Lediglich die Auswirkungen des Dienstaufrufs auf die reale Welt werden in beiden Ansätzen als *Effects* bezeichnet. Die Beschreibung des Interfaces gliedert sich in die Beschreibung der *Choreography* und der *Orchestration*. Die Choreography in WSMO beschreibt nicht etwa den Nachrichtenaustausch mehrerer Parteien aus der globalen Sicht eines außen stehenden Beobachters wie in [BDO05] definiert, sondern lediglich in welcher Reihenfolge der Dienst welche Daten eines Klienten konsumieren kann bzw. dem Klienten bereitstellen kann. Die Orchestration in WSMO beschreibt hingegen die Interaktion des Dienstes mit anderen Diensten, um die angebotene Funktionalität bereitstellen zu

können. Sowohl Choreography als auch Orchestration werden mittels einer *„ontologisierten Abstract State Machine“* (ASM) beschrieben, d.h. einer ASM die auf einer Ontologie als Datenbasis operiert. Das Grounding der Choreography wird in [KMV+07] beschrieben. WSMO gibt die Kommunikationsinfrastruktur nicht vor, mit der Klienten und Dienste miteinander kommunizieren, es ist also technologieneutral. Bislang wird jedoch nur ein Grounding auf Web-Service-Technologie beschrieben. Über eine nichtfunktionale Eigenschaft (nonfunctionalProperty) wird das `<service>`-Element einer WSDL-Beschreibung referenziert, welches den Port, also den Endpunkt des Dienstes enthält. Die einzelnen Nachrichten innerhalb der Choreography werden auf entsprechende Nachrichten innerhalb von WSDL-Operationen abgebildet. Sowohl die Operation mit der die Nachrichten übermittelt werden als auch der Endpunkt der Operation ist somit durch das Grounding festgelegt.

Zur Beschreibung eines Ziels bietet WSMO die gleichen Grundbausteine wie zur Beschreibung der Dienste. Ziele entsprechen den Beschreibungen von Diensten, die den Anforderungen des Klienten genügen. Ein Ziel beschreibt also nicht die Sicht des Klienten auf das zu lösende Problem, sondern beschreibt wie ein Dienst aussehen muss, mit dem der Klient interagieren möchte.

Zudem definiert WSMO neben der Beschreibung eines Dienstes und der Ziele eines Dienstnutzers das Konzept des Mediators (Vermittler). Dieser kann unter anderem dazu verwendet werden, auszudrücken, dass ein Ziel von einem bestimmten Dienst erfüllt wird. Vor allem aber trägt WSMO durch die Definition des Mediators auch der Heterogenität einer Dienstlandschaft Rechnung: Wenn ein Dienst prinzipiell befähigt ist, die Ziele eines Dienstnutzers zu verwirklichen, deren Fähigkeiten bzw. Anforderungen aber durch unterschiedliche Ontologien beschrieben werden bzw. ein unterschiedliches aber nicht vollständig inkompatibles Verhalten vorliegt, können in ihm Datamediation bzw. Prozessmediation definiert werden. Bei der Mediation ist der Einsatz von Ontologien von entscheidender Bedeutung: Eine Datenstruktur kann für viele semantisch unterschiedliche Inhalte verwendet werden. Transformationen von strukturierten Daten, z.B. von einer XML Struktur zu einer anderen, kann

man daher nur für bestimmte Anwendungsfälle definieren. Ein xs:integer kann beispielsweise nicht immer einfach in einen xs:integer kopiert werden, sondern nur dann, wenn man weiß, dass es sich bei beiden um das gleiche handelt, z.B. um einen Betrag in Euro. In BPEL vergibt der Modellierer durch einen geeigneten Variablenname die Semantik der verwendeten Struktur und ermöglicht somit die semantisch korrekte Verwendung der Daten. Die Bedeutung der Daten ist dabei aber nur innerhalb des Prozesses klar. Verwendet man Ontologien, so ist die Bedeutung der Daten global definiert. Dadurch wird es ermöglicht, über eine Anwendung hinaus wiederverwendbare Mediation zu definieren.

WSML zur Modellierung von WSMO

```
wsml            = wsmlvariant? namespace? definition*
definition      = goal | ontology | webservice |mediator | capability | interface
goal            = 'goal' iri? header* nfp* capability? interface*
header          = importsontology | annotations  | usesMediator
usesmediator    = 'usesMediator' irilist
nfp             = 'nfp' attributevaluenfp logDefinitionNfp? |
                  'nonFunctionalProperty' attributevaluenfp logDefinitionNfp?
capability      = 'capability' iri? header* nfp* sharedvardef? prPoAsEf*
sharedvardef    = 'sharedVariables' variablelist
prPoAsEf        = 'precondition' axiomdefinition |'postcondition'
                  axiomdefinition | 'assumption' axiomdefinition |
                  'effect' axiomdefinition
interface       = 'interface' iri? header* nfp* choreography? orchestration?
choreography    = 'choreography' iri? header* state_signature? transitions?
webservice      = 'webService' iri? header* nfp* capability? interface*
mediator        = wgMediator | …
wgmediator      = 'wgMediator' iri? header* nfp* sources? target? useService?
```

Abbildung 21: WSML-Syntax zur Definition von Zielen, Diensten und Mediatoren in BNF Schreibweise

Zur Definition von Diensten, Zielen und Mediatoren in WSMO wird die bereits in Abschnitt 2.2 eingeführte Web Service Modelling Language

(WSML) verwendet. Sie eignet sich nicht nur dazu Ontologien zu definieren, sondern hat darüber hinaus eine native Unterstützung zur Modellierung aller WSMO-Elemente, d.h. sie definiert hierzu eigene Sprachkonstrukte. Die zur Modellierung von WSMO relevanten Sprachkonstrukte sind in Abbildung 21 dargestellt.

WSMX – die Referenzimplementierung von WSMO

Das Web Service Modelling eXecution environment (WSMX) [HCM+05] ist die Referenzimplementierung von WSMO. Ein Klient interagiert nicht direkt mit einem Dienst sondern indirekt über WSMX. Aufgrund der dadurch erlangten zentralen Rolle ist WSMX geeignet, um sowohl Datenmediation als auch Prozessmediation durchzuführen. WSMX ist in Java implementiert und stellt seine Funktionalität als Web-Service-Operationen zur Verfügung. Die Funktionalität von WSMX ist anhand von Ausführungssemantiken [ZaOr05] spezifiziert, die mit Hilfe von UML Aktivitätsdiagrammen [OMG09] beschrieben sind. Die spezifizierte Funktionalität umfasst u.a. die Goal-basierte Suche von Semantischen Web-Services, das anschließende Aufrufen des aufgefundenen Dienstes sowie das Auffinden und Aufrufen eines Dienstes in einem Schritt, auch Goal-basierter Aufruf von Semantischen Web-Services genannt. In allen Fällen werden Nachrichten ausgetauscht, die in WSML serialisierte WSMO-Elemente enthalten. Eine schematische Darstellung des Goal-basierten Aufrufs eines Semantischen Web-Services über die Funktion `achieveGoal(Goal, Data):Data` wird in Abbildung 22 präsentiert.

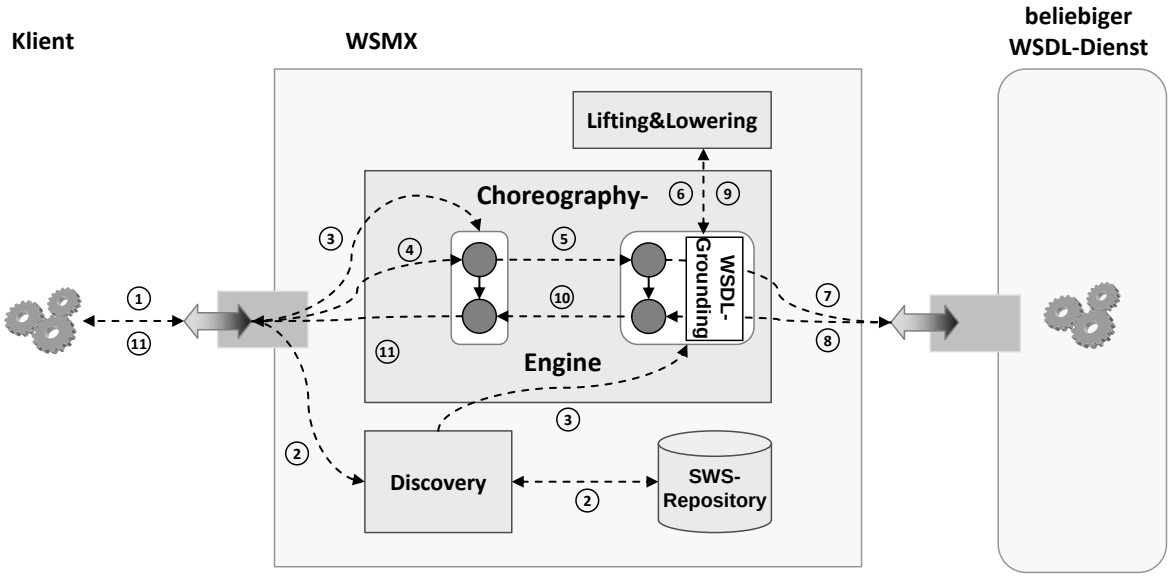


Abbildung 22: Schematische Darstellung der Ausführungssemantik „achieveGoal“

Der Aufruf beginnt mit der Übermittlung eines abstrakten Goals inklusive einer Ontologie mit Instanzen (1). Auf Grundlage des Goals wird ein passender Web-Service aus dem Web-Service-Repository ermittelt (2). Dann wird sowohl für die Goal-Choreography als auch für die Web-Service-Choreography eine der jeweiligen Beschreibung entsprechende Instanz in der Choreography-Engine gestartet (3). Beide Choreography-Instanzen arbeiten auf getrennten Wissensbasen. Die mit dem Goal übermittelten Ontologieinstanzen werden zunächst in die Wissensbasis der Goal-Choreography geladen (4) und von dort dann in die Wissensbasis, auf der die Web-Service-Choreography operiert, übertragen (5). Wird zur Definition der Daten in der Wissensbasis der Web-Service-Choreography eine andere Ontologie bzw. andere Ontologien verwendet als in der Goal-Choreography, so werden die Daten während der Übertragung von einer Choreography-Engine-Instanz zur anderen mittels Ontologie-basierter Mediation entsprechend transformiert. Liegen anschließend in der Datenbasis der Web-Service-Choreography alle Daten vor, die zur Erstellung der Nachricht an den WSDL-Web-Service benötigt werden, so wird eine Request-Response-Operation des Web-Services aufgerufen (7). Hierzu werden die in WSML serialisierten Daten unter zu Hilfenahme der Lifting&Lowering-Komponente in das entsprechende XML-Nachrichten-

format gebracht (6). Die als Antwort eintreffende XML-Nachricht (9) des WSDL-Web-Services wird wiederum unter zu Hilfenahme der Lifting&Lowering-Komponente in Instanzen der Ontologie, mit der die Web-Service-Choreography beschrieben ist, umgewandelt (8). Anschließend werden die neuen Daten auch zurück in die Goal-Choreography übertragen (10). Auch hier kann wieder Datenmediation zum Einsatz kommen. Sind schließlich die geforderten Daten in der Goal-Choreography vorhanden, so wird der WSMX Aufruf durch Übermittlung einer Ontologie mit den Instanzen an den Klienten abgeschlossen (11).

Prinzipiell ermöglicht WSMX durch die Trennung von Goal- und Web-Service-Choreography auch Prozessmediation. Da bisher jedoch nur der eben beschrieben Goal-basierte Aufruf eines Web-Services implementiert ist und WSMX bisher keine komplexeren Interaktionen unterstützt, kommt nur Datenmediation zum Einsatz.

Diskussion

Dadurch, dass in WSMO nicht nur der Dienst sondern auch der Klient Teil des konzeptionellen Models ist, ist es prinzipiell möglich den Nachrichtenaustausch aus der jeweiligen Sicht auszudrücken und jeweils einen Endpunkt anzugeben, an den Nachrichten gesendet werden können. Das hat zur Folge, dass nicht nur der Klient den Dienst aufrufen kann, sondern auch der Dienst den Klienten. Wird zur Kommunikation die Web-Service-Technologie verwendet, so kann der Nachrichtenaustausch also WS-I konform auf Operationen des Dienstes und des Klienten abgebildet werden. Es ist somit auch möglich abzubilden, dass der Dienst dem Klienten mehrere Nachrichten nacheinander schickt, ohne zwischendurch eine Nachricht zu empfangen.

Allerdings beschreibt das Ziel wie bereits erwähnt in der W3C Submission [LPR05] nicht die Sicht des Klienten sondern beschreibt, wie ein Dienst aussehen muss, um dem Ziel zu entsprechen. Das bedeutet auch, dass unter Berücksichtigung dieser Definition kein Endpunkt angegeben werden kann, an dem der Klient Nachrichten empfangen kann und dass der Dienst den Klienten somit nicht aufrufen kann. Momentan wird daher von WSMO-Implementierungen (z.B. WSMX [HCM+05], IRS-III [DCH+04]) auch

lediglich ein RPC unterstützt. Außerdem wird von einer WSMO-Beschreibung lediglich der das `<service>`-Element und nicht das `<port>`-Element referenziert, das den konkreten Endpunkt enthält. Werden für ein `<service>`-Element mehrere `<port>`-Elemente definiert, so bleibt die Auswahl des konkreten Endpunkts dem Dienstnutzer überlassen.

Beispiel Onlineversandhandeldienst

Für das in Abschnitt 2.1.1.2 und 2.1.2.3 vorgestellte Beispielszenario ergibt sich dadurch das Folgende für die Dienstbeschreibung:

Die Beschreibung der Capability befindet sich in Anhang G. Sie macht folgende Aussagen: Als eingehende Nachrichten akzeptiert der Dienst entweder eine Instanz des Konzepts „order“ oder des Konzepts „payment“. Wird eine Instanz des Konzepts „order“ empfangen, so wird eine Instanz des Konzepts „invoice“ verschickt. Wird eine Instanz des Konzepts „payment“ empfangen, so wird sowohl eine Instanz des Konzepts „receipt“ als auch eine Instanz des Konzepts „shippingDetails“ verschickt. Von der Definition von Assumptions und Effects wird in diesem Beispiel abgesehen.

Die Choreography ist in Listing 3 dargestellt. Unter dem Schlüsselwort *in* werden die zu empfangenden Nachrichten („order“ und „payment) beschrieben, unter dem Schlüsselwort *out* die ausgehenden Nachrichten („invoice“, „receipt“ und „shippingDetails“). Unter dem Schlüsselwort *controlled* werden die Konzepte „1stStep“, „2ndStep“ und „3rdStep“ definiert, die nicht von außen manipuliert werden können. Sie werden dazu verwendet die einzelnen Schritte der Choreography explizit zu modellieren. Die *transitionRules* beschreiben schließlich die Reihenfolge, in der die Nachrichten ausgetauscht werden: Ist die Datenbasis der ASM leer, so wird eine Instanz des Konzepts „1stStep“ hinzugefügt. In diesem Zustand der ASM ist der Dienst bereit, eine Instanz des Konzepts „order“ zu empfangen und daraufhin eine Instanz des Konzepts „invoice“ zu verschicken. Um den nächsten Schritt zu ermöglichen, wird die Instanz des Konzepts „1stStep“ gelöscht und eine Instanz des Konzepts „2ndStep“ hinzugefügt. Nun kann die dritte Regel greifen.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-dl"
namespace {os _"http://www.example.org/ontologies/onlineshop#",
           ch _http://www.example.org/ontologies/onlineshop/choreography#,
           wsml _"http://www.wsmo.org/wsml/wsml-syntax#"}
webService os#service
  nonFunctionalProperty
    wsml#endpointDescription hasValue _"http://example.org/onlineshop.wsdl#
                                        wsdl11.service(onlineShop)"
  endNonFunctionalProperty

  capability os#capability

  interface os#interface
    choreography
      stateSignature
        controlled
          ch#1stStep
          ch#2ndStep
          ch#3rdStep
        in
          os#order withGrounding _"http://example.org/onlineshop.wsdl#
            wsdl11.portTypeOperation.input(sellerPT/orderItem)"
          os#payment withGrounding _"http://example.org/onlineshop.wsdl#
            wsdl11.portTypeOperation.input(sellerPT/payBill)"
        out
          os#invoice withGrounding _"http://example.org/onlineshop.wsdl#
            wsdl11.portTypeOperation.output(sellerPT/orderItem)"
          os#receipt withGrounding _"http://example.org/onlineshop.wsdl#
            wsdl11.portTypeOperation.output(sellerPT/payBill)"
          os#shippingDetails
      transitionRules
        if (neg(exists ?1stStep memberOf ch#1stStep) and
              neg(exists ?2ndStep memberOf ch#2ndStep) and
              neg(exists ?3rdStep memberOf ch#3rdStep)) then
            add(_# memberOf ch#1stStep)
        endIf
        if (?input memberOf os#order and
              exists ?1stStep memberOf ch#1stStep then
            add(_# memberOf os#invoice) and
            delete(_# memberOf ch#1stStep) and
            add(_# memberOf ch#2ndStep)
        endIf
        if (?input memberOf os#payment and
              exists ?2ndStep memberOf ch#2ndStep then
            add(_# memberOf os#receipt) and
            delete(_# memberOf ch#2ndStep) and
            add(_# memberOf ch#3rdStep)
        endIf
        If (exists ?3rdStep memberOf ch3#rdStep) then
            add(_# memberOf os#shippingDetails) and
            delete(_# memberOf ch#3rdStep)
        endIf
```

Listing 3: WSML-Beschreibung des Onlineversandhandeldienstes

Wird eine Instanz des Konzepts „payment“ empfangen, so wird eine Instanz des Konzepts „receipt“ verschickt. Zudem wird die Instanz des Konzepts „2ndStep“ gelöscht und eine Instanz des Konzepts „3rdStep“ hinzugefügt. Im letzten Schritt wird nun noch eine Instanz des Konzepts „shippingDetails“ verschickt sobald sie vorhanden ist und die Instanz des Konzepts „3rdStep“ gelöscht.

Für die ersten vier Nachrichten wird jeweils ein Grounding auf Operationen des Dienstes definiert. Für die ersten beiden auf die Operation „orderItem“ und für die beiden folgenden auf die Operation „payBill“. Für die abschließende Nachricht kann jedoch kein Grounding definiert werden, da dafür wegen der Einschränkungen des WS-I Basic Profiles keine Operation auf Seiten des Dienstes zur Verfügung steht. Die Erreichbarkeit des Dienstes wird durch die Attributbelegung „`wsml#endpointDescription hasValue _“http://example.org/onlineshop.wsdl#wsdl11.service (onlineShop)“`“ spezifiziert, durch die auf das WSDL-`<service>`-Element „onlineShop“ verwiesen wird.

2.2.3 SAWSDL

SAWSDL (Semantic Annotations for WSDL and XSD) basiert auf einer früheren Spezifikation namens WSDL-S [AFM+05] und wurde 2007 eine W3C „Recommendation“, also ein offizieller Standard. Im Gegensatz zu OWL-S und WSMO definiert SAWSDL kein eigenes konzeptionelles Modell für Dienste sondern verwendet das konzeptionelle Modell von WSDL (2.0) und ermöglicht es, WSDL- und XML Schema-Elemente mit Ontologiekonzepten zu annotieren.

SAWSDL macht keine Aussage darüber, welche Ontologiesprache zur Beschreibung der Ontologiekonzepte verwendet wird, d.h. es kann sowohl auf OWL oder WSML-Konzepte als auch auf in anderen Sprachen definierte Konzepte verwiesen werden. Es wird lediglich der Annotations-mechanismus spezifiziert. Dieser wird über Erweiterungsattribute definiert, die es erlauben unterschiedliche Bestandteile eines WSDL-Dokumentes, wie die Struktur von Input- und Output-Nachrichten, Interfaces (Porttypen) oder Operationen, mit Ontologiekonzepten zu annotieren. SAWSDL macht somit von den Erweiterungsmöglichkeiten von

WSDL bzw. XML Schema gebrauch. Zwar ist WSDL 1.1 in der 2001 spezifizierten Fassung im Gegensatz zu WSDL 2.0 nicht ausreichend erweiterbar, um semantische Annotationen an den relevanten Stellen vorzunehmen, die im Rahmen des WS-I Basic Profiles überarbeitete Version lässt dies allerdings zu. Die SAWSDL-Erweiterungen sind im Einzelnen:

modelReference

Das Erweiterungsattribut `modelReference` kann verwendet werden, um Bestandteile eines WSDL- oder XML Schema-Dokuments (WSDL/XSD Component) mit einem Konzept in einem semantischen Modell (Ontologie) zu assoziieren. Annotiert werden können unter anderem die Elemente `inputs`, `outputs` und `faults` (Fehler) innerhalb von Operationen sowie der angebotene Service.

liftingSchemaMapping & loweringSchemaMapping

Die beiden Attribute `liftingSchemaMapping` und `loweringSchemaMapping` finden unter anderem bei `input-`, `output-` und `fault-`Definitionen Verwendung. Sie ermöglichen es, auf Abbildungen zu verweisen, die definieren, wie Daten von ihrer XML-Serialisierung in eine ontologische Repräsentation überführt werden können und anders herum. Über die zur Definition der Abbildung verwendete Sprache macht SAWSDL keinerlei Aussage bzw. Einschränkung.

Listing 4 dient zur Veranschaulichung der in SAWSDL definierten Konzepte und greift das Beispiel des Onlineversandhandeldienstes wieder auf. Eine vollständige SAWSDL-Beschreibung sowohl des Dienstes als auch des Klienten findet sich in Anhang D und E. Mit Hilfe der Annotationen wird zwischen der WSDL-Beschreibung und der WSMO-Beschreibung die gleiche Beziehung hergestellt wie in Listing 3. Allerdings verweisen dieses Mal nicht die Konzepte und der WSMO-Web-Service auf die entsprechende WSDL-Beschreibung sondern anders herum: das WSDL-Element `<service>` mit der Bezeichnung „bookingService“ verweist auf den WSMO-Web-Service „http://…/onlineshop#service“, das `<input>`-Element verweist auf das Konzept „http://…/onlineshop#order“, das `<output>`-

Element verweist auf das Konzept „http://.../onlineshop#invoice“, usw. Hinzu kommt jedoch, dass die SAWSDL-Annotationen `liftingSchemaMapping` und `loweringSchemaMapping` Transformationen zum Lifting bzw. Lowering der als Input bzw. Output verwendeten Daten referenzieren und somit die Transformationslogik bekannt ist.

Es können zur semantischen Beschreibung der Nachrichten auch andere Elemente annotiert werden, z.B. die verwendeten Messages selber, die einzelnen Parts oder sogar die XML Schema Definition der Nachricht. Da diese Strukturen aber auch in anderen Operationen verwendet werden können und dort möglicherweise eine andere Semantik haben, wählen wir in unserem Beispiel die Annotation der Elemente `<input>` und `<output>` der Operation. Die Reihenfolge in der ein Klient die Operationen nutzen kann, ist durch die Referenz auf den WSMO-Web-Service und der darin enthaltenen Choreography spezifiziert.

```
<wsdl:definitions ... >
  <wsdl:portType name="sellerPT">
    <wsdl:operation name="orderItem">
      <wsdl:input message="order"
                  sawsdl:liftingSchemaMapping="http://.../tItem2Ont.xslt"
                  sawsdl:loweringSchemaMapping="http://.../Ont2tItem.xslt"
                  sawsdl:modelReference="http://.../onlineshop#order" />
      <wsdl:output message="invoice"
                   sawsdl:liftingSchemaMapping="http://.../tInvoice2Ont.xslt"
                   sawsdl:loweringSchemaMapping="http://.../Ont2tInvoice.xslt"
                   sawsdl:modelReference="http://.../onlineshop#invoice" />
    </wsdl:operation>...
  </wsdl:portType>
  <wsdl:service name="onlineShop"
                sawsdl:modelReference="http://.../onlineshop#service">
    ...
  </wsdl:service>
  ...
</wsdl:definitions>
```

Listing 4: SAWSDL-Beschreibung des Onlineversandhandeldienstes

2.3 Prozessflexibilität

Wie bereits in Abschnitt 2.1.2 erwähnt, umfasst ein Workflow im Allgemeinen drei Dimensionen. Es bietet sich daher an, die Flexibilität von Workflows bzw. Prozessen auch anhand dieser drei Dimensionen zu

betrachten. Diese Arbeit beschäftigt sich mit der Flexibilisierung der ‚womit'-Dimension, d.h. der Wahl der Aktivitätsimplementierungen, der Sprache BPEL. Die Flexibilisierung der ‚wie'-Dimension, d.h. des Kontroll- und Datenflusses, ist nicht Teil dieser Arbeit. Deshalb wird in diesem Abschnitt auf die Flexibilität der ‚womit'-Dimension, nicht aber der ‚wie'-Dimension eingegangen. Auch die Flexibilität der ‚wer'-Dimension ist in dieser Arbeit nicht von Relevanz, da diese von der Kernspezifikation von BPEL nicht direkt unterstützt wird sondern nur über die Erweiterung BPEL4People [AAD+07a], die es ermöglicht, Human Tasks einzubinden, die entsprechend der Spezifikation WS-HumanTask [AAD+07b] defininiert sind.

Im Vergleich zu älteren Workflowsprachen (z. B. FDL (Flow Definition Language) [LeRo00]) verfügt BPEL bereits über eine gewisse Flexibilität bzgl. der Wahl der Aktivitätsimplementierungen, weil nur gegen Porttyp und Operation, d.h. nur gegen die abstrakte Beschreibung eines Dienstes, gebunden wird. Die konkrete Instanz einer Aktivitätsimplementierung kann zur Laufzeit ermittelt werden. Flexibler als BPEL verhalten sich jedoch die direkten Vorgängersprachen von BPEL, WSFL [Le01] und XLANG [Th01], die auch beide von der Web-Service-Technologie zur Spezifikation der ‚womit'-Dimension Gebrauch machen. Bei beiden wird der durch einen Prozess definierte Nachrichtenaustausch vollständig mittels der eigenen Servicebeschreibung abgedeckt. Dazu werden alle von WSDL 1.1 definierten Operationstypen verwendet. Über das Konzept eines Pluglinks in WSFL bzw. einer Portmap in XLANG werden der in der eigenen Servicebeschreibung definierte Porttyp bzw. die darin enthaltenen Operationen auf die Operationen einer potentiellen Aktivitätsimplementierung abgebildet. Somit wird bei WSFL und XLANG im Vergleich zu BPEL eine höhere Entkopplung von Dienstnutzer und Dienstanbieter erreicht, da sich beide nicht nur auf der konkreten Ebene der Endpunktreferenz sondern auch auf abstrakter Ebene nicht aufeinander beziehen. Um dies zu erreichen werden jedoch, wie bereits erwähnt, entgegen der Richtlinien des WS-I Basic Profiles [BEF+06] auch die Operationstypen Notification und Solicit-Response in den Servicebeschreibungen verwendet. Die Ansätze von WSFL bzw. XLANG sind

somit nicht WS-I konform und finden sich auch nicht im Modell des Nachfolgers BPEL wieder.

Im Folgenden werden Ansätze zur Flexibilisierung der womit-Dimension von BPEL-Prozessen vorgestellt, die den der Interoperabilität geschuldeten Einschränkungen der WS-I Organisation genügen.

Mandell und MCIlraith [MAMC03] verfolgen einen Proxy-basierten Ansatz zur Flexibilisierung der womit-Dimension von BPEL-Prozessen. Dienstaufrufe eines BPEL-Prozesses werden an einen dedizierten Dienst zur Auffindung der eigentlichen Aktivitätsimplementierung delegiert. Dieser wird im aufrufenden BPEL-Prozess durch einen Porttypen und eine WSDL-Operation identifiziert. Beim Aufruf des Proxy-Dienstes wird eine OWL-S Beschreibung übermittelt, die verwendet wird, um eine geeignete Aktivitätsimplementierung aufzufinden. BPEL wird also auf einer anderen Abstraktionsebene verwendet. Beim normalen Einsatz von BPEL wird die abstrakte Beschreibung eines Dienstes, der eine Fachlichkeit realisiert, in ein Prozessmodell verdrahtet, das von einem fachlichen Prozessmodell abgeleitet ist. Bei Mandell und McIlraith wird stattdessen die Schnittstelle einer Infrastrukturkomponente in das Prozessmodell kodiert und nicht die Anforderung an eine Aktivitätsimplementierung. Über eine etwaige Spracherweiterung von BPEL wird ebenso wenig eine Aussage getroffen wie über asynchrone und langlaufende Kommunikation zwischen Prozess und Aktivitätsimplementierung. Daher ist auch nicht klar, ob der Ansatz Anpassungen an einer vorhandenen BPEL-Engine erfordert.

Meteor-S [VGS+05] verfolgt ebenfalls einen Proxy-basierten Ansatz und verwendet zur Beschreibung der Aktivitätsimplementierung ebenfalls OWL-S. Der Proxy wird in diesem Fall Prozesskonfigurationsmodul genannt und ermöglicht es, Dienste zum Deployment oder zur Laufzeit zu binden. Das Prozesskonfigurationsmodul ist statusbehaftet. Reicht der Aufruf eines einzelnen Dienstes nicht aus, um die geforderte Funktionalität zu erhalten, so erstellt das Prozesskonfigurationsmodul einen Ausführungsplan, der den Aufruf mehrerer Dienste koordiniert, die gemeinsam die geforderte Funktionalität erbringen können. Die Schnittstelle des Prozesskonfigurationsmoduls wird wie beim Ansatz von

Mandell und MCIlraith in das Modell des aufrufenden Prozesses kodiert. BPEL wird also auch hier auf einer anderen Abstraktionsebene verwendet. Asynchrone und lang laufende Kommunikation zwischen Prozess und Prozesskonfigurationsmodul wird nicht betrachtet. Über eine etwaige Spracherweiterung von BPEL und der daraus resultierenden Notwendigkeit der Erweiterung einer BPEL-Engine wird keine Aussage getroffen.

Auch in [St09] wird ein Proxy-basierten Ansatz verfolgt. Zur Beschreibung der Anforderungen an eine Aktivitätsimplementierung kommen jedoch nicht OWL-S-Beschreibungen sondern WSMO-Goals zum Einsatz. Die Schnittstelle des Proxy, in diesem Fall Semantic Invocation Service (SISi) genannt, wird ebenfalls in den BPEL-Prozess kodiert. SISi nimmt das WSMO-Goal und die Instanzdaten entgegen, findet einen passenden Dienst zur Implementierung der `<bpel:invoke>`-Aktivität, ruft diesen auf und gibt das Ergebnis zurück. Es werden lediglich einmalige, synchrone Aufrufe unterstützt. Auf asynchrone, langlaufende Kommunikation zwischen BPEL-Prozess und Aktivitätsimplementierung, die prinzipiell mit WSMO und einer entsprechenden Implementierung möglich sind, wird nicht eingegangen. Eine Erweiterung der Sprache BPEL findet nicht statt. Es können daher konventionelle BPEL-Engines verwendet werden.

Karastoyanova et al. [KLNW06] erhöhen die Flexibilität von BPEL-Prozessen durch Parametrisierung der `<bpel:invoke>`-Aktivität und entsprechender Implementierung einer BPEL-Engine. Die Parametrisierung ermöglicht es, die Definition von `<bpel:invoke>`-Aktivitäten zur Laufzeit zu ändern. Es werden drei Strategien zur Anpassung von Porttyp und Operation genannt: Das statische Überschreiben der Porttyp- und Operationsdefinition („static"), die manuelle Änderung durch einen Administrator („prompt") sowie das automatische Ändern von Porttyp und Operation nach erfolgter semantischer Suche („query"). Die Strategie „query" verwendet OWL-S Beschreibungen zum Auffinden entsprechender Aktivitätsimplementierungen. Alle Strategien, auch die der semantischen Suche, unterstützen lediglich die Änderung von Porttyp und Operation einer einzelnen synchronen `<bpel:invoke>`-Aktivität.

Ein weiterer Ansatz der Flexibilisierung, insbesondere der womit-Dimension von BPEL-Prozessen ist die Aspektorientierung [KaLe09]. Sie ermöglicht es, zur Laufzeit Aktivitäten wegzulassen, neu hinzuzufügen oder statt einer vordefinierten Aktivität eine andere auszuführen. Eine `<bpel:invoke>`-Aktivität kann also zur Laufzeit gegen eine andere `<bpel:invoke>`-Aktivität ausgetauscht werden, für die ein anderer Porttyp und eine andere Operation definiert ist. Zur Umsetzung der Aspektorientierung ist die Erweiterung einer BPEL-Engine erforderlich.

Alle vorgestellten Ansätze haben gemeinsam, dass Porttyp und Operation einer potentiellen Aktivitätsimplementierung zur Designzeit in das Prozessmodell kodiert werden müssen. Sie gehören gemäß der in [Ka06] eingeführten Kategorisierung der Flexibilität zur Kategorie "Flexibilität durch Änderung". Der in dieser Arbeit verfolgte Ansatz gehört im Gegensatz dazu in die Kategorie "Flexibilität durch Weglassen". Ein weiteres Differenzierungsmerkmal des in dieser Arbeit vorgestellten Ansatzes gegenüber den bisherigen Ansätzen ist die Unterstützung der Kommunikationsarten. Alle bisherigen Ansätze beschränken sich auf die Flexibilisierung einer einzelnen `<bpel:invoke>`-Aktivität, die synchron einen Dienst aufruft. Der in dieser Arbeit vorgestellte Ansatz hingegen ermöglicht asynchrone, lang laufende Kommunikation eines BPEL-Prozesses mit einem Dienst bzw. mit einem Nutzer des vom BPEL-Prozess angebotenen Dienstes.

2.4 Zusammenfassung

In diesem Kapitel wurden die für die Arbeit nötigen Grundlagen gelegt. Es wurden zum einen die Sprachen WSDL und BPEL vorgestellt, die Teil des Web-Service-Schichtenmodells sind. Es wurde ein Beispielszenario eingeführt, welches das Zusammenspiel von WSDL und BPEL verdeutlicht und es wurde gezeigt, welche Interaktionszenarien BPEL unterstützt. Zum anderen wurden die zurzeit populärsten SWS-Frameworks OWL-S und WSMO sowie der Standard SAWSDL vorgestellt, der eine Verbindung zwischen XML-Welt und Ontologiewelt ermöglicht. Außerdem wurden verwandte Arbeiten im Themenkomplex Prozessflexibilität behandelt.

Neben der Einführung in die Grundlagen und der verwandten Arbeiten liefert dieses Kapitel auch einen ersten Beitrag dieser Arbeit. Es wurden die beiden SWS-Frameworks OWL-S und WSMO hinsichtlich ihrer Tauglichkeit für den Einsatz in lang laufenden Geschäftsprozessen analysiert und es wurde gezeigt, bis zu welchem Grad sie sich zur Unterstützung des Beispielszenarios eignen. Diese Analyse findet sich auch in der bereits veröffentlichten Publikation [NLKL07].

KAPITEL 3

BPEL^{LIGHT} – EINE VON WSDL UNABHÄNGIGE PROZESSAUSFÜHRUNGSSPRACHE

Dieses Kapitel stellt eine auf BPEL basierende Workflowsprache namens BPEL^{light} vor. Zunächst werden Defizite von BPEL bezüglich der Flexibilität bei der Wahl potentieller Kommunikationspartner aufgezeigt. Die aufgezeigten Defizite motivieren die Definition einer neuen, von WSDL unabhängigen BPEL-basierten Workflowsprache. Anschließend werden mögliche Ansätze zur Definition einer derartigen Workflowsprache analysiert und bewertet. Es folgt die Formulierung der konkreten Anforderungen an die neu zu definierende Sprache. Diese wird dann in den folgenden Abschnitten sukzessive vorgestellt: Zunächst wird ein von WSDL unabhängiges, nachrichtenbasiertes Kommunikationsmodell präsentiert (Abschnitt 3.1), welches Unterstützung für alle mit BPEL abbildbaren Interaktionsszenarien bietet. Anschließend wird ein Profil für abstraktes BPEL^{light} vorgestellt (Abschnitt 3.2), welches es erlaubt, häufig auftretende Nachrichtenaustauschmuster zu modellieren. In einem weiteren Schritt werden diese Nachrichtenaustauschmuster dazu verwendet, das Verhalten von BPEL^{light}-Aktivitäten (Abschnitt 3.3) sowie BPEL^{light}-Prozessfragmenten (Abschnitt 3.4) zu beschreiben. Es folgt die Modellierung des in Kapitel 2 eingeführten Beispielprozesses (Abschnitt 3.5) sowie die methodologische Betrachtung der Überführung von BPEL-Prozessen zu BPEL^{light}-Prozessen und anders herum (Abschnitt 3.6). Das Kapitel schließt mit einem konzeptionellen Vergleich von BPEL und BPEL^{light} und einer Zusammenfassung (Abschnitt 3.7).

Defizite von BPEL und Motivation für BPEL^{light}

In BPEL sind die ‚wie'-Dimension und die Beschreibung der ‚womit'-Dimension eng miteinander gekoppelt (vergleiche Kapitel 2): Zum einen sind Aktivitäten ein Aspekt der Prozesslogik (‚wie'). Zum anderen verweisen Aktivitäten auf WSDL-Porttypen und –Operationen (‚womit'). Daraus ergeben sich die folgenden Defizite für BPEL:

(i) Die Verwendung von Partnerdiensten, welche die gewünschte Funktionalität bieten, nicht jedoch die spezifizierte Schnittstelle implementieren, wird verhindert, d.h. die Flexibilität bezüglich der Wahl der Kommunikationspartner wird eingeschränkt.

(ii) Komplexe Geschäftsfunktionen, die einen komplexen Nachrichtenaustausch erfordern, können nicht durch eine einzelne Aktivität umgesetzt werden.

(iii) Die Wiederverwendung eines Prozesses oder eines Teils eines Prozesses in einem anderen Zusammenhang wird stark eingeschränkt: Der Fluss zwischen Aktivitäten kann nicht separat wieder verwendet werden, sondern nur in Kombination mit der Schnittstellenspezifikation von Aktivitätsimplementierungen. Dies gilt auch für Subprozesse.

(iv) BPEL wird auf WSDL zur Beschreibung von Aktivitäts-implementierungen eingeschränkt. Andere Beschreibungssprachen sind per Definition ausgeschlossen.

(v) Zur Modellierung eines ausführbaren Geschäftsprozesses reichen fundierte Kenntnisse der Prozessmodellierung nicht aus. Auch Kenntnisse der Web-Service-Technologie sind erforderlich.

Ansätze

Diese Defizite lassen sich beseitigen, indem man die ‚wie'-Dimension vollständig von der ‚womit'-Dimension entkoppelt, d.h. die Aktivitäten innerhalb des Prozessmodells ausschließlich zur Definition der Prozesslogik verwendet und von den Schnittstellen der Implementierung von Aktivitäten abstrahiert. Aus der Definition von BPEL ergeben sich

hierzu zwei verschiedene Ansätze. Zum einen ist es möglich, ein Profil für abstraktes BPEL zu erstellen, welches es erlaubt, bei den existierenden Interaktionsaktivitäten alle WSDL-spezifischen Angaben wegzulassen. Zum anderen können Erweiterungen zu BPEL definiert werden, z.B. neue Interaktionsaktivitäten, die per Definitionem nicht von WSDL-Beschreibungen abhängen. Diese beiden Möglichkeiten werden im Folgenden näher beleuchtet.

Die BPEL-Spezifikation sieht die Möglichkeit vor, Profile für abstraktes BPEL zu erstellen, die es ermöglichen, Informationen auszulassen, die zur Definition eines ausführbaren Prozesses erforderlich sind. Zu diesem Zweck definiert die BPEL-Spezifikation eine grundlegende syntaktische Form (*Common Base*), der alle Profile entsprechen müssen. Diese Common Base definiert beispielsweise, dass Aktivitäten, Ausdrücke (Expressions), Attribute und `<from>`-Spezifikationen unsichtbar gemacht werden können indem man sie durch so genannte „*opake Token*“ ersetzt [AAA+07]. Die Semantik eines abstrakten Prozesses ist durch die Common Base jedoch nicht definiert. Sie bleibt der Definition der Profile überlassen. In der BPEL 2.0-Spezifikation sind bereits zwei Profile für abstraktes BPEL enthalten: Eines für die Beschreibung des von außen sichtbaren Verhaltens eines Prozesses (*Observable Behaviour*) und eines für die Definition von Prozessvorlagen (*Templates*).

Neben der Definition der Semantik spezifiziert ein Profil, welche Informationen angegeben werden müssen und welche Angaben weggelassen bzw. unsichtbar gemacht werden können. Das Profil für das sichtbare Verhalten eines Prozesses gibt beispielsweise vor, dass sowohl Interaktionsaktivitäten als auch deren Attribute `partnerLink` und `operation` nicht durch opake Token ersetzt werden dürfen.

Dem generischen Ansatz der abstrakten Prozesse folgend ist es möglich, ein Profil zu definieren, welches es erlaubt, WSDL-spezifische Details auszulassen und damit die Wiederverwendbarkeit von abstrakten Prozessmodellen zu erhöhen. Dieser Vorgehensweise wurde z.B. in [DKLW07] nachgegangen um BPEL4Chor, eine auf BPEL basierende Sprache zur Beschreibung von Choreographien, zu entwickeln. Um

zusätzlich komplexe Nachrichtenaustauschmuster durch eine Aktivität zu realisieren, wäre eine Anpassung der Erweiterung BPEL-SPE um den Aspekt der Parametrisierung möglich. Dies würde es erlauben, abstrakte Subprozesse zu modellieren, die nach entsprechender Parametrisierung zur Laufzeit mit einer `<spe:call>`-Aktivität aufgerufen werden können.

BPEL ist erweiterbar. In der Version 1.1 kann die Sprache durch Zusatzattribute und –elemente erweitert werden. Allerdings ist die Erweiterbarkeit eingeschränkt, da es keine Möglichkeit gibt, neue Aktivitäten einzuführen, ohne die Spezifikation und das in ihr enthaltene XML Schema zu verletzen und damit die Kompatibilität zu BPEL zu verlieren. BPEL 2.0 trägt dieser Unzulänglichkeit Rechnung und stellt mit der `<extensionActivity>` einen Mechanismus zur Einbindung neu definierter Aktivitätstypen zur Verfügung. Zusätzlich ermöglicht es BPEL 2.0 individuelle Datenmanipulationsoperationen zu definieren. Die Erweiterungen dürfen dabei in keinem Fall die Semantik der bereits existierenden Sprachkonstrukte ändern.

Die Erweiterbarkeit von BPEL ermöglicht es unter anderem, mit Hilfe des `<extensionActivity>` Mechanismus neue Interaktionsaktivitätstypen einzuführen, die keine WSDL-Artefakte referenzieren und sowohl einfache als auch komplexe Nachrichtenaustauschmuster implementieren. Zusätzlich können Mechanismen definiert werden, die es erlauben, Aktivitäten zum Zwecke der Sichtbarkeit der Interaktionsmuster zwischen Partnern in Beziehung zu setzen. Diese Mechanismen können wie auch die Interaktionsaktivitäten unabhängig von WSDL definiert werden.

Zusammenfassend lässt sich feststellen, dass BPEL an sich keine Kommunikation vorsieht, die nicht auf WSDL basiert. Es definiert jedoch mehrere einzelne Erweiterungspunkte, die es – entsprechend miteinander kombiniert – erlauben, zusätzlich zum bereits existierenden, auf WSDL basierenden Interaktionsmodell, ein neues Interaktionsmodell zu definieren, welches ohne Abhängigkeiten zu WSDL auskommt.

Analyse und Bewertung der Ansätze

Ein Profil für abstraktes BPEL zu erstellen ermöglicht es unter anderem, WSDL-spezifische Angaben wegzulassen und damit die Wiederverwendbarkeit abstrakter Prozesse zu erhöhen. Soll ein derartiger abstrakter Prozess allerdings ausführbar gemacht werden, so müssen die Details, von denen im Profil abstrahiert wird, d.h. in diesem Fall WSDL-Porttypen und –Operationen, spezifiziert werden. Das führt wiederum dazu, dass die Dimensionen ‚wie' und ‚womit' in der Definition des ausführbaren Prozesses wieder eng gekoppelt sind. Außerdem sind Aktivitätsbeschreibungen dadurch weiterhin auf WSDL eingeschränkt und somit auch auf den Austausch maximal zweier Nachrichten mit einem Geschäftspartner. Bezieht man eine Anpassung der BPEL-Erweiterung BPEL-SPE in den Ansatz eines Profils für abstraktes BPEL mit ein, so wäre es auch möglich, komplexere Nachrichtenaustauschmuster durch eine Aktivität – die `<spe:call>`-Aktivität – umzusetzen. Dies alles ist bei der Modellierung des Prozesses zu berücksichtigen. Daher sind zwar beim Prozessmodellierer keine detaillierten Kenntnisse der Web-Service-Technologie notwendig, wohl aber Kenntnisse über die Einschränkungen die sie mit sich bringt. Die Erstellung eines Profils für abstraktes BPEL resultiert daher lediglich in einer erhöhten Wiederverwendbarkeit und Flexibilität von Prozessen oder Teilen von Prozessen während der Modellierung. Die Flexibilität während der Laufzeit wird jedoch nicht erhöht. Der Umstand, dass bei einem Profil-basierten Ansatz vor der Ausführung herkömmliches, ausführbares BPEL erzeugt wird, bedeutet aber auch, dass er keinerlei Erweiterungen vorhandener BPEL-Engines impliziert, um Prozesse auszuführen.

Die Erweiterbarkeit von BPEL zu nutzen, um ein Interaktionsmodell zu definieren, welches per Definitionem nicht von WSDL abhängt, erhöht die Flexibilität von Prozessen nicht nur während der Modellierung, sondern auch zur Ausführung. Die Flexibilität wird dadurch erhöht, dass WSDL-Beschreibungen der zu verwendenden Aktivitätsimplementierungen zu keinem Zeitpunkt mehr ein Bestandteil der Prozessbeschreibung sind. Das bedeutet, dass die Beschreibung von Aktivitäten nicht mehr an WSDL gebunden ist, sondern in jeder beliebigen geeigneten IDL vorgenommen

werden kann. Außerdem wird dadurch die Möglichkeit eröffnet, Aktivitäten zu definieren, die nicht auf den Austausch zweier Nachrichten mit einem Geschäftspartner beschränkt sind und wieder verwendbare Prozesse und Teilprozesse zu spezifizieren. Ein Nachteil einer BPEL-Erweiterung besteht jedoch in der Notwendigkeit, vorhandene BPEL-Engines dahingehend anzupassen, dass sie die Erweiterungen verstehen und ausführen können.

Zusammenfassend kann festgestellt werden, dass der Ansatz, BPEL zu erweitern, weitaus viel versprechender und mächtiger ist als das Erstellen eines Profils für abstraktes BPEL. Deshalb konzentriert sich diese Arbeit darauf, BPEL zu erweitern und somit eine Sprache zu definieren, die universell zur Definition von Geschäftsprozessen geeignet ist, und nicht nur für Geschäftsprozesse in Web-Service-Umgebungen. Da es sich um eine leichtgewichtige – weil nicht von WSDL-Beschreibungen abhängende – Sprache handelt, trägt sie den Namen BPEL^{light}.

Anforderungen an BPEL^{light}

In der Sprache BPEL^{light} sollen die Dimensionen ‚wie' und ‚womit' vollständig entkoppelt werden, d.h. Aktivitäten, die Teil des Kontrollflusses sind, sollen nicht auf Schnittstellenbeschreibungen verweisen. Das hat nicht nur die Beseitigung der genannten Schwächen zur Folge, sondern ermöglicht auch eine bessere Spezialisierung der an der Entstehung eines Prozesses beteiligten Personen: Wie in Abbildung 23 deutlich wird, sind für die Definition eines Geschäftsprozesses unter Verwendung von BPEL^{light} lediglich Kenntnisse zur Erstellung der Prozesslogik von Nöten, nicht jedoch wie bei herkömmlichem BPEL Detailwissen über Web-Services, da die Abbildung des Geschäftsprozesses auf die IT-Infrastruktur separat erfolgt. Diese Abbildung kann sowohl indirekt über eine (abstrakte) Schnittstellenbeschreibung erfolgen als auch direkt durch die Zuordnung von Aktivitäten zu konkreten Implementierungen. Die indirekte Abbildung erfordert drei Schritte: Zunächst muss während der Modellierung eine abstrakte Beschreibung der Schnittstelle der potentiell zu verwendenden Implementierung vorgenommen werden, beispielsweise unter Verwendung von WSDL. Zum Deployment bzw. zur Laufzeit müssen dann die Aktivitäten des BPEL^{light}-Prozesses der abstrakten Beschreibung und

der tatsächlichen Implementierung (dem Endpunkt) zugeordnet werden. In Abbildung 23 sind diese Schritte mit grobgestrichelter Umrandung, die direkte Zuordnung von Aktivitäten zu einem Endpunkt mit feingestrichelter Umrandung dargestellt.

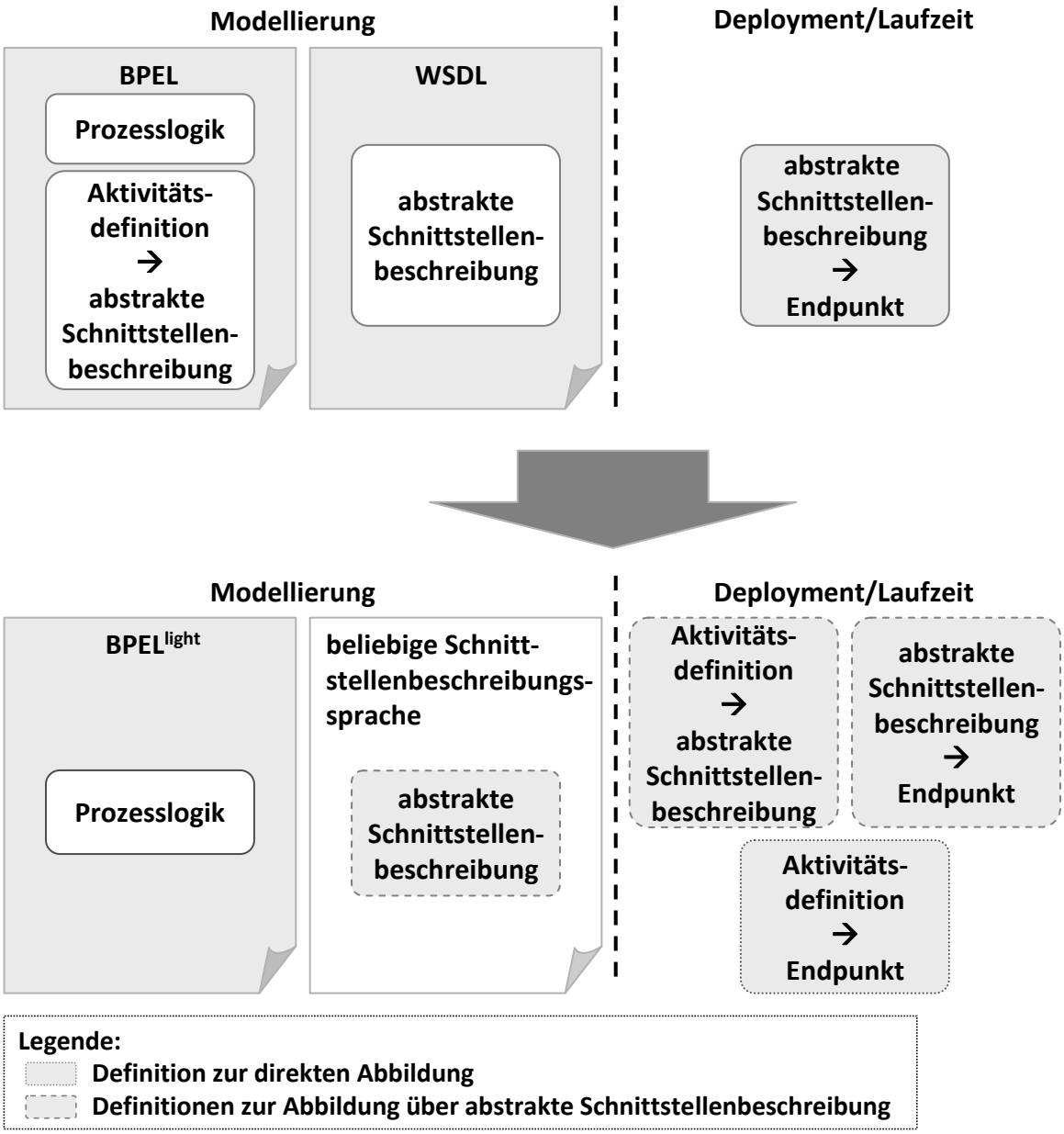


Abbildung 23: Entkopplung der Prozesslogik von der Schnittstellenbeschreibungssprache

BPEL[light] soll als Erweiterung zu BPEL definiert werden, d.h. die von BPEL definierten Erweiterungspunkte nutzen. Damit wird sichergestellt, dass die operationale Semantik von BPEL bestehen bleibt. Das trifft sowohl auf den Kontrollfluss, im Besonderen aber auch auf den Datenfluss zu: Die

Definition und Typisierung von Variablen sowie deren Zugriff und Manipulation bleiben von den Erweiterungen unberührt. Die BPEL-konforme Vorgehensweise bei der Definition der Erweiterungen und die damit verbundene Praxisnähe garantieren eine höchstmögliche Akzeptanz der Konzepte und einen höchstmöglichen Einfluss auf die Weiterentwicklung zukünftiger Workflowsprachen.

BPEL-Interaktionsaktivitäten sind maßgeschneidert auf WSDL-Operationen. Die Aktivitäten `<receive>`, `<pick>` und `<eventHandler>` implementieren eine WSDL-Operation vom Typ One-way oder in Kombination mit der `<reply>`-Aktivität eine Operation des Typs Request-Response. Die `<invoke>`-Aktivität nutzt Operationen des Typs One-way bzw. Request-Response, die von einem Partnerdienst bereitgestellt werden. Die Interaktionsaktivitäten sind also beschränkt auf den Austausch von maximal zwei Nachrichten, die zudem mit lediglich einem einzigen Partner ausgetauscht werden können. Da Aktivitäten in einem Prozess jedoch Geschäftsfunktionalität realisieren sollen, ist diese Einschränkung zu restriktiv. Als Beispiel sei das Einholen eines Angebots angeführt, bei dem eine Anfrage an mehrere Partner gesendet wird und die Angebote entgegengenommen werden. Dabei werden mehr als zwei Nachrichten gesendet bzw. empfangen und es ist nicht nur ein Partnerdienst in den Nachrichtenaustausch involviert. Um solche Szenarien nicht jedes mal neu modellieren zu müssen sondern direkt durch eine Interaktionsaktivität zu unterstützen, soll es BPEL^{light} deshalb ermöglichen, das Verhalten von Aktivitäten über generische Beschreibungen von Nachrichtenaustausch-mustern zu definieren. Dies stellt zusätzlich zur Entkopplung einen weiteren Vorteil von BPEL^{light} gegenüber herkömmlichem BPEL dar.

Das Verhalten von BPEL^{light}-Aktivitäten soll ebenfalls in BPEL^{light} beschrieben werden. Dadurch, dass die gleiche Sprache verwendet wird, um den Fluss in Aktivitäten und zwischen Aktivitäten zu definieren, wird sichergestellt, dass ein Prozessentwickler lediglich eine Sprache beherrschen muss, um einen Prozess modellieren bzw. verstehen zu können. Außerdem werden keine spezialisierten Modellierer für Nachrichtenaustauschmuster benötigt, da Prozessentwickler in der Lage sind, Nachrichtenaustauschmuster zu definieren. BPEL^{light} beschreibt

allerdings konkrete Prozesse, bei denen alle zur Ausführung nötigen Details spezifiziert sind, unter anderem auch, welche Daten, d.h. der Inhalt welcher Variablen, einer Aktivitätsimplementierung zur Verfügung stehen, und welche Daten von ihr erwartet werden. Um jedoch ein allgemeines Verhalten von Aktivitäten zu beschreiben ist es nötig, von konkreten Werten zu abstrahieren. Dazu gehören nicht nur die eben angesprochenen Datentypen, sondern beispielsweise auch die in der Definition des Kontrollflusses verwendeten Ausdrücke. Diese Abstrahierung lässt sich durch die Definition eines abstrakten Profils für BPEL^{light} erreichen. Mit Hilfe eines solchen Profils ist es möglich, häufig vorkommende Nachrichtenaustauschmuster separat zu beschreiben, die dann dazu verwendet werden können, das Verhalten von Interaktionsaktivitäten zu bestimmen. Auch bei der Beschreibung eines Nachrichtenaustauschmusters können Interaktionsaktivitäten mit komplexem Verhalten verwendet werden, die wiederum ein Nachrichtenaustauschmuster repräsentieren. Es ist also möglich Nachrichtenaustauschmuster zu komponieren und dadurch höherwertige Nachrichtenaustauschmuster zu definieren. Zusätzlich zu ausführbarem BPEL^{light} muss also auch ein Profil für abstraktes BPEL^{light} zur Definition von Nachrichtenaustauschmustern definiert werden.

3.1 BPEL^{light} zur Spezifikation ausführbarer Geschäftsprozesse

BPEL^{light} ermöglicht es, multilaterale Interaktionen, die jeweils einem Geschäftsziel dienen, zu modellieren. Um ein Geschäftsziel zu erreichen, ist es im allgemeinen Fall notwendig, mit mehreren Partnern mehrere Nachrichten auszutauschen. Als Beispiel sei hier zum einen das Ermitteln des günstigsten Angebots für ein Produkt angeführt, bei dem eine Aufforderung ein Gebot abzugeben an mehrere Partner verschickt wird und die Gebote bis zu einem vorher bestimmten Termin angenommen werden und zum anderen ein Szenario, in dem eine Zahlungsaufforderung an einen Geschäftspartner versendet wird und in einem späteren Schritt die Bestätigung der Bank über die Tätigung der Überweisung erfolgt. Das erste Szenario wird in der Folge als „*Request for Bid*“ bezeichnet und folgt

dem abstrakten Muster „One-to-Many Send/Receive“ [BDH05]. Das zweite Szenario lässt sich durch das abstrakte Muster eines „*Request with Referral*“ [BDH05] beschreiben. Einfachere Geschäftsziele lassen sich auch durch den Austausch mehrerer Nachrichten mit einem einzigen Partner bzw. mit dem Austausch einer einzelnen Nachricht mit einem einzelnen Partner realisieren.

Um die zur Umsetzung der Geschäftsziele notwendigen Interaktionen realisieren zu können führt BPEL^{light} zur Definition eines von WSDL unabhängigen Interaktionsmodells neue Elemente und Attribute in einem separaten Namensraum[7] ein. Gemeinsam können diese genutzt werden, um Interaktionen eines BPEL^{light}-Prozesses mit Geschäftspartnern unabhängig von WSDL zu definieren. Die Erweiterungen werden in den folgenden Abschnitten sukzessive eingeführt und kontinuierlich bis hin zur vollständigen Syntax ausgearbeitet. Zur besseren Lesbarkeit werden hierbei alle BPEL^{light}-Elemente mit dem Präfix „`bl:`“ versehen und alle BPEL-Elemente mit dem Präfix „`bpel:`“. Die Spezifikation der BPEL^{light}-Konstrukte erfolgt anhand einer informalen Syntax zur Beschreibung der XML Grammatik: Zur Definition der Kardinalität von Elementen und Attributen werden die Zeichen „?“ (Kardinalität 0 oder 1), „*“ (Kardinalität 0 bis unendlich) und „+“ (Kardinalität 1 bis unendlich) angefügt. Elemente und Attribute, die durch „|“ separiert werden und durch „(“ und „)“ gruppiert werden, sind Alternativen.

3.1.1 Das Rahmenwerk: Konversationen mit abstrakten Partnern

In BPEL^{light} wird eine Interaktion zur Erreichung eines Geschäftsziels durch das Konzept der Konversation repräsentiert. Eine Konversation wird durch das Element `<bl:conversation>` repräsentiert, das wie ein `<bpel:partnerLink>` innerhalb eines Prozesses (`<bpel:process>`) oder im Rahmen eines Geltungsbereichs (`<bpel:scope>`) definiert werden kann. Die in eine Konversation involvierten Partner werden durch das `<bl:partner>`-Element repräsentiert, welches ebenfalls entweder für einen ganzen Prozess oder nur einen bestimmten Geltungsbereich definiert

[7] `xmlns:bl=http://iaas.uni-stuttgart.de/BPELlight/2008/`

wird. Ein Partner repräsentiert hierbei einen bestimmten Partnertypus und nicht die Instanz eines Partners. Die Syntax eines BPEL-`<bpel:process>`-Elements mit BPEL[light]-Erweiterungen ist in Listing 5 dargestellt. Neben den neu definierten Elementen sind auch die zur Bekanntmachung der Erweiterungen nötigen Elemente, im Besonderen die Spezifikation eines entsprechendes `<bpel:extension>`-Elements, zu beachten. Die Syntax zur Definition von Konversationen und Partnern innerhalb eines `<bpel:scope>`s sind analog zu der gezeigten Definition in einem `<bpel:process>`.

```
<bpel:process name="NCName"
              targetNamespace="anyURI"
              queryLanguage="anyURI"?
              expressionLanguage="anyURI"?
              suppressJoinFailure="yes|no"?
              exitOnStandardFault="yes|no"?
              xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/
                          executable"
              xmlns:bl="http://iaas.uni-stuttgart.de/BPELlight/2008/"?>

  <bpel:extensions>
    <bpel:extension namespace="http://iaas.uni-stuttgart.de/BPELlight/
                                2008/"
                    mustUnderstand="yes"/>?...
  </bpel:extensions>
  <bl:conversations>?
    <bl:conversation name="NCName"/>+
  </bl:conversations>
  <bl:partners>?
    <bl:partner name="NCName"/>+
  </bl:partners>
  ...
  activity
</bpel:process>
```

Listing 5: Schema der mit BPEL[light]-Elementen erweiterten `<bpel:process>`-Definition

3.1.2 Die Definition des Nachrichtenaustauschs

Um den eigentlichen Nachrichtenaustausch einer Konversation auszudrücken werden zum einen ein neuer `<bl:eventHandler>` definiert und zum anderen neue Aktivitätstypen mit Hilfe des `<bpel:extensionActivity>` Mechanismus' eingeführt.

Dabei realisiert ein einziger Aktivitätstyp, die `<bl:interactionActivity>`, die operationale Semantik, die von den BPEL-Aktivitäten `<bpel:receive>`, `<bpel:reply>` und `<bpel:invoke>` implementiert wird. Sie ist in Abbildung 24 dargestellt. Sie kann sowohl lediglich eine Nachricht senden (linke Abbildung) als auch eine einzelne Nachricht empfangen (rechte Abbildung). In diesen einfachen Fällen spezifiziert sie entweder eine zu sendende Nachricht, d.h. aus Prozesssicht eine Eingabe (engl. Input) für die Aktivität, oder eine zu empfangende Nachricht, d.h. aus Prozesssicht eine Ausgabe (engl. Output) der Aktivität. Außerdem wird die Aktivität durch die Zugehörigkeit zu einer Konversation (engl. Conversation) und das Zuordnen eines Partners (engl. Partner) zu einer Nachricht in das BPEL^light Rahmenwerk eingebettet. Da Konversation und Partner bereits innerhalb eines Prozesses bzw. eines Geltungsbereiches definiert werden und von der Aktivität lediglich verwendet werden, sind diese im Gegensatz zur Eingabe bzw. Ausgabe mit einer gestrichelten Umrandung versehen. Durch die fein gestrichelte Linie werden einer Nachricht abstrakte Adressinformationen eines Partners zugeordnet, die grob gestrichelte Linie symbolisiert einen Nachrichtenaustausch.

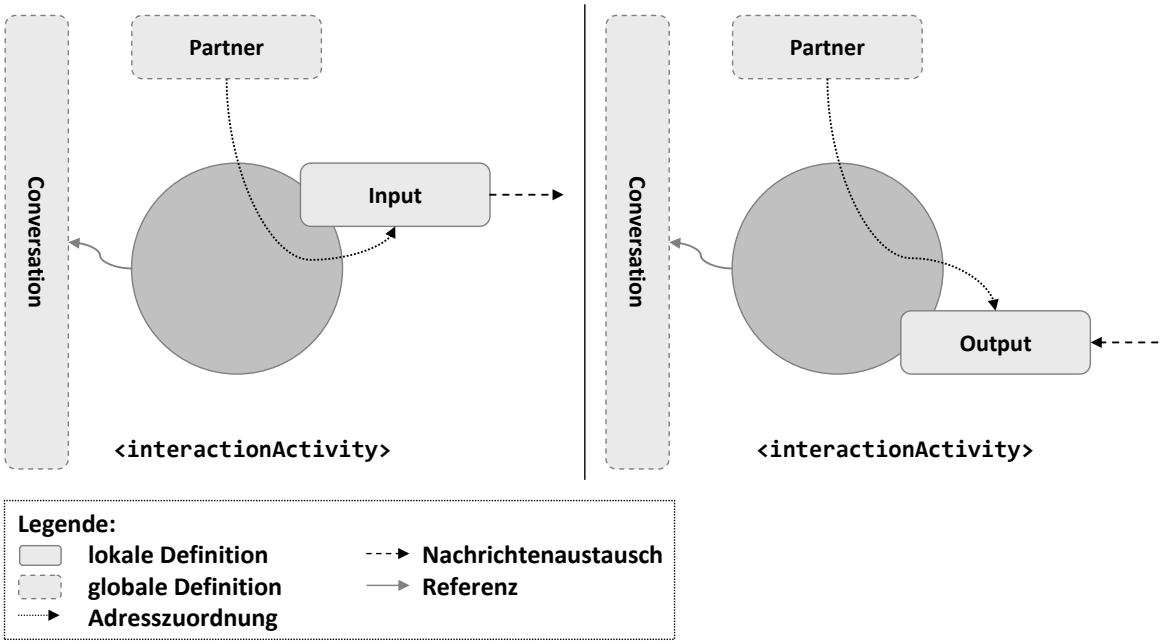


Abbildung 24: Einzelner Aktivitätstyp zum Senden oder Versenden von Nachrichten

Die Syntax der `<bl:interactionActivity>` wird in Listing 6 beschrieben. Über das `conversation`-Attribut werden die über die Aktivität

auszutauschenden Nachrichten zu einer Konversation zugeordnet. Ist lediglich ein `<bl:input>` definiert, so liest die Aktivität sobald sie aktiv ist den Inhalt der im `<bl:input>` angegebenen Variablen bzw. über den angegebenen `<bpel:from>`-Ausdruck aus und verschickt die Daten in einer Nachricht an den durch das `<bl:partner>`-Element spezifizierten Partnertyp. Hierzu muss genau eine der beiden Angaben – Name der Variablen oder `<bpel:from>`-Ausdruck – gemacht werden. Für den Fall, dass nur ein `<bl:output>` angegeben ist, wartet die Aktivität nach Aktivierung so lange bis eine entsprechende Nachricht des durch das `<bl:partner>`-Element spezifizierten Partners eintrifft und speichert die Daten in der im `<bl:output>` angegebenen Variablen bzw. an die über den `<bpel:to>`-Ausdruck spezifizierten Adresse ab. Hierzu muss genau eine der beiden Angaben – Name der Variablen oder `<bpel:to>`-Ausdruck – gemacht werden. Beginnt die Aktivität mit einer eingehenden Nachricht, so kann durch das Setzen des `createInstance`-Attributes auf „`yes`“ angezeigt werden, dass durch die eingehende Nachricht eine neue Prozessinstanz erzeugt werden kann.

```
<bpel:extensionActivity>
  <bl:interactionActivity conversation="NCName"?
                          createInstance="yes|no"?
                          standard-attributes>
    standard-elements
    <bl:input variable="NCName"?...>*
      <bpel:from/>?
    </bl:input>
    <bl:output variable="NCName"?...>*
      <bpel:to/>?
    </bl:output>
    <bl:partner name="NCName"?.../>+
  </bl:interactionActivity>
</bpel:extensionActivity>
```

Listing 6: Schema der `<bl:interactionActivity>`

Eingabe- und Ausgabeparameter sind nicht als Attribute sondern als Elemente der `<bl:interactionActivity>` definiert. Es können auch mehrere Eingabe- und Ausgabeparameter definiert werden. Für den Fall, dass nicht nur ein einzelner Eingabeparameter oder ein einzelner Ausgabeparameter definiert ist, implementiert die

`<bl:interactionActivity>` ein komplexes Nachrichtenaustauschmuster. Dies wird in Abschnitt 3.3 näher betrachtet. Die vollständige Syntax der `<bl:interactionActivity>` ist in Anhang H nachzulesen.

Um die operationale Semantik der `<bpel:pick>`-Aktivität und des `<bpel:eventHandler>`s zu erhalten, müssen den beiden Sprachkonstrukten entsprechende BPEL^{light} Varianten neu definiert werden, d.h. `<bl:pick>` und `<bl:eventHandler>`.

Zwar ist es möglich für `<bpel:pick>` und `<bpel:eventHandler>` entsprechende `<bl:onMessage>` bzw. `<bl:onEvent>`-Erweiterungselemente zu definieren, die keine WSDL-Elemente referenzieren, allerdings gibt es Einschränkungen für beide Sprachkonstrukte, die dazu führen, dass ihre Funktionalität nicht uneingeschränkt für BPEL^{light} verwendet werden kann.

Für `<bpel:pick>`-Aktivitäten ist es vorgeschrieben, dass pro Aktivität mindestens ein von WSDL abhängendes `<bpel:onMessage>`-Element spezifiziert ist. D.h. es ist nicht möglich, einen Prozess mit einem `<bpel:pick>` zu modellieren, der nur ein oder mehrere `<bl:onMessage>`-Elemente enthält. Für `<bpel:eventHandler>` gilt eine zwar nicht ganz so starke, aber dennoch ausreichende Einschränkung. Hier muss mindestens entweder ein `<bpel:onEvent>` oder ein `<bpel:onAlarm>` spezifiziert sein. Es ist also auch nicht möglich einen Prozess mit einem `<bpel:eventHandler>` mit lediglich einem oder mehreren spezifizierten `<bl:onEvent>`s zu modellieren.

```
<bpel:extensionActivity>
  <bl:pick createInstance="yes|no"?
           standard-attributes>
    standard-elements
    (
    <bl:onMessage conversation="NCName"
                  partner="NCName"
                  variable="NCName"?>
      <bpel:to/>?
      activity
    </bl:onMessage>
    |
    <bl:onAlarm/>
    )+
  </bl:pick>
</bpel:extensionActivity>
```

Listing 7: Schema der `<bl:pick>`-Aktivität

Die Syntax der `<bl:pick>`-Aktivität ist in Listing 7 dargestellt. Die `<bl:pick>`-Aktivität wartet wie die `<bpel:pick>`-Aktivität auf das Auftreten eines einzelnen Ereignisses aus einer Menge von Ereignissen. Beim Auftreten eines Ereignisses wird eine dem Ereignis zugeordnete Aktivität ausgeführt. Nach Eintreten eines Ereignisses werden keine weiteren Ereignisse von der `<bl:pick>`-Aktivität verarbeitet. Ereignisse können wie bei der `<bpel:pick>`-Aktivität entweder eintreffende Nachrichten sein oder ein zeitliches Ereignis. Für die Definition des zeitlichen Ereignisses wird ein neues `<bl:onAlarm>`-Element definiert, welches sich nur syntaktisch vom BPEL-Element `<bpel:onAlarm>` unterscheidet und dessen Semantik im Wesentlichen der des BPEL-Elements `<bpel:onAlarm>` entspricht. Auf die Definition des `<bl:onAlarm>`-Elements wird in Kapitel 3.2 genauer eingegangen. Das Ereignis einer eintreffenden Nachricht wird durch das neu definierte `<bl:onMessage>-Element` repräsentiert. Es referenziert keine WSDL-Elemente. Stattdessen wird die eintreffende Nachricht folgendermaßen in das Interaktionsmodell von BPEL[light] eingebettet: Das `<bl:onMessage>` referenziert eine Konversation durch das `conversation`-Attribut und einen Partner durch das `partner`-Attribut. Beide Attribute müssen ebenso spezifiziert werden wie das `variable`-Attribut oder der `<bpel:to>`-Ausdruck, welches angibt in welche Variable die eingehende Nachricht gespeichert werden soll.

Bei der `<bl:pick>`-Aktivität wird die Konversation nicht wie bei der `<bl:interactionActivity>` global an der Aktivität selber angegeben, sondern an der entsprechenden Nachricht. Das liegt in der operationalen Semantik der `<bl:pick>`-Aktivität begründet: Eine `<bl:interactionActivity>` in der bisher vorgestellten Form beinhaltet keine Prozesslogik. Sie empfängt oder sendet eine Nachricht. Diese atomaren Bausteine des Nachrichtenaustauschs lassen sich beliebig miteinander kombinieren durch die Definition von Prozesslogik zwischen `<bl:interactionActivity>`s. Nachrichten unterschiedlicher Konversationen können also miteinander in Beziehung gebracht werden. Implementiert eine `<bl:interactionActivity>` eine komplexe operationale Semantik, so wird diese Semantik durch eine Konversation repräsentiert (vergleiche Abschnitt 3.2 f). Es existiert also nicht die Notwendigkeit, Nachrichten innerhalb einer `<bl:interactionActivity>` unterschiedlichen Konversationen zuzuordnen. Die `<bl:pick>`-Aktivität hingegen beinhaltet Prozesslogik; sie implementiert die operationale Semantik *„Deferred Choice“* [AWBH00]. Wäre die `<bl:pick>`-Aktivität einer Konversation zugeordnet und nicht die einzelnen Nachrichten, so gäbe es keine Möglichkeit zu modellieren, dass der gewählte Pfad einer `<bl:pick>`-Aktivität von Nachrichten unterschiedlicher Konversationen abhängt.

Eine spezielle Form der `<bl:pick>`-Aktivität wird verwendet, wenn eine neue Instanz eines BPEL[light]-Prozesses durch das Eintreffen einer Nachricht erstellt werden soll. In diesem Fall ist das Attribut `createInstance` auf den Wert „`yes`“ zu setzen. Außerdem dürfen dann keine zeitlichen Ereignisse spezifiziert werden, welche die Aktivität auslösen. Wird das Attribut `createInstance` nicht gesetzt, so handelt es sich um eine nicht instantiierende `<bl:pick>`-Aktivität.

Jeder Geltungsbereich, auch der Prozess selbst, kann ein oder mehrere Ereignisse verarbeiten. Es kann maximal ein `<bl:onAlarm>`- und beliebig viele `<bl:onEvent>`-Elemente innerhalb eines `<bl:eventHandlers>` definiert werden. Listing 8 zeigt das um die `<bl:eventHandlers>` erweiterte `<bpel:process>`-Element. Die Definition der `<bl:eventHandlers>` in einer `<bpel:scope>`-Aktivität ist analog.

```
<bpel:process ...>
...
  <bl:eventHandlers>?
    <bl:onEvent conversation="NCName"
                partner="NCName"
                variable="NCName"?>*
      <bpel:to/>?
      <bpel:scope ...>...</scope>
    </bl:onEvent>
    <bl:onAlarm/>?
  </bl:eventHandlers>
...
</bpel:process>
```

Listing 8: Schema der mit einem `<bl:eventHandlers>`-Element erweiterten `<bpel:process>`-Definition

Ein `<bl:eventHandler>` wartet wie eine `<bl:pick>`-Aktivität auf das Eintreten spezifizierter Ereignisse. Diese Ereignisse können sowohl das Empfangen einer Nachricht als auch das Verstreichen einer bestimmten Zeitspanne oder Erreichen eines bestimmten Zeitpunkts sein. Allerdings reagiert der `<bl:eventHandler>` im Gegensatz zur `<bl:pick>`-Aktivität auch nach dem Eintreten des ersten Ereignisses auf weitere Ereignisse. Ein `<bl:eventHandler>` verarbeitet Ereignisse, solange der Geltungsbereich bzw. Prozess für den er definiert wurde aktiv ist. Des Weiteren unterscheiden sich die beiden Sprachkonstrukte darin, dass das Eintreffen einer Nachricht in einem `<bl:pick>` die Ausführung einer beliebigen Aktivität auslöst, wohingegen die auszuführende Aktivität in einem `<bl:eventHandler>` vom Typ `<bpel:scope>` sein muss. Um das Eintreffen einer Nachricht zu modellieren, definiert BPEL^light ein neues `<bl:onEvent>`-Element. Im Unterschied zum `<bpel:onEvent>`-Element referenziert das `<bl:onEvent>`-Element keine WSDL-Elemente, sondern fügt sich in das BPEL^light Interaktionsmodell ein, indem es durch die Attribute `conversation` und `partner` die entsprechenden Konzepte referenziert. Das Attribut `variable` spezifiziert zudem, in welche Variable der Inhalt der eintreffenden Nachricht gespeichert wird. Alternativ kann hierzu auch ein `<bpel:to>`-Ausdruck verwendet werden. Zur Modellierung eines zeitlichen Ereignisses wird das in Kapitel 3.2 näher beschriebene Element `<bl:onAlarm>` verwendet. Die Zugehörigkeit eines `<bl:onEvent>`-Elements zu einer Konversation wird nicht global im `<bl:eventHandler>`

vorgenommen, da in einem Prozess bzw. Geltungsbereich mehrere Konversationen definiert sein können, die alle die Definition eines für den gesamten Prozess bzw. Geltungsbereich gültigen Ereignisses erfordern.

3.1.3 Fault Handling in BPEL^{light}

Im vorangegangenen Abschnitt wurde gezeigt, wie das reguläre Verhalten eines Prozesses in Bezug auf den Nachrichtenaustausch durch BPEL^{light} modelliert werden kann. Dabei wurde von etwaigen Fehlern, die auftreten können und dann auch zwischen Partnern kommuniziert werden müssen, abgesehen.

In BPEL kann es lediglich bei blockierenden Aufrufen zum Austausch einer Fehlernachricht kommen. Dies wird zum einen in einem `<bpel:invoke>` deutlich, welches einen impliziten Geltungsbereich besitzt, der über `<bpel:catch>`- bzw. `<bpel:catchAll>`-Konstrukte einen empfangenen Fehler direkt behandeln kann und zum anderen in einer `<bpel:reply>`-Aktivität, welche im Fehlerfall mit der für die entsprechende Operation definierten Fehlernachricht antwortet.

BPEL^{light} beschreibt lediglich den Nachrichtenaustausch ohne eine Aussage darüber zu machen, wie der Nachrichtenaustausch auf die Kommunikationsinfrastruktur abgebildet wird, d.h. es unterscheidet nicht zwischen synchronen und asynchronen Aufrufen. Trotzdem muss es möglich sein, Fehlernachrichten zu empfangen oder zu senden. Betrachtet man die beiden Szenarien in denen bei BPEL Fehlernachrichten vorkommen können, so ergibt sich für das BPEL^{light} Kommunikationsmodel folgendes.

Zum einen muss die `<bl:interactionActivity>` eine Fehlernachricht versenden können, um die volle Funktionalität der `<bpel:reply>`-Aktivität abzubilden und zum anderen muss das *„Deferred Choice“* Verhalten des `<bpel:invoke>`s modelliert werden können, d.h. dass als Antwort auf eine Nachricht entweder eine reguläre Antwort empfangen werden kann oder alternativ eine Fehlernachricht. Dieses Verhalten lässt sich durch eine sendende `<bl:interactionActivity>` und ein anschließendes `<bl:pick>` realisieren, bei dem ein `<bl:onMessage>` die reguläre Nachricht

repräsentiert und ein `<bl:onFaultMessage>` die Fehlernachricht. Die Umsetzung der Semantik der `<bpel:invoke>`-Aktivität durch zwei aufeinander folgende Aktivitäten bedeutet auch, dass die Kommunikation mit dem Geschäftspartner aus Sicht des Prozesses in BPEL[light] nicht blockierend ist. Die sich daraus ergebene Syntax der beiden Aktivitäten ist in Listing 9 dargestellt.

```
<bpel:extensionActivity>
  <bl:interactionActivity ...>
    standard-elements
    <bl:input .../>*
    <bl:output .../>*
    <bl:inFault faultName="QName"
                variable="NCName"?...>*
      <bpel:from/>?
    </bl:inFault>
    <bl:partner .../>*
  </bl:interactionActivity>
</bpel:extensionActivity>
```

```
<bpel:extensionActivity>
  <bl:pick ...>
    standard-elements
    (
    <bl:onMessage ...>...
      activity
    </bl:onMessage>
    |
    <bl:onAlarm/>
    |
    <bl:onFaultMessage
      partner="NCName"
      faultName="QName-list"/>
    )+
  </bl:pick>
</bpel:extensionActivity>
```

Listing 9: Fehlernachrichten senden bzw. empfangen: Schema der Aktivitäten `<bl:interactionActivity>` und `<bl:pick>`

Neben den Elementen `<bl:input>` und `<bl:output>` können für eine `<bl:interactionActivity>` auch Fehler in Form des `<bl:inFault>`-Elements spezifiziert werden. Ist lediglich ein `<bl:inFault>` angegeben, so liest die `<bl:interactionActivity>` sobald sie aktiviert wird die Fehlernachricht aus der angegebenen Variablen bzw. über den angegebenen `<bpel:from>`-Ausdruck aus und sendet sie als Teil der entsprechenden Konversation dem spezifizierten Partner.

Um in einer `<bl:pick>`-Aktivität auch Fehlernachrichten empfangen zu können, erlaubt es BPEL[light] zusätzlich zu `<bl:onMessage>` und `<bl:onAlarm>`-Elementen auch `<bl:onFaultMessage>`-Elemente zu spezifizieren. Jedes `<bl:onFaultMessage>`-Element spezifiziert dabei über das `partner`-Attribut von welchem Geschäftspartner eine Fehlernachricht empfangen werden kann. Die Art der Fehlernachricht wird über das

Attribut `faultName` in Form einer Qname-Liste angegeben. Dadurch ermöglicht es BPEL^{light} zu unterscheiden von welchem Partner welche Fehlerarten empfangen werden können.

Ein `<bl:onFaultMessage>` spezifiziert anders als ein `<bl:onMessage>` oder `<bl:onAlarm>` keine auszuführende Aktivität. Stattdessen wird implizit eine `<bpel:throw>`-Aktivität aufgerufen, welches die empfangene Fehlernachricht an den zuständigen `<bpel:faultHandler>` delegiert. Der `<bpel:faultHandler>` erlaubt es dann, die unterschiedlichen Fehlerarten unterschiedlich zu bearbeiten oder alle möglichen Fehler durch ein `<bpel:catchAll>` abzufangen. Für jede `<pick>`-Aktivität darf maximal ein zeitliches Ereignis jedoch mehrere eintreffende reguläre Nachrichten und Fehlernachrichten spezifiziert werden. Fehlernachrichten können auch dann spezifiziert werden, wenn keine regulären Nachrichten erwartet werden. Dadurch lässt sich beispielsweise das zum Nachrichtenaustauschmuster „Robust In-Only“ [CHL+07] duale Nachrichtenaustauschmuster „Robust Out-Only“ mit BPEL^{light} umsetzen.

3.1.4 Korrelieren von Nachrichten und Prozessinstanzen

In BPEL werden Correlation Sets dazu verwendet, eintreffende Nachrichten mit Prozessinstanzen zu korrelieren. Das ist immer dann notwendig, wenn die eintreffende Nachricht nicht die Antwort auf eine synchrone `<bpel:invoke>`-Aktivität ist und die Kommunikationsinfrastruktur keine Mechanismen zur Korrelation bietet. BPEL^{light} macht keinerlei Aussage über die Kommunikationsinfrastruktur, die verwendet wird, um mit den Geschäftspartnern zu kommunizieren. Es kann also nicht davon ausgegangen werden, dass die verwendete Kommunikationsinfrastruktur entsprechende Möglichkeiten zur Korrelation von Nachrichten und Prozessinstanzen bietet. Die in BPEL eingeführten Correlation Sets sind somit für den Einsatz in BPEL^{light} von besonderer Bedeutung.

Wie in herkömmlichem BPEL werden `<bpel:correlationSets>` für einen Prozess oder Geltungsbereich definiert. Die in BPEL^{light} neu eingeführten Elemente, die Nachrichten senden oder empfangen, verwenden sie mittels des `<bpel:correlations>`-Elements. In Listing 10 ist das `<bl:input>`-Element der `<bl:interactionActivity>` dargestellt. Korrelation kann

nicht nur für das `<bl:input>`-Element der `<bl:interactionActivity>` definiert werden, sondern auch für das `<bl:onMessage>`-Element der `<bl:pick>`-Aktivität, das `<bl:onEvent>`-Element des `<bl:eventHandler>`s und die `<bl:output>`- und `<bl:inFault>`-Elemente der `<bl:interactionActivity>`. Die Funktionsweise der `<bpel:correlations>` in BPEL^light bleibt unverändert wie in [AAA+07] für BPEL beschrieben.

```
<bpel:extensionActivity>
  <bl:interactionActivity ...>...
    <bl:input ...>*
      <bpel:correlations>?
        <bpel:correlation set="NCName" initiate="yes|join|no"? />+
      </bpel:correlations>
    </bl:input ...>...
  </bl:interactionActivity>
</bpel:extensionActivity>
```

Listing 10: Korrelieren von Nachrichten und Prozessinstanzen: Schema des `<bl:input>`-Elements der `<bl:interactionActivity>`

Die `<bpel:invoke>`-Aktivität kann sowohl eine Nachricht senden als auch empfangen. Die `<bpel:correlations>` werden für die ganze Aktivität und nicht für jede Nachricht selber definiert. Für den Fall, dass eine `<bpel:invoke>`-Aktivität auch eine Nachricht empfängt, muss über das `pattern`-Attribut in der `<bpel:correlation>` angegeben werden, für welche Nachrichten die Korrelation definiert wird. Die `<bl:interactionActivity>` ist, wie bereits erwähnt, wie die `<bpel:invoke>`-Aktivität nicht auf das Senden oder Empfangen einer Nachricht beschränkt, sondern kann auch ein komplexes Verhalten implementieren. Das Verhalten ist jedoch nicht wie bei der `<bpel:invoke>`-Aktivität auf „Solicit-Response“ bzw. „Out-In“ beschränkt, sondern kann beliebig definiert werden. Es ist somit nicht möglich, über vordefinierte Werte des Attributs `pattern` die Nachrichten auszuwählen, für welche die Korrelation definiert ist. Deshalb wird in der `<bl:interactionActivity>` wie auch bei den anderen Interaktionsaktivitäten die Korrelation pro Nachricht definiert.

3.1.5 Zwischenbilanz

Das bis zu diesem Punkt vorgestellte einfache Modell deckt bereits die gesamte Funktionalität von BPEL im Bezug auf die Kommunikation mit Partnern ab. Lediglich der blockierende Aufruf eines Dienstes wird durch die bisher vorgestellten Erweiterungen nicht direkt unterstützt, kann jedoch durch eine sendende und eine darauf folgende empfangende Aktivität abgebildet werden. Die direkte Unterstützung für einen blockierenden Aufruf sowie für weitere beliebige operationale Semantiken wird in Kapitel 3.3 vorgestellt. Das bisher vorgestellte Modell ermöglicht es, unterschiedliche Partnertypen, die unterschiedliche Rollen in einem Geschäftsprozess spielen, zu unterscheiden. Szenarien, in denen lediglich Nachrichten mit einem Partner ausgetauscht werden oder in denen mehrere unterschiedliche Partnertypen – jedoch nicht mehrere Instanzen eines Partnertyps – in den Nachrichtenaustausch involviert sind, können somit modelliert werden. Auch Fehlerbehandlung kann vorgenommen werden. Der Prozess in Listing 11 veranschaulicht die Expressivität des BPEL^{light} Interaktionsmodells anhand einer konkreten Ausprägung des Szenarios „Request with Referral“, in der eine Zahlungsaufforderung an einen Kunden versendet wird und später der Zahlungseingang von der Bank bestätigt wird. Innerhalb des Prozesses sind unter anderem die beiden Partner „customer“ und „bankingHouse“ definiert sowie die Konversation „payInvoice“. Die Aktivität „sendInvoice“ sendet eine zur Konversation „payInvoice“ zugehörige Nachricht mit dem Inhalt der variablen „invoice“ an den Partner „customer“. Die anschließende, zur gleichen Konversation zugehörige `<bl:pick>`-Aktivität, empfängt entweder die Antwortnachricht vom Partner „bankingHouse“ und speichert sie in der Variablen „confirmation“ oder empfängt eine Fehlernachricht. Eine Fehlernachricht kann sowohl vom Partner „customer“ als auch vom Partner „bankingHouse“ empfangen werden. Der Kunde ist möglicherweise nicht der richtige Adressat der Rechnung und verweigert deshalb die Zahlung und die Bank meldet eventuell, dass die Zahlung nicht möglich war.

```
<bpel:process
  xmlns:bpel="http://.../wsbpel/2.0/process/executable/"
  xmlns:bl="http://.../BPELlight/2008/"
  suppressJoinFailure="yes"
  targetNamespace="http://.../samples"
  name="payment">...
  <bpel:variables>
    <bpel:variable name="invoice" .../>
    <bpel:variable name="confirmation" .../>
  </bpel:variables>...
  <bl:conversations>
    <bl:conversation name="payInvoice"/>
  </bl:conversations>
  <bl:partners>
    <bl:partner name="customer"/>
    <bl:partner name="bankingHouse"/>...
  </bl:partners>...
  <bpel:flow>
    <bpel:links>
      <bpel:link name="sendToReceive"/>...
    </bpel:links>
    <bpel:extensionActivity>
      <bl:interactionActivity bpel:name="sendInvoice"
                              conversation="payInvoice"...>
        <bl:input variable="invoice" />
        <bl:partner name="customer"/>
        <bpel:sources>
          <bpel:source linkName="sendToReceive"/>...
        </bpel:sources>...
      </bl:interactionActivity>
    </bpel:extensionActivity>
    <bpel:extensionActivity>
      <bl:pick bpel:name="receiveResponse"...>
        <bpel:targets>
          <bpel:target linkName="sendToReceive"/>...
        </bpel:targets>...
        <bl:onMessage
          conversation="payInvoice"
          variable="confirmation"
          partner="bankingHouse"...>
          ...
        </bl:onMessage>
        <bl:onFaultMessage
          conversation="payInvoice"
          faultName="ns:transferFailed"
          partner="bankingHouse"/>
        <bl:onFaultMessage
          conversation="payInvoice"
          faultName="ns:wrongRecipient"
          partner="customer "/>
      </bl:pick>
    </bpel:extensionActivity>...
  </bpel:flow>
</bpel:process>
```

Listing 11: Szenario „Request with Referral" in BPEL[light]

Das Codefragment abstrahiert von der den Prozess instantiierenden Aktivität sowie von Variablendeklarationen und der Definition der entsprechenden Faulthandler.

An diesem Beispiel lässt sich gut erkennen, dass es möglich ist, die Interaktion eines Prozesses mit unterschiedlichen Partnertypen zu modellieren und zusammengehörende Nachrichten über eine Konversation zu gruppieren. Was allerdings bisher nicht ausreichend unterstützt wird, ist die Interaktion eines Prozesses mit mehreren Instanzen eines einzigen Partnertypen. BPEL bietet für diesen Anwendungsfall keine native Unterstützung. Stattdessen können Variablen verwendet werden, um die EPRs von unterschiedlichen Instanzen innerhalb des Prozesses zu unterscheiden. Wie Interaktionen mit mehreren Instanzen eines Partnertyps in BPEL^{light} realisiert sind, wird im folgenden Abschnitt erläutert.

3.1.6 Konversationen mit mehreren Instanzen des gleichen Partnertyps

Szenarien, in denen mehrere Instanzen eines Partnertyps mit einem Geschäftsprozess interagieren, werden durch das bisher vorgestellte Modell noch nicht unterstützt, da kein Instanzbegriff für Partner existiert. Um dem Abhilfe zu verschaffen, definiert BPEL^{light} einen Container (`partnerInstances`) für das `<bl:partner>`-Element in dem Partnerinstanzinformationen gespeichert werden können. Im Gegensatz zum BPEL-Partnerlink kann jedes `<bl:partner>`-Element nicht nur eine sondern mehrere Referenzen auf Instanzen des Partnertyps halten. Innerhalb des `partnerInstances` Containers sind die Referenzen auf die unterschiedlichen Instanzen in einer Liste angeordnet. Zusätzlich bietet er die Möglichkeit eine der Referenzen zu markieren, um für einen Nachrichtenaustausch zu bestimmen mit welcher konkreten Partnerinstanz die Nachricht ausgetauscht wird. Die Syntax des `partnerInstances` Containers ist in Listing 12 dargestellt.

```
<bl:partnerInstances partnerName="NCName">
  <bl:current>
    <bl:partnerInstance id="xsd:int"/>
  </bl:current>
  <bl:list>
    <bl:partnerInstance id="xsd:int" name="NCName"?/>+
  </bl:list>
</bl:partnerInstances>
```

Listing 12: Schema des `partnerInstances`-Containers

Der `partnerInstances` Container wird über das Attribut `partnerName` dem entsprechenden Partner zugeordnet. Er hat zwei Elemente, das `<current>`-`Element` und das `<list>`-Element. Das `<list>`-Element enthält ein oder mehrere `<partnerInstance>`-Elemente. Sie sind eindeutig durch das Attribut `id` gekennzeichnet. Zusätzlich besteht die Möglichkeit, jede Partnerinstanz über das Attribut `name` mit einem Namen zu versehen. Das `<current>`-Element enthält genau ein `<partnerInstance>`-Element, welches über das Attribut `id` die aktuell ausgewählte Partnerinstanz anzeigt.

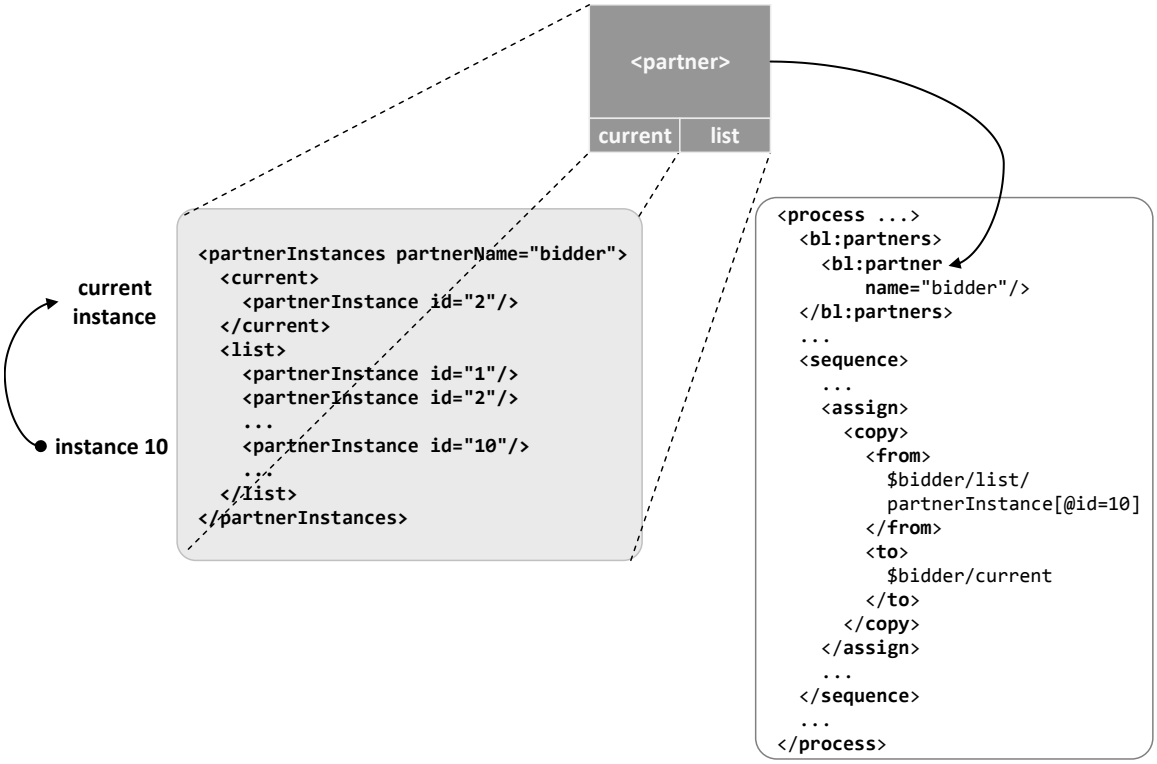


Abbildung 25: Auswahl der zu verwendenden Partnerinstanz

Auf einen partnerInstances Container kann per Xpath-Ausdruck ebenso zugegriffen werden wie auf Variablen. Das <list>-Element verhält sich dabei wie eine Variable, die ein Array enthält. So lässt sich beispielsweise die Liste der Instanzen im Laufe eines Prozesses erweitern oder eine Instanz aus der Liste als aktuelle Instanz markieren. Abbildung 25 zeigt ein Beispielszenario, in dem für einen Partner mehrere abstrakte partnerInstances definiert sind und in dem eine <bpel:assign>-Aktivität in einem Prozess verwendet wird, um die Partnerinstanz mit dem Wert „id="10"" aus der Liste von Partnerinstanzen für den Nachrichtenaustausch auszuwählen.

Wie in BPEL kann auch in BPEL^{light} mit konkreten Partnern gearbeitet werden. Allerdings werden die konkreten Partner nicht in Form von Endpunktreferenzen angegeben sondern stattdessen abstrakte Bezeichner für Instanzen verwendet, um dadurch unabhängig von der IT-Infrastruktur zu bleiben.

Listing 13 illustriert die Verwendung des partnerInstances Containers anhand des Szenarios „Request for Bid". Der Prozess definiert u.a. eine Konversation „requestForBid" und einen Partner „bidder". Für diesen wurden mehrere Instanzen definiert. In einer <bpel:forEach>-Aktivität wird jeder Instanz des Partners „bidder" zunächst eine Nachricht gesendet und anschließend optional eine Nachricht empfangen und an der Instanz entsprechenden Stelle im Array „result" gespeichert. Um eine parallele Ausführung der <bpel:forEach>-Aktivität zu ermöglichen, wird das Attribut parallel auf „yes" gesetzt. Weiterhin wird ein lokaler Partner „rfbBidder" innerhalb eines in das <bpel:forEach> eingebetteten Geltungsbereiches definiert. Für den Geltungsbereich ist das Attribut isolated auf „yes" gesetzt um sicherzustellen, dass sich die Geltungsbereiche der unterschiedlichen Iterationen nicht beeinflussen. In jeder Iteration wird zunächst der globale Partner „bidder" auf die lokale Definition „rfbBidder" kopiert. Anschließend werden die Instanzinformationen für eine Iteration des Schleifendurchlaufs auf das current Feld des „rfbBidders" geschrieben.

```
<bpel:process
  xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable/"
  xmlns:bl="http://iaas.uni-stuttgart.de/BPELlight/2008/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns="http://example.org"
  name="requestForBidSampleProcess"
  suppressJoinFailure="yes"
  targetNamespace="http://.../samples">...
  <bpel:variables>
    <bpel:variable name="counter" type="xsd:long"/>
    <bpel:variable name="result" type="ns:Array"/>...
  </bpel:variables>
  <bl:partners>
    <bl:partner name="bidder"/>...
  </bl:partners>
  <bl:conversations>
    <bl:conversation name="requestForBid"/>...
  </bl:conversations>...
  <bpel:forEach counterName="counter" parallel="yes">
    <bpel:startCounterValue>1</bpel:startCounterValue>
    <bpel:finalCounterValue>
      count($bidder/list/partnerInstance)
    </bpel:finalCounterValue>
    <bpel:scope isolated="yes">
      <bl:partners>
        <bl:partner name="rfbBidder"/>...
      </bl:partners>
      <bpel:sequence>
        <bpel:assign>...
          <bpel:copy>
            <bpel:from>$bidder/list/partnerInstance[$counter]</bpel:from>
            <bpel:to>$rfbBidder/current</bpel:to>
          </bpel:copy>
        </bpel:assign>
        <bpel:extensionActivity>
          <bl:interactionActivity conversation="requestForBid"...>
            <bl:input variable=... />
            <bl:partner name="rfbBidder">
          </bl:interactionActivity>
        </bpel:extensionActivity>
        <bpel:extensionActivity>
          <bl:pick>
            <bl:onMessage conversation="requestForBid"
                          partner="rfbBidder">
              <bpel:to>$result[$counter]</bpel:to>
              ...
            </bl:onMessage>...
            <bl:onAlarm ...>
          </bl:pick>
        </bpel:extensionActivity>
      </bpel:sequence>
    </bpel:scope>
  </bpel:forEach>
</bpel:process>
```

Listing 13: Szenario „Request for Bid" in BPEL[light]

3.2 BPEL^light zur Beschreibung von Nachrichtenaustauschmustern

Geschäftsziele in einem Prozess können sowohl den Austausch einer Nachricht mit einem Partner als auch einen komplexen Nachrichtenaustausch mit mehreren Partnern erfordern. Dabei lassen sich Muster beobachten, die sehr häufig vorkommen, wie beispielsweise die bereits gezeigten Muster „Request with Referral“ und „Request for Bid“. Die direkte Unterstützung dieser Muster durch einzelne Aktivitäten der Geschäftsprozessausführungssprache birgt großes Potential für die Erleichterung der Modellierung, da dadurch die Muster nicht bei jedem neuen Prozess vollständig neu modelliert werden müssen. Um die Muster direkt unterstützen zu können, müssen sie zunächst auf abstrakte Art und Weise beschrieben werden. Anschließend können sie entweder zur Beschreibung der Semantik einzelner Aktivitäten oder auch zur Beschreibung der Semantik eines Prozessfragments verwendet werden.

Im Web-Service-Umfeld werden Nachrichtenaustauschmuster verwendet, um WSDL 2.0-Operationen zu kategorisieren. Das vom W3C für die Definition von Nachrichtenaustauschmustern vorgesehene Template [CHL+07] ist in Listing 14 dargestellt.

```
This pattern consists of [number] message[s, in order] as follows:
[enumeration, specifying, for each message]
A[n optional] message:

1.      indicated by an Interface Message Reference component whose
        message label is "[label]" and direction is "[direction]"
2.      [received from|sent to] ["some" if first mention] node
        [node identifier]

This pattern uses the rule [fault ruleset reference].

An Interface Operation using this message exchange pattern has a
message exchange pattern property with the value "[pattern IRI]".
```

Listing 14: Template für WSDL 2.0-Nachrichtenaustauschmuster [CHL+07]

Innerhalb des Templates sind die eingeklammerten Begriffe Gegenstand von Ersetzungsoperationen. Die Beschreibungen „received from“ und „sent to“ sind jeweils aus der Sicht des Dienstes. Vom Dienst verschiedene an der Kommunikation beteiligte Dienste (Knoten, engl. Node) werden explizit als Sender oder Empfänger einer entsprechenden Nachricht angegeben.

Das Template ermöglicht komplexere Nachrichtenaustauschmuster als die für WSDL 1.1 zugelassenen Muster Request-Response und One-way zu beschreiben. Allerdings ist seine Ausdrucksstärke nicht ausreichend: Es gibt beispielsweise *keine* Möglichkeit *Parallelität* oder *bedingte Verzweigungen* auszudrücken. Außerdem kennt das Template *keinen Instanzbegriff*. Es ist also nicht möglich, Kommunikation mit mehreren Instanzen eines Knotens bzw. eines Knotentyps zu beschreiben. Ein weiterer Nachteil ist, dass Muster mit *optionalen Nachrichten unterspezifiziert* sind: Das Template bietet keine Möglichkeit zu definieren, wie sich ein Dienst tatsächlich verhält, d. h. wie er entscheidet, ob eine optionale Nachricht eintreffen wird oder nicht. Aus den genannten Gründen ist das Template nicht geeignet, um hinreichend komplexe Nachrichtenaustauschmuster eindeutig zu beschreiben; hierzu ist ein mächtigerer Formalismus notwendig.

BPEL bietet mächtige Konstrukte, um den Kontrollfluss zu beschreiben. Die Sprache ist jedoch ungeeignet für die Definition von Nachrichtenaustauschmustern, weil der Kontrollfluss lediglich *zwischen* WSDL-Operationen beschrieben werden kann. BPEL^{light} hingegen abstrahiert von WSDL und damit auch von Operationen und definiert den Kontrollfluss zwischen einzelnen Nachrichten. Damit eignet sich BPEL^{light} auch zur Definition von Nachrichtenaustauschmustern, die unter anderem zur Definition des Flusses *innerhalb* von WSDL-Operationen verwendet werden können.

Nachrichtenaustauschmuster in BPEL^{light} eignen sich allerdings nicht nur zur Definition von WSDL-Operationen, sondern vor allem auch zur Definition des Flusses innerhalb von BPEL^{light}-Aktivitäten, und das nicht zuletzt deswegen, weil in diesem Fall zur Definition des Prozesses selber und zur Definition einzelner Aktivitäten die gleiche Sprache verwendet

wird und somit ein Prozessmodellierer lediglich mit einer Sprache bzw. einem Modellierungsparadigma vertraut sein muss.

BPEL^{light} ermöglicht zunächst allerdings nur die Definition des Flusses zwischen konkreten Nachrichten. Für die Beschreibung von wieder verwendbaren Mustern ist es jedoch notwendig, von konkreten Datentypen zu abstrahieren, also den Fluss zwischen abstrakten untypisierten Nachrichten zu beschreiben. Durch die Abstraktion von den Datentypen wird die Wiederverwendbarkeit dieser Muster extrem erhöht. Das Muster eines „Request for Bid"s kann durch die Abstraktion von Datentypen beispielsweise nicht nur dazu verwendet werden, das günstigste Angebot für ein Produkt einzuholen, sondern auch um eine Abstimmung zu realisieren. Nicht nur die Abstraktion von Datentypen erhöht die Wiederverwendbarkeit von Mustern, sondern auch eine abstrakte Definition des Kontrollflusses. Hat man beispielsweise ein Muster, in dem nach Eintreffen einer Nachricht entweder eine Bestätigung oder eine Stornierung verschickt wird, so ist es von Vorteil das „*Xor*" Verhalten [AWBH00] ausdrücken zu können, ohne dabei angeben zu müssen, von welchen konkreten Werten es abhängt, welcher Zweig genommen wird. Gleiches gilt für die Definition von Iterationen und zeitlichen Ausdrücken.

In BPEL lässt sich eine Abstraktion von Implementierungsdetails durch die Definition eines Profils für abstrakte Prozesse erreichen. Dabei müssen die Regeln der Common Base eingehalten werden, die lauten: Durch opake Token ersetzt werden dürfen Aktivitäten, Ausdrücke, Attribute und `<bpel:from>`-Spezifikationen. Attribute können zudem auch unter folgenden Voraussetzungen weggelassen werden (Omission Shortcuts): (i) Das Attribut wird für die Definition eines ausführbaren Prozessmodels verlangt und (ii) es hat keinen Standardwert [AAA+07].

Dadurch eröffnen sich mehrere Möglichkeiten, um in einem Prozess von Datentypen zu abstrahieren. Zum einen ist es möglich, bei der Definition von Variablen den Datentyp auszulassen bzw. durch ein opakes Token zu ersetzen und auf diese abstrakten Variablen in Nachrichtendefinitionen von Aktivitäten zu referenzieren. Daraus resultiert die Definition eines abstrakten Datenflusses, da zwar nicht angegeben wird von welchem Typ

die Daten sind, aber spezifiziert wird von welchen Aktivitäten die Daten gelesen bzw. geschrieben werden. Die andere Möglichkeit von Datentypen zu abstrahieren ist, bei der Definition der durch Interaktionsaktivitäten zu verschickenden bzw. erwarteten Nachrichten die Referenz auf die Variable, aus der die Daten ausgelesen bzw. in welche die Daten geschrieben werden sollen, wegzulassen bzw. durch ein opakes Token zu ersetzen. Dadurch werden Aktivitäten, die Teil des Kontrollflusses sind, unabhängig von Daten beschrieben und somit der Kontrollfluss vollständig vom Datenfluss getrennt, was in einer erhöhten Wiederverwendbarkeit resultiert.

Schwieriger gestaltet sich das bei der abstrakten Beschreibung des Kontrollflusses. Es ist zwar möglich, beispielsweise eine `transitionCondition` in einem `<bpel:link>` durch ein opakes Token zu ersetzen, allerdings ist es dann nicht mehr möglich, eine Beziehung zwischen `transitionConditions` unterschiedlicher `<bpel:link>`s herzustellen, wie es z.B. zur Definition eines „Xor“ Verhaltens notwendig ist. Dies ist möglich, wenn die Ausdrücke konkret definiert werden und sich auf die gleichen konkreten Variablen beziehen. Da die Verwendung von konkreten Variablen allerdings die Wiederverwendbarkeit des Nachrichtenaustauschmusters einschränkt, führt BPEL^{light} ein neues Konstrukt für Ausdrücke ein, das `<bl:expression>`-Element. Es wird wie eine Variable für einen kompletten Prozess oder lediglich für einen bestimmten Geltungsbereich deklariert und folgt bezüglich der Sichtbarkeit den gleichen Regeln wie Variablen. Es kann in mehreren `transitionConditions` eingesetzt werden und ermöglicht es dadurch, die `transitionConditions` miteinander in Beziehung zu setzen. Dies ist am Beispiel der Umsetzung des „Xors“ in Abbildung 26 dargestellt.

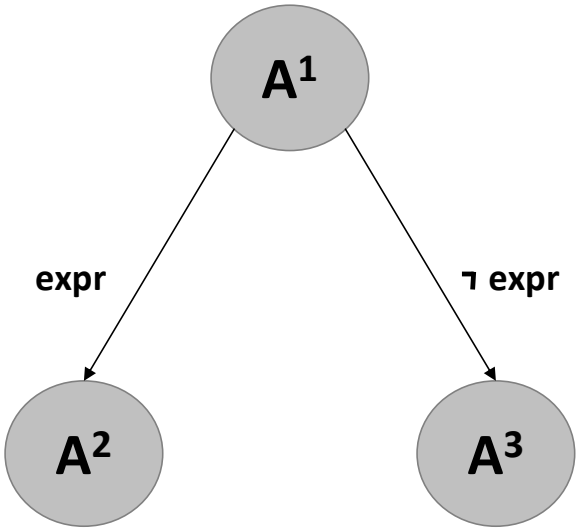


Abbildung 26: Beschreibung eines Xors mit BPEL[light]

Die Syntax des `<bl:expression>`-Element ist in Listing 15 dargestellt. Das `name`-Attribut ermöglicht die eindeutige Identifizierung innerhalb des Prozesses bzw. des Geltungsbereichs. Über das Attribut `expressionLanguage` lässt sich die Ausdruckssprache wählen, das Attribut `value` enthält den eigentlichen Ausdruck.

```
<bpel:process ...>...
  <bl:expression name="NCName"
                 expressionLanguage="anyURI"?
                 value="expr"/>
</bpel:process>
```

Listing 15: Schema der `<bl:expression>`

Auf eine `<bl:expression>` kann in Xpath 1.0 Ausdrücken über so genannte „Xpath Variable Bindings“ mit identischem Namen zugegriffen werden. D.h. für eine `<bl:expression>` mit dem Namen „productAvailable“ existiert ein Xpath Variable Binding namens „productAvailable“. Bei Ausführung eines Xpath 1.0-Ausdrucks, der eine `<bl:expression>` enthält, wird zunächst die `<bl:expression>` ausgewertet und anschließend das Ergebnis in den Ausdruck eingesetzt und verarbeitet. Die eingebettete `<bl:expression>` kann auch mit Hilfe einer anderen Ausdruckssprache verarbeitet werden. Abstrahiert man nun bei der Definition der `<bl:expression>` vom

konkreten Wert (Attribut `value`) des Ausdrucks und der Ausdruckssprache (Attribut `expressionLanguage`) so kann man den abstrakten Ausdruck zur Definition von Bedingungen in Kontrollfluss verwenden. Das Attribut `value` kann hierbei entweder weggelassen werden oder durch ein opakes Token ersetzt werden. Das Attribut `expressionLanguage` kann lediglich durch ein opakes Token ersetzt werden.

```
<bpel:process...>...
  <bl:expressions>
    <bl:expression name="productAvailable"
                   expressionLanguage=...
                   value=.../>
  </bl:expressions>...
  <bpel:flow>
    <bpel:links>
      <bpel:link name="confirmation"/>
      <bpel:link name="withdrawal"/>...
    </bpel:links>...

    <bpel:empty ...>
      <bpel:sources>
        <bpel:source linkName="confirmation">
          <bpel:transitionCondition>
            $productAvailable
          </bpel:transitionCondition>
        </bpel:source>
        <bpel:source linkName="withdrawal">
          <bpel:transitionCondition>
            not($productAvailable)
          </bpel:transitionCondition>
        </bpel:source>
      </bpel:sources>
    </bpel:empty>...
  </bpel:flow>
</bpel:process>
```

Listing 16: Verwendung der `<bl:expression>` zur Definition eines Xor-Verhaltens

Ein Beispiel für die Verwendung der `<bl:expression>` in einem konkreten Prozess ist in Listing 16 abgebildet. Es zeigt einen Ausschnitt eines Prozesses, in dem abhängig von der Verfügbarkeit eines Produktes unterschiedliche Pfade genommen werden sollen. Dazu sind in den `<transitionCondition>`-Elementen der entsprechenden `<bpel:link>`s zwei Xpath Ausdrücke, die sich jeweils auf die `<bl:expression>` „productAvailable“ beziehen, definiert. Der Ausdruck des `<bpel:link>`s „confirmation“ gibt den Wahrheitswert der `<bl:expression>`

„productAvailable“ zurück, der Ausdruck des `<bpel:link>`s „withdrawal“ hingegen gibt die Negation des Wahrheitswertes des Ausdrucks „productAvailable“ zurück.

Eine weitere Besonderheit bei Ausdrücken bilden zeitliche Ausdrücke, da sie in unterschiedlicher Form in einem BPEL-Prozess vorkommen können. Das Ergebnis von bedingten oder numerischen Ausdrücken wird eindeutig als erfüllte bzw. nicht erfüllte Bedingung oder Zähler interpretiert. Das Ergebnis eines zeitlichen Ausdrucks kann einer Zeitspanne, einem bestimmten Zeitpunkt bzw. einem Wiederholungsintervall entsprechen. Die unterschiedlichen Arten zeitlicher Ausdrücke werden in BPEL durch die Elemente `<bpel:for>`, `<bpel:until>` und `<bpel:repeatEvery>` ausgedrückt, die es jeweils ermöglichen, einen entsprechenden Ausdruck anzugeben und über das `expressionLanguage`-Attribut die Ausdruckssprache, in welcher der Ausdruck formuliert ist, zu spezifizieren. Soll nun von zeitlichen Ausdrücken beispielsweise in einem `<bpel:onAlarm>`-Element oder einer `<bpel:wait>`-Aktivität abstrahiert werden, so kann den Regeln der Common Base entsprechend zwar vom verwendeten konkreten Ausdruck abstrahiert werden, indem er entweder weggelassen oder durch ein opakes Token ersetzt wird, nicht jedoch von der Art des Ausdrucks, d.h. ob es sich um eine Zeitspanne, einen Zeitpunkt oder ein Intervall handelt.

Da sich die zwingende Angabe der Art des Ausdrucks negativ auf die Wiederverwendbarkeit eines Nachrichtenaustauschmusters auswirkt, definiert BPEL[light] ein neues `<bl:timingExpression>`-Element, welches es ermöglicht sowohl von zeitlichen Ausdrücken als auch von ihrer Art zu abstrahieren. Dieses Element kommt im ebenfalls neuen `<bl:onAlarm>`-Element und der neuen `<bl:wait>`-Aktivität zum Einsatz. Die Syntax des `<bl:onAlarm>`-Ausdrucks und der `<bl:wait>`-Aktivität, deren Definition jeweils vom `<bl:timingExpression>`-Element Verwendung machen, ist in Listing 17 dargestellt.

```
<bl:onAlarm>
  <bl:timingExpression expressionLanguage="anyURI"?
                       type="for|until|repeatEvery"
                       value="expr"/>
  activity
</bl:onAlarm>

<bpel:extensionActivity>
  <bl:wait standard-attributes>
    standard-elements
    <bl:timingExpression .../>
  </bl:wait>
</bpel:extensionActivity>
```

Listing 17: Schema des `<bl:onAlarm>`-Elements sowie der `<bl:wait>`-Aktivität

Der in einer `<bl:pick>`-Aktivität und einem `<bl:eventHandler>` zum Einsatz kommende Ausdruck `<bl:onAlarm>` enthält ebenso wie die `<bl:wait>`-Aktivität ein `<bl:timingExpression>`-Element. Über das Attribut `expressionLanguage` lässt sich die Ausdruckssprache für den zeitlichen Ausdruck wählen. Das Attribut `type` gibt an, um welche Art Ausdruck es sich handelt (Attributwerte „for“, „until“ oder „repeatEvery“), d.h. welchem Element in einem vergleichbaren `<bpel:onAlarm>` die `<bl:timingExpression>` entspricht. Das Attribut `value` enthält schließlich den eigentlichen Ausdruck. Da die `<bl:timingExpression>` nicht optional ist, kann sie den Regeln der Common Base folgend in einem abstrakten Prozess durch ein opakes token ersetzt werden bzw. komplett weggelassen werden. Da es nicht notwendig ist, Querverweise zwischen unterschiedlichen zeitlichen Ausdrücken herzustellen, werden sie nicht wie eine `<bl:expression>` global für einen Geltungsbereich oder Prozess definiert, sondern lediglich lokal im entsprechenden `<bl:onAlarm>`-Ausdruck bzw. in der `<bl:wait>`-Aktivität.

Profil für Nachrichtenaustauschmuster

Um Nachrichtenaustauschmuster abstrakt definieren zu können, wurde im Rahmen dieser Arbeit unter einem neuen Namensraum[8] ein abstraktes Profil für BPEL^light entwickelt. Da BPEL^light eine standardkonforme

[8] `xmlns:bl=http://iaas.uni-stuttgart.de/BPELlight/abstract/mep/2008/`

Erweiterung von BPEL ist, ist die Grundlage des Profils wie bei Profilen für BPEL die Common Base. Das abstrakte Profil für Nachrichtenaustauschmuster schränkt die Common Base in der folgenden Art und Weise ein:

- Es dürfen keinerlei BPEL-Elemente weggelassen werden.
- Explizite opaque Tokens, d.h. opaque Aktivitäten, opaque Attribute, opaque Ausdrücke und opaque `<bpel:from>`-Spezifikationen dürfen in Prozessen zur Beschreibung abstrakter Nachrichtenaustauschmuster bis auf folgende Ausnahmen nicht verwendet werden:
  - In den Elementen `<bl:input>` und `<bl:output>` innerhalb einer `<bl:interactionActivity>` sowie den Elementen `<bl:onMessage>` und `<bl:onEvent>` innerhalb eines `<bl:pick>`s bzw. `<bl:eventHandler>`s darf das Attribut `variable` durch ein opaques Attribut ersetzt werden.
  - In den Elementen `<bl:onFaultMessage>` innerhalb eines `<bl:pick>`s und `<bl:inFault>` innerhalb der `<bl:interactionActivity>` darf das Attribut `faultName` durch ein opaques Attribut ersetzt werden.
  - Im Element `<bl:expression>` dürfen die Attribute `expressionLanguage` und `value` durch opaque Attribute ersetzt werden.
  - Im Element `<bl:timingExpression>` dürfen die Attribute `expressionLanguage`, `type` und `value` durch opaque Attribute ersetzt werden.
  - Omission shortcuts dürfen bei allen genannten Attributen, die durch opaque Attribute ersetzt werden können, bis auf die folgenden Ausnahmen eingesetzt werden:

    In den Elementen `<bl:expression>` und `<bl:timingExpression>` darf das Attribut expressionLanguage nicht weggelassen werden.

Modellierung von Nachrichtenaustauschmustern

Jedes Nachrichtenaustauschmuster in BPEL^light wird durch eine eigene Prozessdefinition repräsentiert. Allerdings handelt es sich dabei nicht um einen eigenständigen Prozess wie es beispielsweise bei einem BPEL-SPE-Subprozess der Fall ist. Innerhalb des Prozesses zur Definition eines Nachrichtenaustauschmusters werden zum einen BPEL-Kontrollstrukturen dazu verwendet, einen Fluss durch BPEL^light Interaktionsaktivitäten (`<bl:interactionActivity>` und `<bl:pick>`) bzw. Event Handler (`<bl:onEvent>`) zu modellieren. Zum anderen wird eine `<bl:conversation>` dazu verwendet, die Aktivitäten zu einem Nachrichtenaustauschmuster zu gruppieren und es werden potentiell mehrere Partner definiert, welche die Empfänger bzw. Sender der Nachrichten repräsentieren.

Das Nachrichtenaustauschmuster „Request with Referral“ wird als Beispiel in Abbildung 27 dargestellt. Es besteht aus einer einzelnen Konversation, die durch die beiden sequentiellen `<bl:interactionActivity>`s A und B realisiert wird. Aktivität A definiert genau eine zu versendende Nachricht m1 und referenziert den Partner p1, sendet also eine Nachricht an p1. Aktivität B definiert genau eine zu empfangende Nachricht m2 und referenziert den Partner p2, empfängt also eine Nachricht von dem zu p1 verschiedenen Partner p2. Von Fehlernachrichten wird in der Darstellung abstrahiert.

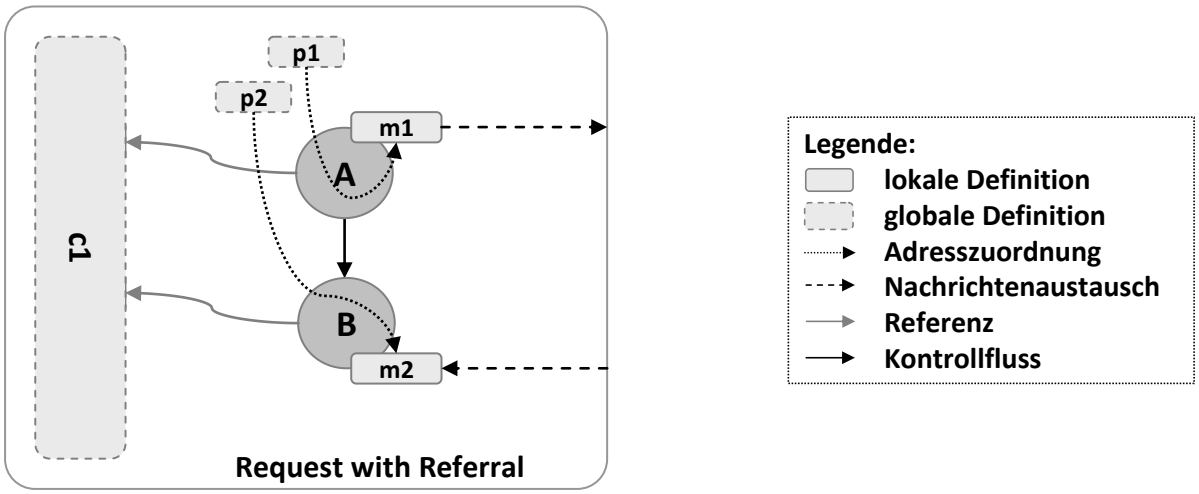


Abbildung 27: Nachrichtenaustauschmuster „Request with Referral“

```
<bpel:process
  xmlns:bpel="http://.../wsbpel/2.0/process/abstract"
  xmlns:bl="http://.../BPELlight/abstract/mep/2008"
  suppressJoinFailure="yes"
  targetNamespace="http://.../mepInBpel"
  name="requestWithReferral">...
  <bl:conversations>
    <bl:conversation name="requestWithReferralConv"/>
  </bl:conversations>
  <bl:partners>
    <bl:partner name="contactedProvider"/>
    <bl:partner name="respondingProvider"/>
  </bl:partners>
  <bpel:flow>
    <bpel:links>
      <bpel:link name="sendToReceive"/>
    </bpel:links>
    <bpel:extensionActivity>
      <bl:interactionActivity bpel:name="sendRequest"
                              conversation="requestWithReferralConv">
        <bl:input                                  />
        <bl:partner name="contactedProvider"/>
        <bpel:sources>
          <bpel:source linkName="sendToReceive "/>
        </bpel:sources>
      </bl:interactionActivity>
    </bpel:extensionActivity>
    <bpel:extensionActivity>
      <bl:pick bpel:name="receiveResponse">
        <bpel:targets>
          <bpel:target linkName="sendToReceive "/>
        </bpel:targets>
        <bl:onMessage
                      conversation="requestWithReferralConv"
                      partner="respondingProvider">
          <bpel:empty/>
        </bl:onMessage>
        <bl:onFaultMessage
                           conversation="requestWithReferralConv"
                           partner="respondingProvider">
        </bl:onFaultMessage>
        <bl:onFaultMessage
                           conversation="requestWithReferralConv"
                           partner="contactedProvider">
        </bl:onFaultMessage>...
      </bl:pick>
    </bpel:extensionActivity>...
  </bpel:flow>
</bpel:process>
```

Abstraktion von Datentypen

Listing 18: Nachrichtenaustauschmuster „Request with Referral" als abstrakter BPEL[light]-Prozess

In Listing 18 ist der das Nachrichtenaustauschmuster spezifizierende BPEL^{light} Code abgebildet. Partner p1 wird hierbei durch den Partner „contactedProvider“ und Partner p2 durch den Partner „respondingProvider“ dargestellt. Aktivität A entspricht der `<bl:interactionActivity>` „sendRequest“, Aktivität B entspricht der `<bl:pick>`-Aktivität „receiveResponse“. Die Nachricht m1 wird durch den `<bl:input>` der „sendRequest“ Aktivität widergespiegelt, m2 durch das `<bl:onMessage>` der „receiveResponse“ Aktivität. Vergleicht man den Code mit dem in Listing 11 dargestellten ausführbaren Prozess, so wird deutlich, dass bei der Definition des Nachrichtenaustauschmusters von Datentypen abstrahiert wurde.

Wie bereits erwähnt, können Nachrichtenaustauschmuster in einem Prozess unterschiedlichen Einsatz finden. Zum einen können sie zur Beschreibung der Semantik von `<bl:interactionActivity>`s dienen und zum anderen können sie verwendet werden, um nicht die Semantik einer einzelnen Aktivität sondern größerer Teile eines Prozesses zu beschreiben. Diese beiden Einsatzmöglichkeiten werden in den folgenden Abschnitten näher betrachtet.

3.3 Nachrichtenaustauschmuster zur Beschreibung des Verhaltens von BPEL^{light} Interaktionsaktivitäten

Ist wie in Abschnitt 3.1.2, Seite 103 ff, lediglich ein Eingabe- oder Ausgabeparameter und ein Partner in einer `<bl:interactionActivity>` definiert, so ist die Semantik der Aktivität eindeutig: Sie sendet dem angegebenen Partner eine Nachricht bzw. Fehlernachricht oder empfängt vom angegebenen Partner eine Nachricht. Diese Art der `<bl:interactionActivity>` wird auch als *einfache* Interaktionsaktivität bezeichnet. Werden jedoch mehrere Eingabe- und/oder Ausgabeparameter und mehrere Partner definiert, da es sich um eine komplexe Geschäftfunktion handelt, die realisiert werden soll, ist die Semantik der Aktivität zunächst unklar. In diesem Fall können die im vorangegangen Abschnitt beschriebenen Nachrichtenaustauschmuster für die Beschreibung der Semantik der Aktivität verwendet werden. Es handelt sich dann um eine *komplexe* Interaktionsaktivität. Abbildung 28 stellt

diesen Sachverhalt schematisch dar. Das Nachrichtenaustauschmuster (engl. *Message Exchange Pattern*, MEP) ermöglicht es, die unterschiedlichen Partner den Nachrichten zuzuordnen und die Reihenfolge festzulegen, in der die Nachrichten gesendet bzw. empfangen werden sollen. Der Übersichtlichkeit halber wird in der Abbildung von Fehlernachrichten sowie zeitlichen und zanderen Ausdrücken abstrahiert.

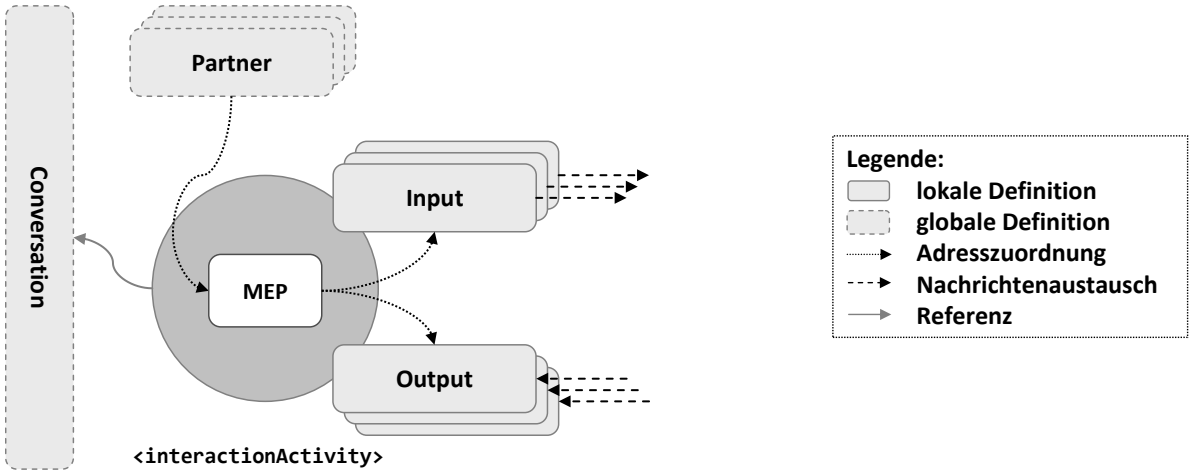


Abbildung 28: `<bl:interactionActivity>` deren Semantik durch ein Nachrichtenaustauschmuster bestimmt wird

Dafür müssen zum einen die Nachrichtenaustauschmuster selber eindeutig identifizierbar sein und zum anderen auch die parametrisierten Größen, die konkretisiert werden müssen, um ausführbar zu sein. Das ist in Abbildung 29 anhand des Nachrichtenaustauschmusters „Request with Referral“ dargestellt: Das Nachrichtenaustauschmuster selber ist wie jeder Prozess nach außen sichtbar. Gleiches gilt für die Partner, hier im Beispiel p1 und p2, sowie die Ein- und Ausgabeparameter m1 und m2.

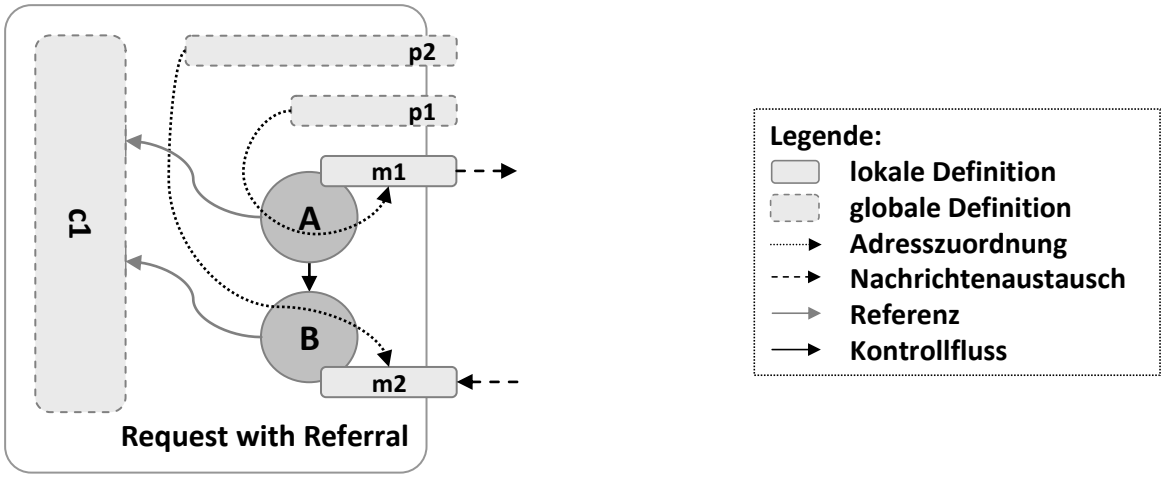


Abbildung 29: Parametrisierbares Nachrichtenaustauschmuster

Die Identifizierung der Nachrichtenaustauschmuster erfolgt anhand einer URI Referenz die sich aus den Werten der Prozessattribute `targetnamespace` und `name` zusammensetzt. Das Nachrichtenaustauschmuster „Request with Referral“ wird beispielsweise entsprechend der Syntax einer URI Referenz durch den Bezeichner „http://.../mepInBpel#requestWithReferral“ referenziert. Um denjenigen Größen, von denen bei der Definition des Nachrichtenaustauschmusters abstrahiert wurde, Werte von außen zuordnen zu können, ist es notwendig, eben diese Größen innerhalb des Nachrichtenaustauschmusters eindeutig zu identifizieren. Die Elemente `<bl:partner>` und `<bl:expression>` werden bereits innerhalb des Prozessmodells referenziert und haben deshalb ein `name`-Attribut, dessen Wert auch für die Referenzierung von außen herangezogen werden kann. Anders verhält es sich mit den Definitionen ausgehender sowie eingehender Nachrichten und zeitlicher Ausdrücke. Sie werden lokal vorgenommen und werden innerhalb eines Prozessmodells nicht referenziert. Diese Elemente haben also in ihrer bisher vorgestellten Form keine eindeutigen Identifikationsmerkmale. Um sie dennoch von außen referenzieren zu können, definiert BPEL[light] für die Elemente `<bl:input>`, `<bl:output>` und `<bl:inFault>` in `<bl:interactionActivity>`s, `<bl:onMessage>` und `<bl:onFaultMessage>` in `<bl:pick>`-Aktivitäten sowie `<bl:timingExpression>` in `<bl:onAlarm>`s oder `<bl:wait>`s das Attribut `messageLabel` bzw.

`timingExpressionLabel`. Das in Listing 18 dargestellte Nachrichtenaustauschmuster „Request with Referral“ wird dementsprechend wie folgt erweitert: Das Element `<bl:input>` der `<bl:interactionActivity>` „sendRequest“ erhält ein Attribut `messageLabel` mit dem Wert „request“ und das `<bl:onMessage>`-Element der Aktivität „receiveResponse“ erhält ein Attribut `messageLabel` mit dem Wert „response“. Die Fehler erhalten entsprechend ihrer Absender ebenfalls über das Attribut `messageLabel` die Bezeichnungen „contactedProviderFault“ bzw. „respondingProviderFault“.

Um die durch abstrakte BPEL^{light}-Prozesse beschriebenen Nachrichtenaustauschmuster zur Definition des Verhaltens von `<bl:interactionActivity>`s verwenden zu können, muss es möglich sein, aus der `<bl:interactionActivity>` sowohl das Nachrichtenaustauschmuster als auch die parametrisierten Größen zu referenzieren.

Das ist in Abbildung 30 anhand eines Beispiels dargestellt. Die abgebildete `<bl:interactionActivity>` definiert einen Eingabeparameter „mA“ und einen Ausgabeparameter „mB“. Außerdem referenziert sie zwei Partner „pA“ und „pB“ sowie eine Konversation „cA“. Das Nachrichtenaustauschmuster „Request with Referral“ wird verwendet, um die operationale Semantik der `<bl:interactionActivity>` zu spezifizieren. Die parametrisierten Größen des Nachrichtenaustauschmusters werden wie folgt belegt: Der Eingabeparameter „m1“ wird auf den Eingabeparameter „mA“ abgebildet und der Ausgabeparameter „m2“ auf den Ausgabeparameter „mB“. Außerdem wird der Partner „p1“ dem Partner „pA“ zugeordnet und der Partner „p2“ dem Partner „pB“. Durch diese Abbildung ist für die `<bl:interactionActivity>` spezifiziert, dass zunächst die Nachricht „mA“ an den Partner „pA“ gesendet wird und anschließend die Nachricht „mB“ vom Partner „pB“ empfangen wird.

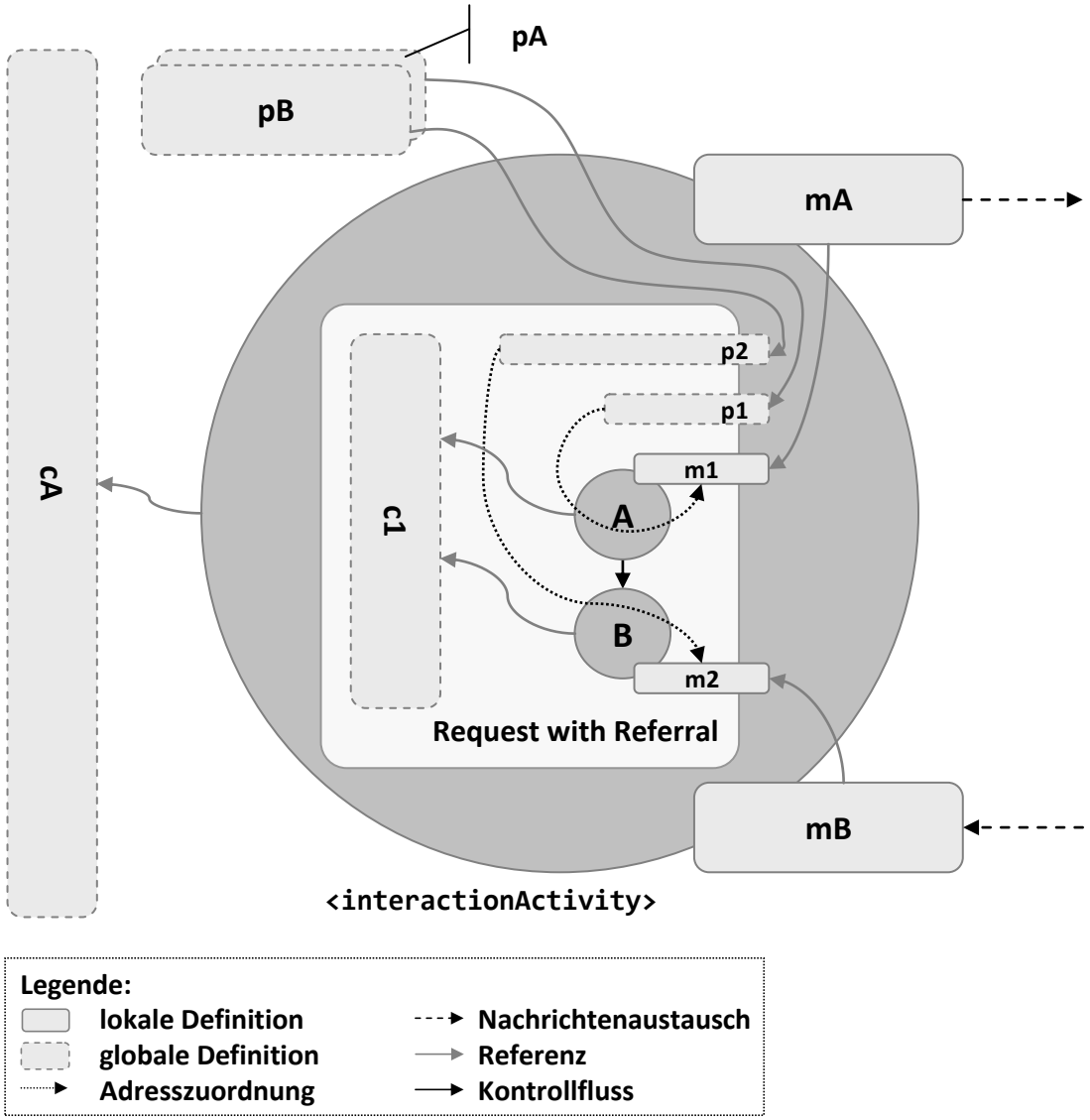


Abbildung 30: `<bl:interactionActivity>` deren Semantik durch das Nachrichtenaustauschmuster „Request with Referral“ bestimmt wird

Um sowohl das Nachrichtenaustauschmuster als auch die parametrisierten Punkte referenzieren zu können, bietet die `<bl:interactionActivity>` und deren Kindelemente weitere, bisher nicht vorgestellte Attribute (siehe Listing 19). Das Attribut `mep` der `<bl:interactionActivity>` ist optional und dient der Referenzierung des die Aktivität definierenden Nachrichtenaustauschmusters. Das Attribut `messageRef` der Elemente `<bl:input>`, `<bl:output>` und `<bl:inFault>` dient dazu, sie entsprechenden Nachrichten bzw. Variablen innerhalb des referenzierten Nachrichtenaustauschmusters zuzuordnen:

- Das Element `<bl:input>` referenziert ein `<bl:input>`-Element einer `<bl:interactionActivity>` des referenzierten Prozessmodells.
- Das Element `<bl:output>` referenziert entweder ein `<bl:output>`-Element einer `<bl:interactionActivity>`, ein `<bl:onMessage>` einer `<bl:pick>`-Aktivität oder ein `<bl:onEvent>` eines `<bl:eventHander>`s des referenzierten Prozessmodells.
- Das Element `<bl:inFault>` referenziert ein `<bl:inFault>`-Element einer `<bl:interactionActivity>` des referenzierten Prozessmodells.

Alternativ können sich die genannten Elemente auch auf Variablen des referenzierten Prozessmodells beziehen.

Nachrichtenaustauschmuster können nicht nur Fehler senden, sondern auch empfangen, d.h. die `<bl:interactionActivity>` muss einen eintreffenden Fehler verarbeiten können. Dies wird durch das bisher nicht vorgestellte Element `<bl:outFault>` realisiert. Es ist analog zum `<bl:inFault>`-Element definiert, ermöglicht also u.a. auch die Verwendung von Korrelation, und referenziert mittels des `messageRef`-Attributs entweder ein `<bl:outFault>`-Element einer `<bl:interactionActivity>` oder ein `<bl:onFaultMessage>` einer `<bl:pick>`-Aktivität des referenzierten Prozessmodells.

Analog zur Zuordnung der Nachrichten der `<bl:interactionActivity>` zu Nachrichten des Nachrichtenaustauschmusters, werden die Partner durch ein `partnerRef`-Attribut des `<bl:partner>`-Elements der `<bl:interactionActivity>` referenziert.

Um auch auf Ausdrücke innerhalb des Nachrichtenaustauschmusters von der `<bl:interactionActivity>` aus verweisen zu können, bietet sie zwei weitere Elemente, das `<bl:expression>` und das `<bl:timimgExpression>`-Element mit den Attributen `expressionRef` bzw. `timingExpressionRef`.

```
<bpel:extensionActivity>
  <bl:interactionActivity mep="anyURI"?...>...
    <bl:input messageRef="NCName"?.../>*
    <bl:output messageRef="NCName"?.../>*
    <bl:inFault messageRef="NCName"?.../>*
    <bl:outFault messageRef="NCName"?.../>*
    <bl:partner partnerRef="NCName"?.../>+
    <bl:expression expressionRef="NCName".../>*
    <bl:timingExpression timingExpressionRef="NCName".../>*
  </bl:interactionActivity>
</bpel:extensionActivity>
```

Listing 19: Schema der `<bl:interactionActivity>`: Elemente und Attribute zur Referenzierung parametrisierter Größen in Nachrichtenaustauschmustern

Referenziert eine `<bl:interactionActivity>` kein Nachrichtenaustauschmuster und definiert lediglich ein `<bl:input>` und ein `<bl:partner>`-Element aber keine weiteren Elemente, so ist dies semantisch äquivalent zu einer Aktivität, die das Nachrichtenaustauschmuster „Out-Only“ referenziert und bei der Eingabeparameter und Partner auf die entsprechenden Elemente im Nachrichtenaustauschmuster abgebildet werden. Analog ist eine `<bl:interactionActivity>`, die ausschließlich einen einzelnen `<bl:output>` und ein `<bl:partner>`-Elements spezifiziert, semantisch äqivalent zu einer das Nachrichtenaustauchmuster „In-Only“ referenzierenden Aktivität. Sind mehr als entweder ein `<bl:partner>` und ein `<bl:input>` bzw. ein `<bl:output>` spezifiziert – handelt es sich also um eine komplexe Interaktionsaktivität – so muss das Attribut `mep` spezifiziert sein.

```
<bl:partners>
  <bl:partner name="customer"/>
  <bl:partner name="bankingHouse"/>
</bl:partners>
...
  <bpel:extensionActivity>
    <bl:interactionActivity mep="http://.../mepInBpel#requestWithReferral"
      <bl:input messageRef="request"...>
      <bl:output messageRef="response"...>
      <bl:outFault messageRef="contactedProviderFault".../>
      <bl:outFault messageRef="respondingProviderFault".../>
      <bl:partner name="customer"
                  partnerRef="contactedProvider"/>
      <bl:partner name="bankingHouse"
                  partnerRef="respondingProvider"/>
    </bl:interactionActivity>
  </bpel:extensionActivity>
...
```

Listing 20: `<interactionActivity>` deren Semantik durch das Nachrichtenaustauschmuster „Request with Referral“ bestimmt wird

Listing 20 zeigt eine `<bl:interactionActivity>` deren Semantik durch das in Listing 18 dargestellte und auf Seite 134 erweiterte Nachrichtenaustauschmuster „Request with Referral“ definiert wird. Hierzu erhält das Attribut `mep` den Wert „http://…/mepInBpel#requestWithReferral“. Der `<bl:input>` referenziert mit Hilfe des Attributs `messageRef` den im Nachrichtenaustauschmuster definierten `<bl:input>` mit der Attributbelegung „`messageLabel=“request““`, der `<bl:output>` referenziert das `<bl:onMessage>` mit der Attributbelegung „`messageLabel=“response““`, Jeweils ein `<bl:outFault>` referenziert mittels `messageRef`-Attribut den Fehler „contactedProviderFault“ sowie „respondingProviderFault“. Der Partner „customer wird über das Attribut `partnerRef` abgebildet auf den Partner „contactedProvider“, der Partner „bankingHouse“ auf den Partner „respondingProvider“.

Setzt man während der Modellierung das Attribut `mep` in einer `<bl:interactionActivity>` so verweist man dadurch auf ein Nachrichtenaustauschmuster, das durch einen abstrakten BPEL[light]-Prozess beschrieben wird. Diese Prozessbeschreibung repräsentiert jedoch keinen eigenständig lauffähigen Prozess wie es beispielsweise bei einem BPEL-

SPE-Subprozess der Fall ist, sondern definiert lediglich das Verhalten, d.h. die operationale Semantik, der einbindenden Aktivität. Über die Implementierung dieser Semantik in einer BPEL^{light}-Engine wird nichts ausgesagt. Hierzu gibt es unterschiedliche Möglichkeiten (siehe Abschnitt 6.1.2.1.2).

Abstrakt definierte Nachrichtenaustauschmuster können zum einen in einem konkreten Prozessmodell verwendet werden. Zum anderen können sie verwendet werden, um Aktivitäten eines komplexeren abstrakten Nachrichtenaustauschmusters zu beschreiben, d.h. sie können als Grundlage für die Definition von Nachrichtenaustauschmustern höherer Ordnung dienen. Diese beiden Szenarien werden in den folgenden beiden Abschnitten näher erläutert.

3.3.1 Interaktionsaktivitäten mit komplexem Verhalten zur Definition höherwertiger Nachrichtenaustauschmuster

Die Verwendung von komplexen Interaktionsaktivitäten bei der Definition von Nachrichtenaustauschmustern trägt dazu bei, den Modellierungsaufwand zu minimieren.

Wird ein BPEL^{light} Nachrichtenaustauschmuster zur Definition eines höherwertigeren BPEL^{light} Nachrichtenaustauschmusters verwendet, so reicht es aus, die parametrisierten Größen in der Definition des neuen Nachrichtenaustauschmusters zur Verfügung zu stellen. Das wird dadurch erreicht, dass die parametrisierten Größen von entsprechenden Elementen per `messageRef-`, `partnerRef-`, `expressionRef-` bzw. `timingExpressionRef`-Attribut referenziert werden und diese Elemente wiederum identifizierbar gemacht werden durch entsprechende Attribute. Partner und Ausdrücke werden über das `name-Attribut`, und sämtliche Nachrichten über das `messageLabel-Attribut` identifiziert. Die `<bl:timingExpression>` ist über das Attribut `timingExpressionLabel` identifizierbar.

In Abbildung 31 ist die rekursive Definition des Nachrichtenaustauschmusters „Request for Bid“ durch weniger komplexe Nachrichtenaustauschmuster dargestellt.

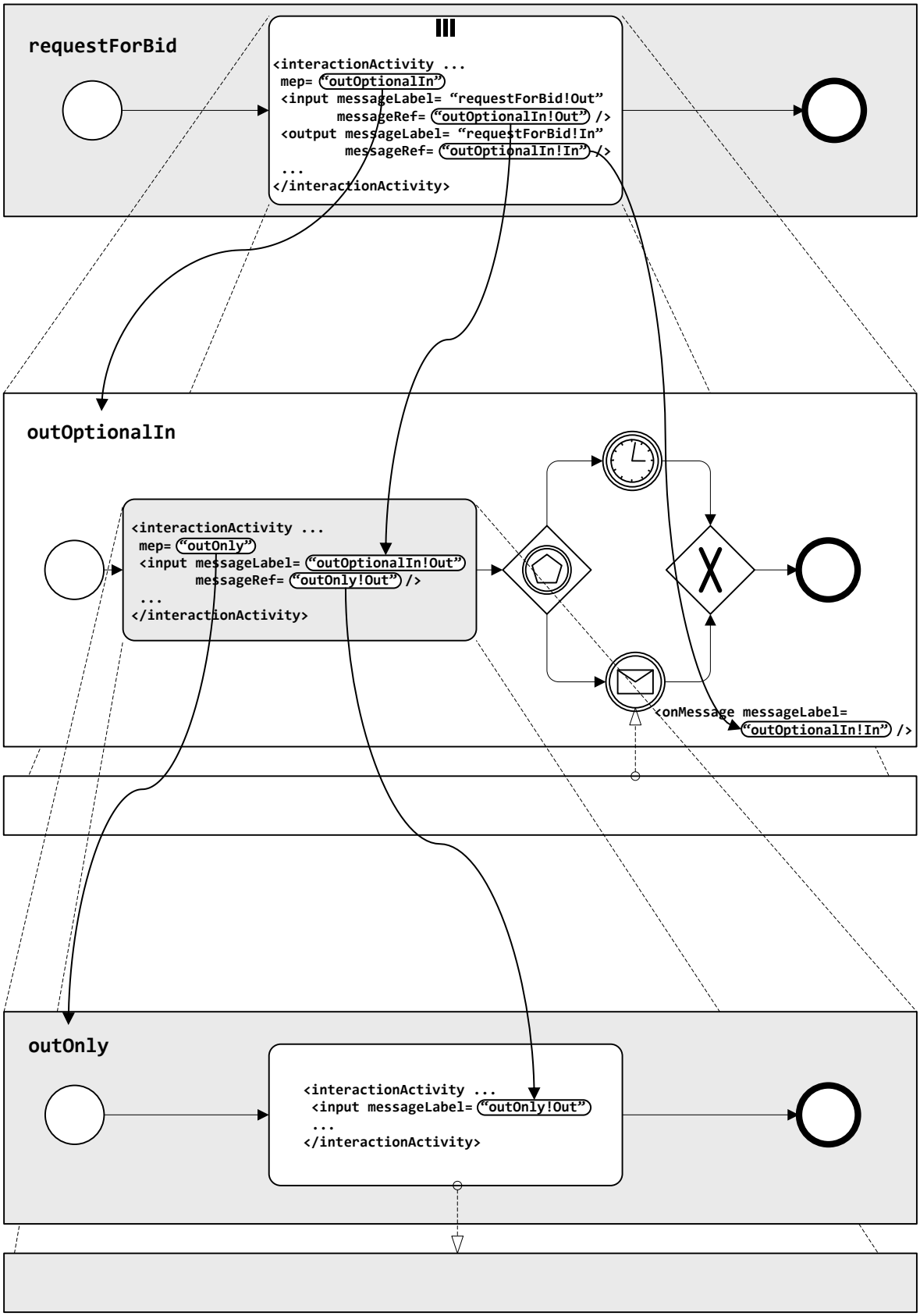


Abbildung 31: Rekursive Definition eines „Request for Bid"s

Zur Visualisierung des Kontrollflusses wird die Business Process Modeling Notation (BPMN) [OMG08] verwendet. Die Abbildung konzentriert sich darauf, die Zusammenhänge der einzelnen Nachrichtenaustauschmuster und der einzelnen Nachrichten aufzuzeigen. Von Fehlernachrichten wird in der Abbildung abstrahiert. An den dafür relevanten Stellen wird das BPMN Diagramm durch BPEL^{light} Code ergänzt.

Zuunterst ist das Nachrichtenaustauschmuster „Out-Only“ dargestellt. Es wird in BPMN repräsentiert durch eine Sequenz bestehend aus einem *Start-Event*, einem *Task* und einem *End-Event*. Der Task spiegelt eine `<bl:interactionActivity>` wider, die lediglich eine Nachricht mit dem `messageLabel` „outOnly!Out“ versendet.

Über dem Nachrichtenaustauschmuster „Out-Only“ ist das Nachrichtenaustauschmuster „Out-Optional-In“ abgebildet. Es wird repräsentiert durch ein Start-Event, einen darauf folgenden Task, einem anschließenden *Event-basierten Gateway* in Kombination mit einem *Xor-Gateway* zur Zusammenführung des Kontrollflusses sowie einem abschließenden End-Event. Zwischen den beiden Gateways sind zwei verschiedene Zweige, ein Zweig mit einem *Timer-Event* und ein anderer Zweig mit einem *Message-Event*. Der Task repräsentiert, wie bereits beim „Out-Only“, eine `<bl:interactionActivity>`, die eine Nachricht mit dem `messageLabel` „outOptionalIn!Out“ versendet. Die Semantik der Aktivität wird hier jedoch durch eine Referenz auf das Nachrichtenaustauschmuster „Out-Only“ festgelegt. Dementsprechend verweist die Nachricht „outOptionalIn!Out“ mit Hilfe des Attributs `messageRef` auf die Nachricht „outOnly!Out“. Die beiden Gateways repräsentieren eine `<bl:pick>` Aktivität und das Timer-Event ein `<bl:onAlarm>`-Element. Das Message-Event entspricht einem `<bl:onMessage>` mit `messageLabel=“outOptionalIn!In“`.

Das oberste BPMN Diagramm stellt schließlich das Nachrichtenaustauschmuster des Typs „Request for Bid“ dar. Es setzt sich zusammen aus einer Sequenz eines Start-Events, eines mehrfach instantiierenden Tasks und eines End-Events. Der Task repräsentiert eine `<bpel:forEach>`-Aktivität, die eine `<bl:interactionActivity>` enthält,

deren Semantik über das Attribut `mep` mit dem Wert „outOptionalIn“ spezifiziert ist. Die Aktivität spezifiziert einen `<bl:input>` mit `messageLabel="requestForBid!Out"` und einen `<bl:output>` mit `messageLabel="requestForBid!In".` Beide referenzieren über das Attribut `messageRef` ihr jeweiliges Gegenstück in der Definition des Nachrichtenaustauschmusters „Out-Optional-In“: „requestForBid!Out“ referenziert „outOptionalIn!Out“ und „requestForBid!In“ referenziert „outOptionalIn!In“.

Die vollständige Beschreibung der Nachrichtenaustauschmuster „Out-Only“, „Out-Optional-In“ und „Request for Bid“ sind in Anhang I, J und K nachzulesen.

3.3.2 Interaktionsaktivitäten mit komplexem Verhalten in ausführbaren Prozessen

Die Verwendung von Nachrichtenaustauschmustern zur Definition komplexer Interaktionsaktivitäten vereinfacht die Modellierung konkreter Prozesse: Prozesse können mit weniger Aufwand und damit schneller modelliert werden. Kommen Nachrichtenaustauschmuster in konkreten Prozessen zum Einsatz, so reicht es nicht aus, die parametrisierten Größen des simpleren Nachrichtenaustauschmusters lediglich im komplexen Nachrichtenaustauschmuster als Parameter identifizierbar zu machen. Stattdessen müssen in diesem Fall entsprechend der Vorgaben für ausführbares BPEL^{light} alle vom Nachrichtenaustauschmuster zur Verfügung gestellten parametrisierten Größen mit konkreten Werten belegt, also vollständig spezifiziert werden, so dass eine Ausführung möglich ist. Zum besseren Verständnis wird hierbei nochmals explizit auf die entsprechenden Elemente und Attribute eingegangen.

Das Attribut `variable` oder das entsprechende `<bpel:to>` bzw. `<bpel:from>`-Element muss in der `<bl:interactionActivity>` für die Elemente `<bl:input>` und `<bl:output>` spezifiziert sein. Ebenfalls in der `<bl:interactionActivity>` muss gegebenenfalls das Attribut `faultName` der Elemente `<bl:inFault>` und `<bl:outFault>` spezifiziert werden sowie die `<bl:timingExpression>`s (siehe Listing 21): Um diese zu spezifizieren,

müssen die Attribute `type` und `value` mit Werten belegt werden und gegebenenfalls das Attribut `expressionLanguage`. Die nicht in der `<bl:interactionActivity>` definierte sondern lediglich referenzierte `<bl:expression>` muss im jeweiligen Prozess bzw. Gültigkeitsbereich durch das Attribut `value` und gegebenenfalls durch das Attribut `expressionLanguage` spezifiziert werden. Der referenzierte `<bl:partner>` bedarf keiner weiteren Spezifikation. Die konkreten Instanzinformationen werden außerhalb des Prozessmodells während des Deployments oder zur Laufzeit festgelegt (siehe Kapitel 4 und Kapitel 5).

```
<bpel:extensionActivity>
  <bl:interactionActivity ...>...
    <bl:timingExpression timingExpressionLabel="NCName"?
                         timingExpressionRef="NCName"?
                         expressionLanguage="anyURI"?
                         type="for|until|repeatEvery"?
                         value="expr"?/>*
  </bl:interactionActivity>
</bpel:extensionActivity>
```

Listing 21: Schema des `<bl:timingExpression>`-Elements der `<bl:interactionActivity>`

In Listing 22 ist ein Ausschnitt eines ausführbaren Prozesses dargestellt in dem eine `<bl:interactionActivity>` eingesetzt wird, deren Semantik durch das Nachrichtenaustauschmuster „Request with Referral“ definiert wird. Im Gegensatz zum Einsatz der `<bl:interactionActivity>` in einem abstrakten Prozess (siehe Abbildung 31) sind hier alle zur Ausführung notwendigen Details spezifiziert: Der `<bl:input>` referenziert die Variable „invoice“ und der `<bl:output>` die Variable „confirmation“. Die Variablen sind vom Typ „ns:tInvoice“ bzw. „ns:tConfirmation“. Der die Fehlernachricht „contactedProviderFault“ referenzierende `<bl:outFault>` spezifiziert den Fehler „ns:wrongProvider“, der die Fehlernachricht „respondingProviderFault“ referenzierende `<bl:outFault>` spezifiziert den Fehler „ns:transferFailed“.

```
<bpel:variables>
  <bpel:variable name="invoice" type="ns:tInvoice".../>
  <bpel:variable name="confirmation" type="ns:tConfirmation" .../>
</bpel:variables>
<bl:partners>
  <bl:partner name="customer"/>
  <bl:partner name="bankingHouse"/>
</bl:partners>
...
  <bpel:extensionActivity>
    <bl:interactionActivity mep="http://.../mepInBpel#requestWithReferral"
      <bl:input variable="invoice"
                messageRef="request".../>
      <bl:output variable="confirmation"
                 messageRef="response".../>
      <bl:outFault faultName="ns:wrongRecipient"
                   messageRef="contactedProviderFault".../>
      <bl:outFault faultName="ns:transferFailed"
                   messageRef="respondingProviderFault".../>
      <bl:partner name="customer"
                  partnerRef="contactedProvider"/>
      <bl:partner name="bankingHouse"
                  partnerRef="respondingProvider"/>
    </bl:interactionActivity>
  </bpel:extensionActivity>
...
```

Listing 22: `<bl:interactionActivity>` in einem konkreten Prozess deren Semantik durch das Nachrichtenaustauschmuster „Request with Referral“ bestimmt wird

3.4 Nachrichtenaustauschmuster zur Kategorisierung von Prozessfragmenten

Nachrichtenaustauschmuster definieren die Reihenfolge, in der Nachrichten gesendet bzw. empfangen werden. Sie definieren den Kontrollfluss zwischen „abstrakten“ Nachrichten aber machen im Allgemeinen keine Aussage über den Datenfluss während des Nachrichtenaustauschs, d.h. es wird nicht spezifiziert, wie die Beziehung der Inhalte der Nachrichten zueinander ist. Wird ein Nachrichtenaustauschmuster nun verwendet, um die Semantik einer `<bl:interactionActivity>` zu definieren, so ist es nicht ohne weiteres möglich einen komplexen Datenfluss in das Nachrichtenaustauschmuster zu injizieren. Soll für einen durch ein Nachrichtenaustauschmuster beschriebenen Nachrichtenaustausch ein komplexer Datenfluss definiert werden, so gibt es hierzu mehrere Möglichkeiten. Zum einen kann Aspekt-

orientiertes Programmieren (AOP) verwendet werden, um `<bpel:assign>`-Aktivitäten in das konkretisierte Nachrichtenaustauschmuster hineinzuweben. Dieser Ansatz wird für BPEL-Prozesse im Allgemeinen in [KaLe09] beschrieben und in dieser Arbeit nicht weiter verfolgt. Zum anderen ist es aber auch möglich, das abstrakte, ein `<bl:interactionActivity>` beschreibendes Nachrichtenaustauschmuster, in dem umgebenden Prozess aufgehen zu lassen und dann den komplexen Datenfluss in den Kontrollfluss des Prozesses hineinzumodellieren. Eine weitere Indikation dafür, ein Nachrichtenaustauschmuster in einen Prozess aufgehen zu lassen, ist die Notwendigkeit, Synchronisationspunkte zwischen dem Nachrichtenaustauschmuster und dem übrigen Prozess zu definieren durch das Einfügen zusätzlicher Kontrolllinks zwischen Aktivitäten des Nachrichtenaustauschmusters und dem Prozess.

Als *Prozessfragment* wird im Kontext dieser Arbeit eine Menge von Aktivitäten sowie deren Vorgänger- und Nachfolgerbeziehung definiert. Ein Nachrichtenaustauschmuster wird definiert durch eine Konversation und entsprechende, die Konversation referenzierende Aktivitäten. Eine Konversation ist somit ein Prozessfragment, welches der Erfüllung eines Geschäftsziels dient und ausschließlich Aktivitäten zur Interaktion mit Geschäftspartnern enthält. Lässt man ein durch eine `<bl:interactionActivity>` konkretisiertes Nachrichtenaustauschmuster in dem umgebenden Prozess aufgehen, so steht an Stelle der `<bl:interactionActivity>` eine durch die in der `<bl:interactionActivity>` angegebenen Werte konkretisierte Konversation, d.h. ein neues Prozessfragment innerhalb des Prozesses.

Wird ein Nachrichtenaustauschmuster von einer `<bl:interactionActivity>` referenziert, so definiert es dessen Verhalten und darf folglich nicht manipuliert werden. Um zu verhindern, dass ein Nachrichtenaustauschmuster, das ein Prozessfragment beschreibt, manipuliert werden kann, muss es folglich möglich sein, festzuhalten, dass es sich bei der entsprechenden Konversation um ein Nachrichtenaustauschmuster handelt. Und da die `<bl:interactionActivity>` vorher Teil einer höherwertigen Konversation war, muss es zudem möglich sein, auch

die nun konkretisierte Konversation des Nachrichtenaustauschmusters dieser höherwertigen Konversation zuzuordnen.

```
<bl:conversations>?
  <bl:conversation name="NCName"
                   mep="anyURI"?
                   parent="NCName"?/>+
</bl:conversations>
```

Listing 23: Vollständiges Schema der `<bl:conversation>`

Dies wird anhand zweier Attribute in der `<bl:conversation>` erreicht (siehe Listing 23). Das Attribut `mep` ist optional und ermöglicht es, das Nachrichtenaustauschmuster zu spezifizieren, dessen operationaler Semantik die Konversation entspricht. Ist das Attribut nicht spezifiziert, so können beliebige Aktivitäten zur Konversation hinzugefügt bzw. von der Konversation entfernt werden. Auch die Reihenfolge der Aktivitäten kann beliebig verändert werden. Ist das Attribut jedoch gesetzt, so wird damit ausgesagt, dass die Konversation einem bestimmten abstrakten Nachrichtenaustauschmuster folgt. In diesem Fall sind keine Modifikationen erlaubt, die zur Folge haben, dass die operationale Semantik der Konversation nicht dem referenzierten Nachrichtenaustauschmuster entspricht. Modifikationen, die das von außen sichtbare Verhalten der Konversation nicht verändern, wie beispielsweise `<bpel:assign>`-Aktivitäten zur Definition des Datenflusses, sind weiterhin möglich. Das Attribut `parent` ist ebenfalls optional und ermöglicht eine Konversation einer höherwertigen Konversation unterzuordnen.

Wie bereits in Abschnitt 3.3.1 gezeigt, kann ein Nachrichtenaustauschmuster rekursiv aus anderen Nachrichtenaustauschmustern aufgebaut sein. Das bedeutet auch, dass man ein solch rekursiv aufgebautes Nachrichtenaustauschmuster schrittweise in einem Prozess aufgehen lassen kann. Jede Ebene eines Nachrichtenaustauschmusters wird dabei durch eine Konversation repräsentiert. Das vollständige aufgehen lassen eines Nachrichtenaustauschmusters kann also in einem unübersichtlichen Prozess resultieren. Es ist daher empfehlenswert, nur so viele Ebenen des Nachrichtenaustauschmusters in den Prozess aufgehen zu lassen wie nötig sind, um den Datenfluss zu definieren.

3.5 Beispiel: BPEL^{light}-Prozess als Klient eines Onlineversandhandeldienstes

Um die Eignung von BPEL^{light} als Metamodell für ausführbare Geschäftsprozesse zu demonstrieren, wird in diesem Abschnitt der in Abschnitt 2.1.2.3 eingeführte Klient eines Onlineversandhandels als BPEL^{light}-Prozess präsentiert.

BPEL^{light} beschreibt den Nachrichtenaustausch unabhängig von der Bündelung der Nachrichten zu WSDL-Operationen. Hierzu stehen die `<bl:interactionActivity>`, das `<bl:pick>` sowie `<bl:eventHandlers>` zur Verfügung. In diesem Beispiel muss kein „Deferred Choice“ Verhalten modelliert werden und es muss keine eintreffende Nachricht parallel zum normalen Kontrollfluss verarbeitet werden. Daher kommt hier nur die `<bl:interactionActivity>` zum Einsatz. Diese kann sowohl lediglich eine Nachricht empfangen bzw. eine Nachricht senden als auch ein komplexes Verhalten implementieren. Daher gibt es mehrere Möglichkeiten den Nachrichtenaustausch zu modellieren. Es können fünf Aktivitäten verwendet werden, die jeweils eine Nachricht senden bzw. empfangen. Es kann aber auch ein der Folge der fünf Nachrichten entsprechendes Nachrichtenaustauschmuster definiert werden, welches dann zur Definition der Semantik einer einzelnen Aktivität verwendet werden kann. Ein Mittelweg, der jeweils zwei Aktivitäten mit dem Verhalten „Out-In“ und eine Aktivität mit dem Verhalten „In“ zur Modellierung des Szenarios verwendet und damit der Granularität des BPEL-Prozesses in Listing 2 entspricht, ist in Listing 24 abgebildet. Von der Kommunikation mit anderen Geschäftspartnern vor, während oder nach der angesprochenen Konversation wird abstrahiert.

```
<bpel:process name="buyProcess"
              targetNamespace="http://example.org/"...>...
  <bpel:variables>
    <bpel:variable name="order".../>
    <bpel:variable name="invoice".../>
    <bpel:variable name="payment".../>
    <bpel:variable name="receipt".../>
    <bpel:variable name="shippingDetails".../>
  </bpel:variables>...
  <bl:partners>
    <bl:partner name="seller" />...
  </bl:partners>...
  <bl:conversations>
    <bl:conversation name="salesConv" />...
  </bl:conversations>...
  ...
    <bpel:sequence>
      <bpel:extensionActivity>
        <bl:interactionActivity bpel:name="order"
                                conversation="salesConv"
                                mep="http://.../outIn>
          <bl:input messageRef="out"
                    variable="order"... />
          <bl:output messageRef="in"
                     variable="invoice"... />
          <bl:partner name="seller"
                      partnerRef="provider" ... />
        </bl:interactionActivity>
      </bpel:extensionActivity>...
      <bpel:extensionActivity>
        <bl:interactionActivity bpel:name="pay"
                                conversation="salesConv"
                                mep="http://.../outIn">
          <bl:input messageRef="out"
                    variable="payment"... />
          <bl:output messageRef="in"
                     variable="receipt"... />
          <bl:partner name="seller"
                      partnerRef="provider" ... />
        </bl:interactionActivity>
      </bpel:extensionActivity>...
      <bpel:extensionActivity>
        <bl:interactionActivity name="receiveShippingDetails"
                                conversation="salesConv" ...>
          <bl:output variable="shippingDetails"... />
          <bl:partner name="seller"... />
        </bl:interactionActivity>
      </bpel:extensionActivity>...
    </bpel:sequence>
  ...
</bpel:process>
```

Listing 24: BPEL[light]-Prozess als Klient eines Onlineversandhandeldienstes

Zunächst versendet die Aktivität „order“ den Inhalt der Variablen „order“ an den Partner „seller“ und speichert die in der Folge vom Partner „seller“ eintreffende Nachricht in der Variablen „invoice“ ab. Anschließend versendet die Aktivität „pay“ den Inhalt der Variablen „payment“ an den Partner „seller“ und speichert die danach vom Partner „seller“ eintreffende Nachricht in der Variablen „receipt“ ab. Schließlich empfängt die Aktivität „receiveShippingDetails“ eine weitere Nachricht vom Partner „seller“ und speichert sie in der Variablen „shippingDetails“ ab. Alle genannten Aktivitäten dienen der Erfüllung eines gemeinsamen Ziels und sind folglich durch die Konversation „salesConv“ gruppiert.

3.6 Methodologie: Transformationen zwischen WS-BPEL und BPEL^{light}

Eine Möglichkeit, einen BPEL^{light}-Prozess zu erhalten, ist es, diesen von Grund auf neu zu modellieren. Sind jedoch schon Prozessmodelle in WS-BPEL vorhanden, so bietet es sich an, diese als Grundlage zu nehmen und sie nach BPEL^{light} zu transformieren. Somit lässt sich der Aufwand zur Erstellung von BPEL^{light}-Prozessen minimieren. Dieser Abschnitt zeigt daher, wie BPEL^{light}-Prozessmodelle aus WS-BPEL-Prozessmodellen generiert werden können Auch die Rückgewinnung von WS-BPEL-konformen Prozessmodellen aus BPEL^{light}-Prozessmodellen wird diskutiert.

3.6.1 Von WS-BPEL zu BPEL^{light}

WS-BPEL definiert die Beziehung eines Prozesses mit seinen Geschäftspartnern auf rein technischer Ebene. Es werden Partnerlinks definiert, die angeben, welcher Porttyp von einem Geschäftspartner bzw. vom Prozess implementiert werden muss. Ein Partnerlink entspricht aufgrund seiner bilateraler Eigenschaft in BPEL^{light} am ehesten dem `<bl:partner>`, der jedoch nicht auf einen Porttypen reduziert ist, sondern allgemein einen Geschäftspartner repräsentiert. Einem Prozess in WS-BPEL hingegen ist es nicht anzusehen, ob mehrere Partnerlinks mit dem gleichen Geschäftspartner etabliert werden sollen. In BPEL4WS 1.1 war dies noch durch das `<bpel:partner>`-Element möglich. Es konnte verwendet werden, um mehrere Partnerlinks zu gruppieren. In WS-BPEL 2.0

allerdings wurde das `<bpel:partner>`-Element ersatzlos gestrichen und gehört damit nicht mehr zum konzeptionellen Modell von BPEL. Die Bündelung von mehreren Partnerlinks ist somit nur noch über das allgemeine Element `<bpel:documentation>` möglich. Es erlaubt, Dokumentation jeglicher Art in ein Prozessmodell einzufügen. Das Zusammenfassen mehrerer `<bpel:partnerLink>`s zu einem `<bl:partner>` kann also nicht automatisch erfolgen, sondern bedarf der menschlichen Interaktion. Soll ein Prozess vollautomatisch transformiert werden, so ist für jeden `<bpel:partnerLink>` ein `<bl:partner>` anzulegen.

BPEL^{light} bietet weiterhin die Möglichkeit, Nachrichten, die zur Erfüllung eines bestimmten Ziels ausgetauscht werden, zu einer Konversation zusammenzufassen. Die Konversationen sind hierbei nicht auf bilaterale Beziehungen beschränkt, sondern können Nachrichten an bzw. von unterschiedlichen Geschäftspartnern umfassen. Das Metamodell von WS-BPEL sieht die Definition eines potentiell multilateralen Nachrichtenaustausches, der notwendig ist, um ein Geschäftsziel zu erreichen, nicht vor. Dies ist wiederum nur über das allgemeine `<bpel:documentation>`-Element möglich. Die Überführung von WS-BPEL-Prozessmodellen zu entsprechenden BPEL^{light}-Prozessmodellen mit komplexen Konversationen ist also nur mit Hilfe menschlicher Interaktion möglich. Rudimentäre Zusammenhänge können allerdings aus BPEL-Interaktionsaktivitäten abgeleitet werden. Sie referenzieren eine WSDL-Operation und implementieren somit ein bestimmtes Nachrichtenaustauschmuster, die in BPEL^{light} durch Konversationen repräsentiert werden.

Die BPEL-Interaktionsaktivitäten `<bpel:receive>`, `<bpel:reply>` und `<bpel:invoke>` können also auf eine `<bl:interactionActivity>` abgebildet werden, dessen Semantik durch das entsprechende Nachrichtenaustauschmuster spezifiziert ist. Alternativ können `<bpel:invoke>`-Aktivitäten, die eine Operation des Typs Request-Response aufrufen, auch auf eine `<bl:interactionActivity>` und eine folgende `<bl:pick>`-Aktivität abgebildet werden, die über eine Konversation gruppiert werden. Implementiert ein `<bpel:receive>`-`<bpel:reply>` Paar eine Operation des Typs Request-Response, so werden die beiden daraus

gewonnenen `<bl:interactionActivity>`s durch eine Konversation gruppiert. Die Transformation der `<bpel:invoke>`-Aktivitäten, die Operationen der `partnerRole` aufrufen, zu BPEL^{light}-Aktivitäten ist in Abbildung 32 und Abbildung 33 dargestellt, die der BPEL-Aktivitäten, die Operationen der Rolle `myRole` implementieren, in Abbildung 34 und Abbildung 35. In den genannten Darstellungen wird diesmal nicht von Fehlern abstrahiert. Fehlernachrichten in BPEL^{light} werden mit „f“ bezeichnet. Die `<bpel:pick>`-Aktivität und die `<bpel:eventHandler>` werden auf `<bl:pick>` bzw. `<bl:eventHandler>` abgebildet. Implementieren sie gemeinsam mit einer `<bpel:reply>`-Aktivität eine Operation des Typs Request-Response, so werden die beiden den BPEL-Aktivitäten entsprechenden BPEL^{light}-Aktivitäten durch eine Konversation gruppiert.

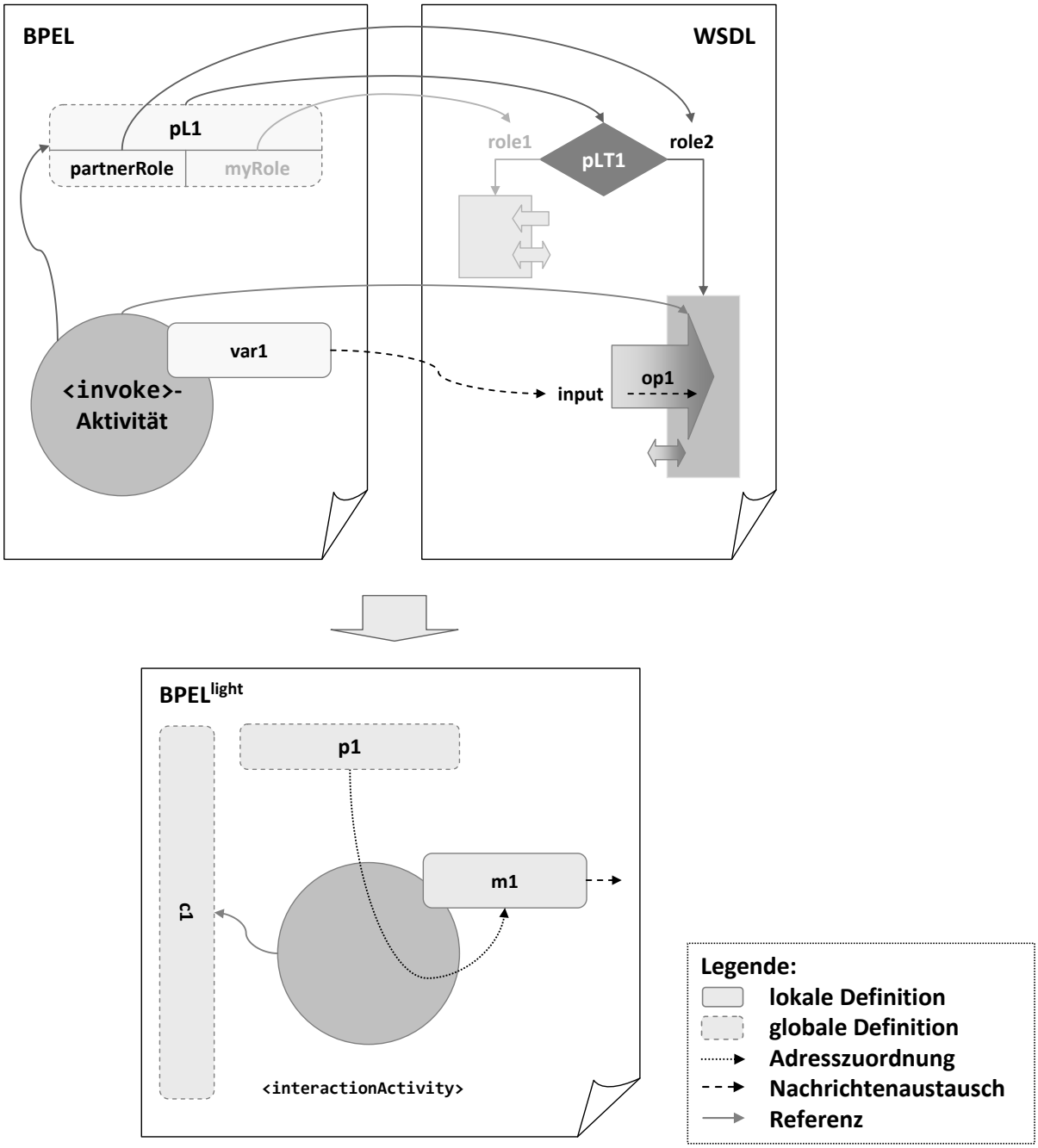


Abbildung 32: Transformation einer One-way `<invoke>`-Aktivität zu BPEL[light]

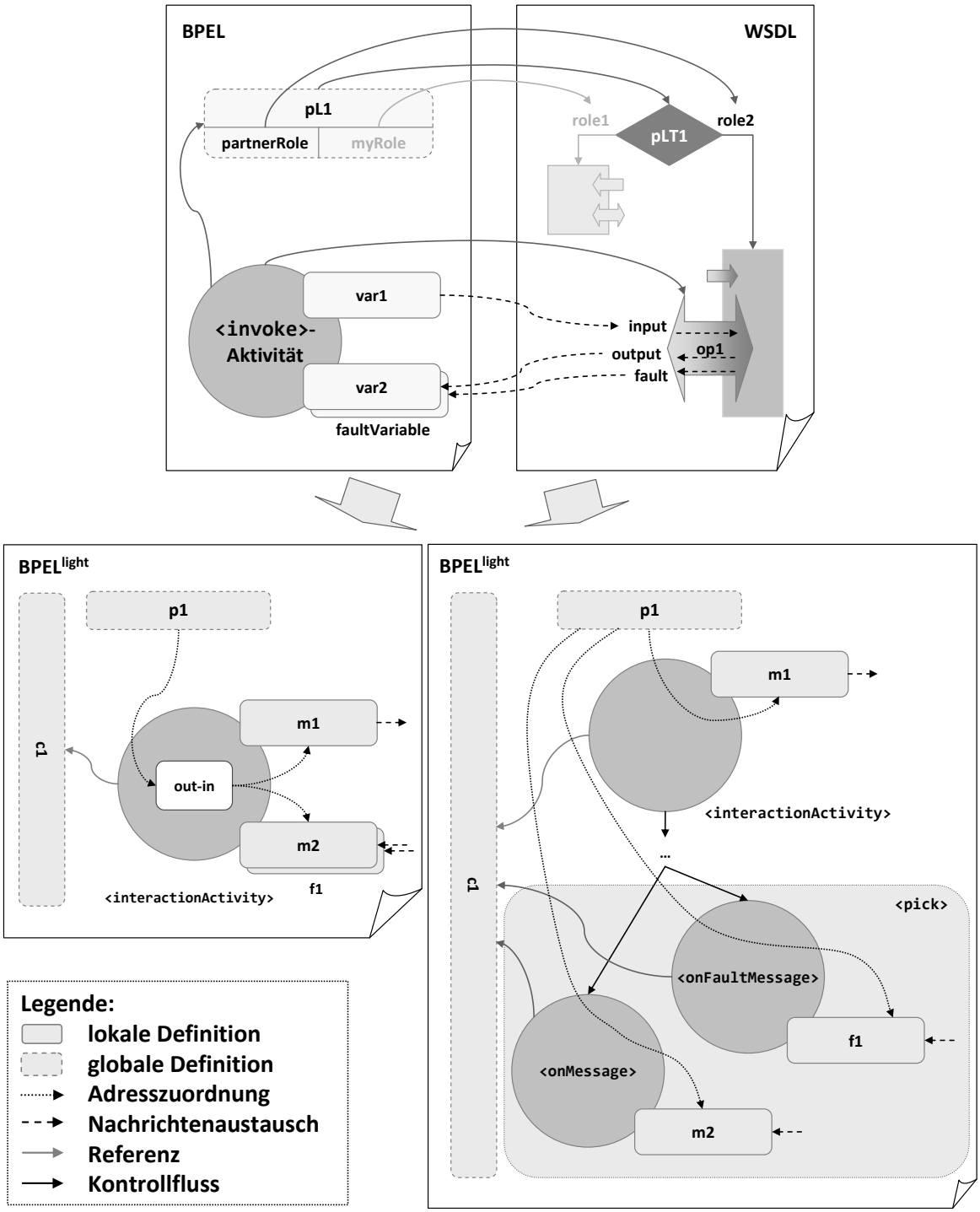


Abbildung 33: Transformation einer Request-Response <invoke>-Aktivität zu BPEL^light

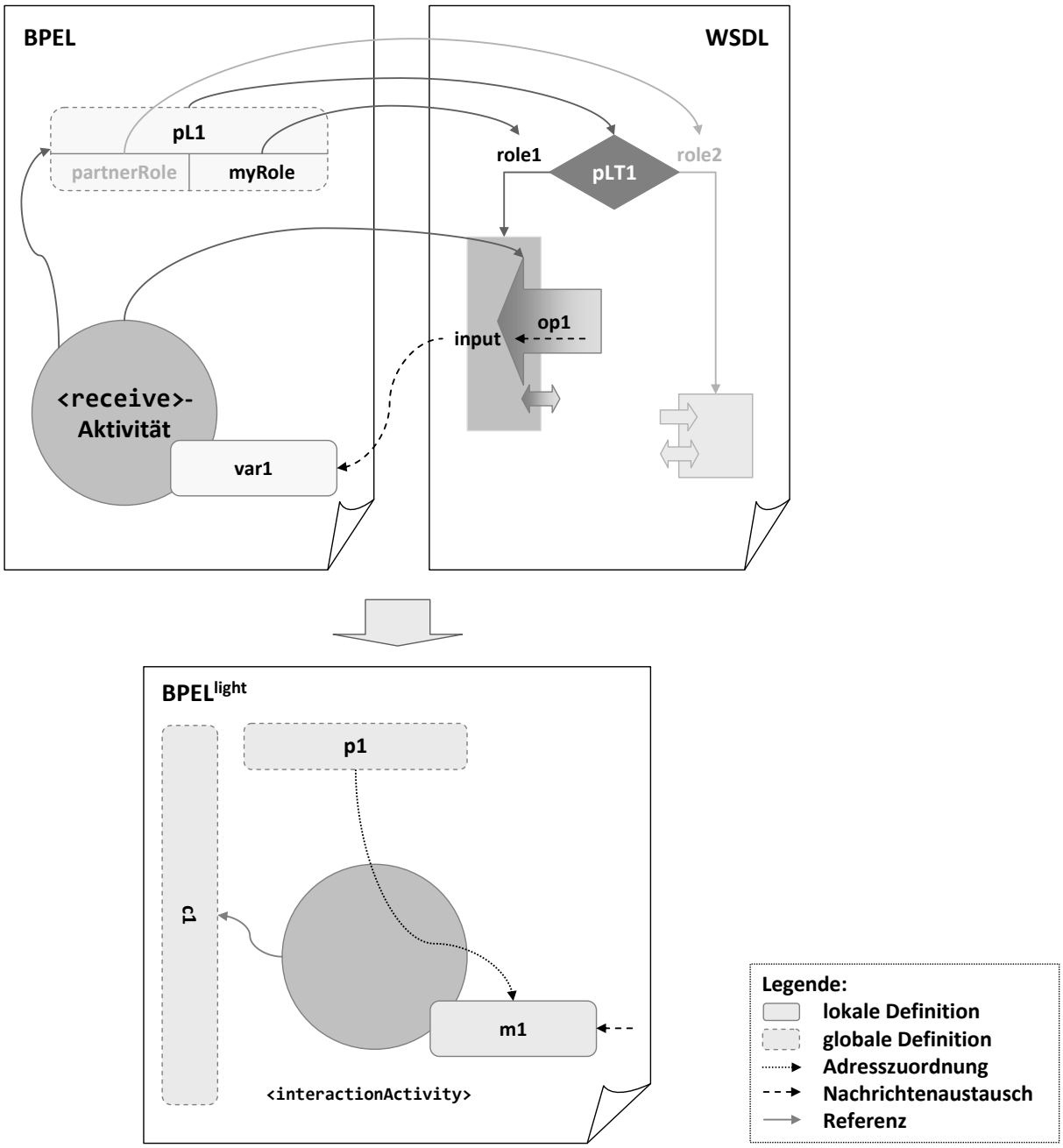


Abbildung 34: Transformation einer `<receive>`-Aktivität, die eine One-way-Operation implementiert, zu BPEL[light]

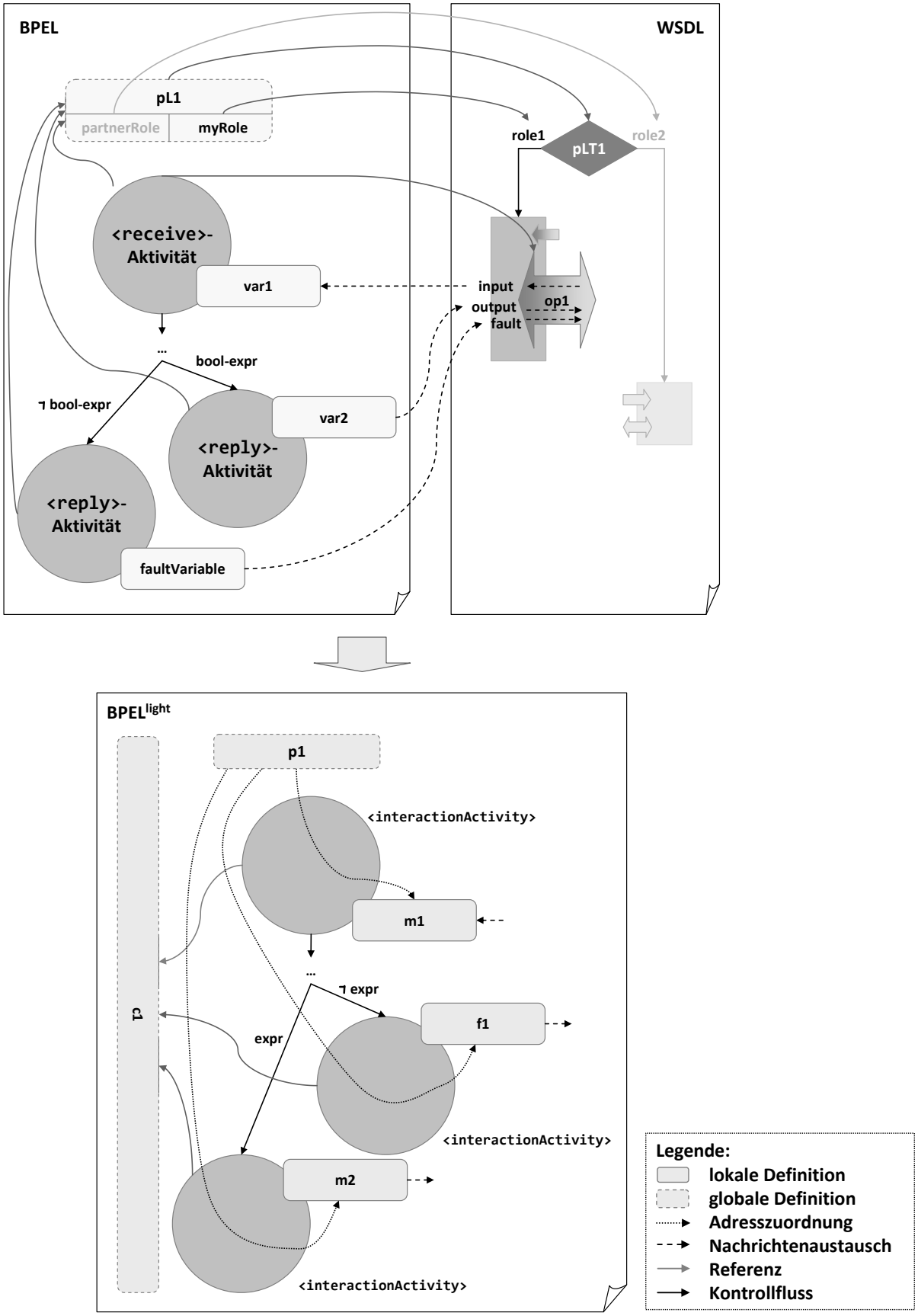


Abbildung 35: Transformation eines <receive>-<reply>-Paares zu BPEL^light

3.6.2 Von BPEL^{light} zu WS-BPEL

BPEL^{light}-Prozesse abstrahieren von WSDL-Porttypen und –Operationen. Da deren Angabe in einem WS-BPEL-Prozess allerdings zwingend ist, ist es im Allgemeinen nicht möglich, aus einem BPEL^{light}-Prozess einen WS-BPEL-Prozess zu gewinnen, der existierende Partnerlinks verwendet, sich also auf existierende Porttypen und Operationen bezieht. Dies ist nur möglich, falls das BPEL^{light} Interaktionsmodell außerhalb der Prozessbeschreibung auf WSDL abgebildet wird. In Kapitel 4 und Kapitel 5 werden zwei Ausprägungen einer solchen Abbildung sowie das Generieren eines WS-BPEL-Prozesses aus einem BPEL^{light}-Prozess unter Verwendung existierender WSDL-Beschreibungen beschrieben.

Existiert keine Abbildung eines BPEL^{light}-Prozesses auf entsprechende WSDL-Beschreibungen, so kann man zwar einen BPEL^{light}-Prozess zu einem WS-BPEL-Prozess überführen, jedoch nur zu einem WS-BPEL-Prozess, der nicht bereits existierende WSDL-Beschreibungen verwendet, sondern WSDL-Beschreibungen, die aus dem der Transformation zu Grunde liegenden BPEL^{light}-Prozess generiert werden. Dies wird im Folgenden genauer erläutert.

In einem BPEL^{light}-Prozess werden Konversationen verwendet, um den logischen Zusammenhang ausgetauschter Nachrichten zu kennzeichnen. Es können auch Nachrichten unterschiedlicher Sender bzw. Empfänger zu einer Konversation gruppiert werden. Außerdem können Geschäftspartner an unterschiedlichen Konversationen partizipieren. Eine Aktivität kann den Austausch einer einzelnen Nachricht (einfache Interaktionsaktivität) aber auch den Austausch mehrerer Nachrichten (komplexe Interaktionsaktivität) durchführen.

Einfache Interaktionsaktivitäten können direkt auf entsprechende BPEL-Konstrukte abgebildet werden: sendende `<bl:interactionActivity>`s auf `<bpel:invoke>`s und empfangende `<bl:interactionActivity>`s auf `<bpel:receive>`s. Sendende `<bl:interactionActivity>`s, die einer empfangenden `<bl:interactionActivity>` folgen, können auch auf `<bpel:reply>`-Aktivitäten abgebildet werden. Allerdings ist dies nur möglich, wenn die Gegebenheiten synchrone Kommunikation zulassen.

Analog zu den einfachen Interaktionsaktivitäten werden `<bl:pick>`-Aktivitäten auf `<bpel:pick>`-Aktivitäten und `<bl:eventHandlers>` auf `<bpel:eventHandlers>` abgebildet. Schwieriger gestaltet sich die Abbildung bei komplexen Interaktionsaktivitäten. Lediglich eine `<bl:interactionActivity>` die das Verhalten „Out-In“ implementiert kann direkt auf eine `<bpel:invoke>-Aktivität` abgebildet werden, unter der Vorraussetzung, dass nicht asynchron kommuniziert werden muss. Alle Aktivitäten, die ein anderes komplexes Nachrichtenaustauschmuster referenzieren, müssen durch ein die entsprechende Semantik implementierendes Prozessfragment ersetzt werden. In diesem Fragment dürfen nur einfache Interaktionsaktivitäten, Interaktionsaktivitäten mit dem Verhalten „Out-In“, `<bl:pick>`-Aktivitäten und `<bl:eventHandlers>` zum Einsatz kommen. Während der Transformation müssen also die von den komplexen Interaktionsaktivitäten referenzierten Nachrichtenaustauschmuster in entsprechende Prozessfragmente überführt werden. Dabei müssen alle neu eingeführten BPEL^{light}-Konstrukte wie z.B. die `<bl:timingExpression>` auf entsprechende BPEL-Konstrukte abgebildet werden. Dies trifft auch zu, wenn die Konstrukte im Prozess selber Verwendung finden.

Die Generierung der WSDL-Beschreibungen wird wie folgt durchgeführt. Es wird für jede Konversation pro Partner ein Partnerlink und ein entsprechender Partnerlinktyp definiert. Die Operationen der Porttypen, welche die Rollen des Partnerlinktyps realisieren, können aus den Interaktionsaktivitäten abgeleitet werden:

- Eine sendende `<bl:interactionActivity>`, die auf ein `<bpel:invoke>` abgebildet wird, impliziert eine Operation des Typs One-way im Porttypen des Partners.

- Eine `<bl:interactionActivity>`, die auf ein `<bpel:invoke>` abgebildet wird und durch das Nachrichtenaustauschmuster „out-in“ charakterisiert ist, impliziert eine Operation des Typs Request-Response im Porttypen des Partners.

- Eine empfangende `<bl:interactionActivity>`, die auf eine `<bpel:receive>`-Aktivität abgebildet wird, sowie das `<bl:onMessage>` Event einer `<bl:pick>`-Aktivität bzw. das `<bl:onEvent>` des `<bl:eventHandlers>` implizieren:
  - eine Operation des Typs One-way im Porttypen des Prozesses, wenn sie nicht in Kombination mit einer darauf folgenden, auf eine `<bpel:reply>`-Aktivität abgebildeten, sendenden `<bl:interactionActivity>` verwendet wird,
  - eine Operation des Typs Request-Response im Porttypen des Prozesses, wenn sie in Kombination mit einer darauf folgenden, auf eine `<bpel:reply>`-Aktivität abgebildeten, sendenden `<bl:interactionActivity>` verwendet wird.

Diese Operationen müssen eindeutig benannt werden und von den entsprechenden Interaktionsaktivitäten des Prozesses referenziert werden.

3.7 Zusammenfassung

In diesem Kapitel wurde eine neue, von WSDL unabhängige BPEL-basierte Workflowsprache namens BPEL^{light} vorgestellt. BPEL^{light} entkoppelt die Prozesslogik vollständig von Schnittstellenspezifikationen und erhöht damit die Flexibilität der Prozesse bezüglich der „womit"-Dimension. Diese Trennung hat zur Folge, dass Prozessmodellierer bei Verwendung von BPEL^{light} lediglich die Definition der Prozesslogik beherrschen müssen. Kenntnisse über Web-Services sind – anders als bei BPEL – nicht erforderlich. Um die Entkopplung zu erreichen, definiert BPEL^{light} ein eigenes Interaktionsmodell als Erweiterung zu BPEL. Dadurch wird sichergestellt, dass die operationale Semantik von BPEL unberührt bleibt und dass entsprechend erweiterte BPEL-Engines verwendet werden können, um BPEL^{light}-Prozesse ausführen. Das Interaktionsmodell besteht zum einen aus Konstrukten zur Spezifikation von Geschäftspartnern eines Prozesses und zur Gruppierung von ausgetauschten Nachrichten, die gemeinsam einem Geschäftsziel dienen, und zum anderen aus Interaktionsaktivitäten zur Definition des Nachrichtenaustauschs. Die Semantik der Interaktionsaktivitäten kann durch Nachrichtenaustausch-

muster spezifiziert werden, die wiederum – dem in dieser Arbeit definierten Profil für abstraktes BPEL^{light} zur Definition von Nachrichtenaustauschmuster folgend – als abstrakte BPEL^{light}-Prozesse modelliert werden können. Dadurch können auch komplexe Geschäftsfunktionen durch eine einzige Aktivität abgebildet werden. Die Verwendung der gleichen Sprache zur Definition eines ausführbaren Geschäftsprozesses und zur Definition eines Nachrichtenaustauschmusters hat den Vorteil, dass ein BPEL^{light}-Prozessmodellierer gleichzeitig auch in der Lage ist, Nachrichtenaustauschmuster zu modellieren und zu verstehen.

Die wichtigsten Eigenschaften von BPEL^{light} werden anhand eines Vergleichs seines Interaktionsmodells mit dem Interaktionsmodell von WS-BPEL in Tabelle 1 zusammengefasst.

Kriterien	WS-BPEL	BPEL^{light}
Entkopplung der Prozesslogik von Schnittstellenspezifikationen	Nein	Ja
Definition bilateraler Kommunikation	Ja / Nein	Ja
Definition multilateraler Kommunikation	Nein	Ja
Zielorientierte Modellierung	Nein	Ja
Frei definierbare Semantik von Interaktionsaktivitäten	Nein	Ja

Tabelle 1: Vergleich von WS-BPEL und BPEL^{light}

Wie bereits erwähnt, ist in BPEL^{light} im Gegensatz zu BPEL die Prozesslogik von der Schnittstellenspezifikation entkoppelt. Die Definition bilateraler Kommunikation zwischen einem Prozess und dessen Geschäftspartner wird sowohl von WS-BPEL als auch BPEL^{light} unterstützt. In BPEL^{light} wird der Geschäftspartner auf konzeptioneller Ebene definiert. Er kann in unterschiedlichen Konversationen zum Einsatz kommen und unterschiedliche Rollen einnehmen. In BPEL wird der Partner lediglich auf

technischer Ebene, d.h. über einen Partnerlink, definiert. Man kann nicht ausdrücken, dass ein Geschäftspartner in mehreren Partnerlinks die Rolle des Kommunikationspartners einnimmt. Daher wird in Tabelle 1 das Kriterium „Definition bilateraler Kommunikation“ mit „ja/nein“ bewertet. Die Modellierung mehrerer bilateraler Beziehungen zwischen dem Prozess und unterschiedlichen Partnern ist in beiden Sprachen möglich. In BPEL über die Definition mehrerer Partnerlinks und in BPEL^{light} über die Definition mehrerer Partner. Das Zusammenwirken mehrerer bilateraler Beziehungen zur Erfüllung eines Geschäftsziels, d.h. die explizite Definition multilateraler Kommunikation, ist hingegen nur in BPEL^{light} mittels einer Konversation möglich. BPEL bietet einen den WSDL 1.1-Operationstypen entsprechenden fixen Satz Interaktionsaktivitäten. BPEL^{light} hingegen ermöglicht es, die Semantik von Interaktionsaktivitäten mit Hilfe von Nachrichtenaustauschmustern frei zu definieren. Trotz der unterschiedlichen Mächtigkeit lassen sich WS-BPEL-Prozessmodelle auf BPEL^{light}-Prozessmodelle abbilden. Auch eine Abbildung von BPEL^{light}-Prozessmodellen auf WS-BPEL-Prozessmodelle ist möglich.

Die in diesem Kapitel präsentierten Ergebnisse finden sich auch in den bereits veröffentlichten Publikationen [NLKL07b, NLL08, LNL08, NLL08b und NHL+08].

KAPITEL 4

BPEL^{LIGHT} MIT WEB-SERVICES

Anders als BPEL ermöglicht es BPEL^{light}, ausführbare Geschäftsprozesse zu definieren, ohne dabei Aussagen über die für die Implementierung einzelner Aktivitäten zur Verfügung stehende IT Infrastruktur zu machen. Anstatt Aktivitäten des Prozessmodells über Porttypen und Operationen zu definieren, wird in BPEL^{light} lediglich spezifiziert, welche Daten einer Aktivitätsimplementierung zur Verfügung stehen und welche Daten zur Weiterverarbeitung im Prozess benötigt werden. Sollen die Prozesse jedoch ausgeführt werden, so müssen spätestens zur Laufzeit den Anforderungen der Aktivitäten entsprechende Implementierungen ausgewählt und an die Aktivitäten gebunden werden.

Web-Services sind eine populäre Technologie zur Integration von Anwendungen. Auch BPEL – mit Web-Services als Aktivitätsimplementierungen – ist weit verbreitet und erfolgreich im Einsatz. Ein Einsatz von Web-Services als Implementierung von BPEL^{light}-Aktivitäten sowie zur Bereitstellung der Funktionalität von BPEL^{light}-Prozessen birgt daher großes Potential: Es ermöglicht zum einen die Verwendung zahlreicher als Web-Service vorhandener Dienste zur Implementierung von BPEL^{light}-Aktivitäten. Zum anderen wird eine für Klienten transparente Umstellung des vom Prozess implementierten Dienstes von BPEL zu BPEL^{light} möglich. Dieses Kapitel stellt Mechanismen vor, die es ermöglichen, BPEL^{light} in einer Web-Service-Umgebung einzusetzen. Die vorgestellten Mechanismen erlauben es sowohl einen BPEL^{light}-Prozess als Web-Service zur Verfügung zu stellen als auch Web-Services als Aktivitätsimplementierungen zu verwenden. Somit stellt BPEL^{light} in Kombination mit den in diesem Kapitel vorgestellten Mechanismen – wie konventionelles BPEL auch – ein rekursives Aggregationsmodell für Web-Services dar. In Abschnitt 4.1 werden zunächst Mechanismen vorgestellt, die nötig sind um BPEL^{light} in

Web-Service-Umgebungen einzusetzen. Anschließend wird in Abschnitt 4.2 gezeigt, wie die von BPEL unterstützten Interaktionsszenarien mit der Kombination von BPEL^{light} und Web-Services umgesetzt werden können. In Abschnitt 4.3 wird dann das durch die Arbeit hinweg verwendete Beispielszenario durch das Gespann BPEL^{light} mit Web-Services realisiert. In Abschnitt 4.4 wird das methodologische Vorgehen beschrieben, um BPEL^{light}-Prozesse mit Web-Services als Aktivitätsimplementierungen aus herkömmlichen BPEL-Prozessen zu gewinnen. Das Kapitel schließt mit einer Zusammenfassung (Abschnitt 4.5).

4.1 Web-Services als Aktivitätsimplementierungen

Sollen Web-Services als Aktivitätsimplementierungen herangezogen werden und dazu verwendet werden, die Funktionalität des Prozesses als Dienst zur Verfügung zu stellen, so muss das BPEL^{light} Interaktionsmodell auf entsprechende Konzepte der Web-Service-Technologie abgebildet werden:

In BPEL^{light} werden Interaktionsaktivitäten zum einen zu Konversationen gruppiert, zum anderen wird für alle Daten einer Interaktionsaktivität angegeben, welchem Kommunikationspartner welche Daten zur Verfügung gestellt werden und welche Daten von ihm erwartet werden (siehe Abschnitt 3.1.1). Zu beachten ist, dass Konversationen sowohl bilateral als auch multilateral sein können.

Um ein sich wechselseitiges Aufrufen zweier Partner über Web-Service-Technologie zu ermöglichen wurde, im Rahmen der BPEL-Spezifikation die WSDL-Erweiterung Partnerlinktyp definiert (siehe Abschnitt 2.1.2). Ein Partnerlinktyp definiert für zwei Interaktionspartner jeweils einen zu implementierenden Porttypen und ermöglicht somit, dass sich die beiden Partner wechselseitig über Operationen des Typs One-way und Request-Response aufrufen können.

Abbildung von BPEL^{light}-Aktivitäten auf Web-Services

Ein Partner in BPEL^{light} wird nicht wie die Partnerrolle eines BPEL-Partnerlinks mit einer EPR assoziiert, sondern mit abstrakt definierten Partnerinstanzen. Sollen Web-Services verwendet werden, um Aktivitäten zu implementieren, können den abstrakten Partnerinstanzen direkt EPRs zugeordnet werden. Es kann durch die Indirektion über die abstrakten Partnerinstanzen jedoch auch eine feingranularere Abbildung vorgenommen werden: Partner nehmen potentiell an unterschiedlichen Konversationen teil und nehmen dabei unterschiedliche Rollen ein. Es ist daher nahe liegend, einem Partner pro Konversation einen Partnerlinktypen zuzuordnen. Konversationen können hierarchisch definiert werden. Wird einem Partner für eine Konversation ein Partnerlinktyp zugeordnet, so gilt dieser auch für alle untergeordneten Konversationen. Neben der Zuordnung von Partnern zu Partnerlinktypen müssen auch die einzelnen Nachrichten eines BPEL^{light}-Prozessmodells der Prozesslogik entsprechend Operationen zugeordnet werden. Da die `<bl:interactionActivity>` im Gegensatz zu BPEL-Interaktionsaktivitäten keine Rückschlüsse darauf zulässt, von welcher Rolle des Partnerlinktypen die verwendete Operation implementiert wird und da Operationen nicht global, sondern lokal für einen Porttypen definiert werden, muss für jede Nachricht neben der Operation auch die Rolle, d.h. „myRole“ oder „partnerRole“ spezifiziert werden. Diese Zusammenhänge sind am Beispiel zweier Konversationen innerhalb eines BPEL^{light}-Prozesses sowie der zugehörigen Partner, Aktivitäten und Nachrichten in Abbildung 36 dargestellt. Wird eine BPEL^{light} Nachricht auf eine One-way-Operation abgebildet, so entspricht sie der durch die Operation empfangenen Nachricht. Soll sie auf eine Request-Response-Operation abgebildet werden, so ist durch ihre Richtung die Zuordnung zum Input bzw. Output der Operation eindeutig. Werden Nachrichten auf Request-Response-Operationen des Kommunikationspartners abgebildet, so muss die Antwortnachricht nicht auf den Input der Aktivität, welche die erste Nachricht sendet, abgebildet werden. Auch eine der sendenden Aktivität folgende Aktivität kann zur Verarbeitung der Antwortnachricht herangezogen werden. Wird ein Nachrichtenaustausch auf eine Request-

Response-Operation des Prozesses abgebildet, so ist darauf zu achten, dass die Zeitspanne zwischen der empfangenden Aktivität und der Aktivität, welche die Antwortnachricht verschickt, nur von geringer Dauer ist, so dass der Aufruf über HTTP nicht mit einem Time-out abbricht.

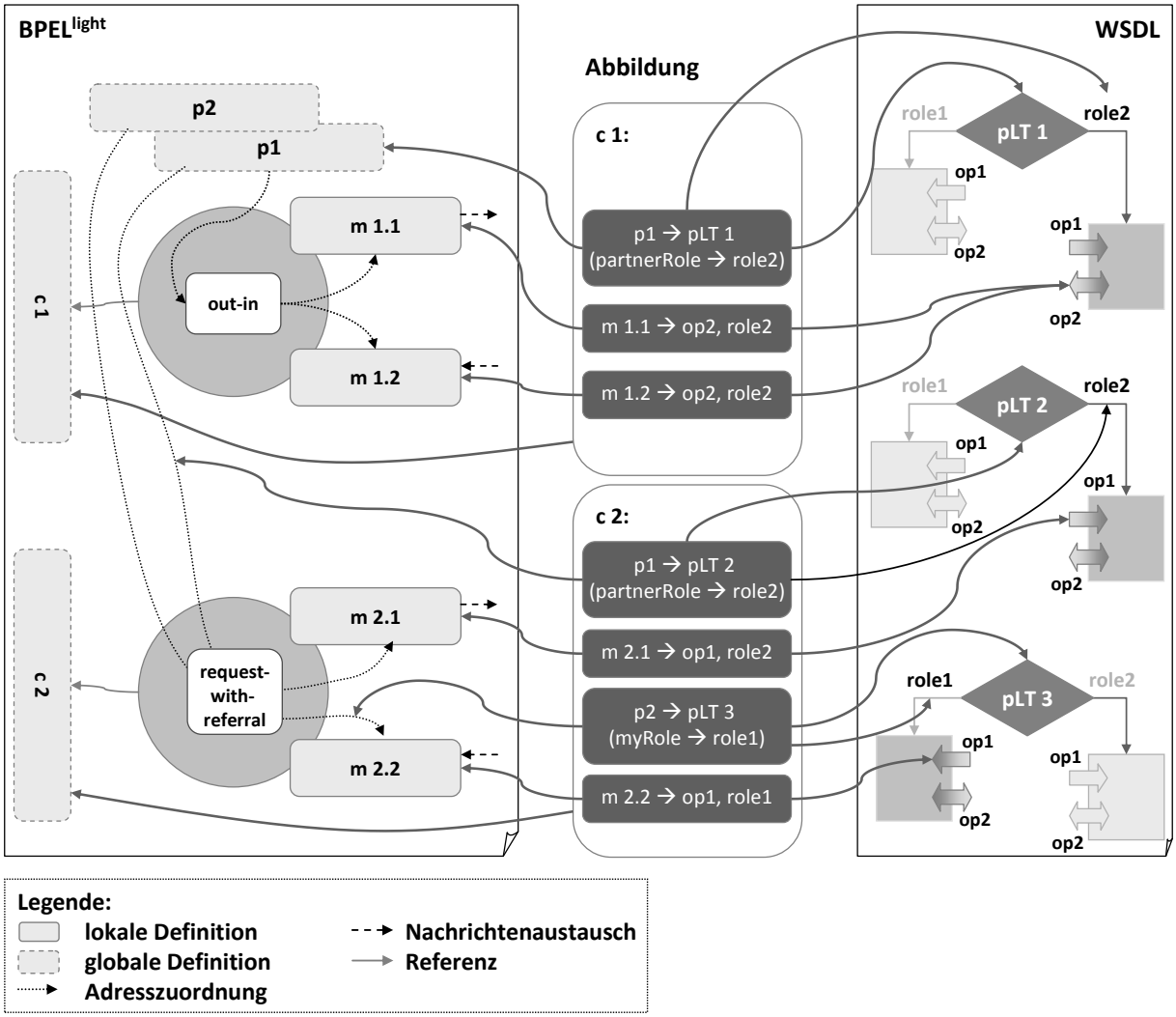


Abbildung 36: Abbildung von Konversationen, Partnern und Nachrichten auf entsprechende WSDL-Elemente

Der auf die Web-Service-Technologie abzubildende Prozess muss nicht nur aus einfachen Interaktionsaktivitäten bestehen bzw. aus Interaktionsaktivitäten, die das Nachrichtenaustauschmuster „Out-In" referenzieren. In einer beliebig komplexen `<bl:interactionActivity>` ist jede Nachricht des referenzierten Nachrichtenaustauschmusters auch im konkreten Prozess eindeutig identifizierbar. Es kann also jede Nachricht

einer Operation zugeordnet werden, in einem über die Kombination Konversation und Partner spezifizierten Partnerlinktypen.

Mögliche Techniken zur Abbildung

Die Spezifikation der Abbildung von BPEL^{light}-Aktivitäten auf Web-Services kann zu unterschiedlichen Zeitpunkten des Lebenszyklus eines Prozesses ausgewertet werden: während der Modellierung, des Deployments oder zur Laufzeit. Die Zuordnung der WSDL-Operationen zu Aktivitäten außerhalb des Prozessmodells wird in dieser Arbeit als Annotation bezeichnet.

Wird die Abbildung direkt im Prozessmodell vorgenommen, so sind bereits während der Modellierung die Schnittstellen der Geschäftspartner festgelegt. Diese Technik ist in BPEL umgesetzt. Sie birgt die für BPEL identifizierten Nachteile und wird deshalb nicht weiter verfolgt.

Soll die Abbildung beim Deployment eines Prozesses ausgewertet werden, so können Deployment Deskriptoren dazu verwendet werden, den Aktivitäten WSDL-Porttypen und –Operationen zuzuordnen. Die Auswahl der tatsächlichen Aktivitätsimplementierungen kann entweder auch während des Deployments vorgenommen werden oder zur Laufzeit unter Verwendung eines Enterprise Service Busses. Die Verwendung von Deployment Deskriptoren zur Spezifikation der Abbildung von Aktivitäten auf Operationen resultiert in einer zu herkömmlichem BPEL vergleichbaren Situation: Zwar sind die Prozessmodelle an sich unabhängig von WSDL definiert, sie werden jedoch beim Deployment an WSDL-Porttypen und –Operationen gebunden. Zur Laufzeit können daher lediglich Dienste ausgewählt werden, welche die beim Deployment gewählte Schnittstelle implementieren. Die Verwendung von Diensten mit anderer Schnittstelle erfordert jedoch lediglich eine Modifikation des Deployment Deskriptors und ein Redeployment des Prozesses wohingegen bei herkömmlichem BPEL das Prozessmodell geändert und der Deployment Deskriptor angepasst werden muss, um den Prozess redeployen zu können.

Eine weitere Möglichkeit ist es, während des Deployments noch keine Zuordnung von Aktivitäten zu Porttypen und Operationen vorzunehmen, sondern beispielsweise WS-PolicyAttachment [BHH+07] zu verwenden, das eine Zuordnung erst zur Laufzeit ermöglicht. Wird diese Option gewählt, so kann die komplette Abbildung der Prozesslogik auf die Kommunikationsinfrastruktur jederzeit durch Änderung der Annotation erfolgen, ohne den Prozess erneut deployen zu müssen. Dazu benötigt man einen neuen ESB (Enterprise Service Bus), dessen Implementierung nicht Gegenstand dieser Arbeit ist.

Im Folgenden wird die Abbildung des BPEL[light] Interaktionsmodells auf WSDL 1.1-Schnittstellenbeschreibungen mittels Deployment Deskriptoren beschrieben. Die Verwendung von WS-PolicyAttachment wird in dieser Arbeit nicht beschrieben und bleibt künftigen Arbeiten vorbehalten.

```
<blDeployment name="NCName"
              targetNamespace="anyURI"
              processName="anyURI"
              xmlns="http://iaas.uni-stuttgart.de/BPELlight/deployment/2008"
              xmlns:sref="http:// iaas.uni-stuttgart.de/BPELlight/serviceref/
                          2008">
              ...
  <partnerInstances partnerName="NCName">
    <partnerInstance id="xsd:int" name="NCName"?/>+
      <sref:service-Ref conversationName="NCName"
                        sref:reference-scheme="anyURI"/>
  </partnerInstances>
  <conversationMapping conversationName="NCName">
    <partnerMapping partnerName="NCName"
                    partnerLinkTypeName="QName"
                    myRole="NCName"?
                    partnerRole="NCName"?/>*
    <messageMapping messageLabel="NCName"
                    operationName="NCName"
                    role="partnerRole/myRole"?/>+
  </conversationMapping>
</blDeployment>
```

Listing 25: Schema des Deployment Deskriptors für BPEL[light] mit Web-Services

Listing 25 zeigt den schematischen Aufbau des Deployment Deskriptors. Das Attribut `processName` des `<blDeployment>`-Elements gibt an, für welches Prozessmodell die Abbildung von BPEL[light] auf Web-Service-Technologie definiert wird. Der Prozess wird über die mit einem „#“ als

Trennzeichen verknüpften Werte der Prozessattribute `targetnamespace` und `name` identifiziert. Innerhalb des Elements `<partnerInstances>` werden Instanzen eines Partnertyps abstrakt definiert: Jedes `<partnerInstance>`-Element repräsentiert eine Instanz, welche durch das Attribut `id` eindeutig identifiziert wird. Zusätzlich kann jeder Instanz über das `name`-Attribut ein Name zugeordnet werden. So lassen sich beispielsweise für den Partnertyp „travelAgent“ zwei Instanzen definieren, die Instanzen „BCD Travel“ und „Opodo.de“ mit den Identifiern „1“ bzw. „2“. Diesen Instanzen können dann pro Konversation eine Endpunktreferenz zugeordnet werden. Werden beispielsweise die Ziele „buchen“ und „stornieren“ über unterschiedliche Konversationen modelliert, so ist es möglich, diese Ziele auch dann mit der Instanz „BCD Travel“ zu verwirklichen, wenn „BCD Travel“ die dazu benötigten Funktionen über unterschiedliche Endpunkte anbietet.

Die Abbildung des Prozessmodells auf WSDL-Schnittstellenbeschreibungen erfolgt über die Elemente `<conversationMapping>`, `<partnerMapping>` und `<messageMapping>`. Die Attribute `conversationName`, `partnerName` und `messageLabel` geben an, für welche Elemente des Prozesses die Abbildung definiert wird. Die Attribute `partnerLinkTypeName` und `operationName` geben das Ziel der Abbildung an. Für den Partnerlinktypen wird mittels der Attribute `myRole` und `partnerRole` zudem spezifiziert, welche Rolle der Prozess und welche Rolle der Kommunikationspartner des Prozesses einnimmt. Für jeden Partner innerhalb einer Konversation wird mit dieser Abbildung ein Partnerlinktyp festgelegt. Außerdem wird jede Nachricht innerhalb der Konversation einer WSDL-Operation zugeordnet. Somit ermöglicht es der Deployment Deskriptor, dass ein Partner in unterschiedlichen Konversationen unterschiedliche Porttypen implementiert.

4.2 Unterstützte Interaktionsszenarien des Gespanns BPEL^{light} mit Web-Services

Um BPEL-Prozesse auf BPEL^{light} umstellen zu können, muss sichergestellt sein, dass BPEL^{light} in Kombination mit Web-Services die gleichen Szenarien realisieren kann wie herkömmliches BPEL. Im Folgenden werden deshalb

die in Abschnitt 2.1.2 eingeführten von BPEL unterstützten Interaktionsszenarien betrachtet: (i) synchroner Aufruf eines kurzlebigen BPEL^{light}-Prozesses, (ii) blockierender, synchroner Aufruf eines Dienstes, (iii) asynchroner Aufruf eines langlebigen BPEL^{light}-Prozesses und (iv) nicht blockierender, asynchroner Aufruf eines Dienstes.

(i) Aufruf eines kurzlebigen BPEL^{light}-Prozesses

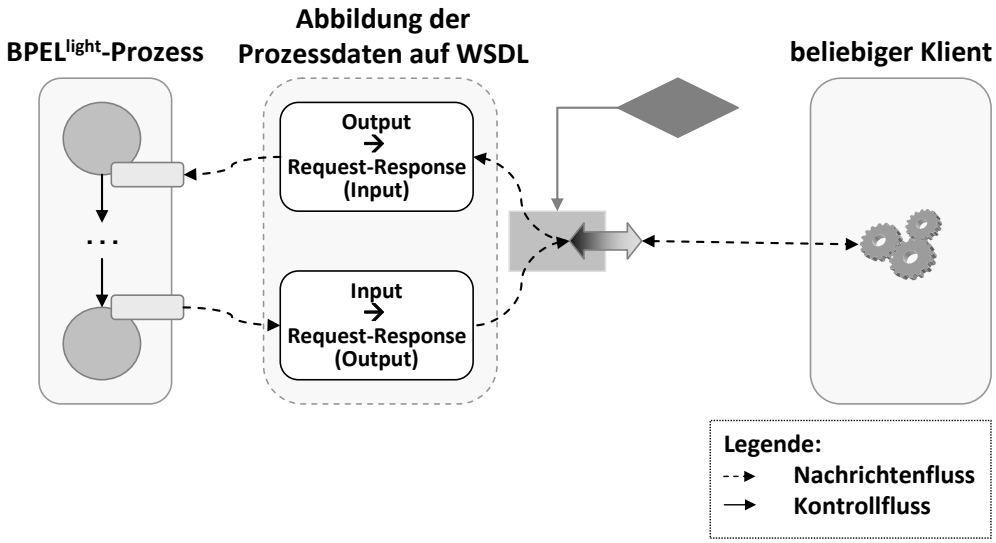


Abbildung 37: Aufruf eines kurzlebigen BPEL^{light}-Prozesses

Abbildung 37 zeigt schematisch die Zusammenhänge bei einem Aufruf eines kurzlebigen BPEL^{light}-Prozesses durch einen beliebigen, Web-Service-Technologie unterstützenden Klienten. Der Prozessausschnitt zeigt eine Konversation mit lediglich einem Partner, bei der zunächst eine Nachricht empfangen wird und im weiteren Verlauf eine Nachricht zurückgesendet wird. Von der Kommunikation des Prozesses mit anderen Geschäftpartnern vor, nach oder während der abgebildeten Konversation wird abstrahiert. Der Prozess stellt über einen Partnerlinktypen eine Operation des Typs Request-Response zur Verfügung. Die vom Prozess angebotenen bzw. benötigten Daten werden auf die über WSDL-Operation gesendeten bzw. empfangenen Nachrichten abgebildet. Der Input der Request-Response-Operation wird auf den Output der empfangenden Aktivität abgebildet und der Input der sendenden Aktivität auf den Output der Request-Response-Operation. Beim Eintreffen der Nachricht über die bereitgestellte WSDL-Operation kann dadurch die verarbeitende BPEL^{light}-Aktivität ermittelt

werden und beim Senden der Nachricht durch die BPEL[light]-Aktivität kann die entsprechende WSDL-Operation bestimmt werden. Die Abbildung des für die Konversation im BPEL[light]-Prozess definierten Partners auf den bereits angesprochenen Partnerlinktypen ist in Abbildung 37 der Übersicht halber nicht dargestellt.

Listing 26 zeigt eine konkrete Ausprägung des eben beschriebenen Szenarios.

```
<bpel:process name="performanceReviewHandlingProcess"
              targetNamespace="http://example.org"...>...
  <bpel:sequence>
    <bpel:extensionActivity>
      <bl:interactionActivity bpel:name="receiveEmployeeDetails"
                              conversation="performanceReviewConv">
        <bl:output messageLabel="employeeDetails"... />
        <bl:partner name="manager".../>
      </bl:interactionActivity>
    </bpel:extensionActivity>...
    <bpel:extensionActivity>
      <bl:interactionActivity bpel:name="sendPerformanceReview"
                              conversation="performanceReviewConv"...>
        <bl:input messageLabel="performanceReview"... />
        <bl:partner name="manager".../>
      </bl:interactionActivity>
    </bpel:extensionActivity>...
  </bpel:sequence>...
</bpel:process>

<blDeployment xmlns:ns1="http://example.org"
              processName="http://...#performanceReviewHandlingProcess"...>
  ...
  <conversationMapping conversationName="performanceReviewConv">
    <partnerMapping partnerName="manager"
                    partnerLinkTypeName="ns1:hrmPLT"
                    myRole="humanResourceManagement"/>
    <messageMapping messageLabel="employeeDetails"
                    operationName="getPerformanceReview"/>
    <messageMapping messageLabel="perfomanceReview"
                    operationName="getPerformanceReview"/>
  </conversationMapping>
</blDeployment>
```

Listing 26: Prozessbeschreibung und Deployment Deskriptor für den kurzlebigen, synchron aufrufbaren BPEL[light]-Prozess *„performanceReviewHandlingProcess“*

Der Prozess „performanceReviewHandlingProcess“ nimmt zunächst im Rahmen der Konversation „performanceReviewConv“ in der Aktivität

„receiveEmployeeDetails“ die Nachricht „employeeDetails“ vom Kommunikationspartner „manager“ entgegen. In der später ausgeführten Aktivität „sendPerformanceReview“ wird im Rahmen der gleichen Konversation dem Partner „manager“ die Nachricht „performanceReview“ übermittelt. In dem darunter dargestellten Deployment Deskriptor wird diese Prozesslogik auf die vom Prozess angebotene Operation „getPerformanceReview“ abgebildet: Dem Partner „manager“ in der Konversation „performanceReviewConv“ ist der Partnerlinktyp „hrmPLT“ zugeordnet. Da lediglich der Prozess vom Dienstnutzer aufgerufen wird, ist nur eine Rolle – die Rolle „humanResourceManagement“ – des Partnerlinktypen definiert. Sie wird über das Attribut `myRole` dem Prozess zugeordnet. Schließlich werden die Nachrichten „employeeDetails“ und „performanceReview“ der vom Prozess angebotenen Operation „getPerformanceReview“ zugeordnet. Die zugehörige WSDL-Beschreibung befindet sich in Anhang A.

(ii) Aufruf eines synchronen Dienstes

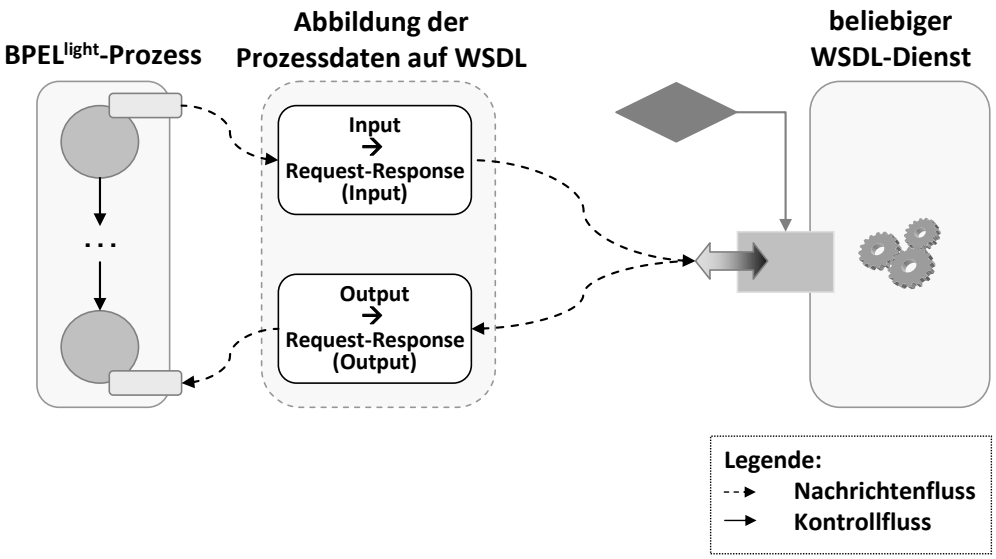


Abbildung 38: Aufruf eines synchronen WSDL-Dienstes

Abbildung 38 zeigt schematisch die Zusammenhänge bei einem Aufruf eines synchronen Web-Services durch einen BPEL[light]-Prozess. Der Prozessausschnitt zeigt eine Konversation mit lediglich einem Partner, bei der zunächst eine Nachricht gesendet wird und im weiteren Verlauf eine

Nachricht empfangen wird. Von der Kommunikation des Prozesses mit anderen Geschäftpartnern vor, nach oder während der abgebildeten Konversation wird wie im vorher bereits gezeigten Szenario abstrahiert. Der Prozess nutzt über einen Partnerlinktypen eine Operation des Typs Request-Response des WSDL-Dienstes.

Listing 27 zeigt eine konkrete Ausprägung des eben beschriebenen Szenarios.

```
<bpel:process name="performanceReviewRequestProcess"
              targetNamespace="http://example.org"...>...
  <bpel:sequence>
    <bpel:extensionActivity>
      <bl:interactionActivity bpel:name="sendEmployeeDetails"
                              conversation="performanceReviewConv">
        <bl:input messageLabel="employeeDetails"... />
        <bl:partner name="hrm" .../>
      </bl:interactionActivity>
    </bpel:extensionActivity>...
    <bpel:extensionActivity>
      <bl:interactionActivity bpel:name="receivePerformanceReview"
                              conversation="performanceReviewConv" ...>
        <bl:output messageLabel="performanceReview"... />
        <bl:partner name="hrm" .../>
      </bl:interactionActivity>
    </bpel:extensionActivity>...
  </bpel:sequence>...
</bpel:process>

<blDeployment xmlns:ns1="http://example.org"
              processName="http://...#performanceReviewRequestProcess"...>
  ...
  <conversationMapping conversationName="performanceReviewConv">
    <partnerMapping partnerName="hmr"
                    partnerLinkTypeName="ns1:hrmPLT"
                    partnerRole="humanResourceManagement"/>
    <messageMapping messageLabel="employeeDetails"
                    operationName="getPerformanceReview"/>
    <messageMapping messageLabel="performanceReview"
                    operationName="getPerformanceReview"/>
  </conversationMapping>
</blDeployment>
```

Listing 27: Prozessbeschreibung und Deployment Deskriptor für den synchron aufrufenden BPEL[light]-Prozess *„performanceReviewRequestProcess“*

Der Prozess „performanceReviewRequestProcess“ übermittelt zunächst im Rahmen der Konversation „performanceReviewConv“ dem Kommunikationspartner „hrm“ die Nachricht „employeeDetails“ durch die

Aktivität „sendEmployeeDetails“. In der später ausgeführten Aktivität „receivePerformanceReview“ wird im Rahmen der gleichen Konversation vom Partner „hrm“ die Nachricht „performanceReview“ empfangen. In dem darunter dargestellten Deployment Deskriptor wird diese Prozesslogik auf die vom Partnerdienst angebotene Operation „getPerformanceReview“ abgebildet: Dem Partner „hrm“ in der Konversation „performanceReviewConv“ wird der Partnerlinktyp „hrmPLT“ zugeordnet. Da lediglich der Dienst vom Prozess aufgerufen wird, ist wie im vorangegangenen Szenario nur eine Rolle – die Rolle „humanResourceManagement“ – des Partnerlinktypen definiert. Sie wird über das Attribut `partnerRole` dem Partnerdienst zugeordnet. Schließlich werden die Nachrichten „employeeDetails“ und „performanceReview“ der vom Dienst angebotenen Operation „getPerformanceReview“ zugeordnet. Die zugehörige WSDL-Datei befindet sich im Anhang A.

(iii) Aufruf eines langlebigen BPEL^{light}-Prozesses

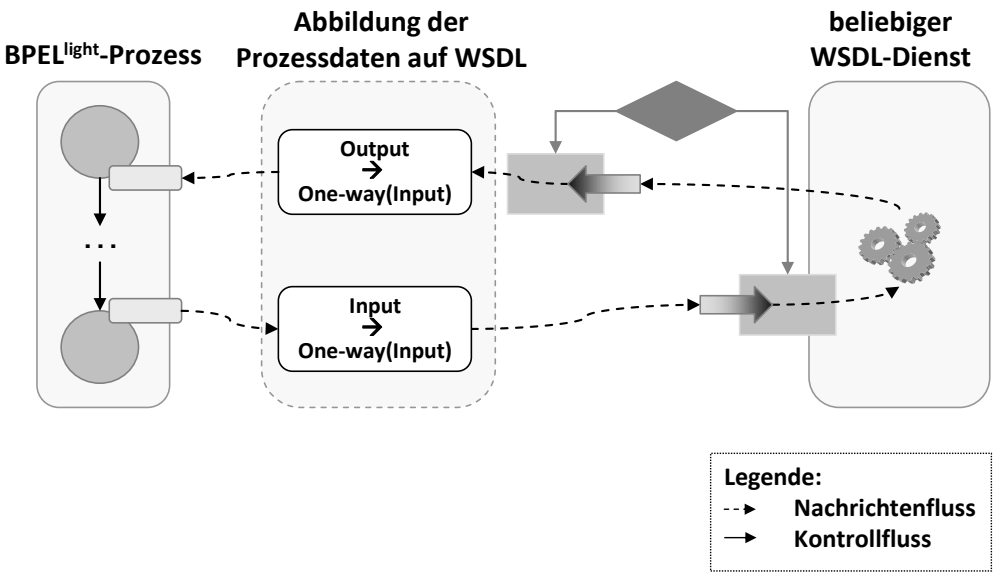


Abbildung 39: Aufruf eines langlebigen BPEL^{light}-Prozesses

Abbildung 39 zeigt schematisch die Zusammenhänge bei einem asynchronen Aufruf eines langlebigen BPEL^{light}-Prozess durch einen beliebigen, Web-Service-Technologie unterstützenden Klienten. Der Prozessausschnitt ist identisch zu dem in Abbildung 37 abgebildeten Prozessausschnitt: Er zeigt eine Konversation mit lediglich einem Partner,

bei der zunächst eine Nachricht empfangen wird und im weiteren Verlauf eine Nachricht zurückgesendet wird. Von der Kommunikation des Prozesses mit anderen Geschäftpartnern vor, nach oder während der abgebildeten Konversation wird wie in den bereits vorher gezeigten Szenarien abstrahiert. Der Prozess bietet über einen Partnerlinktypen eine Operation des Typs One-way an und nutzt anschließend eine über denselben Partnerlinktypen angebotenen Operation des Typs One-way des WSDL-Dienstes.

```
<bpel:process name="bookingHandlingProcess"
              targetNamespace="http://example.org"...>...
  <bpel:sequence>
    <bpel:extensionActivity>
      <bl:interactionActivity bpel:name="receiveTripDetails"
                              conversation="bookTripConv">
        <bl:output messageLabel="tripDetails"... />
        <bl:partner name="customer".../>
      </bl:interactionActivity>
    </bpel:extensionActivity>...
    <bpel:extensionActivity>
      <bl:interactionActivity bpel:name="sendBooking"
                              conversation="bookTripConv"...>
        <bl:input messageLabel="booking"... />
        <bl:partner name="customer".../>
      </bl:interactionActivity>
    </bpel:extensionActivity>...
  </bpel:sequence>...
</bpel:process>

<blDeployment xmlns:ns1="http://example.org"
              processName="http://example.org#bookingHandlingProcess"...>
  ...
  <conversationMapping conversationName="bookTripConv">
    <partnerMapping partnerName="customer"
                    partnerLinkTypeName="ns1:travelAgentPLT"
                    myRole="travelAgent"
                    partnerRole="customer"/>
    <messageMapping messageLabel="tripDetails"
                    operationName="bookTrip"
                    role="myRole"/>
    <messageMapping messageLabel="booking"
                    operationName="receiveBooking"
                    role="partnerRole"/>
  </conversationMapping>
</blDeployment>
```

Listing 28: Prozessbeschreibung und Deployment Deskriptor für den asynchron aufrufbaren BPEL[light]-Prozess *„bookingHandlingProcess“*

Listing 28 zeigt eine konkrete Ausprägung des eben beschriebenen Szenarios. Der Prozess „bookingHandlingProcess“ nimmt zunächst im Rahmen der Konversation „bookTripConv“ in der Aktivität „receiveTripDetails“ die Nachricht „tripDetails“ vom Kommunikationspartner „customer“ entgegen. In der später ausgeführten Aktivität „sendBooking“ wird im Rahmen der gleichen Konversation dem Partner „customer“ die Nachricht „booking“ übermittelt. Der darunter dargestellte Deployment Deskriptor bildet den Prozess auf zwei Operationen des Typs One-way ab, von denen eine vom Prozess selber und die andere vom Kommunikationspartner des Prozesses angeboten wird. Dem Partner „customer“ ist in der Konversation „bookTripConv“ der Partnerlinktyp „travelAgentPLT“ zugeordnet. Da in diesem Szenario beide Kommunikationspartner Operationen anbieten um sich gegenseitig aufzurufen, sind zwei Rollen des Partnerlinktypen definiert, die Rollen „travelAgent“ und „customer“. Die Rolle „travelAgent“ ist über das Attribut `myRole` dem Prozess zugeordnet. Die Rolle „customer“ ist über das Attribut `partnerRole` dem Kommunikationspartner, d.h. dem Klienten, zugewiesen. Schließlich wird die Nachricht „tripDetails“ auf die vom Prozess angebotenen Operation „bookTrip“ und die Nachricht „booking“ auf die vom Klienten angebotenen Operation „receiveBooking“ abgebildet. Die zugehörigen WSDL-Beschreibungen befindet sich in Anhang B und C.

(iv) Aufruf eines asynchronen Dienstes

Abbildung 40 zeigt schematisch die Zusammenhänge bei einem Aufruf eines asynchronen Dienstes durch einen BPEL^{light}-Prozess. Der Prozessausschnitt ist identisch zu dem in Abbildung 38 abgebildeten Prozessausschnitt: Er zeigt eine Konversation mit lediglich einem Partner, bei der zunächst eine Nachricht gesendet wird und im weiteren Verlauf eine Nachricht empfangen wird. Von der Kommunikation des Prozesses mit anderen Geschäftpartnern vor, nach oder während der abgebildeten Konversation wird wie in den bereits vorher gezeigten Szenarien abstrahiert. Der Prozess nutzt über einen Partnerlinktypen eine Operation des Typs One-way des WSDL-Dienstes und bietet eine Operation des Typs One-way über denselben Partnerlinktypen an um anschließend vom Dienst aufgerufen werden zu können.

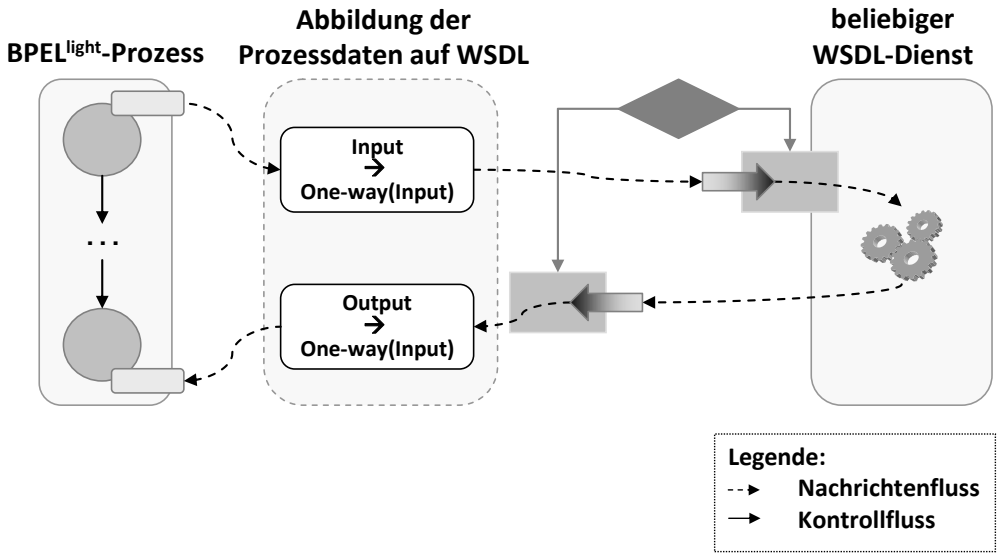


Abbildung 40: Aufruf eines asynchronen WSDL-Dienstes

Listing 29 zeigt eine konkrete Ausprägung des eben beschriebenen Szenarios. Der Prozess „bookingProcess“ übermittelt zunächst im Rahmen der Konversation „bookTripConv“ dem Kommunikationspartner „travelAgent“ die Nachricht „tripDetails“ durch die Aktivität „sendTripDetails“. In der später ausgeführten Aktivität „receiveBooking“ wird im Rahmen der gleichen Konversation vom Partner „travelAgent“ die Nachricht „booking“ empfangen. Der darunter dargestellte Deployment Deskriptor bildet den Prozess auf zwei Operationen des Typs One-way ab, von denen eine vom Prozess selber und die andere vom Kommunikationspartner des Prozesses angeboten wird. Dem Partner „travelAgent“ ist in der Konversation „bookTripConv“ der Partnerlinktyp „travelAgentPLT“ zugeordnet. Da in diesem Szenario beide Kommunikationspartner Operationen anbieten um sich gegenseitig aufzurufen, sind zwei Rollen des Partnerlinktypen definiert, die Rollen „travelAgent“ und „customer“. Die Rolle „customer“ ist über das Attribut `myRole` dem Prozess zugeordnet. Die Rolle „travelAgent“ ist über das Attribut `partnerRole` dem Kommunikationspartner, d.h. dem Dienst, zugewiesen. Schließlich wird die Nachricht „tripDetails“ auf die vom Dienst angebotene Operation „bookTrip“ und die Nachricht „booking“ auf die vom Prozess angebotene Operation „receiveBooking“ abgebildet. Die zugehörigen WSDL-Dateien befinden sich in Anhang B und C.

```
<bpel:process name="bookingProcess"
              targetNamespace="http://example.org"...>...
  <bpel:sequence>
    <bpel:extensionActivity>
      <bl:interactionActivity bpel:name="sendTripDetails"
                              conversation="bookTripConv">
        <bl:output messageLabel="tripDetails"... />
        <bl:partner name="travelAgent" .../>
      </bl:interactionActivity>
    </bpel:extensionActivity>...
    <bpel:extensionActivity>
      <bl:interactionActivity bpel:name="receiveBooking"
                              conversation="bookTripConv" ...>
        <bl:output messageLabel="booking"... />
        <bl:partner name="travelAgent" .../>
      </bl:interactionActivity>
    </bpel:extensionActivity>...
  </bpel:sequence>...
</bpel:process>

<blDeployment xmlns:ns1="http://example.org"
              processName="http://example.org#bookingProcess"...>
  ...
  <conversationMapping conversationName="bookTripConv">
    <partnerMapping partnerName="travelAgent"
                    partnerLinkTypeName="ns1:travelAgentPLT"
                    myRole="customer"
                    partnerRole="travelAgent"/>
    <messageMapping messageLabel="tripDetails"
                    operationName="bookTrip"
                    role="partnerRole"/>
    <messageMapping messageLabel="booking"
                    operationName="receiveBooking"
                    role="myRole"/>
  </conversationMapping>
</blDeployment>
```

Listing 29: Prozessbeschreibung und Deployment Deskriptor für den asynchron aufrufenden BPEL[light]-Prozess *„bookingProcess“*

Dadurch dass der Nachrichtenaustausch einer Konversation auf einen oder mehrere Partnerlinktypen abbildbar ist und sowohl der Prozess als auch seine Kommunikationspartner One-way- und Request-Response-Operationen anbieten können, ist es möglich, einen beliebig komplexen mit BPEL[light] definierten Nachrichtenaustausch über Web-Service-Technologie zu realisieren.

4.3 Beispiel: BPEL^light-Prozess als Klient eines mit WSDL beschriebenen Onlineversandhandeldienstes

Um die Eignung von BPEL^light zur Definition von ausführbaren Geschäftsprozessen, deren Aktivitäten durch Web-Services implementiert werden, zu demonstrieren, wird in diesem Abschnitt gezeigt, wie der in Abschnitt 3.5 präsentierte und in Listing 24 dargestellte BPEL^light-Prozess einen WSDL-Onlineversandhandeldienst verwenden kann, um einen Bestellvorgang vorzunehmen.

Abbildung 41 zeigt eine schematische Darstellung des Szenarios. Der über die Konversation beschriebene Nachrichtenaustausch zur Bestellung ist bilateral, d.h. es ist außer dem Prozess nur ein weiterer Geschäftspartner involviert. Aus diesem Grund kann die komplette Konversation über einen Partnerlinktypen, den „salesPLT“, abgewickelt werden. Die einzelnen Eingabe- und Ausgabeparameter des Prozesses werden dementsprechend auf Operationen des Partnerlinktypen „salesPLT“ abgebildet.

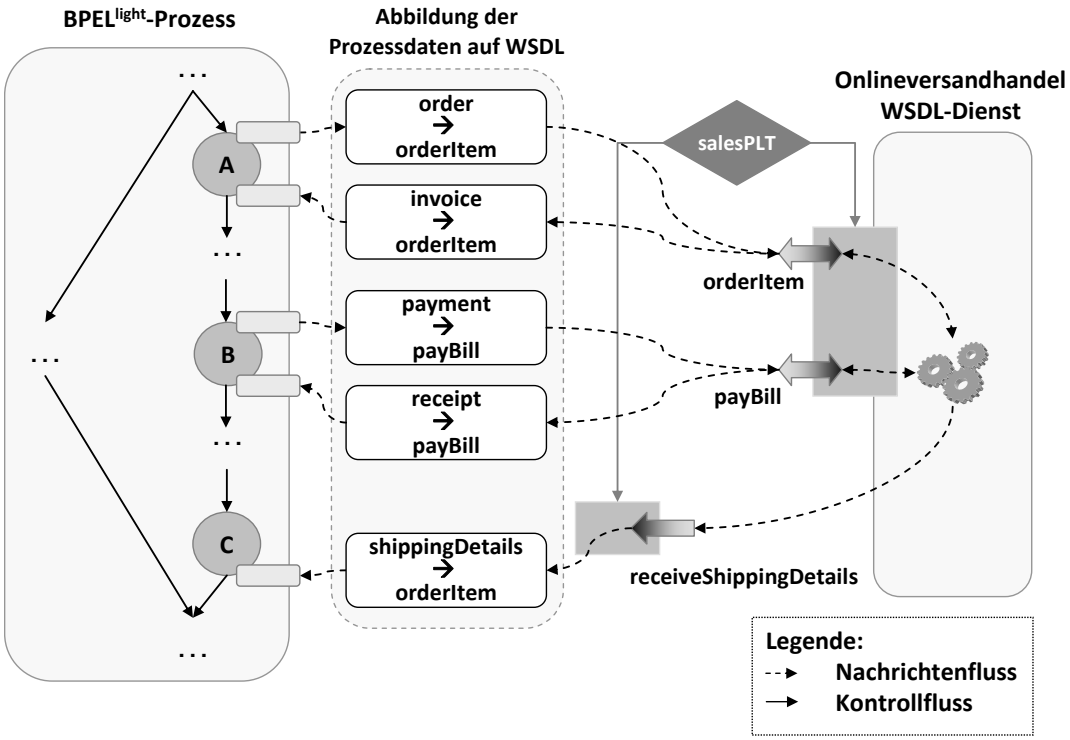


Abbildung 41: BPEL^light-Prozess als Klient eines mit WSDL beschriebenen Onlineversandhandeldienstes

```
<blDeployment xmlns:ns1="http://example.org"
              processName="http://example.org#buyProcess"...>
  <partnerInstances partnerName="seller">
    <partnerInstance id="1" name="myOnlineStore">
      <sref:service-Ref conversationName="salesConv" ...>
        ...
      </sref:serviceRef>
    </partnerInstance>
  </partnerInstances>
  <conversationMapping conversationName="salesConv">
    <partnerMapping partnerName="seller"
                    partnerLinkTypeName="ns1:salesPLT"
                    myRole="buyer"
                    partnerRole="seller"/>
    <messageMapping messageLabel="order"
                    operationName="orderItem"
                    role="partnerRole"/>
    <messageMapping messageLabel="invoice"
                    operationName="orderItem"
                    role="partnerRole"/>
    <messageMapping messageLabel="payment"
                    operationName="payBill"
                    role="partnerRole"/>
    <messageMapping messageLabel="receipt"
                    operationName="payBill"
                    role="partnerRole"/>
    <messageMapping messageLabel="shippingDetails"
                    operationName="receiveShippingDetails"
                    role="myRole"/>
  </conversationMapping>
</blDeployment>
```

Listing 30: Deployment Deskriptor für einen BPEL[light]-Prozess als Klient eines durch WSDL beschriebenen Onlineversandhandeldienstes

Die konkrete Umsetzung der Verknüpfung des Prozesses mit der Kommunikationsinfrastruktur mittels Deployment Deskriptor ist in Listing 30 dargestellt. Das Attribut `processName` des `<blDeployment>`-Elements referenziert den Prozess mittels der den Prozess identifizierenden URI Referenz: „http://example.org#buyProcess“. Die Instanz „myOnlineStore“ wird für den Partnertyp „seller“ über das Element `<partnerInstance>` mit `id='1'` definiert. Die Abbildung des Partners „seller“ in der Konversation „salesConv“ auf den PartnerLinkTypen „salesPLT“ findet über entsprechende `<conversationMapping>` und `<partnerMapping>`-Elemente statt. Schließlich werden die einzelnen Nachrichten mittels entsprechender `<messageMapping>`-Elemente auf Operationen abgebildet: die Nachrichten „order“ und „invoice“ auf die Operation „orderItem“, die Nachrichten

„payment“ und „receipt“ auf die Operation „payBill“ und die Nachricht „shippingDetails“ auf die Operation „receiveShippingDetails“.

4.4 Methodologie: Transformationen zwischen WS-BPEL und BPEL^{light} mit Web-Services als Aktivitätsimplementierungen

In Kapitel 3.6 wurde bereits gezeigt, wie WS-BPEL-Prozessmodelle in BPEL^{light}-Prozessmodelle überführt werden können, und wie aus BPEL^{light}-Prozessmodellen wieder WS-BPEL-Prozessmodelle gewonnen werden können. Bei der beschriebenen Transformation wurde jedoch von den Web-Service spezifischen Informationen in einem WS-BPEL-Prozess kein Gebrauch gemacht. Es wurden lediglich die für den Kontroll- und Datenfluss relevanten Informationen verwendet. Sollen WS-BPEL-Prozessmodelle jedoch in BPEL^{light}-Prozessmodelle transformiert werden, deren Aktivitäten von Web-Services implementiert werden, so können die Web-Service spezifischen Informationen der WS-BPEL-Prozessmodelle dazu verwendet werden, die Abbildung der BPEL^{light}-Prozessbeschreibung auf die Web-Service-Technologie zu spezifizieren. Diese Informationen können dann auch wieder verwendet werden, um aus BPEL^{light}-Prozessmodellenm, deren Aktivitäten durch Web Services implementiert werden, WS-BPEL-Prozessmodelle zu gewinnen.

4.4.1 Von WS-BPEL nach BPEL^{light} mit Web-Services als Aktivitätsimplementierungen

Die in Abschnitt 4.1 präsentierte Abbildung ordnet jedem Partner in einer Konversation einen Partnerlinktypen zu. Außerdem werden die Nachrichten von oder an den Partner auf Operationen des Partnerlinktypen abgebildet. Durch die in Abschnitt 3.6 präsentierte Transformation von WS-BPEL zu BPEL^{light} entstehen ohne menschliches Einwirken pro Partnerlink ein Partner und mehrere bilaterale Konversationen bestehend aus jeweils maximal zwei Nachrichten. Für jede dieser Konversationen kann dem Partner der entsprechende Partnerlinktyp zugeordnet werden. Die maximal zwei Nachrichten werden auf die Operation, die während der Transformation als Grundlage für die Definition der zugehörigen

Konversation dient, abgebildet. Dies ist in Abbildung 42 am Beispiel der Transformation der `<bpel:invoke>`-Aktivität zu einer `<bl:interactionActivity>`, die einen Web-Service über die gleiche Operation aufruft, dargestellt.

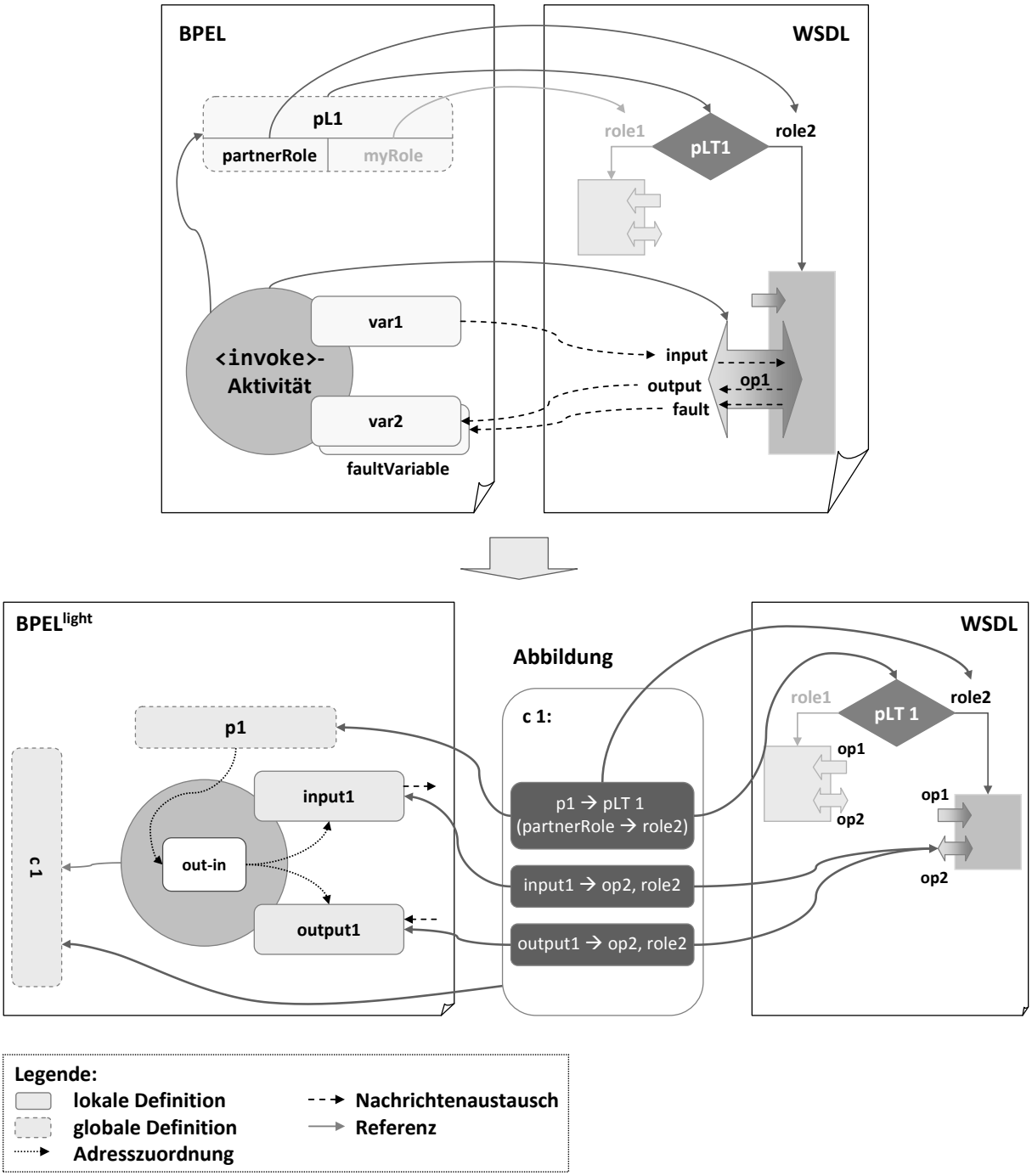


Abbildung 42: Transformation einer `<bpel:invoke>`-Aktivität zu einer `<bl:interactionActivity>`, die einen Web-Service über die gleiche Operation aufruft

Werden durch menschliche Interaktionen während der Transformationen mehrere Partnerlinks zu einem Partner zusammengefasst bzw. komplexe Konversationen eingeführt, so ist die Abbildung entsprechend anzupassen.

4.4.2 Von BPEL^{light} mit Web-Services als Aktivitätsimplementierung zu WS-BPEL

Soll ein aus WS-BPEL automatisch generierter BPEL^{light}-Prozess mit Web-Services als Aktivitätsimplementierung zurücktransformiert werden, so ist dies trivial: Partner sind in allen Konversationen jeweils auf den gleichen Partnerlinktypen abgebildet, der dann dazu verwendet wird, einen Partnerlink zu definieren. Die wiederhergestellten BPEL-Aktivitäten werden durch die Operation definiert, die für die entsprechenden BPEL^{light}-Aktivitäten in der Abbildung spezifiziert sind.

Schwieriger gestaltet sich die Transformation, wenn Partnerlinks zusammengefasst wurden bzw. der BPEL^{light}-Prozess von Grund auf neu modelliert wurde. In diesem Fall muss für jeden Partnerlinktypen, auf den ein Partner in unterschiedlichen Konversationen abgebildet wird, ein Partnerlink definiert werden. Die BPEL-Aktivitäten verwenden wie vorher bereits beschrieben die Operation, die für die entsprechenden BPEL^{light}-Aktivitäten in der Abbildung spezifiziert sind.

4.5 Zusammenfassung

In diesem Kapitel wurde gezeigt, dass Web-Services verwendet werden können, um Aktivitäten von BPEL^{light}-Prozessen zu implementieren und um die Funktionalität des Prozesses als Dienst zur Verfügung zu stellen.

WS-BPEL hängt direkt von WSDL ab. Trotz der Einschränkungen die es für WSDL gibt, ist durch die WSDL-Erweiterung Partnerlinktyp sowohl synchrone als auch asynchrone Kommunikation möglich. Dadurch sind zum einen WS-BPEL-Prozesse von Klienten sowohl synchron als auch asynchron aufrufbar und zum anderen können Dienste vom Prozess sowohl synchron als auch asynchron aufgerufen werden. Wie in diesem Kapitel gezeigt sind alle vier Interaktionsszenarien auch mit BPEL^{light} durch eine geeignete Abbildung seines Interaktionsmodells auf die Web-Service-

Technologie möglich. Bezüglich der Kommunikation ist BPEL^{light} mit Web-Services also gleichwertig zu WS-BPEL. Bezüglich der Flexibilität ist BPEL^{light} mit Web-Services WS-BPEL vorzuziehen, da bei einem Wechsel zu einer Aktivitätsimplementierung, die eine andere Schnittstelle implementiert, lediglich die Abbildungsvorschrift von Prozesslogik und Schnittstellenspezifikation modifiziert werden muss und nicht das Prozessmodell selber. Die Änderung kann entweder durch ein erneutes Deployment oder gar während der Laufzeit vorgenommen werden. Die Migration von WS-BPEL zu BPEL^{light} kann durch eine einfache Transformation vorgenommen werden.

Teile der präsentierten Ergebnisse finden sich auch in der bereits veröffentlichten Publikation [NLKL07b].

KAPITEL 5

BPEL^{LIGHT} MIT (SEMANTISCHEN) WEB-SERVICES – BPEL4SWS

Im vorausgegangenen Kapitel wurde gezeigt, wie BPEL^{light} in einer Web-Service-Umgebung eingesetzt werden kann. Durch die Entkopplung der Prozesslogik und der Schnittstellenspezifikation in WSDL kann der Wechsel zu einer Aktivitätsimplementierung, die eine andere Schnittstelle implementiert, mit weniger Aufwand und schneller vonstatten gehen. Es muss lediglich die Abbildungsvorschrift der Prozesslogik auf die Kommunikationsinfrastruktur geändert werden. Das Auffinden von Diensten, welche die gewünschte Funktionalität implementieren ist jedoch immer noch aufwendig. Maschinen können Dienste anhand unterschiedlicher WSDL-Beschreibungen nicht als semantisch äquivalent erkennen, d.h. die maschinenbasierte Suche geeigneter Dienste ist auf jene Dienste eingeschränkt, die den gleichen Porttypen implementieren. Soll die Suche nach geeigneten Diensten auch auf Dienste, die einen anderen Porttypen implementieren, ausgeweitet werden, so bedarf sie menschlicher Unterstützung zur Interpretation der WSDL-Beschreibungen.

Um in einem weiteren Schritt das Auffinden von geeigneten Aktivitätsimplementierungen zu vereinfachen, kommen Konzepte der Semantischen Web-Services (SWS) zum Einsatz. Sie bieten nicht nur wie Web-Services eine Abstrahierung von der Implementierung eines Dienstes, sondern sie machen zudem keinerlei Aussage über die Realisierung der Kommunikation zwischen Dienstnutzer und Dienst, also z.B. über das Transportprotokoll oder das Format von auszutauschenden Nachrichten. Stattdessen konzentrieren sie sich darauf, die Bedeutung eines Dienstaufrufs mittels Ontologien maschinenverständlich zu beschreiben. Das hat zur Folge, dass Dienstnutzer sich nicht an einen Porttypen binden

müssen, sondern ihre Anforderungen unabhängig von einer Dienstbeschreibung formulieren können. Die unabhängig voneinander beschriebenen Fähigkeiten von Diensten und Anforderungen von Klienten können über semantische Technologien miteinander abgeglichen werden. Genügt ein Dienst den Anforderungen eines Klienten, verwendet allerdings beispielsweise eine andere Ontologie zur Beschreibung der auszutauschenden Nachrichten bzw. ein anderes Datenformat, so können gegebenenfalls Daten über semantische Technologien automatisch von einer Ontologie zur anderen bzw. von einem Format in das andere überführt werden. Zur Kommunikation zwischen Klient und Dienst kann unter anderem die Web-Service-Technologie verwendet werden. Semantische Web-Services können somit als eine Integrationsschicht oberhalb der Web-Service-Technologie angesehen werden.

Im Folgenden wird gezeigt, wie Semantische Web-Services als Aktivitätsimplementierungen von BPEL^{light}-Prozessen eingesetzt werden können und wie sie dazu verwendet werden können, die Funktionalität eines Prozesses als Dienst zur Verfügung zu stellen. Die sich daraus ergebende Sprache trägt den Namen BPEL4SWS (BPEL für Semantische Web-Services). Wie bereits in Abschnitt 2.2 gezeigt, sind die SWS-Frameworks bislang auf einfache synchrone Dienstaufrufe ausgelegt. Darüber hinausgehende Kommunikation wird nicht unterstützt. In Abschnitt 5.1 wird deshalb zunächst beschrieben, wie das WSMO/X Framework dahingehend erweitert werden kann, um Web-Service-basierte, WS-I konforme, langlaufende, asynchrone Kommunikation zu unterstützen. Anschließend wird in Abschnitt 5.2 die konzeptionelle Abbildung von BPEL^{light} auf WSMO-Beschreibungen, d.h. WSMO-Web-Services und WSMO-Goals, vorgenommen und es werden Techniken zur Verfügung gestellt, die sowohl den Prozess selber als auch Semantische Web-Services zur Implementierung einzelner Aktivitäten des Prozesses auffindbar machen. Darüber hinaus werden Techniken vorgestellt, die es dem Prozess ermöglichen, über Web-Service-Technologie mit Geschäftspartnern zu kommunizieren. Es folgt die Umsetzung der vier Benchmarkszenarien in Abschnitt 5.3: Hier wird gezeigt, dass auch ein BPEL4SWS-Prozess, der seine Funktionalität über eine Semantische Web-

Service-Beschreibung anbietet, über herkömmliche Web-Service-Technologie aufgefunden und aufgerufen werden kann. Die Realisierung des in Abschnitt 2.1.2.3 vorgestellten Beispielsszenarios in BPEL4SWS wird in Abschnitt 5.4 präsentiert. Abschnitt 5.5 stellt weitere, semantische Erweiterungen zu BPEL vor, die davon Gebrauch machen, dass die Daten in einem BPEL4SWS-Prozess durch die semantische Annotation auch in Ontologieform vorliegen. Abschließend werden in Abschnitt 5.6 Transformationen von WS-BPEL zu BPEL4SWS und anders herum gezeigt und in Abschnitt 5.7 die wichtigsten Ergebnisse des Kapitels zusammengefasst.

5.1 Semantische Web-Services für lang laufende Prozesse

Beschreiben Semantische Web-Services einen Dienst der über Web-Service-Technologie kommuniziert, so müssen dem WS-I Basic Profile entsprechend folgende Einschränkungen beachtet werden: (i) Es dürfen nur Operationen des Typs One-way und Request-Response verwendet werden, (ii) das in WSDL 1.1 definierte SOAP-Binding muss verwendet werden und (iii) die Kommunikation muss über HTTP erfolgen.

Die im Kapitel 2.2 vorgestellten SWS-Frameworks OWL-S und WSMO verwenden bisher nur Operationen des Typs Request-Response zur Umsetzung eines Dienstaufrufs. Geschäftsprozesse sind jedoch im Allgemeinen lang laufende Prozesse, die sich über mehrere Stunden, Tage oder sogar Wochen erstrecken können. Wird ein Web-Service-basierter Dienst durch solch einen Prozess implementiert, so ist es nicht möglich diesen Dienst über eine Request-Response-Operation – also synchron – aufzurufen, da der Aufruf mit hoher Wahrscheinlichkeit einen Time-out auslöst und mit einer Fehlermeldung abbricht. Zur Verwirklichung Web-Service-basierter asynchroner Kommunikation ist es aufgrund der Vorgaben des WS-I Basic Profiles notwendig, die beiden Nachrichten der Request-Response-Operation auf zwei One-way-Operationen zu verteilen, von denen eine vom Dienst implementiert werden muss und eine vom Dienstnutzer als Call-back. Wie bereits in Abschnitt 2.2.1 beschrieben, umfasst OWL-S lediglich die semantische Beschreibung des Dienstes, jedoch nicht des Dienstnutzers. Der Umstand, dass der Dienstnutzer einen

Call-back anbietet, kann also nicht ausgedrückt werden. Damit ist es ausgeschlossen, mit OWL-S WS-I konforme asynchrone Kommunikation zwischen Dienst und Dienstnutzer zu beschreiben.

Mehr Potential bietet hier WSMO, da in WSMO sowohl die Anforderungen eines Dienstnutzers (WSMO-Goal) als auch die Fähigkeiten eines Dienstes (WSMO-Web-Service) separat beschrieben werden. Allerdings arden Goals in WSMO folgendermaßen beschrieben: „Goals can be descriptions of Web-Services that would potentially satisfy the user desires“[LPR05]. Das bedeutet, dass sie lediglich als Templates für Dienste betrachtet werden, die den Anforderungen des Dienstnutzers entsprechen. Um jedoch die Kommunikation zwischen durch WSMO beschriebenen Diensten und Dienstnutzern Web-Service-basiert und asynchron durchführen zu können, ist es notwendig, die Bedürfnisse eines Klienten aus dessen Sicht zu beschreiben. Das hat auf die Definition der Zielbeschreibung die folgenden Auswirkungen:

1) Die Goal-Capability beschreibt nicht etwa die Fähigkeiten eines Dienstes aus dessen Sicht, sondern vielmehr die Anforderungen an einen Dienst aus der Sicht des Dienstnutzers. Assumptions und Effects bleiben unverändert, da diese Bedingungen an den Zustand der Welt vor bzw. nach dem Dienstaufruf stellen, also die globale Sicht beschreiben. Die Bedeutung der Preconditions und Postconditions ändert sich jedoch, da eine beim Dienst eintreffende Nachricht vom Dienstnutzer gesendet wird, also aus dessen Sicht eine gesendete Nachricht ist und eine vom Dienst gesendete Nachricht vom Dienstnutzer empfangen wird, aus dessen Sicht also eine empfangene Nachricht darstellt.

2) Die Choreography in Goals beschreibt den vom Dienstnutzer beabsichtigten Nachrichtenaustausch aus der Sicht des Dienstnutzers. Werden vom Dienstnutzer WSDL-Operationen zum Empfangen (und Senden) von Nachrichten angeboten, so kann für die entsprechenden, unter den Schlüsselwörtern `in` (und `out`) spezifizierten, die Nachrichten repräsentierenden Konzepte ein Grounding auf diese Operationen spezifiziert werden.

3) Auch Zielbeschreibungen müssen, falls sie eigene Operationen anbieten, einen Endpunkt zur Verfügung stellen. Deshalb kann auch in einem WSMO-Goal über die nichtfunktionale Eigenschaft `wsml#endpointDescription` auf ein WSDL-`<service>`-Element verwiesen werden.

Mit dieser Definition der Choreography ist es nun möglich, die Sicht des Dienstes und des Dienstnutzers separat zu beschreiben und eine für Geschäftsprozesse typische lang laufende asynchrone Kommunikation über Web-Service-Technologie zwischen Dienst und Dienstnutzer unter Beachtung der vom WS-I getroffenen Einschränkungen zu realisieren. Das wird anhand von Abbildung 43 genauer erläutert.

Ein WSMO-Web-Service beschreibt zum einen seine Funktionalität in Form der Capability und zum anderen den Nachrichtenaustausch, der nötig ist, um seine Funktionalität abzurufen in der WSMO-Choreography. Der Nachrichtenaustausch wird dabei aus Sicht des Dienstes beschrieben. Ähnliches gilt für das WSMO-Goal: Die gewünschte Funktionalität wird anhand der Capability beschrieben und die WSMO-Choreography beschreibt aus der Sicht des Dienstnutzers welche Nachrichten ausgetauscht werden sollen. Sowohl WSMO-Goal als auch WSMO-Web-Service abstrahieren dabei von den verwendeten Kommunikationsmechanismen. Wie bereits angesprochen, ist eine Möglichkeit zur Realisierung der Kommunikation die Web-Service-Technologie. Wird sie zur Implementierung eines Dienstnutzers bzw. Dienstes herangezogen, so können Nachrichten lediglich auf Operationen des Typs One-way oder Request-Response abgebildet werden. Das hat zur Folge, dass auf beiden Seiten jeweils nur eine partielle Abbildung des Nachrichtenaustausches auf WSDL-Operationen stattfindet, da Anforderung und Dienst unabhängig voneinander beschrieben werden und sich daher nicht auf Operationen der jeweiligen Gegenüber beziehen können. Es sind also sowohl auf Seite des Dienstnutzers als auch des Dienstes manche Nachrichten auf die jeweils eigenen Operationen, den eigenen Porttypen, abgebildet. Jedoch wird anders als bei der im Rahmen der BPEL-Spezifikation definierten WSDL-Erweiterung Partnerlinktyp nicht vorgegeben, welche Porttypen der jeweilige Kommunikationspartner implementieren muss. Ein Paar aus WSMO-Goal und einem dazu passenden

WSMO-Web-Service kann daher als „dynamischer Partnerlinktyp“ betrachtet werden. Somit kann auch Web-Service-Technologie verwendet werden, um asynchrone Kommunikation zwischen Dienstnutzer und Dienst in WSMO zu realisieren.

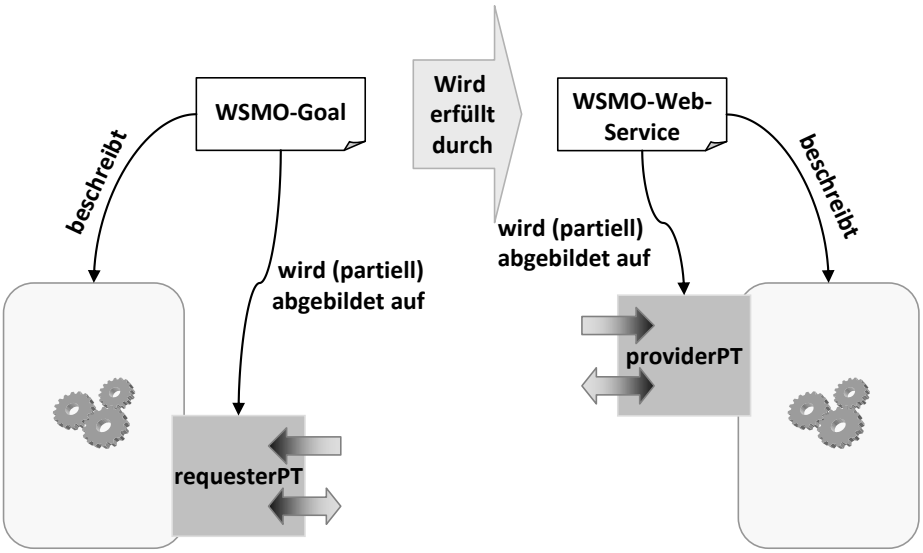


Abbildung 43: „Dynamischer Partnerlinktyp“ durch WSMO-Goal und -Web-Service

Listing 31 zeigt ein WSMO-Goal, das die Anforderungen des Prozesses an einen Onlineversandhandeldienst formuliert. Von der Beschreibung der Capability wird an dieser Stelle abgesehen. Die Choreography ist wie folgt definiert: Unter dem Schlüsselwort *out* werden die zu sendenden Nachrichten („order“ und „creditCard) beschrieben, unter dem Schlüsselwort *in* die zu empfangenden Nachrichten („invoice“, „receipt“ und „shippingDetails“). Unter dem Schlüsselwort *controlled* werden die Konzepte „2ndStep“, „3rdStep“ und „4thStep“ definiert, die nicht von außen manipuliert werden können. Sie werden dazu verwendet, die einzelnen Schritte der Choreography explizit zu modellieren. Unter dem Schlüsselwort *transitionRules* beschreiben Regeln schließlich die Reihenfolge, in der die Nachrichten ausgetauscht werden: Existiert in der Datenbasis der ASM weder eine Instanz des Konzepts „2ndStep“ noch eine Instanz der Konzepte „3rdStep“ oder „4thStep“, so wird eine Instanz des Konzepts „order“ und eine Instanz des Konzepts „2ndStep“ hinzugefügt.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace {oc _"http://www.example.org/ontologies/onlineShop#",
          ch _http://www.example.org/ontologies/onlineShopConsumer/
              choreography#,
          wsml _"http://www.wsmo.org/wsml/wsml-syntax#"}
goal oc#onlineshopClient
  nonFunctionalProperty
    wsml#endpointDescription hasValue _"http://example.org/onlineShopConsumer
                                       .wsdl#wsdl11.service(onlineShopConsumer)"
  endNonFunctionalProperty

  capability oc#capability

  interface oc#interface
    choreography
      stateSignature
        controlled
          ch#2ndStep
          ch#3rdStep
          ch#4thStep
        out
          oc#order
          oc#payment
        in
          oc#invoice
          oc#receipt
          oc#shippingDetails withGrounding _"http://example.org/onlineShopClient
                                            .wsdl#wsdl11.portTypeOperation.input
                                            (buyerPT/getShippingDetails)"
      transitionRules
        if (neg(_# memberOf ch#2ndStep or _# memberOf ch#3rdStep or
                _# memberOf ch#4thStep)) then
            add(_# memberOf oc#order and
            add(_# memberOf ch#2ndStep)
        if (?response memberOf oc#invoice and
              exists ? 2ndStep memberOf ch#2ndStep) then
            add(_# memberOf oc#payment) and
            delete(_# memberOf ch2ndStep) and
            add(_# memberOf ch#3rdStep)
        endIf
        if (?response memberOf oc#receipt and
              exists ?3rdStep memberOf ch#3rdStep) then
            delete(_# memberOf ch#3rdStep) and
            add(_# memberOf ch#4thStep)
        endIf
        if (?notification memberOf oc#shippingDetails and
              exists ?4thStep memberOf ch#4thStep) then
            delete(_# memberOf ch#4thStep)
        endIf
```

Listing 31: WSML-Beschreibung eines WSMO-Goals zur Bestellung bei einem Onlineversandhandeldienst

In diesem Zustand der ASM hat der Dienst bereits die erste Nachricht („order“) gesendet und ist bereit, eine Instanz des Konzepts „invoice“ zu

empfangen und daraufhin eine Instanz des Konzepts „creditCard“ zu verschicken. Um den nächsten Schritt zu ermöglichen wird die Instanz des Konzepts „2ndStep“ gelöscht und eine Instanz des Konzepts „3rdStep“ hinzugefügt. Nun kann die dritte Regel greifen. Wird eine Instanz des Konzepts „receipt“ empfangen, so wird die Instanz des Konzepts „3rdStep“ gelöscht und eine Instanz des Konzepts „4thStep“ hinzugefügt. In diesem Zustand der ASM ist der Dienst bereit die letzte Nachricht, eine Instanz des Konzepts „shippingDetails“, zu empfangen. Wird sie schließlich empfangen, so wird die Instanz des Konzepts „4thStep“ gelöscht. Der Nachrichtenaustausch ist beendet.

Für die ersten vier Nachrichten wird jeweils kein Grounding definiert. Lediglich für die abschließende Nachricht wird ein Grounding auf den Input der Operation „getShippingDetails“ definiert. Außerdem verweist die Zielbeschreibung mittels der nichtfunktionalen Eigenschaft `endpointDescription` auf das WSDL-`<service>`-Element „onlineshopClient“ und stellt damit einen oder mehrere Endpunkte zur Verfügung, an den die letzte Nachricht verschickt werden kann.Bislang unterstützt die Referenzimplementierung WSMX lediglich blockierende Aufrufe von Klienten, d.h. eine einkommende Nachricht gefolgt von einer Antwortnachricht [He08]. Folgt die Choreography des Klienten und des Dienstes allerdings einem komplexen Protokoll, so reicht ein einzelner blockierender Aufruf nicht aus. Stattdessen wird es notwendig, Klienten und Diensten die Möglichkeit zu bieten, über einen längeren Zeitraum miteinander zu kommunizieren. Hierzu muss eine WSMO-Implementierung die folgende Funktionalität bieten:

- Initialisierung einer Konversation zwischen Klient und Dienst auf Basis eines WSMO-Goals,
- Entgegennahme einer Nachricht zur Auslieferung beim jeweiligen Partner (entspricht einer WSDL-Operation des Typs One-way),
- Entgegennahme einer Nachricht zur Auslieferung beim Partner mit darauf folgender Zustellung der Antwortnachricht (entspricht einer WSDL-Operation des Typs Request-Response).

Diese können dann wie folgt verwendet werden: Zur Initialisierung schickt der Dienstnutzer ein WSMO-Goal an die WSMO-Implementierung. Diese gleicht das WSMO-Goal mit den ihr bekannten WSMO-Web-Services ab, wählt den geeignetsten Dienst aus, initialisiert sowohl die Choreography des WSMO-Goals als auch des WSMO-Web-Services und vergibt der dadurch instantiierten Konversation einen eindeutigen Bezeichner. Der Bezeichner wird dann dem Klienten zurückgeschickt. Der Klient kann nun mit Hilfe des eindeutigen Bezeichners seiner Choreography entsprechend Nachrichten an die WSMO-Implementierung übermitteln. Wird der Dienst das erste Mal aufgerufen, so wird auch ihm der Bezeichner übermittelt, z.B. durch entsprechende Felder im Kopf der Nachricht. Ist den Kommunikationspartnern der eindeutige Bezeichner bekannt, so können sie dem jeweiligen Partner über die WSMO-Implementierung Nachrichten zukommen lassen. Nachrichten, die in der eigenen Choreography nicht auf WSDL-Operationen abgebildet sind, werden mit der WSMO-Implementierung unter Zuhilfenahme der von ihr zur Verfügung gestellten Funktionalität ausgetauscht, d.h. der Klient bzw. Dienst ruft die WSMO-Implementierung auf. Nachrichten, die in der Choreography des Dienstes bzw. Klienten auf WSDL-Operationen abgebildet sind, werden zwischen WSMO-Implementierung und dem entsprechenden Dienst bzw. Klient über die angegebene WSDL-Operation ausgetauscht, d.h. die WSMO-Implementierung ruft den Dienst bzw. Klienten auf. Da die WSMO-Implementierung stellvertretend für den Klienten bzw. Dienst mit dem jeweiligen Kommunikationspartner interagiert, kann in ihr sowohl Daten- als auch Protokollmediation stattfinden.

Wird die Funktionalität der WSMO-Implementierung als Web-Service-Operationen angeboten, so ist eine langlaufende Interaktion zwischen WSMO-basierten Klienten und Diensten auf Basis der Web-Service-Technologie möglich, die dem WS-I Basic Profil entspricht.

Im Folgenden wird eine Ausführungsumgebung für Semantische Web-Services, d.h. auch für WSMO, als Semantischer Service Bus (SSB) bezeichnet, sofern sie die in diesem Abschnitt präsentierte Funktionalität bietet und mit Dienstnutzern und Diensten über Web-Service-Technologie kommuniziert.

5.2 Semantische Web-Services als Aktivitätsimplementierungen

BPEL^{light} definiert ein Interaktionsmodell, welches ermöglicht, sowohl mehrere Nachrichten durch eine Konversation zu logischen Verbünden zu gruppieren als auch für jede einzelne Nachricht den Kommunikationspartner anzugeben. Um die Funktionalität eines BPEL^{light}-Prozesses oder die Anforderungen eines BPEL^{light}-Prozesses an einen Dienst mittels einer WSMO-Beschreibung auszudrücken und somit den Nachrichtenaustausch des Prozesses durch eine Choreography eines WSMO-Web-Services oder WSMO-Goals zu beschreiben, ist zunächst eine weitergehende Analyse der Mächtigkeit der WSMO-Beschreibungen, im Besonderen der Choreography, notwendig. Dabei ist folgendes von Relevanz:

- In der WSMO-Choreography wird keine Aussage darüber gemacht, mit welchem Kommunikationspartner Nachrichten ausgetauscht werden. Es wird nur definiert, welche Daten dem Dienst zur Verfügung gestellt werden müssen, jedoch nicht von wem. Es ist also nicht möglich, eine multilaterale Kommunikation zu spezifizieren. In der Referenzimplementierung WSMX ist bisher zudem lediglich der blockierende Aufruf eines WSMO-Web-Services, der einer WSDL-Operation entspricht, durch einen Klienten implementiert.

- Der Begriff eines Geschäftspartners ist Teil des konzeptionellen Modells von WSMO. Ist für einen WSMO-Web-Service das Attribut `owner` spezifiziert, so kann man einem Dienst ansehen, von welcher Geschäftseinheit er implementiert wird. Allerdings bietet die Referenzimplementierung WSMX keine Funktionalität an, die von dieser Information Gebrauch macht. Es ist weder eine Ausführungssemantik zur Ermittlung eines Geschäftspartners, der mehrere Ziele erfüllen kann, definiert noch zur Erfüllung eines Ziels durch einen spezifizierten Geschäftspartner. Die Notwendigkeit definieren zu können, dass mehrere Ziele von einer Geschäftseinheit erfüllt werden müssen, wird am folgenden Beispiel deutlich: Eine Anwendung zur Buchung von Flügen möchte zunächst den Preis für einen Flug ermitteln und anschließend, falls der Anwender dem zustimmt, den Flug tatsächlich

buchen. Werden hierfür die beiden WSMO-Goals „getPrice“ und „bookFlight“ formuliert, so muss die Ausführungsumgebung sicherstellen, dass nicht etwa der Preis für einen Air Berlin Flug ermittelt wird, und dann ein Lufthansa Flug gebucht wird.

Abbildung von BPEL^{light}-Konversationen auf WSMO-Beschreibungen

Die Ausdrucksstärke von WSMO wirkt sich wie folgt auf den Einsatz in BPEL^{light}-Prozessen aus. Die WSMO-Choreography eignet sich zur Beschreibung einfacher bis komplexer bilateraler Interaktionen. Sie entspricht in BPEL^{light} von der Bedeutung her am ehesten der `<bl:conversation>`, auch wenn diese nicht nur die Definition bilateraler sondern auch multilateraler Interaktion unterstützt: Sowohl WSMO-Choreography als auch `<bl:conversation>` beschreiben den Nachrichtenaustausch zur Erlangung eines Ziels bzw. zur Bereitstellung einer bestimmten Funktionalität. Ihrer Bedeutung entsprechend unterscheidet man bei der `<bl:conversation>` zwischen Funktionalität *konsumierenden Konversationen* und Funktionalität *anbietenden Konversationen.* WSMO-Goals eignen sich dementsprechend zur Beschreibung konsumierender Konversationen und WSMO-Web-Services zur Beschreibung anbietender Konversationen. Dies ist in Abbildung 44 dargestellt. Der aus den Aktivitäten „A“ und „D“ bestehenden Konversation „conversationA“ wird ein WSMO-Web-Service zugeordnet, dessen Choreography den gleichen Nachrichtenaustausch spezifiziert, wie die BPEL^{light} Konversation. Der aus den Aktivitäten „B“ und „C“ bestehenden Konversation „conversationB“ ist hingegen ein WSMO-Goal zugeordnet.

Werden Konversationen von BPEL^{light}-Prozessen mit WSMO-Web-Services bzw. WSMO-Goals beschrieben, so müssen diese Konversationen derart modelliert sein, dass sie Funktionalität ausschließlich auf bilateraler Basis anbieten bzw. Anforderungen an Dienste stellen, die auf bilateraler Basis zu erfüllen sind.

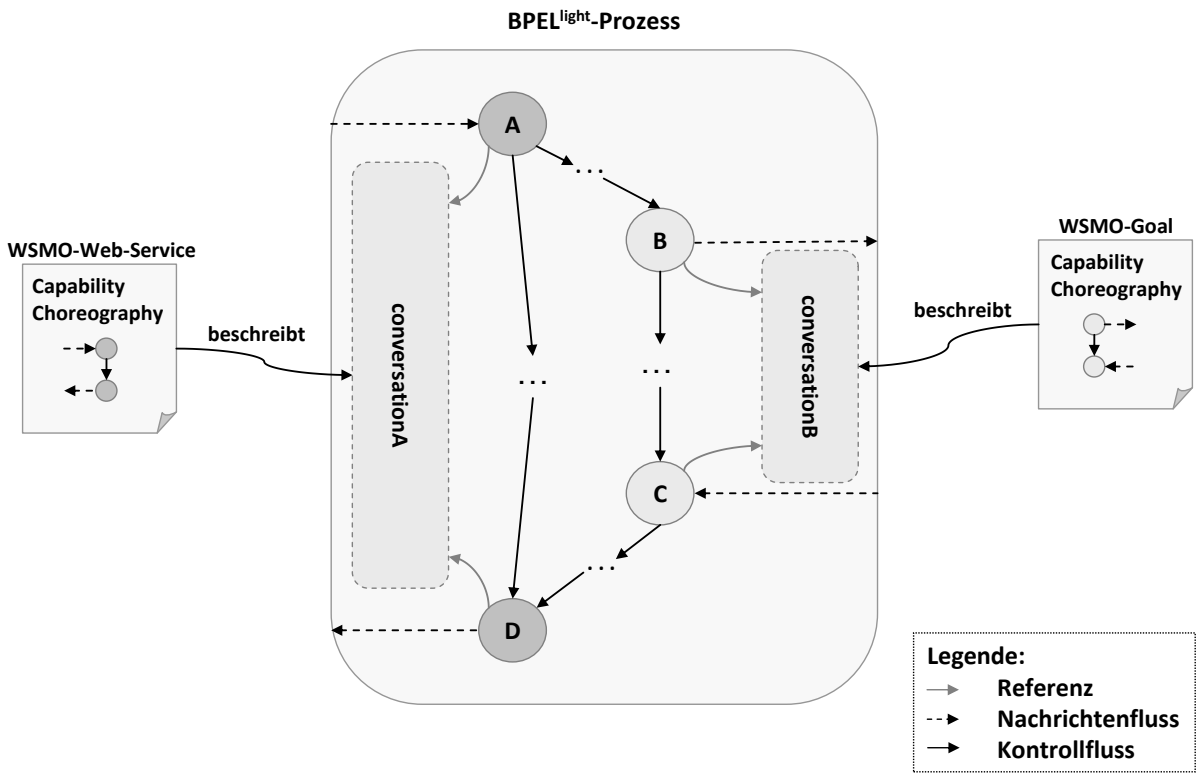


Abbildung 44: Mit WSMO-Konzepten annotierter BPEL[light]-Prozess

Durch die Zuordnung von BPEL[light]-Konversationen zu WSMO-Goals bzw. WSMO-Web-Services wird der Prozess nicht wie im vorigen Kapitel direkt auf die Web-Service-Technologie, d.h. einen statischen Partnerlink abgebildet. Stattdessen wird der in Abbildung 43 gezeigte über WSMO-Beschreibungen realisierte „dynamische Partnerlinktyp“ zur Kommunikation verwendet. Dabei wird jeweils nur die eigene Rolle des Partnerlinks inklusive der vom Prozess angebotenen Operationen angegeben. Die andere Rolle wird durch semantische Technologien dynamisch ermittelt.

Die Zuordnung von BPEL[light]-Konversationen zu WSMO-Goals bzw. WSMO-Web-Services kann wie auch die Abbildung von BPEL[light]-Aktivitäten auf Web-Service-Technologie durch unterschiedliche Techniken realisiert werden, die zu unterschiedlichen Zeitpunkten des Prozesslebenszyklus greifen. Auch hier beschränkt sich die Arbeit auf die Beschreibung der Annotation mittels Deployment Deskriptor. Diese wird in Listing 32 illustriert. Die Attribute `goal` und `service` innerhalb des `<conversationMapping>`-Elements ermöglichen es, auf semantische

Beschreibungen zu verweisen. Bei diesen semantischen Beschreibungen kann es sich um WSMO-Beschreibungen handeln. Prinzipiell ist aber auch die Verwendung anderer Semantischer Web-Service-Frameworks denkbar.

```
<blDeployment processName="anyURI"...>
  ...
  <conversationMapping conversationName="NCName"
                       goal="anyURI"?
                       service="anyURI"?/>...
</blDeployment>
```

Listing 32: Erweitertes Schema des Deployment Deskriptors zur Annotation von Prozessen mit semantischen Beschreibungen

Funktionalität eines Semantischen Service Busses zur Geschäftspartner-ermittlung

Zusätzlich zur Modellierung von Konversationen zur Erfüllung von Geschäftszielen ermöglicht das konzeptionelle Modell von BPEL^light auch zu definieren, mit welchen Geschäftspartnern welche Geschäftsziele erreicht werden sollen. Jede Nachricht wird in BPEL^light unabhängig von Konversationen einem Partner zugeordnet. Wird in mehreren unterschiedlichen konsumierenden Konversationen mit dem gleichen Partner kommuniziert, so müssen sich die durch diese Konversationen repräsentierten Geschäftsziele mit dem gleichen Partner realisieren lassen. Um dies bei der Wahl der geeigneten Dienste berücksichtigen zu können, muss eine WSMO-Implementierung um folgende Funktionalität erweitert werden:

- Ermittlung eines Geschäftspartners, der mehrere Ziele erfüllen kann,
- Initialisierung der Konversation zwischen dem Klienten und dem Dienst eines bestimmten Geschäftspartners auf Basis eines WSMO-Goals.

Web-Service-Technologie als Kommunikationsgrundlage für BPEL4SWS

Durch die Abbildung auf WSMO-Web-Services bzw. WSMO-Goals sind die entsprechenden Konversationen eines BPEL^light-Prozesses semantisch

beschrieben. Um jedoch letztendlich WS-I konforme Kommunikation über Web-Service-Technologie zu ermöglichen, müssen sowohl die WSMO-Beschreibungen als auch die BPEL^{light}-Prozessbeschreibung geeignet auf Web-Services abgebildet werden. Die Abbildung der WSMO-Beschreibungen wurde bereits in Abschnitt 5.1 diskutiert.

Für konsumierende Konversationen, die einem Goal zugeordnet sind, wird lediglich eine *partielle* Abbildung spezifiziert: Es wird spezifiziert, welche Aktivitäten des BPEL^{light}-Prozesses die in der WSMO-Beschreibung angegebenen Operationen implementieren, d.h. welchen BPEL^{light} Nachrichten die Inputs und Outputs der vom Prozess implementierten Operationen entsprechen. Alle Nachrichten, für die in der WSMO-Beschreibungen kein Grounding auf WSDL-Operationen angegeben ist, werden mittels der vom SSB zur Verfügung gestellten Funktionalität ausgetauscht (vergleiche Abschnitt 5.3(ii) und (iv)).

Für konsumierende Konversationen, die keinem Goal zugeordnet sind, und für anbietende Konversationen generell ist die Abbildung komplexer. In diesem Fall wird für jeden Partner pro Konversation ein Partnerlinktyp angegeben. Zusätzlich müssen auch, wie bereits in Abschnitt 4.1 beschrieben, alle zugehörigen Nachrichten des BPEL^{light}-Prozesses WSDL-Operationen zugeordnet werden.

Wird diese *vollständige* Abbildung für konsumierende Konversation spezifiziert, so wird, wie in Abschnitt 4.1 beschrieben, direkt zwischen Prozess und Dienst über Web-Service-Technologie kommuniziert. In diesem Fall darf die Konversation nicht mit einem Goal verknüpft sein.

Dadurch dass bei einer anbietenden Konversation auch die Partnerrolle des Prozesses spezifiziert ist, kann die Funktionalität eines BPEL4SWS-Prozesses, die als semantischer Web-Service bereitgestellt wird, auch über herkömmliche Web-Service-Technologie konsumiert werden (vergleiche Abschnitt 4.1 (i) und (iii)). Es ist nicht nur definiert, welche Aktivitäten die vom Prozess angebotenen Operationen implementieren, sondern auch welche Operationen des Partnerdienstes verwendet werden können, um ihn aufzurufen. Wird der Prozess über seine semantische Beschreibung aufgefunden, so ruft der Prozess den Kommunikationspartner im Fall

asynchroner Kommunikation nicht direkt unter Verwendung der in der Abbildung spezifizierten Operation auf, sondern verwendet stattdessen die Funktionalität des SSBs, um mit dem Partner Nachrichten auszutauschen.

Einen Prozess sowohl als Web-Service als auch als Semantischen Web-Service bereitzustellen ist von größter Bedeutung, da die meisten potentiellen Klienten eines durch einen Prozess implementierten Dienstes keine Semantischen Web-Service-Technologien unterstützen und daher nicht in der Lage wären, die Funktionalität eines BPEL4SWS-Prozesses zu nutzen, wenn dieser nur über Semantische Web-Service-Technologien verfügbar ist. Wären BPEL4SWS-Prozesse nicht gleichzeitig auch über herkömmliche Web-Service-Technologie nutzbar, so würde die Migration eines BPEL-Prozesses zu einem BPEL4SWS-Prozess zwar zu mehr Flexibilität führen, hätte aber die unerwünschte Begleiterscheinung, dass sich die Anzahl der möglichen Klienten nicht steigert, sondern verringert.

```
<blDeployment processName="anyURI"...>
  ...
  <conversationMapping conversationName="NCName"...>
    <partnerMapping partnerName="NCName"
                    partnerLinkTypeName="QName"?
                    myRole="NCName"?
                    partnerRole="NCName"?>*
    <messageMapping messageLabel="NCName"
                    operationName="NCName"
                    role="myRole/partnerRole"?
                    portTypeName="QName"?/>?
  </conversationMapping>
</blDeployment>
```

Listing 33: Erweitertes Schema des Deployment Deskriptors zur partiellen Abbildung von Prozessen auf Web-Service-Technologie

In Listing 33 ist eine Erweiterung des bereits in Listing 25 präsentierten Deployment Deskriptors dargestellt, der neben der vollständigen Abbildung auch die partielle Abbildung ermöglicht. Für die partielle Abbildung muss kein `<partnerMapping>`-Element spezifiziert werden, d.h. es muss kein Partnerlinktyp spezifiziert werden über den kommuniziert wird. Es reicht aus, Nachrichten über das `<messageMapping>`-Element auf vom Prozess angebotene Operationen abzubilden. Da Operationen jedoch nur innerhalb eines Porttypen definiert sind, muss zur eindeutigen

Identifikation der Operation der zugehörige Porttyp über das Attribut `portTypeName` spezifiziert werden. Die vollständige Abbildung ist wie in Abschnitt 4.1 durchzuführen. Die Verwendung der Attribute `partnerLinkTypeName`, `myRole`, `partnerRole` und `role` zur Spezifikation einer vollständigen Abbildung und des Attributs `portTypeName` zur Spezifikation einer partiellen Abbildung sind Alternativen.

Die Abbildung der Prozesslogik von BPEL^{light}-Prozessen auf Semantische Web-Services ist eine Erweiterung der Abbildung der Prozesslogik auf Web-Services. Mit BPEL4SWS lassen sich daher zusätzlich zur Interaktion mit Diensten und Dienstnutzern unter Verwendung eines SSBs auch alle in Abschnitt 4.2 beschriebenen Interaktionsszenarien umsetzen, die ohne semantische Technologien möglich sind.

Dualismus der Datenrepräsentation durch SAWSDL

Durch die Entkopplung der Dienstbeschreibung von den Anforderungen eines Dienstnutzers wird Mediation zum zentralen Thema: Sie ermöglicht die Kommunikation zwischen Dienstnutzern und Diensten, auch wenn die auszutauschenden Nachrichten unterschiedlich definiert werden bzw. wenn Dienstnutzer und Dienst ein jeweils anderes, aber nicht inkompatibles Geschäftsprotokoll erfordern. Wie bereits in Kapitel 2.2.2 beschrieben, ist die Ontologie-basierte Mediation besonders geeignet, um Heterogenitäten aufzulösen. Soll sie in einem SSB zum Einsatz kommen, so müssen sowohl die von einem Dienstnutzer über den SSB gesendete Nachrichten als auch die Nachrichten eines Dienstes in Ontologieform vorliegen.

In BPEL und auch in BPEL^{light} werden Variablen mittels XML Schema typisiert. XML Schema kommt auch in der Regel zur Definition des Nachrichtenformats von Web-Services zum Einsatz. Werden Dienste durch WSMO beschrieben, so werden diesen XML-basierten Nachrichten durch das Grounding in der Choreography ontologische Konzepte zugeordnet. Auch die SAWSDL-Annotation `modelReference` ermöglicht das Verknüpfen von XML Schemata mit Ontologiekonzepten bzw. Instanzen.

Durch die Zuordnung von XML Nachrichtentypen zu Konzepten in einer Ontologie wird jedoch nur die Bedeutung der Nachricht definiert. Um aus einer Instanz eines XML Nachrichtentyps jedoch Ontologieinstanzen zu erzeugen bedarf es der Definition von Transformationsregeln. Auf diese kann mittels der SAWSDL-Annotationen `liftingSchemaMapping` bzw. `loweringSchemaMapping` verwiesen werden.

Festlegen eines Geschäftspartners

Wie bereits erwähnt, kann in BPEL4SWS spezifiziert werden, dass mehrere Ziele mit demselben Geschäftspartner zu verwirklichen sind. Hierzu müssen mehrere konsumierende Konversationen, die alle einen Nachrichtenaustausch mit dem gleichen `<bl:partner>` spezifizieren, mit WSMO-Goals annotiert werden. Soll ein konkreter Geschäftspartner oder eine konkrete Geschäftseinheit definiert werden, mit der ein oder mehrere Ziele verwirklicht werden sollen, so kann dies durch die Abbildung einer Instanz eines `<bl:partner>`s auf eine entsprechende Instanz aus einer Ontologie modelliert werden. Hierzu kann die SAWSDL-Annotation `modelReference` verwendet werden.

```
<blDeployment processName="http://example.org#buyProcess">...
  <partnerInstances partnerName="seller">
    <partnerInstance id="1"
                     sawsdl:modelReference="http://.../Onlinestores#amazon"/>
  </partnerInstances>
  ...
</blDeployment>
```

Listing 34: Erweitertes Schema des Deployment Deskriptors zur semantischen Abbildung auf Partnerinstanzen

In Listing 34 ist exemplarisch die Abbildung mittels Deployment Deskriptor für den Prozess „http://example.org#buyProcess“ dargestellt. Es wird für den Partnertyp „seller“ eine Partnerinstanz mit dem Wert „`id=“1“`“ definiert. Diese Partnerinstanz ist mit der Instanz „amazon“ aus der Ontologie „http://.../Onlinestores“ annotiert. Dadurch wird spezifiziert, dass für die Konversation „salesConv“, sofern sie mit einem Goal verknüpft

ist, nur Dienste in Frage kommen, deren Annotation `owner` auf die entsprechende Instanz verweist.

5.3 Unterstützte Interaktionsszenarien von BPEL4SWS

Werden Semantische Web-Services verwendet, um Aktivitäten zu implementieren bzw. die Funktionalität eines Prozesses zu beschreiben, so resultiert daraus eine höhere Flexibilität von Prozessen bezüglich der Wahl der Kommunikationpartner. Allerdings darf dieses Mehr an Flexibilität nicht heißen, dass wichtige Interaktionsszenarien nicht unterstützt werden. Im Folgenden wird nun zum Nachweis der Abdeckung der in Abschnitt 2.1.2 eingeführten von BPEL unterstützten Interaktionsszenarien das Zusammenspiel der Mechanismen von BPEL4SWS in den Szenarien gezeigt. Diese sind im Einzelnen: (i) Synchroner Aufruf eines kurzlebigen BPEL4SWS-Prozesses, (ii) Blockierender, synchroner Aufruf eines Dienstes, (iii) Asynchroner Aufruf eines langlebigen BPEL4SWS-Prozesses und (iv) Nicht blockierender, asynchroner Aufruf eines Dienstes. Zur Unterstützung der Szenarien bedarf es einer entsprechenden Infrastruktur (siehe 6.1.4).

(i) Aufruf eines kurzlebigen BPEL4SWS-Prozesses

Empfängt ein Prozess im Rahmen einer anbietenden Konversation zunächst eine Nachricht, um daraufhin eine Antwort zu versenden und ist diese Konversation von kurzer Dauer, so kann sie durch eine WSDL-Operation des Typs Request-Response verfügbar gemacht werden. In diesem Fall wird, wie bereits in Abschnitt 4.2 (i) beschrieben, der `<bl:output>` einer empfangenden Aktivität und der `<bl:input>` einer sendenden Aktivität auf diese Operation abgebildet.

Wird die WSDL-Operation aufgerufen, so wird die eintreffende Nachricht an die empfangende Aktivität entsprechend der Abbildung übergeben. Die spätere Ausführung der sendenden Aktivität vervollständigt den Aufruf unter Verwendung des `<bl:input>`s als Antwortnachricht.

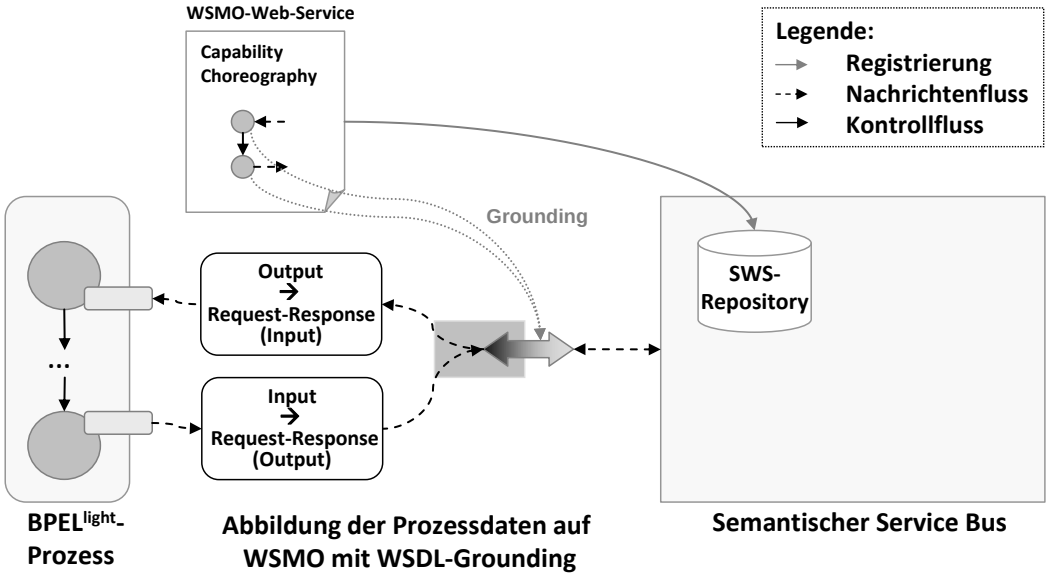


Abbildung 45: Aufruf eines kurzlebigen BPEL4SWS-Prozesses

Wie der Prozess zusätzlich auch als Semantischer Web-Service verfügbar gemacht werden kann ist in Abbildung 45 dargestellt. Zunächst muss ein WSMO-Web-Service modelliert werden, der die Choreography des Prozesses und dessen Capability beschreibt und ein Grounding auf die angebotene WSDL-Operation spezifiziert. Zusätzlich muss die WSDL-Beschreibung des Prozesses mit SAWSDL-Annotationen angereichert werden, um eine Transformation der Daten von ihrer XML Repräsentation zur ontologischen Repräsentation und zurück zu ermöglichen. Die WSMO-Beschreibung des durch den BPEL4SWS implementierten Dienstes muss anschließend durch Registrierung im SWS-Repository in den SSB deployt werden.

Nachdem der durch den Prozess implementierte Dienst durch den Abgleich seiner WSMO-Web-Service-Beschreibung mit einem von einem Dienstnutzer übermittelten WSMO-Goal aufgefunden wurde, wird die Kommunikation zwischen Dienst und Dienstnutzer innerhalb des SSBs initialisiert. Als nächster Schritt müssen die dem SSB vom Dienstnutzer zur Verfügung gestellten Daten in das für den Dienst richtige Format überführt werden. Wird zur Beschreibung der Anforderungen im WSMO-Goal eine andere Ontologie verwendet als zur Beschreibung der Fähigkeiten im WSMO-Web-Service, so kommt zunächst Ontologie-basierte Mediation zum

Einsatz. Anschließend wird die SAWSDL-Annotation `loweringSchemaMapping` der WSDL-Beschreibung des Prozesses verwendet, um die Ontologieinstanzen in eine der WSDL-Beschreibung entsprechende XML Darstellung zu überführen. Nach der Datentransformation wird der BPEL4SWS-Prozess anhand des in der WSMO-Web-Service-Beschreibung spezifizierten WSDL-Groundings aufgerufen. Der Aufruf wird vom Prozess, wie oben bereits beschrieben, verarbeitet. Nach Beendigung des Aufrufs werden die Daten durch den SSB von ihrer XML Darstellung mittels der SAWSDL-Annotation `liftingSchemaMapping` in Ontologieinstanzen überführt. Diese können dann gegebenenfalls mittels Ontologie-basierter Mediation in Instanzen der Ontologie überführt werden, in der die Nachrichten in der Choreography des Dienstnutzers beschrieben sind. Im Kopf der Nachricht, die vom SSB an den Prozess geschickt wird, befindet sich ein Kontext, der die Kommunikation zwischen Dienst und Dienstnutzer innerhalb des SSBs eindeutig identifiziert. Dieser wird benötigt, falls der Prozess den SSB oder der SSB den Prozess während der gleichen Konversation noch einmal aufrufen muss. Ruft der Prozess während der gleichen Konversation den SSB auf, so kann die entsprechende Instanz der Kommunikation zwischen Dienst und Dienstnutzer ermittelt werden. Bei einem erneuten Aufruf des Prozesses durch den SSB kann durch den Kontext die entsprechende Prozessinstanz ermittelt werden.

Ein BPEL4SWS-Prozess kann zwar durch die vorhandene oder nicht vorhandene Kontextinformation unterscheiden, ob er über semantische Technologien oder ausschließlich über Web-Service-Technologie aufgefunden und aufgerufen wurde, für die Verarbeitung des Aufrufs ist dies jedoch nicht von Relevanz. Ein BPEL4SWS-Prozess kann also im synchronen Fall sowohl über Web-Service-Technologien als auch über semantische Technologien aufgefunden und aufgerufen werden. Da die Art des Auffindens in diesem Szenario für den Prozess selber keine Rolle spielt, kann auch ein herkömmlicher WS-BPEL-Prozess, der mit einem `<bpel:receive>`-`<bpel:reply>`-Paar eine Request-Response-Operation implementiert, durch eine SWS-Beschreibung beschrieben werden und aufgefunden werden.

```
<blDeployment xmlns:ns1="http://example.org"
              processName="”http://...#performancedReviewHandlingProcess...>
  ...
  <conversationMapping
      conversationName="performanceReviewConv"
      service="http://example.org/service/getPerformanceReview">
    <messageMapping messageLabel="employeeDetails"
                    operationName="getPerformanceReview"
                    portTypeName="ns1:hrmPT"/>
    <messageMapping messageLabel="performanceReview"
                    operationName="getPerformanceReview"
                    portTypeName="ns1:hrmPT"/>
  </conversationMapping>
</blDeployment>


webService _"http://example.org/service/getPerformanceReview"...
  capability _"http://www.example.org/service/getPerformanceReview#capability"
  interface _"http://www.example.org/service/getPerformanceReview#interface"
    choreography ...
      stateSignature ...
        in
          hrm#performanceReviewRequest withGrounding _"http://example.org/hrm.
            wsdl#wsdl11.portTypeOperation.input(hrmPT/getPerformanceReview)"
        out
          hrm#performanceReview withGrounding _"http://example.org/hrm.wsdl#
            wsdl11.portTypeOperation.output(hrmPT/getPerformanceReview)"
      transitionRules
        (?input memberOf hrm#performanceReviewRequest implies
          exists ?# memberOf hrm#performanceReview
```

Listing 35: Deployment Deskriptor und WSMO-Web-Service-Beschreibung für den kurzlebigen, synchron aufrufbaren BPEL4SWS-Prozess *„performanceReviewHandlingProcess“*

Listing 35 präsentiert eine konkrete Ausprägung des Szenarios. Es umfasst sowohl eine WSMO-Web-Service-Beschreibung der anbietenden Konversation „performanceReviewConv“ des bereits in Listing 26 präsentierten BPEL[light]-Prozesses „performanceReviewHandlingProcess“ als auch die Abbildung der Konversation auf WSMO und WSDL mittels eines Deployment Deskriptors. Die zugehörige WSDL-Datei inklusive der SAWSDL-Annotationen befindet sich in Anhang A. Die WSMO-Beschreibung spezifiziert, dass über die Operation „getPerformanceReview“ eine Nachricht, die dem Ontologiekonzept „performanceReviewRequest“ entspricht, empfangen wird und anschließend über die gleiche Operation eine Nachricht verschickt wird, die dem Ontologiekonzept „performanceReview“ entspricht. Der Deployment Deskriptor definiert

zudem, dass die eingehende Nachricht der Operation „getPerformanceReview“ vom `<bl:output>` „employeeDetails“ des Prozesses verarbeitet wird und dass der `<bl:input>` die Daten für die Antwortnachricht der Operation „getPerformanceReview“ liefert.

(ii) Aufruf eines kurzlebigen Semantischen Web Services

Sendet ein Prozess im Rahmen einer konsumierenden Konversation zunächst eine Nachricht an einen Dienst, um daraufhin eine Antwort zu empfangen und ist diese Konversation von kurzer Dauer, so kann sie zum einen auf eine Request-Response-Operation des zu verwendenden Dienstes abgebildet werden. Dies wird in Abschnitt 4.2 (ii) beschrieben. Zum anderen kann sie aber auch mit einem WSMO-Goal annotiert werden.

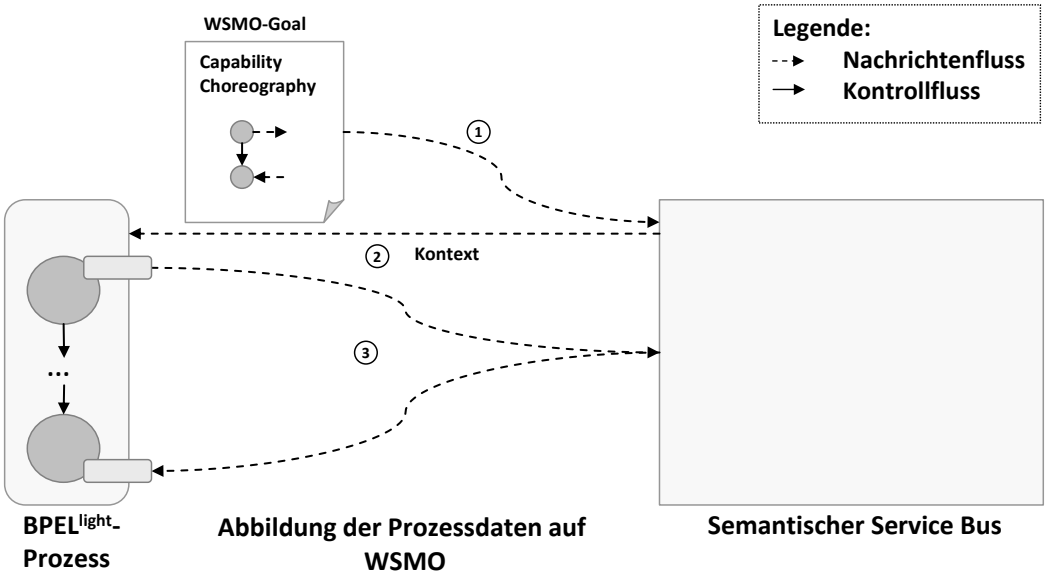


Abbildung 46: Aufruf eines kurzlebigen Semantischen Web-Services

Der Ablauf der Ausführung bei einer durch ein Goal annotierten Konversation ist in Abbildung 46 dargestellt. Bei Beginn der Konversation wird das Goal an den SSB übermittelt. Dieser ermittelt zunächst durch Abgleich der WSMO-Goal Beschreibung mit verfügbaren WSMO-Web-Service-Beschreibungen einen passenden Dienst. Anschließend initialisiert er die Kommunikation zwischen Dienst und Dienstnutzer und übermittelt einen die Kommunikation identifizierenden Kontext zur Korrelation an den Dienstnutzer. Dieser Kontext kann, wie bereits in (i) beschrieben,

verwendet werden, um zum einen die richtige BPEL4SWS-Prozessinstanz zu ermitteln im Falle eines Aufrufs des Prozesses durch den SSB. Zum anderen kann auch die entsprechende Instanz der Kommunikation zwischen Dienst und Dienstnutzer ermittelt werden, wenn der Prozess den SSB aufruft. Im vorliegenden Szenario wird der Inhalt des `<bl:input>`s einer sendenden Aktivität dem SSB zusammen mit dem Kontext übermittelt. Da der Prozess auf Daten in XML Darstellung und der SSB intern auf Ontologien operiert, wird die SAWSDL-Annotation `liftingSchemaMapping` dazu verwendet, die Daten von ihrer XML Repräsentation in entsprechende Ontologieinstanzen zu überführen. Der SSB führt gegebenenfalls Ontologie-basierte Mediation durch, ruft den Dienst auf und schickt die Antwortnachricht an den Prozess zurück. Die Antwortnachricht wird mittels der SAWSDL-Annotation `loweringSchemaMapping` in die innerhalb des BPEL4SWS-Prozesses verwendete XML Darstellung überführt und im `<bl:output>` der empfangenden Aktivität abgespeichert. Das Senden und das Empfangen der Nachrichten kann wie in Abbildung 46 dargestellt durch zwei Aktivitäten realisiert werden. Es kann allerdings auch durch eine einzelne `<bl:interactionActivity>` realisiert werden, die das Nachrichtenaustauschmuster „Out-In“ implementiert.

Listing 36 präsentiert eine konkrete Ausprägung des Szenarios. Es umfasst sowohl eine WSMO-Goal Beschreibung der konsumierenden Konversation „performanceReviewConv“ des bereits in Listing 27 präsentierten BPEL^{light}-Prozesses „performanceReviewRequestProcess“ als auch die Abbildung der Konversation auf das WSMO-Goal mittels eines Deployment Deskriptors. Die zugehörige WSDL-Datei zur Spezifikation der im Prozess verwendeten Datentypen inklusive der SAWSDL-Annotationen befindet sich in Anhang A.

```
<blDeployment processName="http://...#performancedReviewRequestProcess...>
  ...
  <conversationMapping
      conversationName="performanceReviewConv"
      goal="http://example.org/goal/getPerformanceReview"/>
</blDeployment>


goal _"http://example.org/goal/getPerformanceReview"...
  capability _"http://www.example.org/goal/getPerformanceReview#capability"
  interface _"http://www.example.org/goal/getPerformanceReview#interface"
    choreography
      stateSignature ...
        out
          hrm#performanceReviewRequest
        in
          hrm#performanceReview
      transitionRules
        add(_# memberOf hrm#performanceReviewRequest)
```

Listing 36: Deployment Deskriptor und WSMO-Goal für den synchron aufrufenden BPEL4SWS-Prozess *„performanceReviewRequestProcess“*

Die WSMO-Beschreibung spezifiziert, dass eine Nachricht, die dem Ontologiekonzept „performanceReviewRequest“ entspricht, gesendet wird und anschließend eine Nachricht empfangen wird, die dem Ontologiekonzept „performanceReview“ entspricht. Es ist keine Abbildung der `<bl:input>`s und `<bl:output>`s auf WSDL-Operationen spezifiziert. Wird die sendende Aktivität ausgeführt, so sendet sie die spezifizierte Nachricht deshalb an den SSB und nicht direkt an den Dienstnutzer.

(iii) Aufruf eines langlebigen BPEL4SWS-Prozesses

Ein Prozess, der im Rahmen einer anbietenden Konversation zunächst eine Nachricht empfängt, um daraufhin eine Antwort zu versenden, kann dann nicht als eine Request-Response-Operation zur Verfügung gestellt werden, wenn nicht davon auszugehen ist, dass die Antwort kurz nach dem Eintreffen der ersten Nachricht verschickt werden kann. In diesem Fall würde ein Aufruf entsprechend dem WS-I konformen HTTP-Binding in einem durch einen Timeout verursachten Fehler resultieren. Stattdessen muss das Empfangen einer Nachricht und spätere Senden einer Antwortnachricht wie in Abschnitt 4.2 (iii) beschrieben auf zwei durch

einen Partnerlinktypen verbundene One-way-Operationen abgebildet werden, um über Web-Services kommunizieren zu können.

Allerdings kann ein solcher Prozess auch einem SSB als Semantischer Web-Service durch Registrierung einer entsprechenden WSMO-Web-Service-Beschreibung zur Verfügung gestellt werden. Dieser WSMO-Web-Service definiert für die eintreffende Nachricht ein Grounding auf die von Prozess angebotene One-way-Operation. Für die zu versendende Nachricht wird jedoch kein Grounding definiert, da die WSDL-Operation und der Endpunkt des Dienstnutzers zum Zeitpunkt der Modellierung nicht bekannt sind. Auch wenn der Prozess als Semantischer Web-Service aufgerufen werden kann, wird sowohl für den `<bl:output>` der empfangenden Aktivität als auch für den `<bl:input>` der sendenden Aktivität eine Abbildung auf eine One-way-Operation entsprechend Abschnitt 4.2 (iii) spezifiziert. Es ist also bei einer anbietenden Konversation eines BPEL4SWS-Prozesses gleichzeitig ein Partnerlinktyp spezifiziert, über den kommuniziert werden kann, und die Prozessseite des „dynamischen Partnerlinks“.

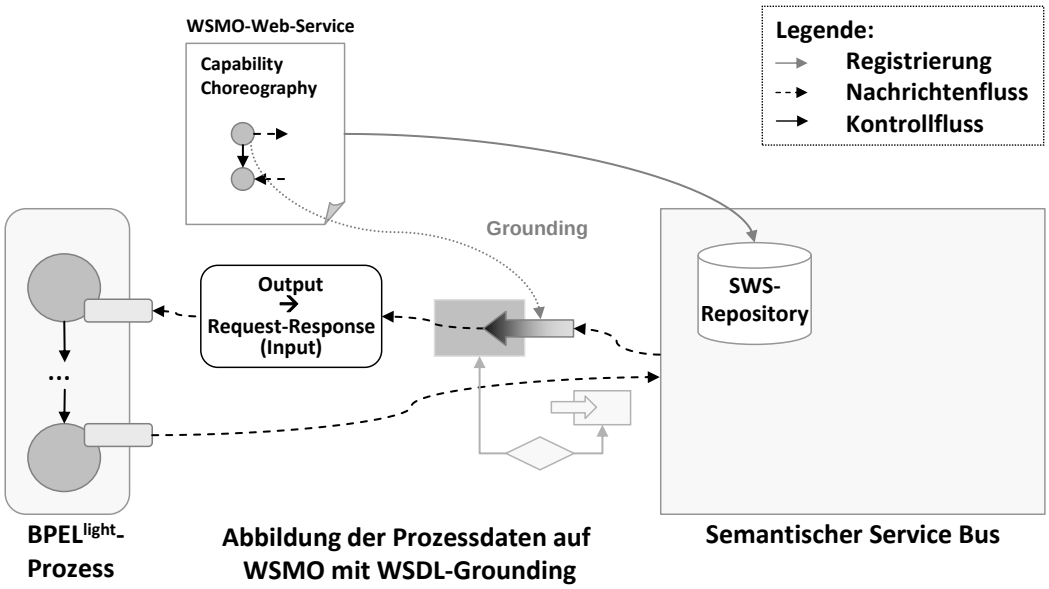


Abbildung 47: Aufruf eines langlebigen BPEL4SWS-Prozesses

Wird der BPEL4SWS-Prozess wie in Abbildung 47 gezeigt über seine WSMO-Beschreibung aufgefunden, so wird er nicht vom Klienten direkt sondern vom SSB aufgerufen. Dazu werden über das Grounding die

Operation und der Endpunkt ermittelt und mit Hilfe Ontologie-basierter Mediation und der SAWSDL-Annotation `loweringSchemaMapping` das benötigte Nachrichtenformat hergestellt. Im Nachrichtenkopf übermittelt der SSB beim Aufruf die Kontextinformation, welche die Kommunikation zwischen Dienstnutzer und Dienst innerhalb des SSBs eindeutig identifiziert. Anhand dieser Kontextinformation ist es für den Prozess ersichtlich, dass er vom SSB aufgerufen wurde. Wird nun die zugehörige sendende Aktivität erreicht, so wird nicht die in Abbildung 47 hellgrau dargestellte dem `<bl:input>` der sendenden Aktivität zugeordnete Operation verwendet, sondern stattdessen die vom SSB angebotene Funktionalität zum Entgegennehmen einer Nachricht im Rahmen einer Kommunikation. Da die Daten innerhalb des SSBs in ontologischer Form vorliegen müssen, werden sie von ihrer XML Repräsentation mittels der SAWSDL-Annotation `liftingSchemaMapping` in Ontologieinstanzen überführt.

Wird der BPEL4SWS-Prozess jedoch nicht vom SSB, sondern direkt von einem Web-Service unterstützenden Klienten aufgerufen, so fehlt die Kontextinformation im Nachrichtenkopf. In diesem Fall antwortet der Prozess dem Klienten unter Verwendung der in der Abbildung des `<bl:input>`s der sendenden Aktivität spezifizierten Operation und des beim Aufruf übermittelten Endpunktes wie in Abschnitt 4.2 (iii) beschrieben.

Ein und derselbe Prozess kann also auch im asynchronen Fall sowohl über Web-Service als auch über semantische Technologien aufgefunden und aufgerufen werden.

```
<blDeployment xmlns:ns1="http://example.org"
              processName="”http://example.org#bookingHandlingProcess”...>
  ...
  <conversationMapping conversationName="bookTripConv"
                       service="http://example.org/service/bookTrip">
      <partnerMapping partnerName="customer"
                      partnerLinkTypeName="ns1:travelAgentPLT"
                      myRole="travelAgent"
                       partnerRole="cusomer"/>
    <messageMapping messageLabel="tripDetails"
                    operationName="bookTrip"
                                   role="myRole"/>
    <messageMapping messageLabel="booking"
                    operationName="receiveBooking"
                    role="partnerRole"/>
  </conversationMapping>
</blDeployment>


webService _"http://example.org/service/bookTrip"...
  capability _"http://www.example.org/service/bookTrip#capability"
  interface _"http://www.example.org/service/bookTrip#interface"
    choreography ...
      stateSignature ...
        in
          vta#bookTripRequest withGrounding _"http://example.org/vta.wsdl#
            wsdl11.portTypeOperation.input(vtaPT/bookTrip)"
        out
          vta#booking
      transitionRules
        (?input memberOf vta#bookTripRequest implies
          exists ?# memberOf vta#booking
```

Listing 37: Deployment Deskriptor und WSMO-Web-Service für den asynchron aufrufbaren BPEL4SWS-Prozess *„bookingHandlingProcess“*

Listing 37 präsentiert eine konkrete Ausprägung des Szenarios. Es umfasst sowohl eine WSMO-Web-Service-Beschreibung der anbietenden Konversation „bookTripConv“ des bereits in Listing 28 präsentierten BPEL[light]-Prozesses „bookingHandlingProcess“ als auch die Abbildung der Konversation auf WSMO und WSDL mittels eines Deployment Deskriptors. Die zugehörige WSDL-Beschreibungen inklusive der SAWSDL-Annotationen befinden sich in Anhang B und C. Die WSMO-Beschreibung spezifiziert, dass über die Operation „bookTrip“ eine Nachricht, die dem ontologischen Konzept „bookTripRequest“ entspricht, empfangen wird und anschließend eine Nachricht verschickt wird, die dem Ontologiekonzept „booking“ entspricht. Der Deployment Deskriptor definiert zudem, dass die eingehende Nachricht der Operation „bookTrip“ vom `<bl:output>`

„tripDetails“ des Prozesses verarbeitet wird und dass der `<bl:input>` „booking“ die Daten für die Antwortnachricht liefert, die im Falle der direkten Kommunikation zwischen Dienst und Dienstnutzer über die vom Dienstnutzer angebotene Operation „receiveBooking“ gesendet wird.

(iv) Aufruf eines langlebigen Semantischen Web-Services

Sendet ein Prozess im Rahmen einer konsumierenden Konversation zunächst eine Nachricht an einen Dienst um daraufhin eine Antwort zu empfangen, so kann sie nicht auf eine Request-Response-Operation des zu verwendenden Dienstes abgebildet werden, wenn nicht davon ausgegangen werden kann, dass die Antwort kurz nach dem Verschicken der ersten Nachricht empfangen wird. Wird, wie in Abschnitt 4.2 (iv) beschrieben, zwischen Dienst und Prozess direkt über Web-Service-Technologie kommuniziert, so wird der Nachrichtenaustausch auf zwei One-way-Operationen abgebildet. Soll der Dienst jedoch über semantische Technologien ermittelt werden, so wird die Konversation mit einem WSMO-Goal annotiert. Im Gegensatz zum blockierenden Goal-basierten Aufruf eines Dienstes, wird jedoch für die einkommende Nachricht im Goal ein Grounding auf eine vom Prozess angebotene One-way-Operation spezifiziert, um die Antwort des Dienstes asynchron empfangen zu können. Außerdem wird der `<bl:output>` der empfangenden Aktivität auf diese One-way-Operation abgebildet.

Wie beim Goal-basierten blockierenden Aufruf eines Dienstes beginnt der nicht-blockierende Aufruf eines Dienstes mit der Übermittlung des WSMO-Goals an den SSB (siehe Abbildung 48). Anschließend ermittelt dieser zunächst durch Abgleich der WSMO-Goal Beschreibung mit verfügbaren WSMO-Web-Service-Beschreibungen einen passenden Dienst, initialisiert die Kommunikation zwischen Dienst und Dienstnutzer und übermittelt einen, die Kommunikation identifizierenden Kontext zur Korrelation an den Dienstnutzer. Dann wird der Inhalt des `<bl:input>`s einer sendenden Aktivität dem SSB zusammen mit dem Kontext übermittelt.

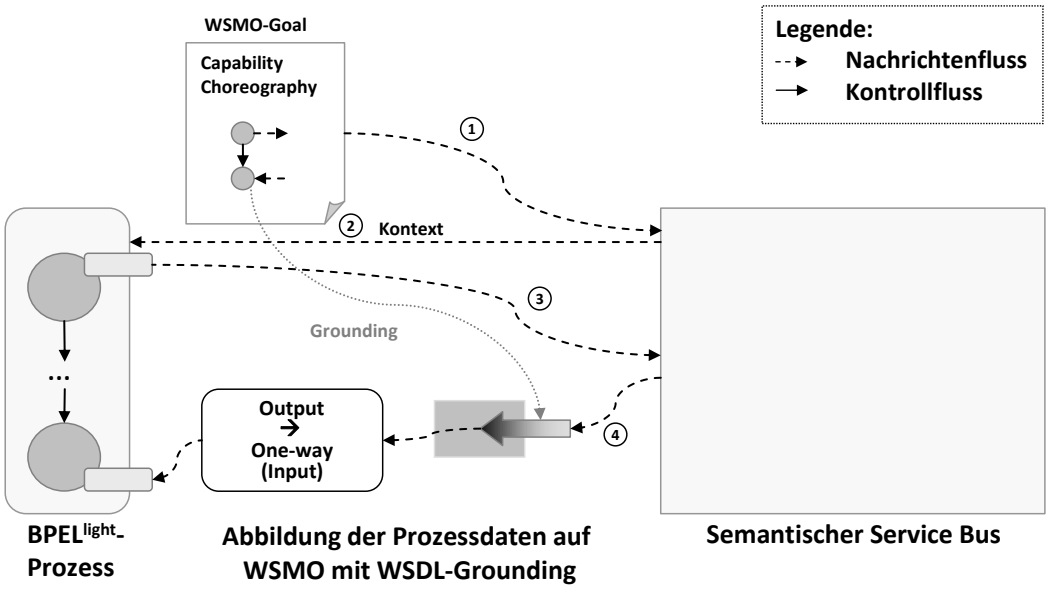


Abbildung 48: Aufruf eines langlebigen Semantischen Web-Services

Da der Prozess auf Daten in XML Darstellung und der SSB intern auf Ontologien operiert, wird die SAWSDL-Annotation `liftingSchemaMapping` dazu verwendet, die Daten von ihrer XML Repräsentation in entsprechende Ontologieinstanzen zu überführen. Der SSB führt gegebenenfalls Ontologie-basierte Mediation durch, ruft den Dienst auf und schickt die Antwortnachricht an den Prozess zurück. Dazu werden über das Grounding die Operation und der Endpunkt ermittelt und mit Hilfe Ontologie-basierter Mediation und der SAWSDL-Annotation `loweringSchemaMapping` das benötigte Nachrichtenformat hergestellt. Die Antwortnachricht wird schließlich im `<bl:output>` der empfangenden Aktivität abgespeichert.

```
<blDeployment xmlns:ns1="http://example.org"
              processName="”http://example.org#bookingProcess”...>
  ...
  <conversationMapping conversationName="bookTripConv"
                       goal="http://example.org/goal/bookTrip">
    <messageMapping messageLabel="booking"
                    operationName="receiveBooking"
                    portTypeName="ns1:customerPT"/>
  </conversationMapping>
</blDeployment>


goal _"http://example.org/goal/bookTrip"...
  capability _"http://www.example.org/goal/bookTrip#capability"
  interface _"http://www.example.org/goal/bookTrip#interface"
    choreography
      stateSignature ...
        out
          vta#bookTripRequest
        in
          vta#booking withGrounding _"http://example.org/vtaCustomer.wsdl#
            wsdl11.portTypeOperation.input(vtaCustomerPT/receiveBooking)"
      transitionRules
        add(_# memberOf vta#bookTripRequest)
```

Listing 38: Deployment Deskriptor und WSMO-Goal für den asynchron aufrufenden BPEL4SWS-Prozess *„bookingProcess“*

Listing 38 präsentiert eine konkrete Ausprägung des Szenarios. Es umfasst sowohl eine WSMO-Web-Service-Beschreibung der konsumierenden Konversation „bookTripConv“ des bereits in Listing 29 präsentierten BPEL[light]-Prozesses „bookingProcess“ als auch die Abbildung der Konversation auf WSMO und WSDL mittels eines Deployment Deskriptors. Die zugehörigen WSDL-Beschreibungen inklusive der SAWSDL-Annotationen befinden sich in Anhang B und C. Die WSMO-Beschreibung spezifiziert, dass zunächst eine Nachricht, die dem ontologischen Konzept „bookTripRequest“ entspricht, gesendet wird und anschließend eine Nachricht über die Operation „receiveBooking“ empfangen wird, die dem ontologiekonzept „booking“ entspricht. Der Deployment Deskriptor definiert zudem, dass die Nachricht der Operation „receiveBooking“ vom `<bl:output>` „booking“ des Prozesses verarbeitet wird.

5.4 Beispiel: BPEL4SWS-Prozess als Klient eines semantisch beschriebenen Onlineversandhandeldienstes

Um die Eignung von BPEL4SWS zur Definition von ausführbaren Geschäftsprozessen zu demonstrieren, wird in diesem Abschnitt gezeigt, wie der in Abschnitt 3.5 präsentierte BPEL^light-Prozess durch ein geeignetes WSMO-Goal seine Anforderungen an eine durch WSMO beschriebene Dienstimplementierung spezifizieren kann, bei der ein Bestellvorgang vorgenommen werden soll.

```
<blDeployment xmlns:ns1="http://example.org"
              processName="http://example.org#buyProcess">...
  <conversationMapping conversationName="salesConv"
                       goal="http://example.org/goal/onlineshopclient">
    <messageMapping messageLabel="shippingDetails"
                    operationName="getShippingDetails"
                    portTypeName="ns1:buyerPT"/>
  </conversationMapping>
</blDeployment>
```

Listing 39: Deployment Deskriptor für einen BPEL4SWS-Prozess, der Anforderungen an einen Onlineversandhandeldienst über ein WSMO-Goal stellt

In Listing 39 ist der zur Abbildung auf WSMO notwendige Deployment Deskriptor abgebildet. Der über die Konversation „salesConv“ beschriebene Nachrichtenaustausch zur Bestellung ist bilateral, d.h. es ist außer dem Prozess „http://example.org#buyProcess“ nur ein weiterer Geschäftspartner involviert. Aus diesem Grund kann die komplette Konversation mit einem WSMO-Goal beschrieben werden. Durch das Attribut `goal` für das `<conversationMapping>`-Element, das sich auf die Konversation „salesConv“ bezieht, wird dementsprechend das bereits in Listing 31 dargestellte WSMO-Goal „http://example.org/goal/onlineshopclient“ referenziert Außerdem wird die einkommende Nachricht „shippingDetails“ dem Grounding des WSMO-Goals entsprechend auf die Operation „getShippingDetails“ des Porttyps „ns1:buyerPT“ abgebildet.

Abbildung 49 zeigt eine schematische Darstellung des Prozesses, der einen Bestellvorgang realisiert. Seine Anforderungen, die durch ein WSMO-Goal beschrieben werden, werden durch den in Abschnitt 2.1.1.2 eingeführten

und in Abschnitt 2.2.2 mit WSMO beschriebenen Onlineversand-handeldienst erfüllt.

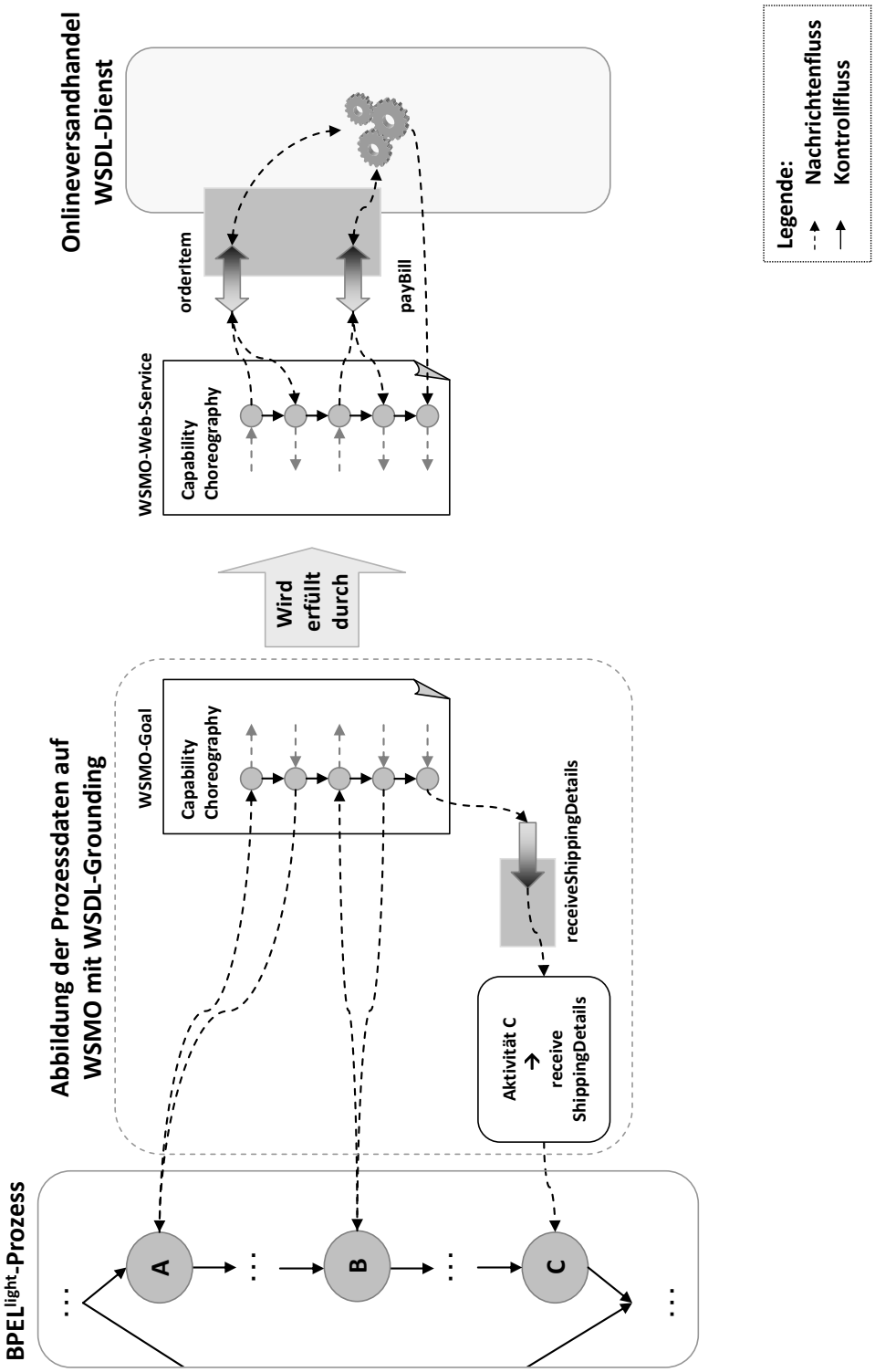


Abbildung 49: BPEL4SWS-Prozess als Klient eines mit WSMO beschriebenen Onlineversandhandeldienstes

5.5 Ontologiewissen über Daten in BPEL4SWS

In BPEL4SWS ist nicht wie bei herkömmlichem BPEL nur die Struktur der Daten durch ein XSD definiert, sondern auch die Bedeutung der Daten bekannt: Die SAWSDL-Annotation `modelReference` wird verwendet, um einer Datenstruktur ein Konzept in einer Ontologie zuzuweisen. Weiterhin ist definiert, wie Daten von ihrer XML Repräsentation in eine von der Struktur abstrahierende ontologische Darstellung überführt werden können und anders herum. Dadurch können auch innerhalb eines Prozesses semantische Technologien zum Einsatz kommen, unter anderem zur Definition von Datenmanipulation und zur Auswertung von Bedingungen auf Basis ontologischen Wissens.

5.5.1 Vereinfachung der Datenmanipulation auf Basis ontologischen Wissens

Datenfluss wird in BPEL über Variablen realisiert, auf die von allen Aktivitäten im Prozess bzw. Geltungsbereich zugegriffen werden kann. Sie können zum einen als Input- oder Outputcontainer von Interaktionsaktivitäten und zum anderen als Quelle bzw. Ziel von Datenmanipulationsoperationen verwendet werden: In einer `<bpel:assign>`-Aktivität können über `<bpel:copy>`-Anweisungen Variablen als ganzes oder auch Teile von Variablen auf andere Variablen oder Teile davon kopiert werden.

Abbildung 50 zeigt einen Ausschnitt einer Variante des in Listing 24 dargestellten Bestellprozesses. In diesem Fall tätigt der Prozess eine Überweisung, um die übermittelte Rechnung zu bezahlen. Dazu sind folgende Schritte notwendig: Zunächst wird die empfangene Rechnung in die Variable „invoice“ geschrieben. Anschließend liest eine `<bpel:assign>`-Aktivität die Variable „invoice“ aus, extrahiert die relevanten Informationen und schreibt sie in die Variable „sellerAccount“. Diese wird wiederum verwendet, um die Variable „creditTransfer“ zu füllen. Anschließend kommt statt eines einfachen „Out-In“ Nachrichtenaustauschmusters, ein „Request with Referral“ zum Einsatz, d.h. die Variable „creditTransfer“ wird ausgelesen und an eine Bank übermittelt, die dem Onlineversandhandeldienst den Geldeingang bestätigt.

Dieser schickt dann wiederum eine Bestätigungsnachricht an den bestellenden Prozess.

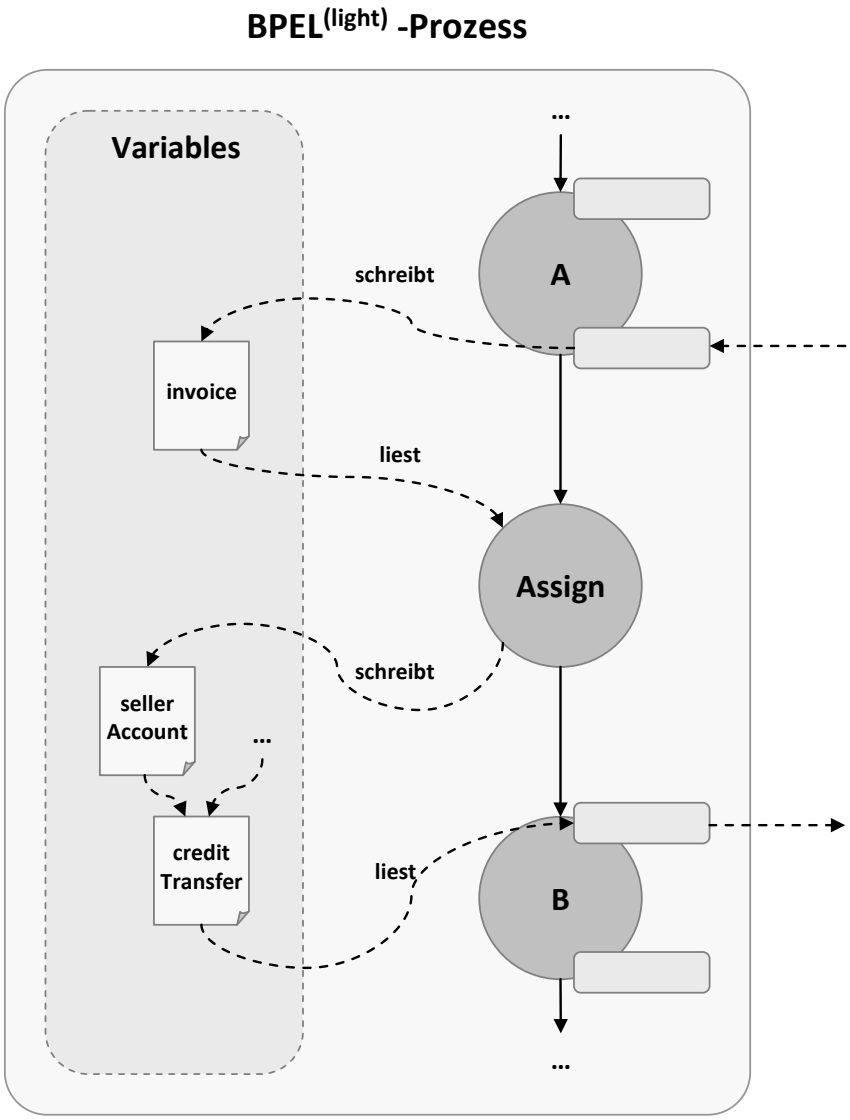


Abbildung 50: Datenmanipulation in BPEL

Listing 40 zeigt das für das Vorgehen relevante Codefragment. Über die Aktivität „order“ wird eine Nachricht des Typs „tInvoice“ empfangen, die in die Variable „invoice“ gespeichert wird. Sie enthält neben einer Liste der bestellten Waren und dem Rechnungsbetrag unter anderem auch die Bankdaten des Rechnungsstellers kodiert in die Elemente „bankAccountNumber“, „accountHolder“ – welches wiederum die Unterelemente „firstName“ und „lastName“ enthält – und „bankCode“. Der Inhalt dieser Elemente wird durch `<bpel:copy>` Anweisungen in der

`<bpel:assign>`-Aktivität in die Felder „accountNumber“, „depositor“ – bzw. dessen Unterelemente „forename“ und „surname“ – und „bankNumber“ der Variablen „sellerAccount“ übertragen. Diese wird anschließend in der Aktivität „pay“ zusammen mit der Bankverbindung des Einkäufers und des zu überweisenden Betrages der Bank als Überweisungsauftrag als Nachricht „creditTransfer“ übermittelt.

```
<!-- process definition -->
<process ...
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsl="http://www.example.org/wholesaler/"
    xmlns:bnk="http://www.example.org/banking/" />...
<!-- variable definition -->
  <variables>
    <variable name="Invoice" messageType="wsl:tInvoice"/>
    <variable name="SellerAccount" messageType="bnk:tAccount"/>...
  </variables>...
<bpel:sequence>...
  <bpel:extensionActivity>
    <bl:interactionActivity bpel:name="order" ... >
          <bl:output variable="invoice"... />
    </bl:interactionActivity>
  <bpel:extensionActivity>
<!-- assign activity -->
  <assign>
    <copy>
      <from>$Invoice/accountholder/firstName</from>
      <to>$SellerAccount/depositor/forename</to>
    </copy>
    <copy>
      <from>$Invoice/accountholder/lastName</from>
      <to>$SellerAccount/depositor/surname</to>
    </copy>
    <copy>
      <from>$Invoice/bankAccountNumber</from>
      <to>$SellerAccount/accountNumber</to>
    </copy>
    <copy>
      <from>$Invoice/bankCode</from>
      <to>$SellerAccount/bankNumber</to>
    </copy>...
  </assign>...
  <bpel:extensionActivity>
    <bl:interactionActivity bpel:name="pay" ... >
          <bl:output variable="creditTransfer"... />
    </bl:interactionActivity>
  <bpel:extensionActivity>...
</process>
```

Listing 40: Extrahierung der Bankdaten über herkömmliches Assignment

Das Kopieren von einer XML Struktur in eine andere ist jedoch nur möglich, da über die Variablen in BPEL den Datenstrukturen eine lokale Semantik zugeordnet wird, die beim Modellieren vom Prozessmodellierer festgelegt wird. Sollen die gleichen Kopieroperationen an einer anderen Stelle im Prozess oder in einem anderen Prozess wieder verwendet werden, so ist dies nur über „Copy & Paste“ möglich. Soll eine Wiederverwendung von Transformationen im eigentlichen Sinne stattfinden, so kann statt der `<bpel:copy>` Anweisungen die Xpath erweiternde Funktion „doXslTransform“ verwendet werden, die es ermöglicht, XSLTs zur Transformation von XML-basierten Daten zu verwenden. Allerdings muss bei dieser Wiederverwendung durch den Prozessmodellierer gewährleistet sein, dass das XSLT im Kontext des Prozesses die Daten semantisch korrekt von der Quellvariablen zur Zielvariablen überführt.

Ist jedoch die Bedeutung der Daten wie in BPEL4SWS durch semantische Annotation bekannt, so kann die Entscheidung ob eine Transformation geeignet ist, um Daten von einem Quellformat in ein Zielformat zu überführen, automatisch getroffen werden. Es reicht daher aus, die Quell- und Zielvariable zu spezifizieren, um einen entsprechenden Transformator bzw. Mediator zu finden. Diese Mediatoren können unabhängig von einem Prozess definiert werden und sind somit wieder verwendbar. Basiert eine Transformation auf einer bestimmten syntaktischen Ausprägung von Daten, so ist die Wiederverwendbarkeit eingeschränkt, da nicht nur die Semantik der Quell- und Zielvariable passend sein muss, sondern zusätzlich ihre syntaktische Ausprägung. Ontologien abstrahieren von der syntaktischen Ausprägung und bilden ausschließlich die Semantik der Daten ab. Werden Transformationen also auf Basis von Ontologien, d.h. strukturunabhängig spezifiziert, so ist für sie folgerichtig eine maximale Wiederverwendbarkeit gewährleistet. Um diese Art von Mediatoren in einem Prozess einsetzen zu können, definiert BPEL4SWS das `<b4s:mediate>`-Element als eine `<bpel:extensionAssignOperation>` (siehe Listing 41). Es müssen lediglich über die Attribute `inputVariable` und `outputVariable` die Quell- bzw. Zielvariablen der Transformation angegeben werden. Optional kann auch der zu verwendende Mediator über

das Attribut `mediatorURI` und eine Bezeichnung des Elements über das Attribut `name` angegeben werden.

```
<assign validate="yes|no"? standard-attributes>
    standard-elements
  (
  <copy keepSrcElementName="yes|no"?>from-spec to-spec</copy>
  |
  <extensionAssignOperation>
    <b4s:mediate name="NCName"?
                 mediatorURI="anyURI"?
                 inputVariable="NCName"+
                 outputVariable="NCName"+/>
  </extensionAssignOperation>
  )+
</assign>
```

Listing 41: Schema der Erweiterung `<b4s:mediate>`

Ist der Mediator spezifiziert, so wird bei der Ausführung der `<b4s:mediate>`-Operation zunächst die Quellvariable mittels der SAWSDL-Annotation `liftingSchemaMapping` in ontologische Form überführt. Anschließend wird der Mediator über entsprechende, vom SSB angebotene Funktionalität angewendet und das Ergebnis der Mediation über die SAWSDL-Annotation `loweringSchemaMapping` in XML Form überführt.

Ist das Attribut `mediatorURI` nicht spezifiziert, so kann durch die Infrastruktur mittels der Quell- und Zielvariablen ein WSMO-Goal formuliert werden, welches keine Seiteneffekte, d.h. weder Assumptions noch Effects definiert. Ein passender Mediator, der über eine WSMO-Web-Service-Beschreibung verfügt, kann somit ermittelt werden. Zu diesem Zweck muss ein SSB Funktionalität anbieten, die anhand einer WSMO-Goal Beschreibung und der zu vermittelnden Instanzdaten einen entsprechenden Mediator auffindet, aufruft und die vermittelten Instanzdaten als Ergebnis zurückgibt. Auch eine Verkettung mehrerer Mediatoren, d.h. Diensten, die keine Seiteneffekte haben, zur Erfüllung des Mediationsziels ist möglich.

Im Folgenden ist die Datenmanipulation des in Abbildung 50 und Listing 40 präsentierten Beispiels mit semantischen Technologien beschrieben. Das in Listing 4 abgebildete XML Schema „tInvoice“ der Variablen „Invoice“ ist

per SAWSDL-Attribut `modelReference` mit dem Konzept „invoice“ aus der Ontologie „onlineshop“ (siehe Listing 42) annotiert.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace _"http://www.example.org/ontologies/onlineshop#"

ontology onlineshop

concept invoice
  hasNumber ofType _integer
  belongsToShoppingCart ofType shoppingCart
  hasInvoicingParty ofType legallyCompetentPerson
  hasAccountNumber ofType _string
  hasBankIdentificationCode ofType _string


concept legallyCompetentPerson
  hasSurname ofType _string
  hasPrename ofType _string
```

Listing 42: Die „onlineshop“-Ontologie

Analog ist das ebenfalls in Listing 4 abgebildete XML Schema „tAccount“ der Variablen „sellerAccount“ mit dem Konzept „account“ aus der Ontologie „banking“ (siehe Listing 43) annotiert. Für beide ist sowohl ein `liftingSchemaMapping` als auch ein `loweringSchemaMapping` definiert.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace _"http://www.example.org/ontologies/banking#"

ontology banking

concept account
  hasName ofType _string
  hasFirstName ofType _string
  hasAccountNumber ofType _string
  hasBankCode ofType _string
```

Listing 43: Die „banking“-Ontologie

In Listing 44 ist schließlich das BPEL4SWS-Codefragment zur Spezifikation der Datenmanipulation auf Basis ontologischen Wissen dargestellt. Anstatt mehrere `<bpel:copy>`-Anweisungen spezifizieren zu müssen, reicht es aus, ein `<b4s:mediate>`-Element mit der Variablen „invoice“ als Eingabe- und

der Variablen „sellerAccount“ als Ausgabeparameter zu spezifizieren. Da kein Mediator über das Attribut `mediatorURI` spezifiziert ist, wird zur Laufzeit eine, den geforderten Ein- und Ausgaben entsprechender Mediator gesucht.

```
<assign>
  ...
  <extensionAssignOperation>
    <b4s:mediate name="extractBankAccount"
                 inputVariable="invoice"
                 outputVariable="sellerAccount"/>
  </extensionAssignOperation>
</assign>
```

Listing 44: Extrahierung der Bankdaten über Mediation

Die Implementierung des Mediators kann in einer beliebigen Sprache erfolgen. So kann die Transformation von einer Ontologieinstanz zur anderen beispielsweise über XSLTs definiert werden, die auf der XML Serialisierung von Ontologien operieren. Eine andere Möglichkeit der Implementierung ist die Definition geeigneter WSML-Axiome. In Listing 45 ist ein WSML-Axiom zur Extrahierung der Bankdaten aus einer Rechnung und zur Erstellung eines Bankkontodatensatzes dargestellt.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace {ba _"http://www.example.org/ontologies/banking#",
           os _"http://www.example.org/ontologies/onlineshop#"}

ontology banking2tradingMediator

axiom humanDefinition
  definedBy
    ?a[os#hasAccountNumber hasValue ?b,
       os#hasBankIdentificationCode hasValue ?c,
       os#hasInvoicingParty hasValue ?d] memberOf os#invoice
    and ?d[os#hasSurname hasValue ?e,
           os#hasPrename hasValue ?f] memberOf os#legallyCompetentPerson
    implies exists
    ?z[ba#hasName hasValue ?e,
       ba#hasFirstName hasValue ?f,
       ba#hasAccountNumber hasValue ?b,
       ba#hasBankCode hasValue ?c] memberOf ba#account.
```

Listing 45: Mediation mit WSML

Es definiert, dass für jede Instanz „?a“des Konzepts „invoice“ mit den Attributwerten „hasAccountNumber = ?b“, „hasBankIdentificationCode = ?c“ und „invoicingParty = ?d“ – wobei ?d wiederum eine Instanz des Konzepts „legallyCompetentPerson“ mit den Attributwerten „hasSurname = ?e“ und „hasPrename = ?f ist – eine Instanz „?z“ des Konzepts „account“ mit den Attributwerten „hasName = ?e“, „hasFirstName = ?f“, „hasAccountNumber = ?b“ und „hasBankCode = ?c“ existiert. Durch dieses Axiom wird einer Ontology, die eine Instanz des Konzepts „invoice“ beinhaltet, automatisch eine entsprechende Instanz des Konzepts „account“ hinzugefügt.

Ontologien beschreiben jeweils eine bestimmte Domäne, d.h. einen Teilausschnitt der Welt. Wie bereits in Abschnitt 2.2.2 gezeigt, findet Mediation zwischen Prozessen, die zur Beschreibung der gleichen Domäne unterschiedliche Ontologien verwenden, im SSB statt.

Erstrecken sich jedoch Prozesse über mehrere Domänen, so ist aus den folgenden Gründen auch Mediation innerhalb des Prozesses nötig: Domänen werden in der Regel unabhängig voneinander modelliert. Das hat zur Folge, dass sie zum Teil auch gleiche oder zumindest ähnliche Dinge beschreiben, allerdings meist aus unterschiedlichen Blickwinkeln. Für Dinge, die in mehreren Ontologien beschrieben werden, kann über Mediation definiert werden, was ihre Bedeutung in einem anderen Zusammenhang, d.h. in einer anderen Ontologie ist. Da es nicht möglich bzw. sinnvoll ist, für jeden Prozess eine passende Ontologie zu definieren, die alle relevanten Aspekte potentiell unterschiedlicher Domänen enthält, kommen bei Prozessen, die sich über mehrere Domänen erstrecken zur Annotation der Daten und Aktivitäten unterschiedliche, domänenspezifische Ontologien zum Einsatz, zwischen denen entsprechende Mediatoren existieren. Die Verwendung dieser Mediatoren ist durch die präsentierte Erweiterung der `<bpel:assign>`-Aktivität möglich.

5.5.2 Entscheidungen im Kontrollfluss auf Basis ontologischen Wissens

Neben der Datenmanipulation kann ontologisches Wissen über die Bedeutung von Variablen auch verwendet werden, um Entscheidungen im Kontrollfluss zu treffen, die über Boolesche Ausdrücke spezifiziert werden. Im Einzelnen sind dies die Ausdrücke in `<joinCondition>`- und `<transitionCondition>`-Elementen, Abbruchbedingung in `<bpel:while>`- oder `<bpel:repeatUntil>`-Aktivitäten und die Bedingung zur Spezifikation des auszuführenden Pfades einer `<bpel:if>`-Aktivität. Standardmäßig wird in WS-BPEL Xpath zur Definition von Ausdrücken verwendet. Zur Auswertung der Ausdrücke lässt sich jedoch auch eine andere Sprache spezifizieren. In Listing 46 ist die Syntax für die Sprache WSML4BPEL dargestellt. Soll sie zur Auswertung von Ausdrücken verwendet werden, so muss das Attribut expressionLanguage den Wert „http://www.iaas.unistuttgart.de/bpel4sws/wsml4bpel“ aufweisen.

```
wsml4bpel = varList ‘;’ http://www.wsml.org/wsml/wsml-syntax#wsmlVariant
            http://www.wsml.org/wsml/wsml-syntax#namespace
            http://www.wsml.org/wsml/wsml-syntax#log_expr
varList = var ‘ ’ varList | var
```

Listing 46: Syntax von WSML4BPEL in BNF

WSML4BPEL verwendet in WSML formulierte logische Ausdrücke (`http://www.wsml.org/wsml/wsml-syntax#log_expr`), um Anfragen an eine Wissensbasis zu stellen. Die vorliegende WSML-Variante kann über `http://www.wsml.org/wsml/wsml-syntax#wsmlVariant` festgelegt werden. Die verwendeten Namensräume werden über `http://www.wsml.org/wsml/wsml-syntax#namespace` definiert. Die Wissensbasis wird durch eine Ontologie definiert, die folgendermaßen aufgebaut ist:

Sie enthält Instanzen, die dem Inhalt der über den Bezeichner `varList` angegebenen Variablen entsprechen. Diese werden aus den Variablen durch Transformationsvorschriften erstellt, auf die mittels der SAWSDL-Annotation `liftingSchemaMapping` verwiesen wird und sind somit

Instanzen der mittels der SAWSDL-Annotation `modelReference` referenzierten Konzepte. Außerdem werden die Ontologien, in denen die referenzierten Konzepte definiert sind, importiert.

Die Variablen, die als Eingabe für den logischen Ausdruck angegeben sind, müssen in dem Geltungsbereich, in dem der Ausdruck definiert und verwendet wird, sichtbar sein. Außerdem müssen die für die Variablendefinition verwendeten Datentypen mit der SAWSDL-Annotation `modelReference` versehen sein und ein `liftingSchemaMapping` definiert haben. Sind die aus den Eingabevariablen generierten Instanzen Teil der Ergebnismenge, so wird die Bedingung zu „true“ ausgewertet. Sind sie jedoch nicht Teil der Ergebnismenge, so wird die Bedingung zu „false“ ausgewertet. Listing 47 zeigt einen Ausschnitt des Prozesses „performanceReviewHandlingProcess“, der WSML4BPEL zur Auswertung zweier `transitionConditions` verwendet. Die Aktivität „receiveEmployeeDetails“ empfängt eine Nachricht vom Typ „ns:Employee“, der über die SAWSDL-Annotation `modelReference` auf das Konzept „performanceReviewRequest“ in der Ontologie „humanResourceManagement“ verweist und speichert sie in der Variablen „employee“. Sie hat zwei ausgehende `<bpel:link>`s: „receive-to-man“ und „receive-to-woman“. Das Attribut `expressionLanguage` des Elements `<bpel:transitionCondition>` der beiden `<bpel:link>`s hat den Wert „http://www.iaas.uni-stuttgart.de/bpel4sws/wsml4bpel“, d.h. zur Auswertung der Ausdrücke kommt WSML4BPEL zum Einsatz. Für den link „receive-to-man“ ist der Ausdruck „employee; wsmlVariant _“http://www.wsmo.org/wsml/wsml-syntax/wsml-rule“ namespace {ho _“http://example.org/ontologies/humanOntology#humanOntology“} ?x memberOf ho#Man“ angegeben. Es wird zur Evaluierung des Ausdrucks eine Ontologie erzeugt, die aus der Variablen „employee“ generierte Instanzen des in der Ontology „humanResourceManagement“ definierten Konzepts „performanceReviewRequest“ enthält und die Ontologie „humanOntology“ importiert. Handelt es sich um die Personaldaten eines Mannes, so evaluiert der Ausdruck zu „true“. Die Auswertung der `transitionCondition` des `<bpel:link>`s „receive-to-woman“ erfolgt analog.

```
<process xmlns:ns="http://example.org/humanResourceManagement" ...>
  ...
  <variables>
    <variable name="employee" element="ns:Employee" />
    ...
  </variables>
  ...
  <flow>
    <links>
      <link name="receive-to-man" />
      <link name="receive-to-woman" />
    </links>
    ...
    <extensionActivity>
      <bl:interactionActivity name="receiveEmployeeDetails"
                              conversation="perfomanceReviewConv">
        <bl:output variable="employee"
                   messageLabel="employeeDetails" />
        <sources>
          <source linkName="receive-to-man" >
            <transitionCondition
              expressionLanguage="http://www.iaas.unistuttgart.de/bpel4sws/
                                  wsml4bpel">
              employee; wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/
              wsml-rule" namespace {ho _"http://example.org/ontologies/
              humanOntology#humanOntology"} ?x memberOf ho#Man
            </transitionCondition>
          </source>
          <source linkName="receive-to-woman" >
            <transitionCondition
              expressionLanguage="http://www.iaas.unistuttgart.de/bpel4sws/
                                  wsml4bpel" >
              employee; wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/
              wsml-rule" namespace {ho _"http://example.org/ontologies/
              humanOntology#humanOntology"} ?x memberOf ho#Woman
            </transitionCondition>
          </source>
        </sources>
      </b4s:interactionActivity>
    </extensionActivity>
    ...
  </flow>
</process>
```

Listing 47: BPEL4SWS-Prozess mit in WSML formulierten logischen Ausdrücken zur Spezifikation von `transitionConditions`

Durch die gezeigte Erweiterung ist es unter der Vorraussetzung, dass die ontologischen Grundlagen existieren, d.h. dass entsprechende Domänenmodelle existieren und deren Beziehung zueinander definiert ist, möglich, über implizites Wissen Entscheidungen zu fällen. Im vorliegenden Beispiel ist für den Prozessmodellierer beispielsweise irrelevant wie einer

Instanz der Variablen „employee“ zu entnehmen ist, ob sie eine Frau oder einen Mann repräsentiert. Er kann seine Abfrage ontologiebasiert formulieren, die miteinander vernetzten Ontologien nutzen und somit zur Laufzeit Entscheidungen auf Basis ontologischen Wissens und nicht auf Basis struktureller Informationen herbeiführen.

5.6 Methodologie: Transformationen zwischen WS-BPEL und BPEL4SWS

In Kapitel 3.6 wurde bereits die Transformation zwischen WS-BPEL-Prozessmodellen und allgemeinen BPEL^{light}-Prozessmodellen gezeigt. Dieses Kapitel geht darauf ein worauf bei der Transformation geachtet werden muss, wenn es sich nicht um einen allgemeinen BPEL^{light}-Prozess handelt, sondern um einen BPEL4SWS-Prozess, d.h. einen BPEL^{light}-Prozess, dessen Aktivitäten entweder durch Semantische Web-Services implementiert werden oder die Funktionalität des Prozesses als Semantischen Web Service zur Verfügung zu stellen.

5.6.1 Von WS-BPEL zu BPEL4SWS

Durch die in Abschnitt 3.6 präsentierte Transformation von WS-BPEL zu BPEL^{light} entstehen ohne menschliches Einwirken pro Partnerlink ein Partner und mehrere bilaterale Konversationen bestehend aus jeweils maximal zwei Nachrichten. Durch menschliches Eingreifen ist es jedoch möglich, die Konversationen durch übergeordnete Konversationen zusammenzufassen, so dass Konversationen aus mindestens zwei Nachrichten entstehen, von denen mindestens eine gesendet und eine empfangen wird. Je nach Granularität des ursprünglichen Partnerlinks, kann auch der ganze Partnerlink über eine Konversation repräsentiert werden. Die dadurch entstehenden Konversationen können mit einem WSMO-Goal bzw WSMO-Web-Service annotiert werden. Außerdem kann für jede der Konversationen dem Partner der entsprechende Partnerlinktyp zugeordnet werden und somit die vollständige Abbildung auf Web-Service-Technologie spezifiziert werden. Alternativ kann auch nur diejenige Rolle des Partnerlinks, die vom BPEL-Prozess implementiert wird, verwendet werden, um eine partielle Abbildung zu spezifizieren. Die

Nachrichten der ursprünglich generierten Konversationen werden auf die Operation, die während der Transformation als Grundlage für die Definition der generierten Konversation dient, abgebildet. Wird eine vollständige Abbildung benötigt, so wird auch auf die vom Kommunikationspartner anzubietenden Operationen abgebildet. Im Fall einer partiellen Abbildung wird nur auf die vom BPEL-Prozess implementierten Operationen abgebildet. Man kann also einen Partnerlink in BPEL durch einen oder mehrere dynamische Partnerlinks in BPEL4SWS austauschen. Dies ist allerdings im Allgemeinen nicht vollständig automatisierbar. Es müssen zum einen die Transformation von BPEL nach BPEL^{light} dahingehend beeinflusst werden, dass zur Annotation geeignete Konversationen entstehen. Zum anderen müssen WSMO-Beschreibungen erstellt und entsprechende Konversationen mit ihnen annotiert werden. Schließlich müssen Transformationen spezifiziert werden, die ein Lifting und Lowering der in XML beschriebenen Daten zu Ontologieinstanzen und zurück ermöglichen. Auf diese Transformationen muss mit Hilfe der entsprechenden SAWSDL-Annotationen verwiesen werden. Soll zusätzlich von Ontologie-basierter Mediation bzw. Reasoning Gebrauch gemacht werden, so muss dies ebenfalls von Hand implementiert bzw. modelliert werden.

5.6.2 Von BPEL4SWS zu WS-BPEL

Ebenso wenig wie ein WS-BPEL-Prozess ohne menschliches Eingreifen in einen BPEL4SWS-Prozess transformiert werden kann, ist es möglich, einen allgemeinen BPEL4SWS-Prozess vollautomatisch in einen WS-BPEL-Prozess zu überführen. Für diejenigen Konversationen, für die eine vollständige Abbildung auf WSDL definiert wurde, ist die Transformation analog zur Beschreibung in Abschnitt 4.4 durchzuführen. Für die Konversationen, für die eine partielle Abbildung auf WSDL definiert wurde, muss zunächst die WSDL-Beschreibung eines passenden Kommunikationspartners ermittelt werden. Dies ist u.a. möglich, indem man die semantische Beschreibung der Konversation dazu verwendet, einen entsprechenden Dienst zu finden, dessen semantische Beschreibung ein WSDL-Grounding spezifiziert. Anhand dieser Informationen kann dann ein entsprechender Partnerlinktyp definiert werden. Dieser Partnerlinktyp

wird im Partnerlink verwendet, der nach der Transformation die ursprüngliche Konversation repräsentiert. Die Aktivitäten, die den Nachrichtenaustausch der BPEL4SWS-Konversation spezifizieren, werden unter Berücksichtigung der zu verwendenden Operationen bzw. deren Typ in entsprechende WS-BPEL-Aktivitäten überführt. Wird im BPEL4SWS-Prozess von Reasoning bzw. Ontologie-basierter Mediation Gebrauch gemacht, so müssen die entsprechenden Ausdrücke bzw. Transformationen auf XML-Basis definiert werden.

5.7 Zusammenfassung

In diesem Kapitel wurde die Sprache BPEL4SWS vorgestellt. Da existierende SWS-Frameworks keine für Geschäftsprozesse typische, asynchrone Kommunikation auf Basis der Web-Service-Technologie entsprechend des WS-I Basic Profiles unterstützen, wurde zunächst das WSMO-Framework modifiziert, um derartige Kommunikation zu ermöglichen und die notwendige Funktionalität eines semantischen Service Busses definiert. Anschließend wurde gezeigt, wie Aktivitäten von BPEL4SWS-Prozessen mit semantischen Beschreibungen derart annotiert werden können, dass mittels dieser Beschreibungen Dienste aufgefunden werden können, die Aktivitäten von BPEL4SWS-Prozessen implementieren, bzw. dass mittels dieser Beschreibungen der als Dienst angebotene, Funktionalität realisierende Prozess aufgefunden werden kann.

In Tabelle 2 sind die Kommunikationsszenarien, die sich aus dem Einsatz semantischer Technologien ergeben sowie deren Unterstützung von WS-BPEL und BPEL4SWS aufgelistet.

Kriterien	WS-BPEL	BPEL4SWS
Auffinden des Prozesses mittels semantischer Technologien und anschließender synchroner Aufruf	Ja	Ja
Auffinden eines als Aktivitätsimplementierung geeigneten Dienstes mittels semantischer Technologien mit anschließendem synchronen Aufruf	Nein	Ja
Auffinden des Prozesses mittels semantischer Technologien und anschließend asynchroner Aufruf	Nein	Ja
Auffinden eines als Aktivitätsimplementierung geeigneten Dienstes mittels semantischer Technologien mit anschließendem asynchroner Aufruf	Nein	Ja

Tabelle 2: Vergleich von WS-BPEL und BPEL4SWS

Wird ein Dienst über eine Request-Response-Operation aufgerufen, so ist es für den aufgerufenen Dienst zwar erkennbar – für die Ausführung jedoch irrelevant – ob die Operation über semantische Technologien aufgefunden wurde oder nicht. Aus diesem Grund sind nicht nur BPEL4SWS-Prozesse in der Lage einen synchronen Aufruf einer Request-Response-Operation, dem semantische Suche vorausging, durch zwei `<bl:interactionActivity>`s zu verarbeiten sondern auch WS-BPEL-Prozesse durch ein `<bpel:receive>`-`<bpel:reply>` Paar. Anders verhält es sich wenn ein WS-BPEL-Prozess asynchron aufgerufen wird. Für den Call-back wird in diesem Fall eine `<bpel:invoke>`-Aktivität verwendet, welche die Angabe eines Porttypen und einer Operation zwingend vorsieht. Porttyp und Operation

können also nicht über semantische Technologien dynamisch zur Laufzeit ermittelt werden, sondern werden schon bei der Modellierung fixiert. Die `<bpel:invoke>`-Aktivität kommt auch in den Szenarien zum Einsatz, in denen der Prozess einen Dienst aufruft. Daher wird von WS-BPEL weder der asynchrone Aufruf eines Prozesses noch der synchrone bzw. asynchrone Aufruf eines Dienstes im Anschluss an erfolgreiche semantische Suche unterstützt.

BPEL4SWS baut auf das von WSDL unabhängige Interaktionsmodel von BPEL^{light} auf, erweitert die in Kapitel 4 präsentierte Zuordnungslogik der Aktivitäten zur Kommunikationsinfrastruktur und verwendet SWS-Ansätze (z.B. SAWSDL und WSMO) zur semantischen Beschreibung der Interaktion des Prozesses mit seinen Geschäftspartnern. Aufgrund der Abstraktion von WSDL ist BPEL4SWS in der Lage, alle vier genannten Szenarien zu unterstützen. Dadurch, dass BPEL4SWS auf der Zuordnungslogik von BPEL^{light} zur Web-Service-Technologie aufbaut und diese lediglich erweitert, können mit BPEL4SWS auch alle in Kapitel 4 vorgestellten Szenarien umgesetzt werden: wie WS-BPEL ermöglicht auch BPEL4SWS synchrone und asynchrone Kommunikation mit Geschäftspartnern basierend auf Web-Service-Technologie, unabhängig davon, ob die Kommunikation vom Klienten oder vom Prozess ausgeht.

Bezüglich der Flexibilität ist das in diesem Kapitel vorgestellte BPEL4SWS dem herkömmlichen WS-BPEL vorzuziehen. Bei BPEL4SWS kann, wie beim im vorigen Kapitel vorgestellten BPEL^{light} mit Web-Services, durch eine händische Modifikation der Abbildungsvorschrift von Prozesslogik auf Schnittstellenspezifikation eine Aktivitätsimplementierung verwendet werden, die einen anderen Porttypen und eine andere Operation implementiert als die zuvor verwendete. Das Prozessmodell selber muss hierzu nicht angepasst werden. Die Änderung kann entweder durch ein erneutes Deployment oder gar während der Laufzeit vorgenommen werden. Darüber hinaus können Aktivitätsimplementierungen durch den Einsatz semantischer Technologien automatisch während der Laufzeit unabhängig von ihrer implementierten Schnittstelle (Porttyp und Operation) aufgefunden, gebunden und aufgerufen werden.

Durch den Einsatz semantischer Technologien erhöht sich jedoch nicht nur die Flexibilität des Prozesses bezüglich der Kommunikationspartner. Sie kann – wie in diesem Kapitel gezeigt – auch verwendet werden, um Datenmanipulation innerhalb eines Prozesses zu vereinfachen und Entscheidungen im Kontrollfluss unabhängig von der Struktur der Daten, auf Basis ontologischen Wissens zu formulieren.

Die Interaktionsmodelle und ggf. Abbildungsvorschriften der Prozesslogik auf die Kommunikationsinfrastruktur von WS-BPEL und BPEL4SWS lassen sich aufeinander abbilden. Das bedeutet, dass eine Migration von WS-BPEL-Prozessen zu BPEL4SWS durch eine Transformation vorgenommen werden kann. Auch nach der Migration, können die Prozesse immer noch konventionell aufgerufen werden – sowohl synchron als auch asynchron – auch wenn sie über eine semantische Beschreibung verfügen. Sie sind also gleichzeitig sowohl als semantischer Web-Service als auch als herkömmlicher Web-Service verfügbar. Klienten, die gegen den vom Prozess angebotenen Porttypen gebunden sind, können mit dem Prozess somit auch nach der Migration kommunizieren. Dadurch ist gewährleistet, dass durch die Migration keine potentiellen Klienten verloren gehen.

Die in diesem Kapitel präsentierten Ergebnisse finden sich auch zum Teil in den bereits veröffentlichten Publikationen [NLKL07c, LDN+07, NiNo08, KLL+08, NLKL09].

KAPITEL 6

ARCHITEKTUR UND IMPLEMENTIERUNG

In den vorangegangenen Kapiteln wurde die Sprache BPEL^{light} eingeführt und das Zusammenspiel von BPEL^{light} mit unterschiedlichen Servicetechnologien auf konzeptioneller Ebene betrachtet. In diesem Kapitel wird die Architektur und Implementierung einer Infrastruktur gezeigt, die es ermöglicht BPEL^{light}-Prozesse zu modellieren und anschließend auszuführen.

In Abbildung 51 ist diese die Modellierung und Ausführung von BPEL^{light}-Prozessen unterstützende Infrastruktur dargestellt. Sowohl in der Modellierungsumgebung als auch in der Ausführungsumgebung gibt es BPEL^{light} spezifische Komponenten und Komponenten, die spezifisch für die verwendete Servicetechnologie sind. Die Modellierungsumgebung gliedert sich in einen BPEL^{light} Editor, die Deployment Komponente und Komponenten, die von der verwendeten Servicetechnologie abhängen: Editoren zur Erstellung der IDLs und Editoren zur Definition des Mappings von BPEL^{light} auf die entsprechende IDL in einem Deployment Deskriptor (DD). Eine ähnliche Aufteilung findet sich in der Ausführungsumgebung wieder: Neben der BPEL^{light}-spezifischen Engine gibt es Komponenten des Service Busses, die von der jeweiligen Servicetechnologie abhängen, u.a. die Discovery Komponente, das Repository, die Vermittlungs- (bzw. Mediation-) Komponente, die zur Verfügung stehenden Dienste, die fachliche Funktionalität realisieren sowie die Integrationsschicht der BPEL^{light}-Engine, welche die Abbildung von BPEL^{light} auf die Servicetechnologie übernimmt und somit die BPEL^{light}-Engine in die Ausführungsinfrastruktur einbettet.

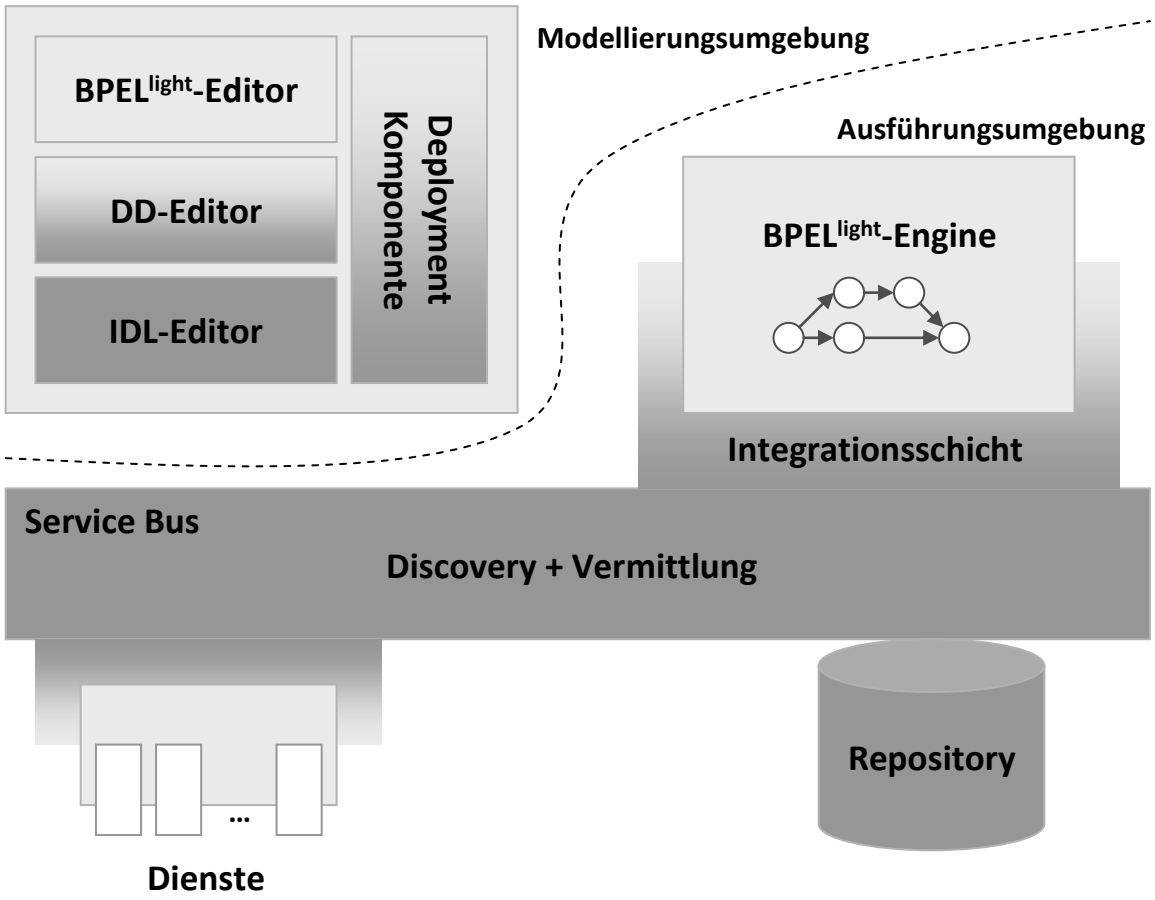


Abbildung 51: Gesamtarchitektur

Das Kapitel ist wie folgt aufgebaut: Abschnitt 6.1 widmet sich der Beschreibung der Ausführungsumgebung für BPEL^{light}-Prozesse, Abschnitt 6.2 der Beschreibung der Modellierungsumgebung. Die Architektur und Implementierung der BPEL^{light} spezifischen Komponenten, d.h. der BPEL^{light}-Engine und des BPEL^{light} Editors ist in Abschnitt 6.1.2 bzw. Abschnitt 6.2.2 beschrieben. Je nach verwendeter Servicetechnologie müssen im Modellierungstool neben dem BPEL^{light}-Prozess weitere Artefakte generiert und auf die Infrastruktur deployt werden. Das Deployment und die Implementierung der Integrationsschicht und damit die Anbindung an unterschiedliche Servicetechnologien wird in Abschnitt 6.1.3 und 6.1.4 beschrieben, die Modellierung der entsprechenden Artefakte wird in Abschnitt 6.2.3 und 6.2.4 beschrieben.

6.1 Ausführungsumgebung für BPEL^{light}-Prozesse

Zur Ausführung von BPEL^{light}-Prozessen wird eine Middleware benötigt, die in der Lage ist, BPEL^{light}-Prozessmodelle einzulesen, zu interpretieren und Instanzen des Prozessmodells auszuführen. Die Ausführung umfasst das Navigieren durch den Kontrollfluss des Prozesses und die Kommunikation mit den angegebenen Partnerdiensten. Da die Prozessausführung meist eine unternehmenskritische Angelegenheit ist und die meisten Geschäftprozesse langlebig sind, d.h. mehrere Stunden, Tage oder gar Wochen andauern, ist zudem nicht nur die Persistierung der Modelldaten sondern auch der Instanzdaten unabdingbar, um im Falle eines Systemabsturzes nach Systemneustart die Fortführung der begonnenen Prozesse gewährleisten zu können.

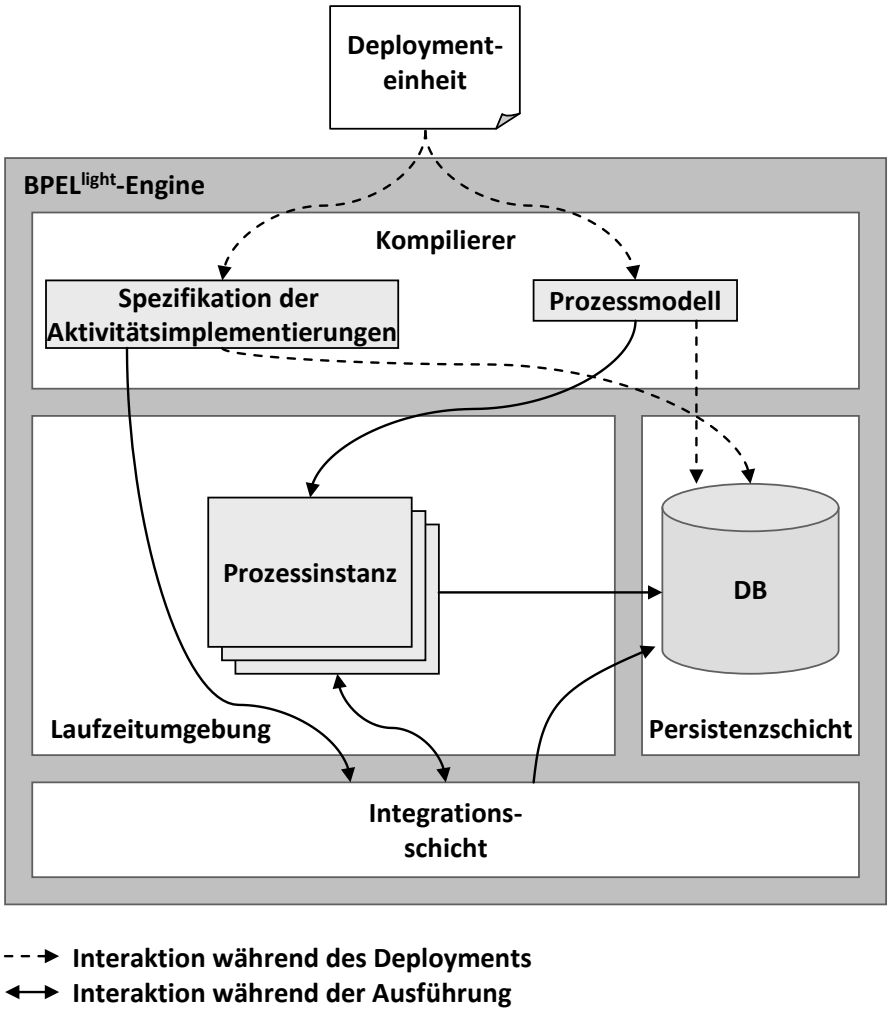


Abbildung 52: Allgemeine Architektur einer BPEL^{light}-Engine

Die sich daraus ergebende allgemeine Architektur einer BPEL^{light}-Engine ist in Abbildung 52 dargestellt. Der Kompilierer erstellt aus einer Deploymenteinheit zu jeder BPEL^{light}-Prozessbeschreibung eine interne Repräsentation des Prozessmodells und stellt der Integrationsschicht die beim Deployment angegebenen Informationen zur Abbildung von BPEL^{light}-Aktivitäten zu Aktivitätsimplementierungen zur Verfügung. Aus dem Prozessmodell werden Prozessinstanzen erzeugt, durch die während der Ausführung navigiert wird. Die Laufzeitumgebung ermöglicht das Navigieren durch den Kontrollfluss einer Prozessinstanz. Sie implementiert u.a. die strukturierten BPEL-Aktivitäten und die BPEL^{light} Interaktionsaktivitäten. Die Integrationsschicht realisiert die Kommunikation der Prozessinstanzen mit den Partnerdiensten, d.h. sie nimmt die Zuordnung von Interaktionsaktivitäten zu Kommunikationspartnern vor. Die Persistenzschicht persistiert schließlich alle relevanten Modell- und Instanzinformationen. Aufgrund der Komplexität von BPEL^{light} und der Fülle an Anforderungen an eine Ausführungsumgebung für BPEL^{light}-Prozesse wurde im Rahmen dieser Arbeit von einer Neuimplementierung einer BPEL^{light}-Engine abgesehen. Stattdessen wurde eine existierende Open-Source BPEL-Engine zu einer BPEL^{light}-Engine erweitert. Zum Zeitpunkt der Implementierung waren lediglich zwei Open-Source BPEL-Engines vorhanden: die unter GPL[9] (Gnu General Public License) stehende ActiveBPEL-Engine[10] und die unter der Apache License 2.0[11] stehende Orchestration Director Engine (ODE)[12]. ActiveBPEL ist jedoch im Gegensatz zu ODE lediglich in der Lage, BPEL-Prozesse der Version 1.1 zu interpretieren und auszuführen. Da BPEL^{light} sich jedoch der erst in Version 2.0 eingeführten Erweiterungsmechanismen bedient, fiel die Wahl bezüglich der zu erweiternden Engine auf ODE. Im Folgenden werden die vorgenommenen Erweiterungen von ODE beschrieben. Weitere Einzelheiten sind [So08] zu entnehmen.

---

[9] `http://www.gnu.org/licenses/gpl.html`

[10] `http://activebpel.org`

[11] `http://www.apache.org/licenses/LICENSE-2.0.html`

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

6.1.1 Apache ODE

Die Apache Orchestration Director Engine (ODE) ist eine Open-Source Workflowengine zur Ausführung von BPEL-Prozessen. ODE kann sowohl Prozesse ausführen, die dem OASIS Standard WS-BPEL 2.0 [AAA+07] folgen, als auch Prozesse, die der Spezifikation BPEL4WS 1.1 [CGKL03] entsprechen. Die Prozesse werden als Teil einer Deploymenteinheit (siehe Abbildung 53) in ODE deployt. Die Artefakte der Deploymenteinheit für ODE sind im Einzelnen:

- Eine `.bpel`-Dateien, d.h. ein Prozessmodel in BPEL 1.1 oder 2.0.
- Eine oder mehrere `.wsdl`-Dateien, aus denen Elemente im Prozessmodell verwendet werden.
- Optional eine oder mehrere `.xsd` Dateien, aus denen Beschreibungen zur Definition der Datentypen von Variablen im BPEL-Prozess und Nachrichten in WSDL-Beschreibungen verwendet werden. Die `.xsd`-Dateien sind optional, weil Datentypdefinitionen auch in `.wsdl`-Dateien vorgenommen werden können. BPEL-Prozesse können `.xsd`-Dateien importieren, um die Datentypen von Variablen zu definieren. Sie können aber auch Datentypdefinitionen aus einer `.wsdl`-Datei verwenden.
- Eine `.dd`-Dateien, d.h. ein Deployment Deskriptor, der weitere zur Kompilierung nötige Informationen beinhaltet. Dies ist u.a. die Abbildung von Endpunkten (`<wsdl:port>`) auf Partnerlinks.

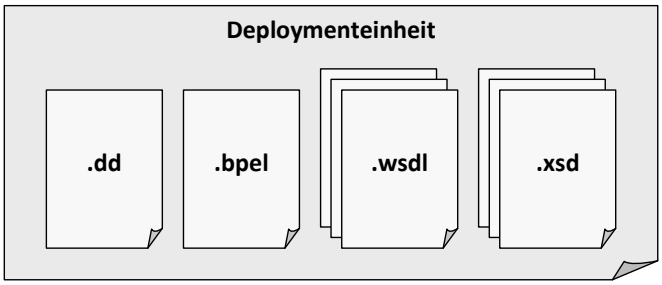


Abbildung 53: Deploymenteinheit für ODE

6.1.1.1 ODE Architektur

Die Architektur von ODE ist in Abbildung 54 dargestellt. Sie gliedert sich in: (i) den Kompilierer, der BPEL-Prozesse in eine interne Objektrepräsentation (O-Model) überführt, (ii) die Laufzeitumgebung, die Instanzen der BPEL-Prozesse ausführt, (iii) die Integrationsschicht, welche die Kommunikation mit Partnerdiensten ermöglicht, und (iv) die Persistenzschicht, die unter anderem den aktuellen Status der Prozessinstanzen speichert und somit die Wiederaufnahme von Prozessinstanzen nach einem Systemabsturz ermöglicht.

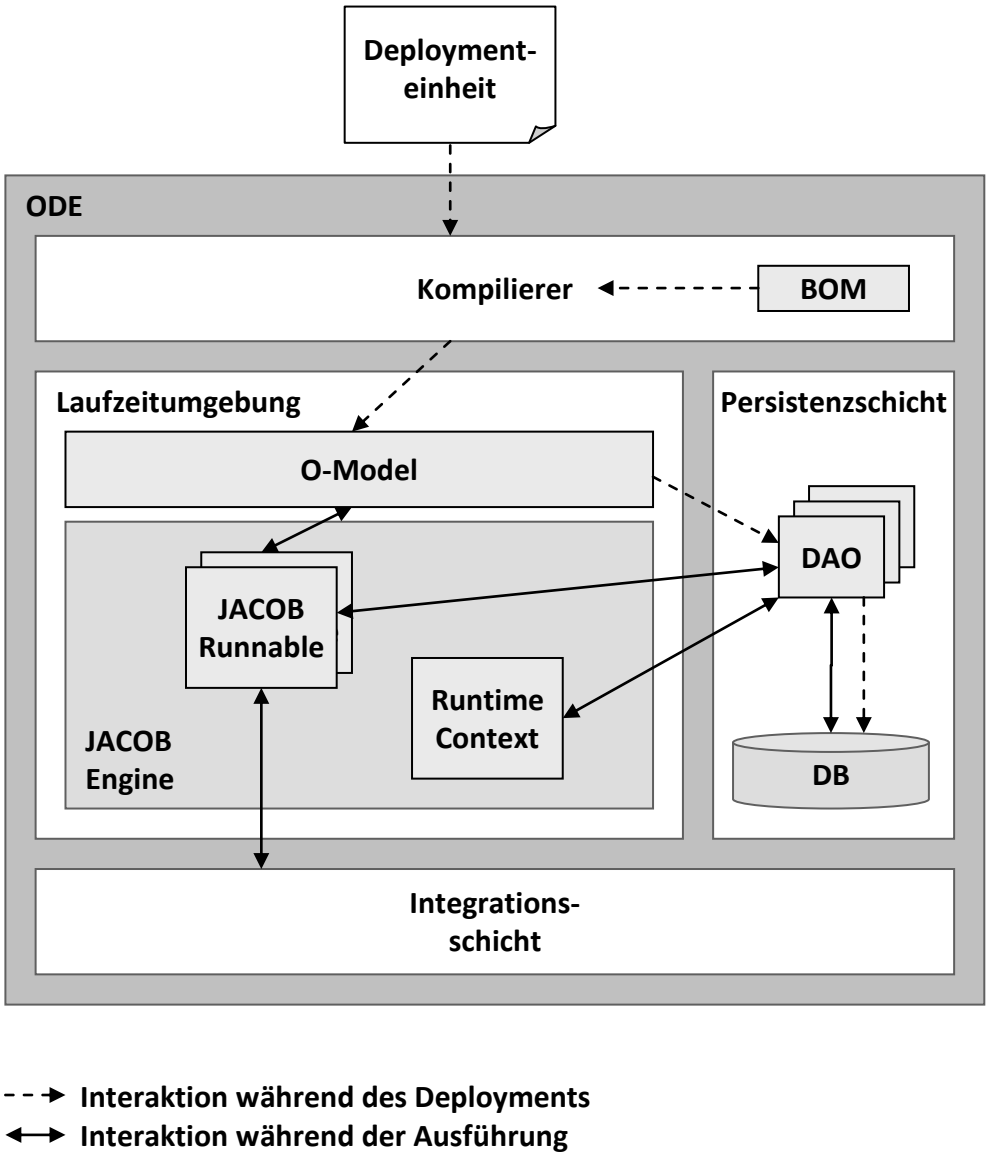


Abbildung 54: Architektur von ODE

(i) Kompilierer

Der BPEL-Kompilierer überführt einen in einer `.bpel`-Datei serialisierten BPEL-Prozess mit Hilfe eines BPEL Objekt Models (BOM) in eine interne Objektrepräsentation, das sogenannte O-Model. Das O-Model vereint alle in der `.bpel`-Datei und den referenzierten `.wsdl`- und `.xsd`-Dateien enthaltenen Informationen in einem Model und enthält somit alle zur Ausführung benötigten (statischen) Informationen für ein Prozessmodell. Das BOM repräsentiert im Gegensatz zum O-Model nicht ein konkretes Prozessmodell, sondern definiert für jedes BPEL-Sprachelement die implementierende Klasse, ermöglicht somit also die Erzeugung von O-Model-Objekten, die dem in der `.bpel`-Datei serialisierten BPEL-Prozess entsprechen. Der Kompilierer verfügt über eine Web-Service-API und ermöglicht somit, dass unterschiedliche Modellierungswerkzeuge Prozesse in die Workflowengine deployen können. Eine weitere Möglichkeit Prozesse in ODE zu deployen besteht darin, die Deploymenteinheit in das Verzeichnis `%DEPLOYMENT_ROOT%/WEB-INF/processes` zu kopieren. Dieses Verzeichnis wird vom Kompilierer überwacht. Wird eine neue Deploymenteinheit hineinkopiert, so wird der enthaltene Prozess deployt. Das Löschen einer Deploymenteinheit aus dem Verzeichnis verursacht ein Undeployment des entsprechenden Prozesses.

(ii) Laufzeitumgebung

Die Laufzeitumgebung ist die Kernkomponente von ODE und führt schließlich die kompilierten Prozesse aus. Die Ausführung von BPEL-Prozessen basiert auf *JACOB* (Java Concurrent Objects). JACOB implementiert Nebenläufigkeit auf Applikationsebene, d.h. in einem Thread, ist also in der Lage, die Ausführung einer Prozessinstanz zu unterbrechen, ihren aktuellen Status zu persistieren, anschließend andere Prozessinstanzen auszuführen, um später mit der Verarbeitung der unterbrochenen Instanz fortzufahren. Die nebenläufige Ausführung unterschiedlicher Instanzen geschieht dabei transparent für die einzelnen Instanzen. JACOB ist eine Java Implementierung des ACTORS [HBS73, Cl81, Ag86] Nebenläufigkeitmodels, d.h. es nutzt Kommunikationskanäle zwischen Aktivitäten, um den Kontrollfluss zu realisieren. Wird

beispielsweise die Ausführung einer Aktivität innerhalb einer `<bpel:sequence>` beendet, so wird dies der `<bpel:sequence>` von der beendeten Aktivität über einen Kanal, den *TerminationChannel*, mitgeteilt. Die `<bpel:sequence>` entscheidet dann, wie weiter zu verfahren ist. War die beendete Aktivität die letzte Aktivität der `<bpel:sequence>`, so wird die `<bpel:sequence>` ebenfalls beendet. Andernfalls wird die nächste Aktivität der `<bpel:sequence>` ausgeführt. Zur Persistierung von Prozessinstanzen bedient sich JACOB der Persistenzschicht.

Beim Eintreffen einer entsprechenden Nachricht erstellt die Laufzeitumgebung aus dem das Prozessmodell repräsentierenden O-Model eine neue Instanz eines Prozessmodells in Form von Implementierungsobjekten, so genannten *JACOB Runnables*, und einen sogenannte *Runtime Context*, der u.a. den Zugriff auf Variablenwerte ermöglicht. Die JACOB Runnables werden zusammen mit dem Runtime Context an JACOB übergeben und anschließend ausgeführt.

(iii) Integrationsschicht

Die Integrationsschicht ermöglicht es, ODE mit der Außenwelt zu kommunizieren. Dies ist nicht nur zum Nachrichtenaustausch von Prozessinstanzen mit Partnerdiensten notwendig, sondern auch um Prozesse in die Engine deployen oder Managementfunktionen anbieten zu können. Im Gegensatz zur Laufzeitumgebung verwendet die Integrationsschicht auch den konkreten Teil der WSDL-Beschreibungen, d.h. `<binding>`- und `<service>`-Informationen. ODE bietet zwei Implementierungen der Integrationsschicht, Axis 2 and Java Business Integration, die im Folgenden beschrieben werden.

Apache Axis 2[13] ist eine Open-Source SOAP Implementierung, welche sowohl SOAP in den Versionen 1.1 und 1.2 unterstützt als auch REST-basierte Web-Service-Aufrufe. Die Axis 2 Integrationsschicht ermöglicht es, ODE als Web Archive (WAR) in einen Servletcontainer, z.B. in Apache

[13] `http://ws.apache.org/axis/`

Tomcat[14], eine Open-Source Implementierung der Java Servlet[15] und Java Server Pages (JSP)[16] Technologien, einzubetten.

Java Business Integration (JBI) ist eine Standard-basierte Architektur für EAI und B2B Integration [TeWa05]. Den Konzepten der SOA entsprechend stellt sie eine Infrastruktur zur Verfügung, um lose gekoppelte Komponenten (Dienste) verlässlich miteinander interagieren zu lassen. Die Interaktion beruht auf einem Nachrichtenaustauschmodel, das die in WSDL 2.0 definierten Nachrichtenaustauschmuster unterstützt und in dem Nachrichten zwischen Komponenten nie direkt ausgetauscht werden, sondern immer über einen so genannten „Normalized Message Broker" vermittelt werden. Die JBI Integrationsschicht bettet ODE als JBI Komponente in einen JBI Container wie beispielsweise Apache ServiceMix[17], ein Open-Source ESB mit integrierten JBI Komponenten für HTTP, JMS, SOAP, etc. ein.

(iv) Persistenzschicht

Um die Prozessausführung verlässlich zu gestalten, ist die Persistierung der Instanzdaten unabdingbar. Die zu sichernden Daten sind unter anderem Variablenwerte, EPRs in Partnerlinks und der Stand der Ausführung.

In ODE werden Data Access Objects (DAO) als Schnittstelle zur Persistenzschicht verwendet. Dadurch wird von der konkreten Implementierung der Persistenzschicht abstrahiert. Dieses Vorgehen ermöglicht es, unterschiedliche Persistierungsmechanismen zu verwenden. ODE bietet zwei unterschiedliche DAO Implementierungen. Die erste verwendet Hibernate[18], ein Open-Source Rahmenwerk, welches Objekte in eine relationale Datenbank persistiert. Die zweite basiert auf Apache OpenJPA[19], ein Projekt, welches die Java Persistence API (JPA)

[14] http://tomcat.apache.org/

[15] http://java.sun.com/products/servlet/

[16] http://java.sun.com/products/jsp/

[17] http://servicemix.apache.org

[18] https://www.hibernate.org/

[19] http://openjpa.apache.org/

implementiert, um Java Objekte abzuspeichern [BiOr06]. Analog zu Hibernate kann bei JPA mit den zu sichernden Objekten „normal“ gearbeitet werden, d.h. die Persistierung erfolgt im Hintergrund. Beide Implementierungen verwenden zur Persistierung ein Apache Derby DBMS[20].

Neben der Persistierung von Prozessinstanzen unterstützt ODE jedoch auch die nicht-persistente Ausführung von Prozessen. In diesem Fall wird ein Prozess durch sogenannte In-Memory-Objekte realisiert und die zu sichernden Informationen werden in der Laufzeitumgebung gehalten, werden also nicht persistiert. Dies erhöht zwar auf der einen Seite die Ausführungsgeschwindigkeit, hat aber auf der anderen Seite einen vollständigen Datenverlust bei Systemabsturz zur Folge. Für Prozesse, die In-Memory laufen gibt es einige Einschränkungen. So kann beispielsweise ihr Status nicht über Managementfunktionen abgefragt werden. Außerdem dürfen sie nur genau eine `<bpel:receive>`-Aktivität enthalten, und zwar diejenige, die den Prozess instantiiert. Es sind also nur kurzlebige Prozesse, sogenannte *Microflows* [LeRo00], erlaubt.

6.1.1.2 Erweiterbarkeit von ODE

BPEL ist als erweiterbare Sprache konzipiert und stellt neben einer allgemeinen Erweiterbarkeit mit der `<bpel:extensionActivity>` und der `<bpel:extensionAssignOperation>` zwei explizite Erweiterungspunkte zur Verfügung. ODE trägt dieser Erweiterbarkeit über explizite Erweiterungspunkte Rechnung und ermöglicht es, über sogenannte *Extension Bundles* Implementierungen für neue Aktivitätstypen und Datenmanipulationsmechanismen in ODE zu registrieren, ohne den ODE Quellcode anpassen zu müssen. Die Extension Bundles müssen sowohl die Funktionalität des BOM als auch des O-Models und der JACOB Runnables implementieren, d.h. sie müssen das Parsen und die Ausführung der entsprechenden Erweiterungselemente ermöglichen. Erweiterungen, die nicht über die beiden Erweiterungspunkte in die Engine eingebracht

[20] `http://db.apache.org/derby/`

werden können, erfordern eine Anpassung und Erweiterung des Quellcodes.

6.1.2 Erweiterung von Apache ODE zu einer BPEL^{light}-Engine

Um BPEL^{light}-Prozesse ausführen zu können, müssen an der Architektur von ODE keine grundlegenden Änderungen vorgenommen werden, sondern kleinere Anpassungen der Implementierung genügen. Es müssen u.a. die Sprachelemente `<bl:conversation>` und `<bl:partner>` aber auch die Aktivitäten `<bl:pick>` und `<bl:interactionActivity>` in die vorhandene Implementierung eingebracht werden. Die erforderlichen Erweiterungen der Implementierung betreffen alle Schichten von ODE. Das aus der Deploymenteinheit erzeugte Prozessmodell, das O-Model, ist frei von WSDL-Informationen, wenn zur Kommunikation mit Partnern lediglich das BPEL^{light} Interaktionsmodell verwendet wird. Gleiches gilt für die JACOB Runnables und den Runtime Context. Die Abbildung auf die Kommunikationsinfrastruktur wird ausschließlich in der Integrationsschicht umgesetzt. Die Erweiterung von ODE bewirkt also, dass die Dimensionen ‚wie' und ‚womit' durch unterschiedliche Komponenten der Architektur unterstützt werden. Die relevanten Änderungen am Kompilierer und an der Ausführungsumgebung sind Gegenstand dieses Abschnitts. Sie sind zum Teil in einem Extension Bundle für BPEL^{light} realisiert und zum Teil direkt im ODE Quellcode implementiert. Die Änderungen an der Integrationsschicht wird den Abschnitten 6.1.3 bzw. 6.1.4. beschrieben. Zunächst wird jedoch die erweiterte Deploymenteinheit beschrieben.

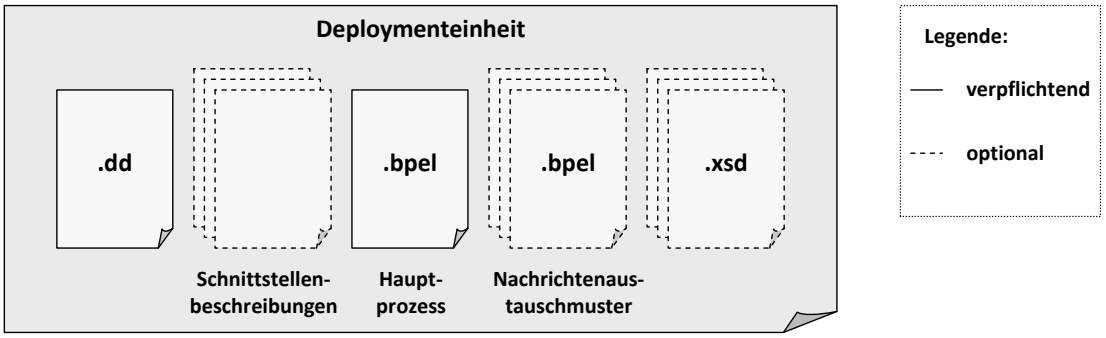


Abbildung 55: Deploymenteinheit für die BPEL^{light}-Engine

Abbildung 55 zeigt die erweiterte Deploymenteinheit für die auf ODE basierende BPEL^{light}-Engine. Die Artefakte der Deploymenteinheit sind im Einzelnen:

- Eine `.bpel`-Datei mit einer BPEL^{light}-Prozessbeschreibung (Hauptprozess). Optional weitere `.bpel`-Dateien mit Definitionen von Nachrichtenaustauschmustern in Form von abstrakten BPEL^{light}-Prozessen.

- Optional eine oder mehrere `.xsd` Dateien. Die `.xsd` Dateien sind optional, da BPEL^{light}-Prozesse `.xsd` Dateien importieren können, um die Datentypen von Variablen zu definieren. Sie können aber auch Datentypdefinitionen aus Schnittstellenbeschreibungen verwenden.

- Eine `.dd`-Dateien, d.h. ein Deployment Deskriptor, der weitere zur Kompilierung nötige Informationen beinhaltet. Dies umfasst u.a. eine Abbildung der BPEL^{light}-Prozesslogik auf eine abstrakte Schnittstellenbeschreibung (wie z.B. WSDL oder WSMO) und gegebenenfalls auch eine Abbildung auf konkrete Partnerinstanzen.

- Eine oder mehrere Dateien mit Schnittstellenbeschreibungen (z.B. WSDL oder WSMO). Auf die benötigten Schnittstellenbeschreibungen wird in Abschnitt 6.1.3 und 6.1.4 genauer eingegangen.

6.1.2.1 Umsetzung der BPEL^{light}-Aktivitäten

BPEL^{light}-Aktivitäten sind reguläre BPEL-Erweiterungen und nutzen den Erweiterungspunkt `<bpel:extensionActivity>`. Dementsprechend kann ihre Implementierung über die in Abschnitt 6.1.1.2 vorgestellten von ODE für die BPEL-Erweiterungspunkte bereitgestellten Erweiterungsmechanismen vorgenommen werden.

Das BPEL^{light} Extension Bundle wurde als eigenes Eclipse Projekt mit dem Namen “bpel-light“ implementiert und in ODE registriert. Die Klassenstruktur ist in Abbildung 56 dargestellt. Von weniger wichtigen Methoden, Attributen und Parametern wird in der Darstellung zur besseren Lesbarkeit abstrahiert. Bis auf die beiden Klassen `AbstractAsyncExtensionOperation` und `AbstractExtensionBundle`, die

Teil von ODE's Erweiterungspunktes sind, sind alle dargestellten Klassen im Paket `org.apache.ode.extension.bpellight` enthalten.

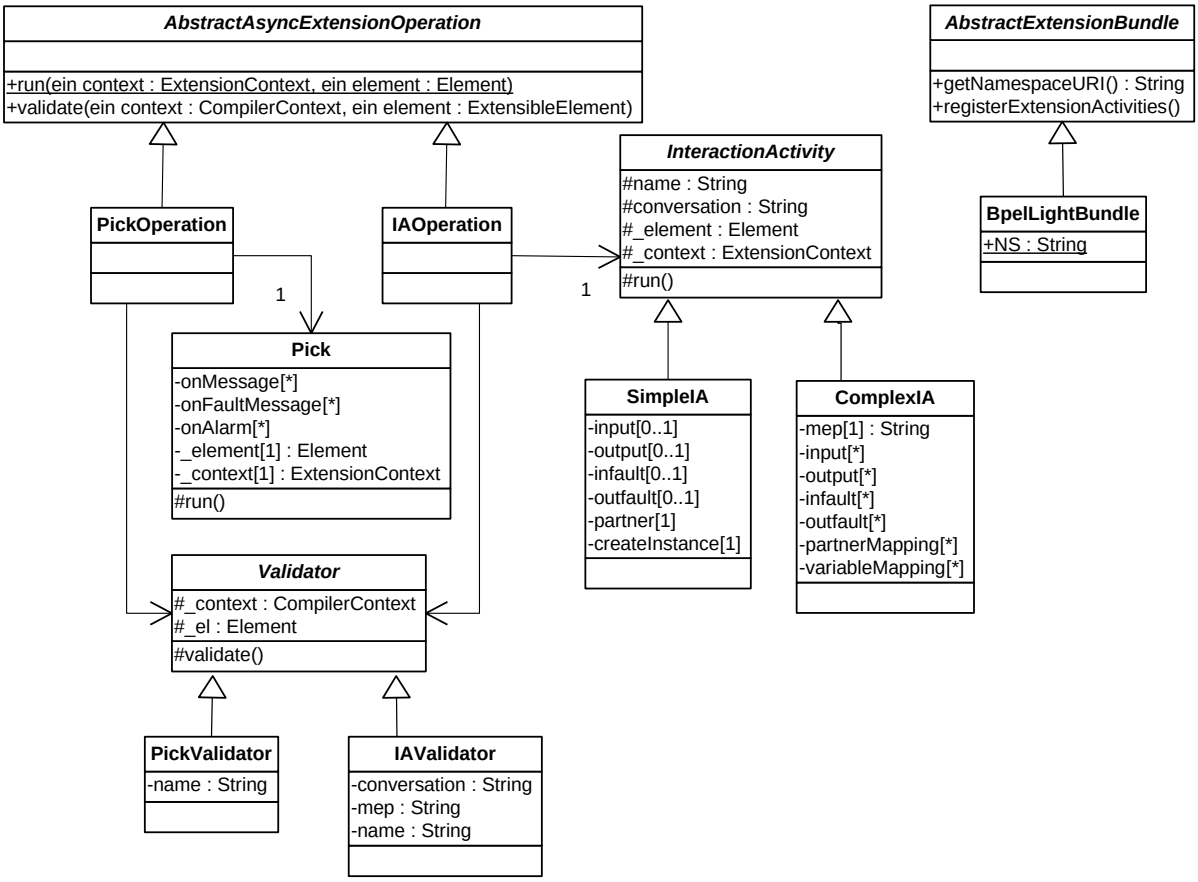


Abbildung 56: Klassenstruktur des ExtensionBundles „bpel-light" [So08]

Die Klasse `BpelLightBundle` bildet die BPEL^{light}-Aktivitäten `<bl:interactionActivity>` und `<bl:pick>` auf die Klassen `IAOperation` bzw. `PickOperation` ab. Diese Abbildung ermöglicht es, dem für die `<bpel:extensionActivity>` zuständigen BOM-Objekt während des Deployments entsprechende Objekte zu erzeugen, welche die BPEL^{light}-Aktivitäten im O-Model repräsentieren. Beide Klassen erweitern die Klasse `AbstractAsynchExtensionOperation`, die als Einstiegspunkt für asynchrone Aktivitäten dient. Von ihr werden zwei Methoden definiert: `validate()` zur Validierung des Erweiterungselements während der Kompilierung und `run()` zur Ausführung der Aktivität zur Laufzeit.

Beide Klassen delegieren die zu implementierende Funktionalität an Hilfsklassen. Die Validierung wird durch die Klasse `Validator`, die

gemeinsame Attribute validiert, sowie die Klassen `IAValidator` und `PickValidator` realisiert. Die Ausführungssemantik der `<bl:interactionActivity>` wird durch die Klasse `InteractionActivity` unter Einbeziehung der Klasse `SimpleIA` bzw. `ComplexIA` realisiert. Die Ausführungssemantik der `<bl:pick>`-Aktivität wird durch die Klasse `Pick` realisiert.

Die Klasse `InteractionActivity` hält alle der einfachen und komplexen `<bl:interactionActivity>` gemeinsamen Attribute und bietet unter anderem eine Methode, welche die kompilierte Repräsentation der referenzierten `<bl:conversation>` bzw. der referenzierten `<bl:partner>` zurückgibt. Außerdem implementiert sie die Methode `run()`, welche die Ausführungssemantik der Aktivität implementiert, ist also ein JACOB Runnable. Je nachdem ob das Attribut `mep` gesetzt ist oder nicht, wird zur Ausführung der Ausführungssemantik ein Objekt der Klasse `SimpleIA` oder `ComplexIA` herangezogen. Die Implementierung der Ausführungssemantik der `<bl:pick>`-Aktivität, d.h. die Implementierung der Methode `run()`, wird vollständig von der Klasse `Pick` übernommen. Sie hält außerdem alle Attribute der `<bl:pick>`-Aktivität und ermöglicht den Zugriff auf die kompilierte Repräsentation der referenzierten `<bl:conversation>` bzw. `<bl:partner>`.

Die `<bpel:correlation>` für BPEL^{light}-Aktivitäten ist nicht implementiert. Stattdessen wird auf die implizite Korrelation von ODE zurückgegriffen. Diese basiert auf SessionIDs, die zwischen Partnern ausgetauscht werden.

6.1.2.1.1 Implementierung der Ausführungssemantik der einfachen `<bl:interactionActivity>`

Die im Rahmen dieser Arbeit vorgenommene Implementierung der einfachen `<bl:interactionActivity>` beschränkt sich auf das Senden bzw. Empfangen regulärer Nachrichten. Es werden also generell ein empfangender Fall und ein sendender Fall unterschieden.

Die Implementierung der empfangenden `<bl:interactionActivity>` realisiert die folgende Funktionalität: Trifft eine Nachricht ein, so wird versucht, diese anhand der impliziten Korrelation von ODE, die auf unter

Partnern ausgetauschten SessionIDs basiert, einer vorhandenen Prozessinstanz zuzuordnen. Schlägt die Korrelation fehl, so wird eine neue Prozessinstanz gestartet, sofern es sich bei der empfangenden Aktivität um eine instantiierende Aktivität handelt. Der Inhalt der eintreffenden Nachricht wird in der vom `<bl:output>` referenzierten Variable gespeichert. Ist in der Nachricht neben den Nutzdaten auch die EPR des aufrufenden Partners als Call-back enthalten, so wird sie der entsprechenden Partnerinstanz hinzugefügt. Ist zusätzlich eine SessionID vorhanden, so wird diese ebenfalls in der entsprechenden Partnerinstanz abgelegt, sodass eine Folgenachricht der entsprechenden Partnerinstanz zugestellt werden kann. Wird eine empfangende `<bl:interactionActivity>` aktiviert bevor die Nachricht eintrifft, so wartet sie auf das Eintreffen der Nachricht.

Die sendende `<bl:interactionActivity>` sendet nach Aktivierung den Inhalt der durch das `<bl:input>`-Element referenzierten Variable an den angegebenen Partner. Ist der Partnerinstanz eine SessionID zugeordnet, so wird diese in die Nachricht injiziert.

6.1.2.1.2 Implementierung der Ausführungssemantik der komplexen `<bl:interactionActivity>`

Im Gegensatz zur einfachen `<bl:interactionActivity>`, deren Implementierung bereits in Abschnitt 6.1.2.1.1 beschrieben ist, gibt es für die komplexe `<bl:interactionActivity>` auf konzeptioneller Ebene mehrere Möglichkeiten sie zu implementieren. Eine Möglichkeit ist die direkte Implementierung einiger ausgewählter Nachrichtenaustauschmuster im ODE Quellcode. Dies ermöglicht zwar eine effiziente Ausführung der Nachrichtenaustauschmuster, führt aber dazu, dass die Ausführbarkeit von BPEL^{light}-Prozessen von der jeweiligen Engineimplementierung abhängig ist. Eine weitere Implementierungsmöglichkeit ist es, die Parameter des Nachrichtenaustauschmusters durch die entsprechenden Inhalte der `<bl:interactionActivity>` zu belegen und anschließend die `<bl:interactionActivity>` durch den nun nicht mehr abstrakten Code des Nachrichtenaustauschmusters zu ersetzen. Anstatt die

`<bl:interactionActivity>` durch den Code des Nachrichtenaustauschmusters zu ersetzen ist es auch möglich, das konkretisierte Nachrichtenaustauschmuster zu einem vollständigen, eigenständig lauffähigen Prozess – einem Subprozess – zu erweitern und durch die komplexe `<bl:interactionActivity>` aufrufen zu lassen. Dadurch dass beim letztgenannten Ansatz das Nachrichtenaustauschmuster nicht in den eigentlichen Prozess eingebettet wird sondern eigenständig lauffähig ist, können die entstandenen Prozesse auch zur Implementierung anderer `<bl:interactionActivity>`s des Prozesses oder auch anderer Prozesse verwendet werden. Der Grad der Wiederverwendung ist in diesem Fall also höher. Die Parametrisierung kann während der Modellierung, während des Deployments oder zur Laufzeit durchgeführt werden. Die generischen Ansätze der Ausführung komplexer Aktivitäten unterstützen alle mit BPEL^{light} modellierbaren Nachrichtenaustauschmuster. Allerdings ist gegenüber der direkten Implementierung der Nachrichtenaustauschmuster im ODE Quellcode mit Performanzeinbußen zu rechnen. Da die direkte Implementierung der Nachrichtenaustauschmuster eine Einschränkung der Sprache nach sich zieht, sind die generischen Ansätze vorzuziehen. Optional können zusätzlich häufig verwendete Nachrichtenaustauschmuster zur Verbesserung der Performanz direkt im Quellcode implementiert werden.

Die vorliegende Implementierung konzentriert sich auf die Realisierung der komplexen `<bl:interactionActivity>`, die einen BPEL^{light}-Prozess aufruft, der das Nachrichtenaustauschmuster implementiert. Ein solcher Prozess wird in der Folge auch als *MEP-Prozess* bezeichnet, ein Prozess, der einen solchen MEP-Prozess aufruft als *Hauptprozess*. Aufgrund der operationalen Semantik von BPEL, die beispielsweise auch explizite Termination vorsieht, ist die Koordination der beiden Prozesse – wie in BPEL-SPE definiert [KKL+05] – notwendig. Dies wurde allerdings im Rahmen dieser Arbeit nicht implementiert. Die komplexe `<bl:interactionActivity>` des Hauptprozesses liest beim Aufruf die Daten aus den durch ihre Kindelemente (`<bl:input>`s, `<bl:partner>`s etc.) referenzierten Elementen aus, übergibt sie an den MEP-Prozess, wartet die Ausführung des MEP-Prozesses ab, empfängt anschließend die vom MEP-

Prozess produzierten Daten und kopiert sie letztendlich in die über die Kindelemente `<bl:output>`s referenzierten Variablen.

BPEL^{light}-Prozesse, die Nachrichtenaustauschmuster beschreiben, sind per Definitionem abstrakte Prozesse und nicht ausführbar. Sie müssen durch Parametrisierung ausführbar gemacht werden. Es reicht jedoch nicht aus, Informationen von denen bei der Beschreibung eines Nachrichtenaustauschmusters abstrahiert wurde, wie z.B. Datentypen, zu konkretisieren, da Nachrichtenaustauschmuster lediglich die Kommunikation mit Partnerdiensten beschreiben [LNL08]. Ein MEP-Prozess muss außerdem auch mit der aufrufenden komplexen `<bl:interactionActivity>` des Hauptprozesses interagieren.

Parametrisierung von Nachrichtenaustauschmustern

Zur Parametrisierung der Nachrichtenaustauschmuster sind die folgenden Schritte implementiert:

- Der Namensraum „`http://iaas.uni-stuttgart.de/BPELlight/abstract/mep/2008/`“, in dem die BPEL^{light}-Konstrukte definiert sind, wird durch den Wert „`http://iaas.uni-stuttgart.de/BPELlight/2008/`“ ersetzt.

- Das Attribut `abstractProcessProfile` des `<process>`-Elements wird entfernt.

- Ein neuer Prozessname wird generiert. Dazu wird das Schema [Name des Nachrichtenaustauschmusters]4[Name der aufrufenden komplexen `<bl:interactionActivity>`] verwendet.

- Opaque Attribute (Referenzen auf Variablen und Datentypen von Variablen) werden durch konkrete Informationen der referenzierenden `<bl:interactionActivity>` ersetzt (Fehler und Ausdrücke sind im Prototyp nicht implementiert):

  - Ein `<bl:input>`-Element im Hauptprozess steht für eine durch den MEP-Prozess zu sendende Nachricht, d.h. für eine sendende einfache `<bl:interactionActivity>`. Es wird innerhalb des MEP-

Prozesses eine neue Variable erstellt, die entsprechend der Variable typisiert wird, die im Hauptprozess vom `<bl:input>`-Element referenziert wird. Die einfache sendende `<bl:interactionActivity>` innerhalb des MEP-Prozesses nutzt die neue Variable als Container für die zu versendende Nachricht, d.h. das Attribut `variable`, das vorher entweder nicht gesetzt war oder den Wert „`##opaque`“ hatte, referenziert nun die neue Variable.

- Ein `<bl:output>`-Element im Hauptprozess steht für eine durch den MEP-Prozess zu empfangende Nachricht, d.h. eine einfache empfangende `<bl:interactionActivity>`, ein `<bl:onMessage>` einer `<bl:pick>`-Aktivität oder ein `<bl:onEvent>` eines `<bl:eventHandlers>`. Analog zum `<bl:input>` wird eine neue Variable erstellt, auf die das entsprechende Attribut verweist.

- Da ein MEP-Prozess lediglich das Verhalten eines Nachrichtenaustauschmusters realisiert, ist die Kommunikation mit einem aufrufenden Hauptprozess nicht Teil des MEP-Prozesses. Es werden daher bei der Parametrisierung Aktivitäten eingefügt, die zum einen die Kommunikation mit dem Hauptprozess übernehmen und zum anderen die Daten entsprechend aufbereiten. Abbildung 57 zeigt die notwendigen Anreicherungen des MEP-Prozesses, um die Kommunikation mit dem Hauptprozess zu ermöglichen. Neue Aktivitäten und Elemente sind mit gestrichelter Umrandung dargestellt. Im einzelnen sind die folgenden Schritte implementiert:

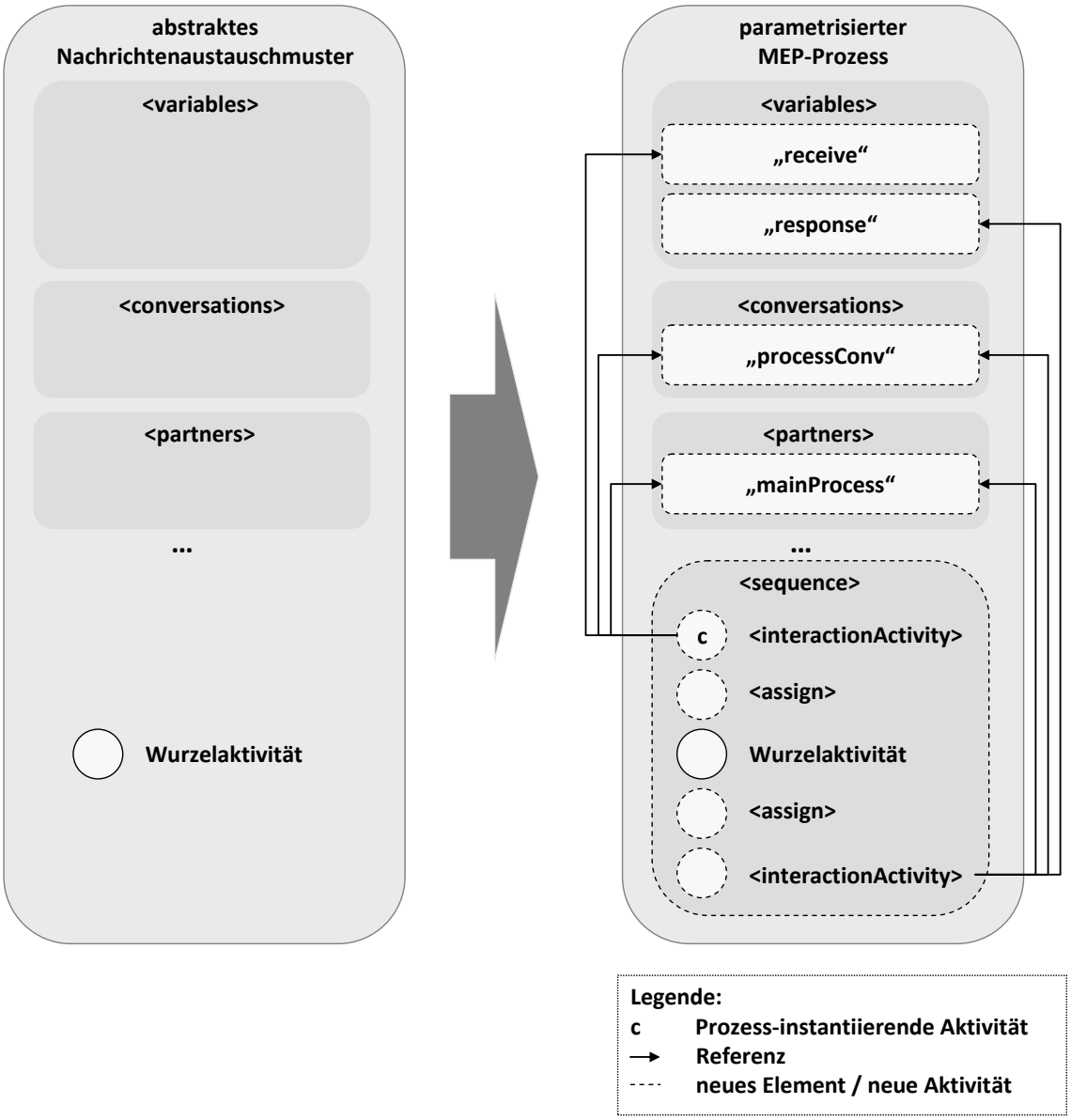


Abbildung 57: Parametrisierung eines MEP-Prozesses

- Die Wurzelaktivität wird eingebettet in eine neue `<bpel:sequence>`-Aktivität. Dies ermöglicht es der Wurzelaktivität, zusätzliche Aktivitäten voranzustellen bzw. anzufügen.

- Der ursprünglichen Wurzelaktivität wird eine einfache, `<bl:interactionActivity>` mit dem Attributwert „`createInstance="yes"`“ vorangestellt.

- Um das Ergebnis des MEP-Prozesses an den Hauptprozess zurückgeben zu können, wird hinter der ursprünglichen Wurzelaktivität eine einfache, sendende `<bl:interactionActivity>` eingefügt.

- Beide `<bl:interactionActivity>`s benötigen jeweils eine Variable – die instantiierende Aktivität eine Variable, in welche die empfangene Nachricht abgelegt werden kann, und die zweite `<bl:interactionActivity>` eine Variable, aus der die zu sendende Nachricht ausgelesen werden kann. Es werden daher zwei neue Variablen definiert, die von den `<bl:interactionActivity>`s entsprechend verwendet werden.

- Um die Daten dem MEP-Prozess übergeben zu können, wird für sie ein Container (`mepCallBundleRequest`) definiert (siehe Listing 48), so dass der MEP-Prozess die Daten auslesen und korrekt verwenden kann. Der Container `mepCallBundleRequest` besteht aus zwei Kindelementen, die beide vom gleichen Typ sind und jeweils mindestens einmal auftreten müssen:

  - Das Element `<input>` wird verwendet, um die Instanz einer Variablen des Hauptprozesses auf eine ausgehende Nachricht im MEP-Prozess zu kopieren.

  - Das Element `<partner>` wird verwendet, um eine Partnerinstanz aus dem Hauptprozess auf eine Partnerinstanz des MEP-Prozesses oder anders herum zu kopieren.

  Das Attribut `mepContainer` bestimmt die Variable bzw. den Partner im MEP-Prozess dem beim Aufruf der spezifizierte Wert zugewiesen werden soll.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://iaas.uni-stuttgart.de/mep-dt"
           xmlns:tns="http://iaas.uni-stuttgart.de/mep-dt"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="mepCallBundleRequest">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="input" minOccurs="1" maxOccurs="unbounded"
                    type="tns:tRequestContent"/>
        <xs:element name="partner" minOccurs="1" maxOccurs="unbounded"
                    type="tns:tRequestContent"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="tRequestContent">
    <xs:all>
      <xs:element name="content" minOccurs="1" maxOccurs="1">
        <xs:complexType>
          <xs:all>
            <xs:any namespace="##other" processContents="lax"/>
          </xs:all>
        </xs:complexType>
      </xs:element>
    </xs:all>
    <xs:attribute name="mepContainer" type="xs:string" use="required"/>
  </xs:complexType>
</xs:schema>
```

Listing 48: XSD für den Request von einem Hauptprozess an einen MEP-Prozess

- Außerdem muss der MEP-Prozess nach dessen Ausführung dem Hauptprozess die von Partnerdiensten empfangenen Daten zur Verfügung stellen. Hierfür wird der in Listing 49 abgebildete Container `mepCallBundleResponse` verwendet. Der Container `mepCallBundleResponse` besteht aus lediglich einem Kindelement, das mindestens einmal auftreten muss:

  - Das Element `<output>` wird verwendet um eine im MEP-Prozess eingehende Nachricht auf eine Variable des Hauptprozesses zu kopieren.

  Das Attribut `ref` identifiziert das Element in der aufrufenden komplexen `<bl:interactionActivity>`, in welches das Ergebnis zurückgeschrieben werden soll.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://iaas.uni-stuttgart.de/mep-dt"
           xmlns:tns="http://iaas.uni-stuttgart.de/mep-dt"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="mepCallBundleResponse">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="output" minOccurs="1" maxOccurs="unbounded"
                    type="tns:tResponseContent"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="tResponseContent">
    <xs:all>
      <xs:element name="content" minOccurs="1" maxOccurs="1">
        <xs:complexType>
          <xs:all>
            <xs:any namespace="##other" processContents="lax"/>
          </xs:all>
        </xs:complexType>
      </xs:element>
    </xs:all>
    <xs:attribute name="ref" type="xs:string" use="required"/>
  </xs:complexType>
</xs:schema>
```

Listing 49: XSD für die Response von einem MEP-Prozess an einen Hauptprozess

- Die instantiierende einfache `<bl:interactionActivity>` empfängt vom Hauptprozess eine Nachricht mit Partnerinstanzdaten und Variableninhalten. Um die Daten aus der Nachricht extrahieren zu können und in die entsprechenden `<bl:partner>` bzw. `<bpel:variable>` Definitionen übertragen zu können, wird eine entsprechende `<bpel:assign>`-Aktivität zwischen die empfangende Aktivität und der ursprünglichen Wurzelaktivität eingefügt. Um die Daten an den Hauptprozess zurücksenden zu können müssen sie aus den entsprechenden `<bpel:variable>` Definitionen extrahiert und in eine Nachricht geschrieben werden. Zu diesem Zweck wird eine entsprechende `<bpel:assign>`-Aktivität zwischen die ursprüngliche Wurzelaktivität und die sendende `<bl:interactionActivity>` eingefügt. Die `<bpel:assign>`-Aktivität zum Entpacken der Hauptprozessinformationen im MEP-Prozess enthält eine `<copy>`-Anweisung pro Partner und pro zu sendender Nachricht. Jede `<copy>`-Anweisung weist dem spezifizierten Ziel

Instanzinformationen zu. Ein Beispielszenario einer entpackenden `<bpel:assign>`-Aktivität ist in Abbildung 58 dargestellt. Die komplexe `<bl:interactionActivity>` des Hauptprozesses definiert drei Kindelemente: ein `<bl:input>`, ein `<bl:output>` und ein `<bl:partner>`. Für die entpackende `<bpel:assign>`-Aktivität sind jedoch nur die Kindelemente `<bl:input>` und `<bl:partner>` relevant. Die vom MEP-Prozess empfangenen Daten werden in der Variablen „receive“ abgespeichert. Die erste `<bpel:copy>`-Anweisung initialisiert die Variable „outOptionalIn!InVar”. Dies ist notwendig, da das Empfangen des Inhalts der Variablen durch den MEP-Prozess nur optional ist. Empfängt der MEP-Prozess keine entsprechende Nachricht und wird die Initialisierung der Variablen nicht vorgenommen, so resultiert der Zugriff auf die Variable beim Packen der Antwortnachricht an die aufrufende komplexe `<bl:interactionActivity>` in einem Fehler. Die zweite `<bpel:copy>`-Anweisung liest die Daten aus der vom `<bl:input>` referenzierten Variable „inVar” aus und weist sie der Variablen „outOptionalIn!OutVar” zu, die von der sendenden `<bl:interactionActivity>` des MEP-Prozesses referenziert wird. Die dritte `<bpel:copy>`-Anweisung liest die Instanzinformationen des `<bl:partner>`s des Hauptprozesses aus und weist sie dem entsprechenden Partner im MEP-Prozess zu.

Hauptprozess

```
...
<bpel:extensionActivitiy>
  <bl:interactionActivity>
    mep=„…out-optional-in“>
    <bl:input variable=„inVar“
              messageRef=„outOptionalIn!Out“
    <bl:output variable=„outVar“
               messageRef=„outOptionalIn!In“
    <bl:partner name=„businessPartner“
                partnerRef=„outOptionalIn!Partner“
  </bl:interactionActivity>
<bpel:extensionActivity>
...
```

```
<mepdt:mepCallBundleRequest>
  <mepdt:input mepContainer=„outOptionalIn!OutVar“>
    <mepdt:content>
      <!-- Inhalt der Variablen inVar -->
    </mepdt:content>
  </mepdt:input>
  <mepdt:partner mepContainer=„outOptionalIn!Partner“>
    <mepdt:content>
      <!-- Instanzdaten des Partners businessPartner -->
    </mepdt:content>
  </mepdt:input>
</mepdt:mepCallBundleRequest>
```

mep-Prozess

```
...
<bpel:variables>
  <bpel:variable name=„outOptionalIn!OutVar“.../>
  <bpel:variable name=„outOptionalIn!InVar“.../>
  <bpel:variable name=„receive“.../>...
</bpel:variables>
...
  <bpel:assign name=„unpack“>
    <bpel:copy ...>
      <bpel:from>
        <bpel:literal/>
      </bpel:from>
      <bpel:to>$outOptionalIn!OutVar</bpel:to>
    </bpel:copy>
    <bpel:copy>
      <bpel:from>
        $receive/mepdt:input[@mepContainer=„outOptionalIn!OutVar“]
                /mepdt:content/element()
      </bpel:from>
      <bpel:to>$outOptionalIn!OutVar</bpel:to>
    </bpel:copy>
    <bpel:copy ...>
      <bpel:from>
        $receive/mepdt:partner[@mepContainer=„outOptionalIn!Partner“]
                /mepdt:content/element()
      </bpel:from>
      <bpel:to>$outOptionalIn!Partner</bpel:to>
    </bpel:copy>
  </bpel:assign>
```

Abbildung 58: Beispiel für eine entpackende `<bpel:assign>`-Aktivität

Die `<bpel:assign>`-Aktivität, welche die Antwortnachricht an den Hauptprozess zusammenstellt, besteht aus mindestens zwei `<bpel:copy>`-Anweisungen. Die erste `<bpel:copy>`-Anweisung initialisiert die für die Antwortnachricht verwendete Variable mit einem `<mepCallBundleResponse>`-Element, das eine bestimmte Anzahl von `<output>`-Elementen enthält. Die Anzahl der `<output>`-Elemente hängt von der Anzahl der Nachrichten ab, die vom MEP-Prozess empfangen wurde und dem Hauptprozess

übermittelt werden soll. Außerdem wird für jedes `<output>`-Element bereits das Attribut `ref` gesetzt, welches es der komplexen `<bl:interactionActivity>` ermöglicht, die einkommenden Daten mit den Variablen der entsprechenden `<bl:output>`-Elemente zu korrelieren. Ab der zweiten `<bpel:copy>`-Anweisung werden schließlich die Variablenwerte in das `<mepCallBundleResponse>`-Element kopiert.

Abbildung 59 greift das in Abbildung 58 eingeführte Beispiel auf. Die dargestellte `<bpel:assign>`-Aktivität initialisiert zunächst die Variable „response" mit einem `<mepCallBundleResponse>`-Element, das lediglich ein `<output>`-Element enthält. Dessen Attribut `ref` wird auf den Wert „outOptionalIn!In" gesetzt, so dass der Hauptprozess den Inhalt mit dem entsprechenden `<bl:output>` mit der Attributbelegung `messageRef=„outOptionalIn!In"` korrelieren kann. Der eigentliche Inhalt des `<output>`-Elements wird erst später gesetzt. Bei der Initialisierung wird ein Platzhalter (`<mepdt:placeholder>`) verwendet.

Schließlich werden die in der Variable „outOptionaIn!InVar" gespeicherten Instanzdaten in das `<mepdt:content>`-Element des `<output>`s kopiert. Der Platzhalter wird dadurch überschrieben.

mep-Prozess

```
...
<bpel:variables>
  <bpel:variable name=„outOptionalIn!OutVar“.../>
  <bpel:variable name=„outOptionalIn!InVar“.../>
  <bpel:variable name=„response“.../>...
</bpel:variables>
...
  <bpel:assign name=„pack“>
    <bpel:copy ...>
      <bpel:from>
        <bpel:literal>
          <mepdt:mepCallBundleResponse>
            <mepdt:output ref=„outOptionalIn!In“>
              <mepdt:content>
                <mepdt:placeholder/>
              </mepdt:content>
            </mepdt:output>
          </mepdt:mepCallBundleResponse>
        </bpel:literal>
      </bpel:from>
      <bpel:to>$response</bpel:to>
    </bpel:copy>
    <bpel:copy ...>
      <bpel:from>$outOptionalIn!InVar</bpel:from>
      <bpel:to>$response/mepdt:output[ref=„outOptionalIn!In“]
                        /mepdt:content/child::element()
      </bpel:to>
    </bpel:copy>
  </bpel:assign>
```

```
<mepdt:mepCallBundleResponse>
  <mepdt:input ref=„outOptionalIn!In“>
    <mepdt:content>
      <!-- Inhalt der Variablen outOptionalIn!InVar -->
    </mepdt:content>
  </mepdt:input>
</mepdt:mepCallBundleResponse>
```

Hauptprozess

```
...
<bpel:extensionActivitiy>
  <bl:interactionActivity>
    mep=„…out-optional-in“>
    <bl:input variable=„inVar“
              messageRef=„outOptionalIn!Out“
    <bl:output variable=„outVar“
               messageRef=„outOptionalIn!In“
    <bl:partner name=„businessPartner“
                partnerRef=„outOptionalIn!Partner“
  </bl:interactionActivity>
<bpel:extensionActivity>
...
```

Abbildung 59: Beispiel für eine zusammenpackende `<bpel:assign>`-Aktivität

- Um der BPEL^{light} Syntax zu genügen, werden die beiden neuen `<bl:interactionActivity>`s einer `<bl:conversation>` zugeordnet. Da die beiden Aktivitäten nicht Teil des eigentlichen Nachrichtenaustauschmusters sind, wird eine neue `<bl:conversation>` definiert, welche die Kommunikation des MEP-Prozesses mit dem Hauptprozess repräsentiert.

- Darüber hinaus wird ein neuer `<bl:partner>` erstellt, der stellvertretend für den Hauptprozess steht und den beiden neuen `<bl:interactionActivity>`s zugeordnet wird.

Sind die Nachrichtenaustauschmuster parametrisiert, muss anschließend auch der Wert des Attributs `mep` der komplexen `<bl:interactionActivity>`s auf den Namen der entstandenen MEP-Prozesse gesetzt werden.

Auch wenn mehrere komplexe `<bl:interactionActivity>`s eines Prozesses das gleiche Nachrichtenaustauschmuster referenzieren, so kann die Parametrisierung aufgrund der Verwendung unterschiedlicher Nachrichtentypen, Fehlertypen oder Ausdrücke in den `<bl:interactionActivity>`s zu mehreren unterschiedlichen Ausprägungen des Nachrichtenaustauschmusters führen.

Kommunikation zwischen Hauptprozess und MEP-Prozess

Während der Ausführung kommunizieren der Hauptprozess und der MEP-Prozess über den Austausch zweier Nachrichten, die einen `mepCallBundleRequest` bzw. `mepCallBundleResponse` enthalten. Zur Korrellation der beiden Prozesse wird ODEs impliziter Korrelationsmechanismus verwendet, der auf dem Austausch von SessionIDs basiert. Der aufrufende Prozess generiert für die Kommunikation mit dem Partner eine SessionID und verschickt diese mit der Nachricht. Der Antwortende Prozess verwendet diese zusätzliche Adressinformation beim Senden der Antwortnachricht. Außerdem generiert er seinerseits eine SessionID, die er mit der Antwort übermittelt. Anschließend können beide Partner gezielt Nachrichten an ihre Partnerinstanz verschicken.

6.1.2.1.3 Implementierung der Ausführungssemantik der `<bl:pick>`-Aktivität

Die Implementierung der `<bl:pick>`-Aktivität beschränkt sich auf das Empfangen regulärer Nachrichten. Die Aktivität wartet auf das Eintreffen einer Nachricht, die zu einer als Kindelement definierten `<bl:onMessage>` passt. Trifft eine Nachricht für ein instantiierendes `<bl:onMessage>` ein, die keiner laufenden Prozessinstanz zugeordnet werden kann, so wird eine neue Prozessinstanz erstellt. Der Inhalt der Nachricht wird in der angegebenen Variablen gespeichert und die Aktivität beendet. Enthält die

Nachricht eine EPR oder eine SessionID, so wird diese Information in die entsprechende Partnerinstanz geschrieben. Ist die eintreffende Nachricht für ein nicht instantiierendes `<bl:onMessage>` bestimmt, wird sie über ODE's implizite Korrelation der Instanz zugewiesen. Die `<bpel:correlation>` für BPEL^{light}-Aktivitäten ist nicht implementiert. Nach erfolgreicher Korrelation verfährt die Engine wie im instantiierenden Fall.

Wird innerhalb der `<bl:pick>`-Aktivität ein `<bpel:onAlarm>` definiert, so kann durch ein zeitliches Ereignis das Warten auf Nachrichten abgebrochen werden. In diesem Fall werden die Aktivitäten der `<bl:onMessage>`-Pfade durch Dead-Path-Elemination behandelt.

6.1.2.2 Umsetzung der Erweiterungen `<bl:partner>` und `<bl:conversation>`

Die Elemente `<bl:partner>`, `<bl:conversation>` und `<bl:expression>` sowie deren Containerelemente können nicht über einen Erweiterungspunkt von ODE implementiert werden. Sie müssen daher in das bestehende BOM und O-Model integriert werden, um eingelesen und ausgeführt werden zu können.

Die Basis des BOM ist die Klasse `BpelObject`. Es ermöglicht den Zugriff auf ein BPEL-Element, seine Attribute und Kindelemente. Alle weiteren Klassen des BOM sind Spezialisierungen der Klasse `BpelObject`. In Abbildung 60 ist ein Klassendiagramm mit den implementierten BPEL^{light}-Erweiterungen dargestellt: Für jedes BPEL^{light}-Element existiert eine Klasse, die von `BpelObject` erbt. Die Klassen `Conversations` und `Partners` ermöglichen es lediglich auf die Kindelemente `<bl:conversation>` bzw. `<bl:partner>` der entsprechenden Elemente zuzugreifen. Die Klassen `Partner` und `Conversation` ermöglichen den Zugriff auf die jeweiligen Attribute. Alle neuen BOM Klassen sind in der Klasse `BpelObjectFactory` registriert und bilden jeweils ein Element auf ein Objekt ab. Dies ermöglicht es dem Kompilierer, für die genannten BPEL^{light}-Elemente einer `.bpel`-Datei entsprechende Objekte im O-Model zu erzeugen. In der prototypischen Implementierung wurde von der Umsetzung der BPEL^{light}-Erweiterungen `<bl:expressions>` and `<bl:expression>` abgesehen. Ihre Implementierung kann jedoch analog zur Implementierung der

Erweiterungen `<bl:conversation>` und `<bl:partner>` sowie deren Containerelemente vorgenommen werden.

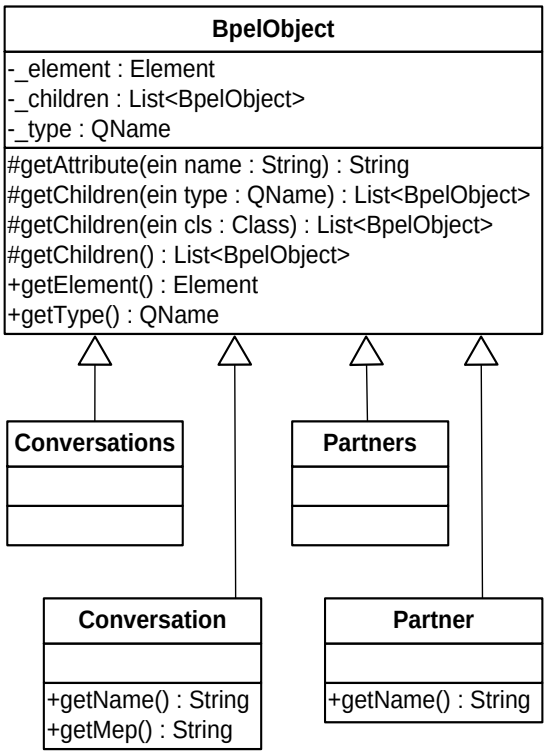


Abbildung 60: Klassendiagramm der implementierten BPEL-Erweiterungen im BOM

Die Erweiterung des O-Models ist in Abbildung 61 dargestellt. Wie bereits beim BOM umfasst die Erweiterung die Repräsentation der Elemente `<bl:conversation>` und `<bl:partner>`, d.h. die Klassen `Oconversation` und `Opartner`. Im Klassendiagramm sind nur die neuen Attribute der bereits existierenden Klassen `Obase`, `Oscope` und `Oprocess` dargestellt. Von Methoden und Attributtypen wird – zugunsten der besseren Lesbarkeit – abstrahiert. Die gestrichelt dargestellte Generalisierung der Klasse `Oscope` ist nicht UML konform. Sie deutet eine indirekte Vererbung über mehrere Klassen hinweg an, die für die Darstellung irrelevant sind. Partner und Konversationen können sowohl für den ganzen Prozess als auch für einen eingeschränkten Geltungsbereich definiert werden. Die Klasse `Omessages` ist ein Container für reguläre Nachrichten und Fehlernachrichten, die während einer Konversation ausgetauscht werden. Wie schon bei der Erweiterung des BOM wurde bei der prototypischen Implementierung von

der Umsetzung der BPEL^{light}-Erweiterung `<bl:expression>` sowie deren Containerelement abgesehen.

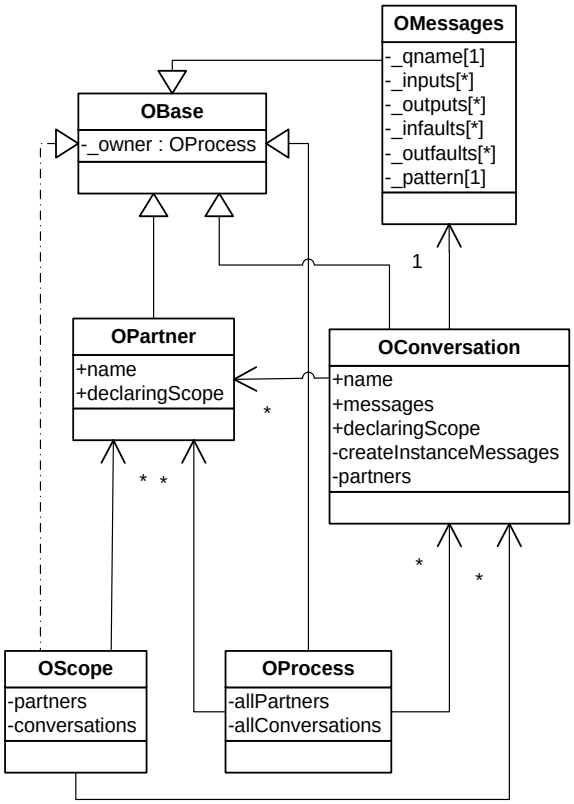


Abbildung 61: Klassendiagramm des erweiterten O-Models

Da das Interaktionsmodell von BPEL^{light} im Gegensatz zu WS-BPEL auf beliebig komplexen Konversationen basiert, in die mehrere Partner involviert sein können, unterliegt die Ausführungsumgebung von ODE signifikanten Veränderungen. Das erweiterte Klassendiagramm der Ausführungsumgebung ist in Abbildung 62 dargestellt. Es enthält sowohl neue als auch bereits vor der Erweiterung existierende Klassen. Die Kernklassen, d.h. das Interface `BpelServer` und seine Implementierung, das Interface `BpelRuntimeContext` und seine Implementierung, sowie die Klasse `BpelProcess`, wurden erweitert, um BPEL^{light} zu unterstützen. Die Fähigkeit, WS-BPEL-Prozesse auszuführen, ist dadurch nicht beeinträchtigt. Die Abbildung zeigt lediglich die wichtigsten Klassen, Attribute und Methoden. Um die Lesbarkeit des Diagramms zu erhöhen, wird von Methodenparametern, Rückgabewerten und einigen bereits vor der Erweiterung existierenden Attributen und Methoden abstrahiert.

Die Klasse `BpelProcess` ist der Mittelpunkt der Ausführungsumgebung. Für jeden deployten Prozess existiert ein `BpelProcess`-Objekt, welches die Koordination aller entsprechenden auszuführenden Prozessinstanzen übernimmt. Die Klasse `BpelServerImpl` ist der Einstiegspunkt, um auf BPEL-Prozesse zugreifen zu können. Sie leitet eintreffende Nachrichten an das entsprechende `BpelProcess`-Objekt weiter, das ein Objekt der Klasse `IncomingMessageHolderImpl` kreiert. Die Funktion dieser Hilfsklasse ist es – wie bereits der Name nahe legt – eine eintreffende Nachricht und die entsprechende Kontextinformation zu halten. Sie triggert die Nachrichtenkorrelation und den Aufruf der adressierten Prozessinstanz.

Die einzelnen Prozessinstanzen werden jeweils durch ein Objekt der Klasse `BpelRuntimeContextImpl` verwaltet, welches JACOB verwendet, um die entsprechende Instanz auszuführen. Die Klasse `BpelRuntimeContextImpl` bietet u.a. Methoden an, um auf Instanzdaten zugreifen zu können (z.B. `getMySessionId()`), einen Partnerdienst oder MEP-Prozess aufrufen zu können (`invokeMep()` und `invokePartnerBpelLight()`) und viele mehr. ODE's Interaktionsmodel basiert auf unterschiedlichen Implementierungen des Interfaces `MessageExchange`, die jeweils eine Kombination von Partnerlink und Operation repräsentieren, d.h. maximal zwei Nachrichten bündeln (eine Anfrage und eine optionale Antwort). Abhängig von der Richtung der ersten Nachricht unterscheidet ODE zwischen `partnerRole` (zu sendende Anfrage) und `myRole` (eintreffende Anfrage) `MessageExchange`-Objekten. Das Ziel von zu sendenden Nachrichten wird in EPRs gespeichert, die den Partnerlinks (bzw. der Rolle partnerRole) zugeordnet sind.

Dieses Paradigma wird durch BPEL^{light} geändert. Eine `<bl:conversation>` kann einen beliebig komplexen Nachrichtenaustausch mit beliebig vielen Partnerdiensten definieren, d.h. sie muss mehrere EPRs verwalten. Es gibt daher keine Unterscheidung mehr zwischen `myRole` und `partnerRole` `MessageExchanges`. Stattdessen wird eine neues Interface `ConversationMessageExchange` eingeführt und eine zugehörige Implementierung, die Klasse `ConversationMessageExchangeImpl`, die alle für die Interaktion relevanten Daten enthält. Die Klasse `ConversationImpl` implementiert die Ausführungssemantik einer `<bl:conversation>`, d.h. die

Erstellung einer neuen Prozessinstanz, die Korrelation eingehender Nachrichten (mittels ODE impliziteer Korrelation) zu Aktivitäten und das Senden von Nachrichten an Partnerdienste. Die Implementierung eines Partners, d.h. die Klasse `PartnerImpl` verwaltet die kompilierte Partnerdefinition. Objekte der Klasse `PartnerInstance` enthalten nicht direkt Partnerinstanz-spezifische Informationen wie beispielsweise eine EPR. Sie kann jedoch über den Geltungsbereich, für den der zugehörige Partner definiert wurde, auf die entsprechenden Informationen zugreifen.

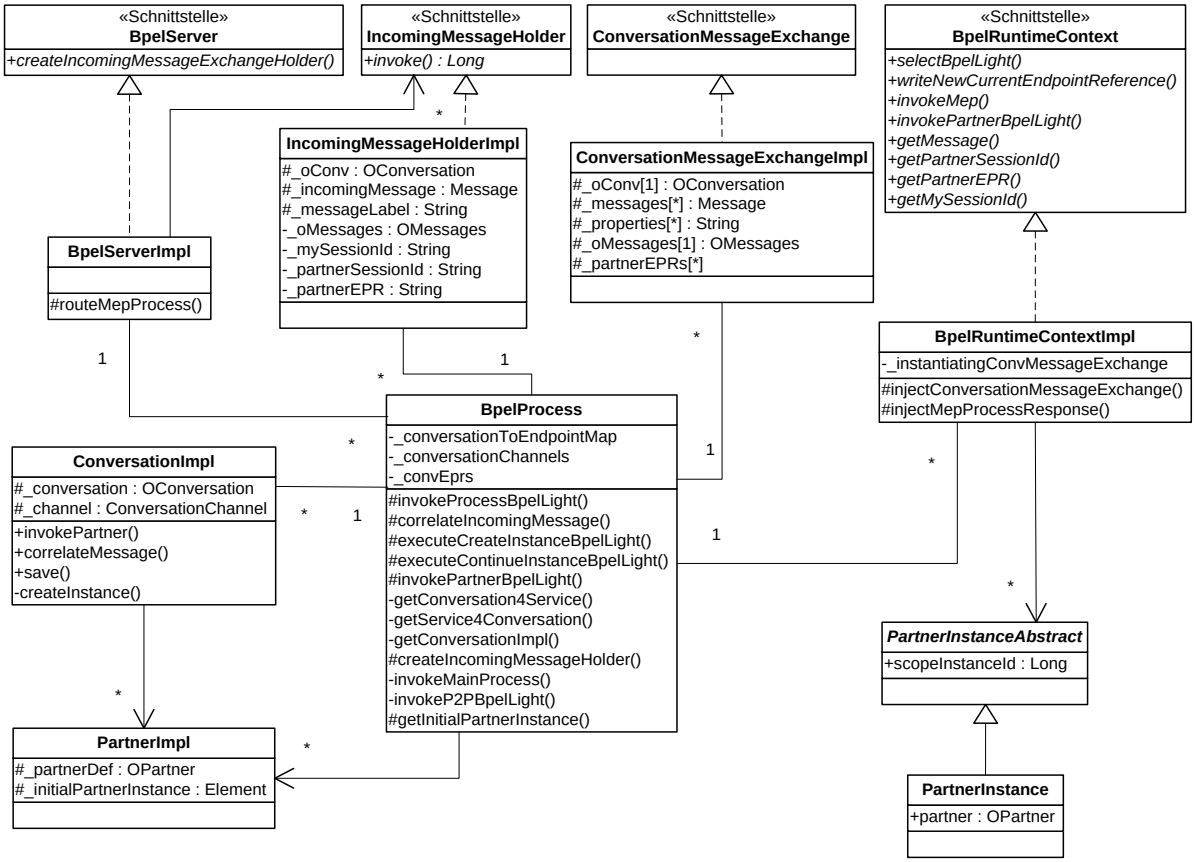


Abbildung 62: Klassendiagramm der erweiterten Ausführungsumgebung

6.1.3 WSDL 1.1-Integrationsschicht – BPEL^{light}-Ausführung in Web-Service-Umgebungen

Um BPEL^{light}-Prozesse in einer Web-Service-Umgebung ausführen zu können, bedarf es einer entsprechenden Implementierung der Integrationsschicht der BPEL^{light}-Engine. Diese muss es ermöglichen, den Prozess selber als Web-Service anzubieten und Web-Services zur

Implementierung von BPEL^{light}-Aktivitäten zu verwenden, so dass alle in Abschnitt 4.2 dargestellten Szenarien unterstützt werden. Diese Anforderung kann dadurch erfüllt werden, dass der durch BPEL^{light} definierte Nachrichtenaustausch innerhalb der Integrationsschicht entsprechend des während der Modellierung angegebenen Mechanismus' auf WSDL-Operationen abgebildet wird. Das bedeutet, dass sich die Architektur für die Verwendung von BPEL^{light} in einer Web-Service-Ausführungsumgebung von der Architektur einer Web-Service-Ausführungsumgebung mit WS-BPEL als Serviceorchestrierungssprache lediglich darin unterscheidet, dass die Abbildung der Prozesslogik auf die Schnittstellenbeschreibungen erst in der Integrationsschicht des verwendeten Busses stattfinden kann und nicht innerhalb der eigentlichen Prozessausführungsengine.

In Abbildung 63 ist anhand der Architektur der Ausführungsumgebung für BPEL^{light}-Prozesse in einer Web-Service-Umgebung das Szenario dargestellt, in dem der Prozess zur Realisierung seiner Funktionalität einen Web-Service nutzt: Zunächst findet die Discovery Komponente des Busses anhand der von der Integrationsschicht der BPEL^{light}-Engine übermittelten WSDL-Informationen einen vorher im Repository registrierten Dienst. Anschließend ruft der Service Bus den Dienst auf und gibt gegebenenfalls die Antwort an den Prozess zurück.

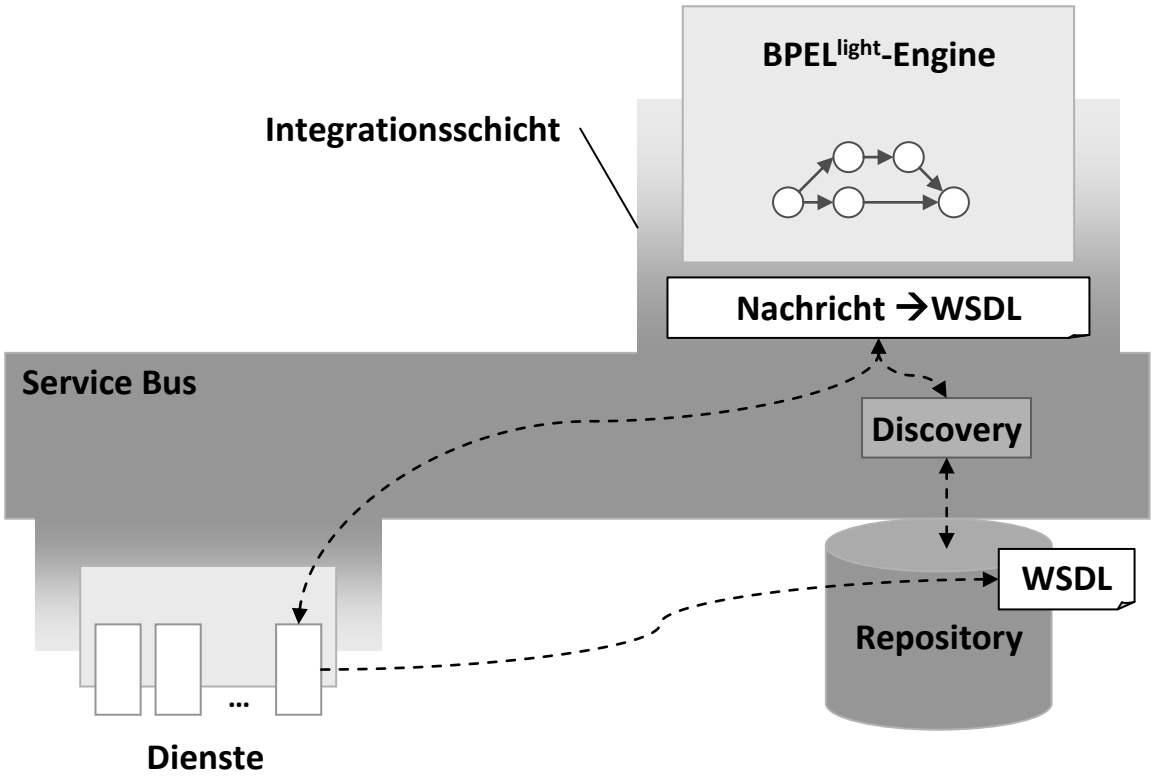


Abbildung 63: BPEL^{light}-Prozess als Klient in einer Web-Service-Ausführungsumgebung

Beim Deployment eines BPEL^{light}-Prozesses, der von einer Web-Service-Umgebung Gebrauch macht, enthält die Deployment-Einheit neben der BPEL^{light}-Prozessbeschreibung und dem Deployment Deskriptor eine oder mehrere WSDL-Dateien der aufzurufenden Dienste und die WSDL-Beschreibung des Prozesses. Alle Artefakte werden in die BPEL^{light}-Engine bzw. in deren Integrationsschicht deployt; die WSDL-Beschreibung des Prozesses zusätzlich in das Repository des Busses.

Die Implementierung der beschriebenen Infrastruktur wurde im Rahmen dieser Arbeit nicht vorgenommen.

6.1.4 WSMO-Integrationsschicht – BPEL^{light}-Ausführung in Semantischen Web-Service-Umgebungen

Wie beim Einsatz von BPEL^{light}-Prozessen in einer Web-Service-Umgebung, bedarf es auch beim Einsatz in einer Semantischen Web-Service-Umgebung einer entsprechenden Implementierung der Integrationsschicht der BPEL^{light}-Engine. Diese muss es ermöglichen, den Prozess selber als Semantischen Web-Service anzubieten und Goal-Beschreibungen

verwenden zu können, um zur Laufzeit durch den semantischen Service Bus entsprechende Semantische Web-Services auffinden und aufrufen lassen zu können.

Der Semantische Service Bus muss über die in Abschnitt 5.1 und 5.2 spezifizierte, Funktionalität verfügen, um alle in Abschnitt 5.3 dargestellten Szenarien unterstützen zu können. Die Funktionalität geht über die Funktionalität existierender Ausführungsumgebungen für Semantische Web-Services hinaus. Wird als semantisches Web-Service-Framework WSMO verwendet, kann diese zusätzliche Funktionalität durch die Erweiterung ausgewählter Komponenten der in Abbildung 22 Abschnitt 2.2.2 präsentierten Architektur von WSMX und deren API realisiert werden. Die betroffenen Komponenten sind in diesem Fall die Discovery und die Choreography-Engine. Die Discovery muss zusätzlich zum bisherigen Funktionsumfang die Funktionen

(i) `findPartner(GoalSet):Partner` und

(ii) `initializeCommunication(Goal):Context` bzw.
`initializeCommunication(Goal, Partner):Context`

anbieten, die Choreography-Engine die Funktion

(iii) `exchangeMessage(Context, Data)` bzw.
`exchangeMessages(Context, Data):Data`.

(i) findPartner

Die Funktion `findPartner(GoalSet):Partner` wird dann benötigt, wenn ein Klient mehrere Geschäftsziele definiert hat, die alle von ein und demselben Geschäftspartner erfüllt werden sollen. Dieses Szenario tritt, je nachdem in welcher Granularität die Geschäftsziele definiert sind, häufiger oder weniger häufig auf. Der Aufruf der Funktion erfolgt durch die Übermittlung einer oder mehrerer WSMO-Goals. Als Antwort wird eine Ontologie zurückgesendet, die eine Instanz eines Ontologiekonzepts enthält. Diese repräsentiert den angeforderten Partner, der in der Lage ist, alle übermittelten Geschäftsziele zu erfüllen.

(ii) initializeCommunication

Die Funktion `initializeCommunication(Goal):Context` wird verwendet, um die Kommunikation zu einem Dienst aufzubauen der in der Lage ist, das angegebene Geschäftsziel zu erfüllen. Bei Verwendung der Funktion `initializeCommunication(Goal, Partner):Context` wird die Suche auf jene Dienste eingeschränkt, die vom angegebenen Geschäftspartner angeboten werden. Der Aufruf der Funktionen erfolgt durch die Übermittlung einer Nachricht, die ein Goal und optional eine Ontologie mit der Partnerdefinition in Form einer Instanz eines Ontologiekonzepts enthält. Als Antwort wird ein Kontext zurückgesendet, der zur Korrelation innerhalb des semantischen Service Busses benötigt wird.

(iii) exchangeMessage/exchangeMessages

Wie bereits in Abschnitt 2.2.2 beschrieben, findet der Nachrichtenaustausch zwischen Klient und Dienst nicht direkt, sondern über den semantischen Service Bus statt. Es gibt also Kommunikation zwischen dem Klienten und dem SSB und dem Dienst und dem SSB. Geht die Interaktion vom SSB aus, so verwendet er die WSDL-Operationen, auf die Goal- bzw. Dienstbeschreibung gegroundet sind, um Nachrichten zu übermitteln und gegebenenfalls zu empfangen. Geht die Interaktion jedoch vom Klienten bzw. vom Dienst aus, so verwenden sie die Funktion `exchangeMessage(Context, Data)` falls lediglich eine Nachricht übermittelt werden soll bzw. `exchangeMessages(Context, Data):Data`, falls im Anschluss an die Übermittlung einer Nachricht eine Antwortnachricht erwartet wird. Der zu übermittelnde Kontext ist dem Klienten durch den Aufruf der Funktion `initializeCommunication()` bekannt, dem Dienst wird der Kontext vom SSB beim Aufruf einer seiner WSDL-Operationen übermittelt.

Aus Gründen der Interoperabilität werden die genannten Funktionen als Web-Service-Operationen angeboten. Die auszutauschenden Nachrichten enthalten in WSML serialisierte WSMO-Elemente.

In Abbildung 64 ist anhand der Architektur der Ausführungsumgebung für BPEL^{light}-Prozesse in einer Semantischen Web-Service-Umgebung das

Szenario dargestellt, in dem ein Prozess die Anforderung an eine Aktivitätsimplementierung in Form eines WSMO-Goals beschreibt und ein entsprechender Dienst vom SSB gefunden und aufgerufen wird. Bezüglich des zu verwendenden Geschäftspartners gibt es im vorliegenden Szenario keinerlei Einschränkungen. Zunächst findet die Discovery Komponente des Busses anhand des von der Integrationsschicht der BPEL^{light}-Engine im Rahmen der Operation `initializeCommunication()` übermittelten WSMO-Goals (2) einen vorher im Repository registrierten Dienst (1). Anschließend wird in der Choreography-Engine des Busses jeweils eine Instanz der Goal- und Web-Service-Choreography gestartet (3) und miteinander assoziiert. Außerdem wird für beide Choreography-Instanzen jeweils ein Kontext erstellt, der es dem Bus ermöglicht, eintreffende Nachrichten von Klient und Dienst mit den korrekten Choreography-Instanzen zu korrelieren. Dem Klienten, d.h. der BPEL^{light}-Engine, wird sein Kontext im Rahmen der Operation `initializeCommunication()` zurückgegeben (4). Dem Dienst wird sein Kontext beim ersten Aufruf durch den SSB übermittelt. Über den nun dynamisch definierten Kanal innerhalb des Busses können Klient und Dienst Nachrichten austauschen. Sowohl Klient als auch Dienst können über die Operation `exchangeMessage()` bzw. `exchangeMessages()` der Wissenbasis ihrer Choreography-Instanz neue Informationen hinzufügen (6). Liegen die Daten beim Klienten bzw. beim Dienst in XML Form vor, so müssen sie vor der Übermittlung in eine entsprechende Ontologieform überführt werden. Hierzu wird die Lifting&Lowering-Komponente des Busses verwendet (5). Innerhalb des Busses werden diese Informationen – gegebenenfalls unter Verwendung Ontologie-basierter Mediation – in die Wissensbasis der Choreography des entsprechenden Kommunikationspartners kopiert (7). Dieser Vorgang kann nun eine Regel in der mit Informationen angereicherten Choreography auslösen, die den Aufruf des mit der Choreography assoziierten Kommunikationspartners nach sich zieht. Der Aufruf geschieht über eine im WSMO-Grounding spezifizierte WSDL-Operation (9). Die in Ontologieform vorliegenden Daten müssen vor dem Aufruf in die entsprechende XML Repräsentation gebracht werden. Auch hierzu wird die Lifting&Lowering-Komponente des Busses verwendet (8).

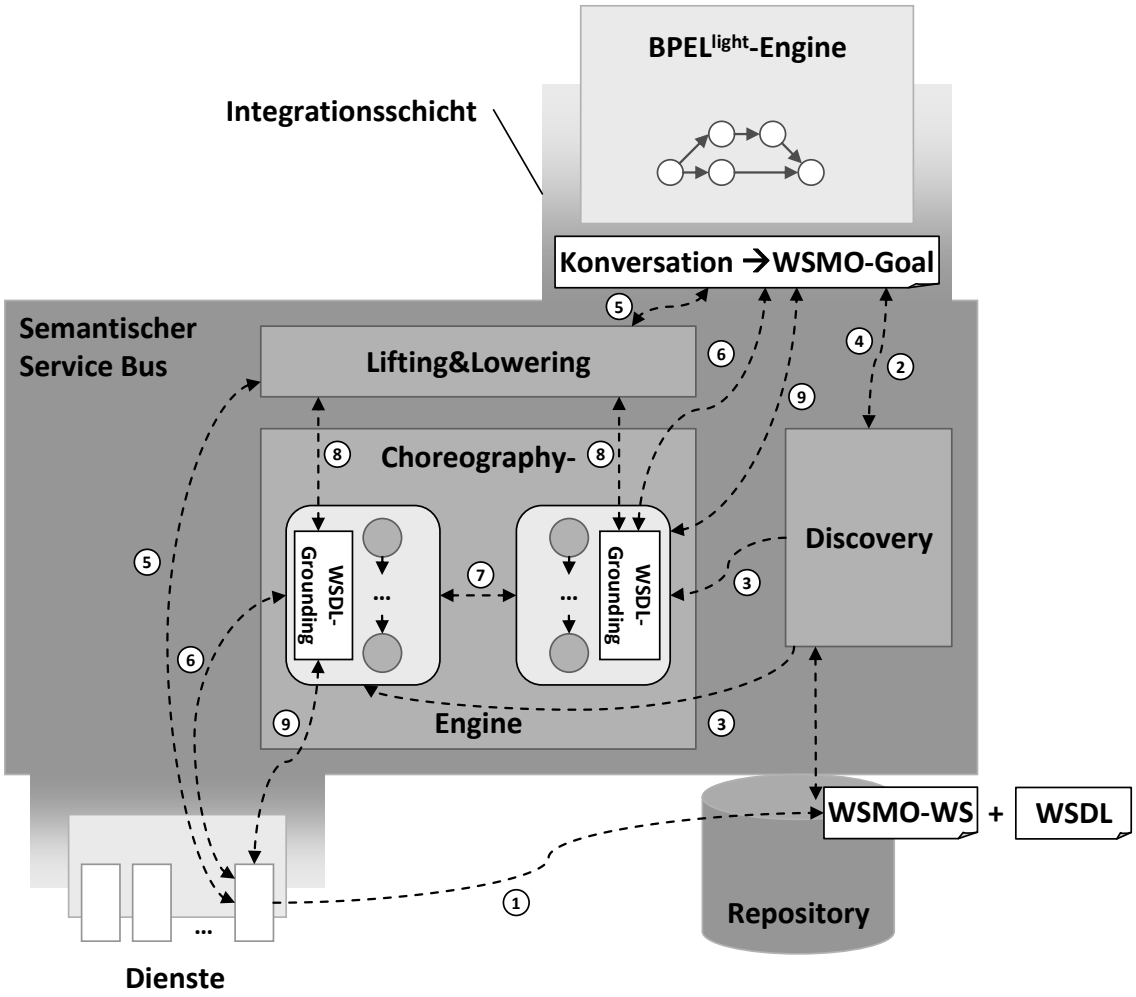


Abbildung 64: BPEL^light-Prozess als Klient in einer Semantischen Web-Service-Ausführungsumgebung

Beim Deployment eines BPEL^light-Prozesses, der von einer Semantischen Web-Service-Umgebung Gebrauch macht, enthält die Deployment-Einheit neben der BPEL^light-Prozessbeschreibung und dem Deployment Deskriptor folgende Artefakte:

- eine oder mehrere WSDL-Dateien der aufzurufenden Dienste inklusive der benötigten SAWSDL-Annotationen,
- die WSDL-Beschreibung des Prozesses, ebenfalls inklusive der benötigten SAWSDL-Annotationen,
- gegebenenfalls eine WSMO-Web-Service-Beschreibung des Prozesses und

- WSMO-Goal-Beschreibungen zur Goal-basierten Suche passender Aktivitätsimplementierungen.

Alle Artefakte werden in die BPEL^{light}-Engine bzw. in deren Integrationsschicht deployt. Die WSDL-Beschreibung des Prozesses inklusive der SAWSDL-Annotationen und die WSMO-Web-Service-Beschreibung des Prozesses werden zusätzlich in das Repository des Busses deployt.

Die Implementierung der in diesem Abschnitt beschriebenen Infrastruktur wurde im Rahmen des Projekts SUPER durchgeführt. Die BPEL4SWS-Engine, d.h. die Integrationsschicht der auf ODE basierenden BPEL^{light}-Engine, wurde durch eine entsprechende Erweiterung von ODE umgesetzt [KLF+07, CLK08]. Ebenfalls implementiert wurde eine Komponente zur Durchführung semantischer Mediation. Der Semantische Service Bus wurde durch die Integration der benötigten WSMX Komponenten in Apache ServiceMix implementiert [LNP+08]. Da die WSMX Komponenten jedoch nur einen Goal-basierten Aufruf einer Request-Response-Operation eines Web-Services unterstützen, ist asynchrone Kommunikation konzeptionell zwar möglich, wird aber durch die Implementierung des semantischen Service Busses nicht unterstützt.

Das Deployment wurde über eine zentrale Deployment-Komponente im semantischen Service Bus implementiert. Die Deployment Komponente nimmt die Deploymenteinheit entgegen und verteilt die enthaltenen Artefakte an die BPEL4SWS-Engine bzw. die betroffenen Komponenten des semantischen Service Busses über entsprechende von der Engine bzw. den Komponenten des Busses angebotenen APIs [LNP+08].

6.2 Modellierungsumgebung für BPEL^{light}-Prozesse

BPEL^{light}-Prozesse lassen sich aufgrund der in dieser Arbeit definierten XML-Serialisierung durch herkömmliche Texteditoren definieren. Da dies jedoch sehr aufwendig und fehleranfällig ist, empfiehlt sich die Verwendung eines Modellierungswerkzeugs, welches es ermöglicht, Prozesse zu verwalten, sie in eine BPEL^{light}-Engine zu deployen und vor allem sie grafisch zu modellieren, d.h. sowohl Elemente von WS-BPEL – z.B.

Kontrollflusselemente und `<bpel:assign>`-Aktivitäten – als auch BPEL^{light}-Interaktionsaktivitäten auf einer Zeichnungsfläche zu platzieren und miteinander zu einem Prozessmodell zu kombinieren sowie deren Eigenschaften zu modifizieren. Wie auch bei der Implementierung der Ausführungsumgebung für BPEL^{light} wurde aufgrund der Komplexität von BPEL^{light} und der sich daraus ergebenden Komplexität für die Implementierung der Modellierungsumgebung eine vollständige Eigenentwicklung ausgeschlossen. Stattdessen wurde bei der Implementierung der Modellierungsumgebung auf exisiterende Editoren zurückgegriffen und diese gegebenenfalls erweitert. Als Basis für den BPEL^{light} Editor kamen unter anderem der Netbeans BPEL-Designer[21] und der Eclipse BPEL-Designer[22] in Frage. Auf den Eclipse BPEL-Designer fiel schließlich die Wahl bezüglich des zu erweiternden Editors, da er zum Zeitpunkt der Entscheidung die BPEL-Spezifikation am Vollständigsten implementierte und bereits über eine Anbindung an Apache ODE verfügte. Im Folgenden werden in Auszügen die vorgenommenen Erweiterungen des Eclipse BPEL-Designers beschrieben. Weitere Einzelheiten sind [Ho08] zu entnehmen.

6.2.1 Einführung in den Eclipse BPEL-Designer

Der Eclipse BPEL-Designer wurde im Mai 2005 von den Firmen IBM und Oracle, Hauptentwicklern des BPEL-Editors, als Open-Source-Projekt freigegeben. Er setzt das MVC-Konzept (Model-View-Controller, deutsch: Modell-Präsentation-Steuerung) [GHJV95] um und verwendet hierzu die Technologien EMF (Eclipse Modeling Framework)[23] und GEF (Graphical Editing Framework)[24]. EMF ermöglicht, dem Model Driven Development (MDD) Ansatz folgend, die Quellcodegenerierung aus einem Modell und implementiert die Modell-Komponente des MVC-Konzepts. GEF ermöglicht die Erstellung grafischer Elemente auf Basis eines Modells und ergänzt

[21] `http://soa.netbeans.org/soa/`

[22] `http://www.eclipse.org/bpel/`

[23] `http://www.eclipse.org/modeling/emf/`

[24] `http://www.eclipse.org/gef/`

EMF somit bezüglich des MVC-Konzepts durch die Implementierung der Präsentation und Steuerung.

6.2.1.1 Architektur des BPEL-Editors

Der Aufbau des BPEL-Editors ist in Abbildung 65 dargestellt. Er besteht aus den folgenden fünf Komponenten:

- Das *Modell* ist mittels EMF umgesetzt und repräsentiert die WS-BPEL-Spezifikation.
- Der *Designer* ist ein auf GEF basierender Editor, der es ermöglicht, WS-BPEL-Prozesse zu erstellen.
- Der *Validator* arbeitet auf dem Modell und erzeugt Warnungen bei Verstößen gegen die WS-BPEL-Spezifikation.
- Die *Runtime* ist ein erweiterbares Rahmenwerk, das es ermöglicht, einen BPEL-Prozess in eine Workflowengine zu deployen.
- Der *Debugger* ist ein Rahmenwerk, das es dem Benutzer ermöglicht, schrittweise durch die Ausführung eines Prozesses zu gehen. Diese Komponente wurde jedoch noch nicht in das Open-Source Projekt aufgenommen.

Die einzelnen Komponenten des BPEL-Editors werden jeweils durch ein bis zwei Java-Pakete realisiert.

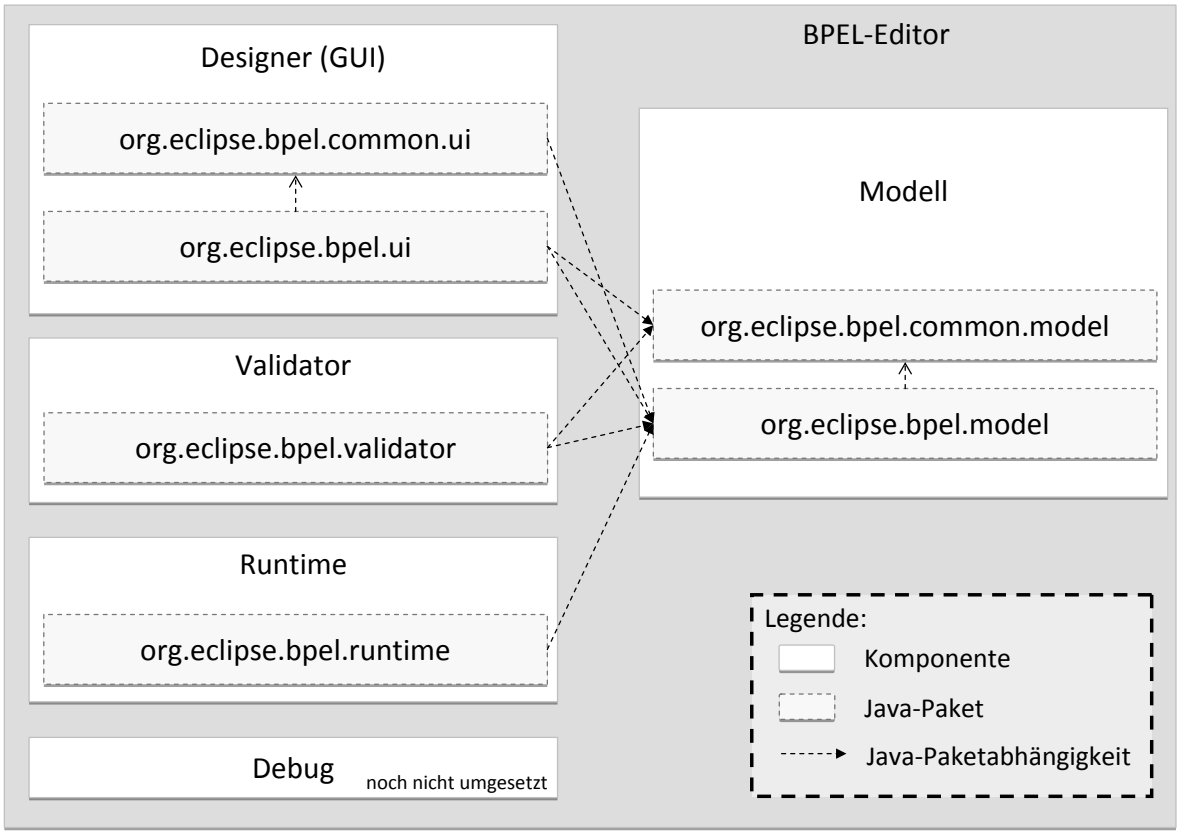


Abbildung 65: Architektur des BPEL-Editors

Die wichtigsten Pakete für die Entwicklung des BPEL^{light} Editors, die Pakete `org.eclipse.bpel.model` und `org.eclipse.bpel.ui` werden im Folgenden kurz betrachtet. Eine detaillierte Beschreibung der Pakete findet sich in [Fe07].

Das Paket `org.eclipse.bpel.model`

Das Paket `org.eclipse.bpel.model` enthält das EMF-Datenmodell und die Objektrepräsentation dieses Modells. Zusätzlich befinden sich dort die Klassen für das Deserialisieren (Einlesen) und das Serialisieren (Schreiben) der `.bpel`-Dateien. Die Klasse `BPELReader` ist für das Einlesen einer `.bpel`-Datei zuständig. Ihr Gegenstück, die Klasse `BPELWriter`, schreibt das aktuelle Objektmodell des aktuellen Prozesses in eine `.bpel`-Datei.

Das Paket `org.eclipse.bpel.ui`

Das Paket `org.eclipse.bpel.ui` beinhaltet die grafische Darstellung des BPEL-Editors. Neben den Klassen, die für das Einbinden in das Eclipse

Framework notwendig sind, wie beispielsweise die Klasse `BPELEditor`, die den Einstiegspunkt in den Editor bildet, sind hier insbesondere die Steuerung und Präsentation des MVC-Konzepts implementiert.

Außer der grafischen Repräsentation der Objekte, d.h. der WS-BPEL-Konstrukte (Aktivitäten, PartnerLinks, Variablen, usw.), findet sich in diesem Paket die Umsetzung für Actions, Dialoge, Wizards, Commands, Properties sowie weitere Hilfsklassen.

6.2.1.2 Grafischer Aufbau des BPEL-Editors

Der Grafische Aufbau des BPEL-Editors ist in Abbildung 66 dargestellt. Er besteht aus folgenden Bereichen:

1. Die *Workspaceadministration* (deutsch: Projekt- und Dateiverwaltung) ermöglicht das Navigieren durch Projekte und Dateien.

2. Die *Editor-View* ermöglicht das Anzeigen und Bearbeiten eines Prozesses. Sie besteht aus zwei sogenannten View-Tabs, dem Design-View-Tab und dem Source-View-Tab. Das Source-View-Tab stellt den aktuellen Quellcode des angezeigten Prozesses dar. Das Design-View-Tab setzt sich aus dem Editor-Canvas, der Palette sowie dem Process Tray zusammen, die im Folgenden beschrieben sind:

    a. Im *Editor-Canvas* (deutsch: Editor-Leinwand) erfolgt der Aufbau des Prozesses. In ihm können u.a. Aktivitäten erstellt und angeordnet und ein Kontrollfluss zwischen ihnen definiert werden.

    b. Die *Palette* ist eine Auflistung der für die Prozessmodellierung zur Verfügung stehenden Aktivitätstypen. Das Erstellen neuer Aktivitäten im Editor-Canvas ist u.a. durch Drag&Drop von Aktivitätstypen aus der Palette möglich. Zusätzlich bietet sie Platz für allgemeine Funktionen wie Selektionswerkzeuge oder Zoomfunktion.

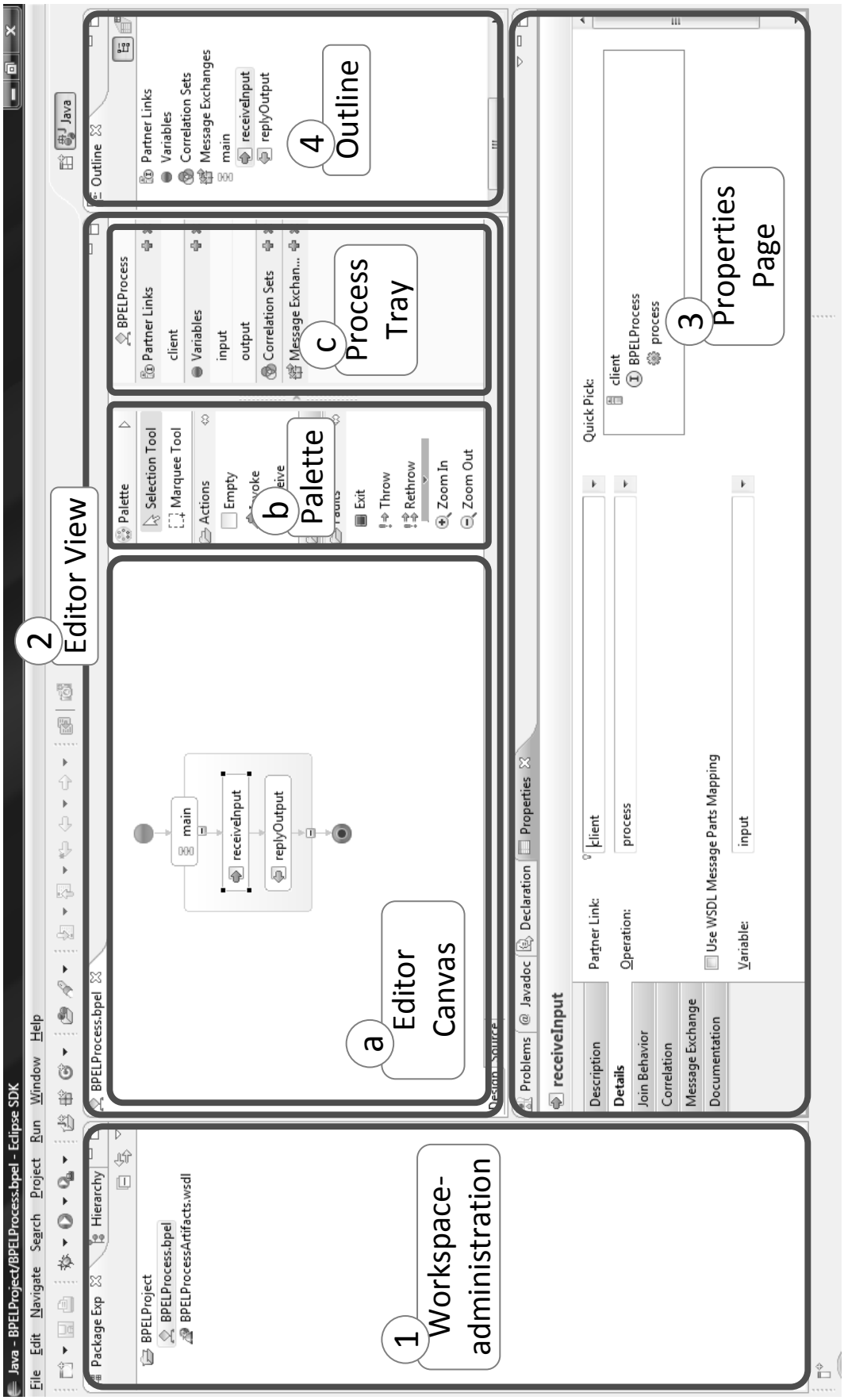


Abbildung 66: Grafischer Aufbau des BPEL-Editors

c. Der *Process Tray* (deutsch: Prozessablage) dient dazu, das Anzeigen, Erstellen, Bearbeiten und Löschen von WS-BPEL-Sprachkonstrukten zu ermöglichen, die keinen direkten Einfluss auf den Kontrollfluss besitzen, also nicht im Editor-Canvas platziert werden können.

3. Die *Properties Page* bildet zusammen mit der Editor-View den Kern der Prozessesmodellierung. Sie ermöglicht u.a. das Setzen von Werten der Sprachkonstrukte.

4. Die *Outline* bietet eine als Baum strukturierte Ansicht des Prozesses.

6.2.2 Erweiterung des Eclipse BPEL-Designers zu einem BPEL^{light}-Editor

Um die Konzepte umsetzen zu können, die BPEL^{light} zugrunde liegen, musste der Eclipse BPEL-Designer um die Modellierung von Nachrichtenaustauschmustern, d.h. von abstrakten BPEL^{light}-Prozessen, und um die Modellierung von ausführbaren BPEL^{light}-Prozessen erweitert werden. Die Modellierung der beiden BPEL^{light}-Prozessarten setzt wiederum voraus, dass u.a. die Sprachelemente `<bl:conversation>` und `<bl:partner>` aber auch die Aktivitäten `<bl:pick>` und `<bl:interactionActivity>` im Editor zur Verfügung stehen.

Die Umsetzung der genannten Anforderungen im Eclipse BPEL-Editor ist Gegenstand dieses Abschnitts. Zunächst wird auf das Erstellen eines BPEL^{light} Projekts und eines BPEL^{light}-Prozesses eingegangen. Anschließend wird das erweiterte Datenmodell vorgestellt und das Einlesen bzw. Schreiben einer BPEL Datei beschrieben. Schließlich wird auf die Implementierung der Elemente `<bl:partner>`, `<bl:conversation>` und `<bl:expression>` sowie der Aktivitäten `<bl:interactionActivity>` und `<bl:pick>` eingegangen.

6.2.2.1 Erstellen eines BPEL^{light} Projektes

Im Zuge der Implementierung des BPEL^{light}-Editors wurde der BPEL-Editor um die Projektart „BPEL^{light}“ erweitert. Hierzu wurde das Paket `org.eclipse.bpel.runtime` in dem die Projektkategorien definiert sind,

angepasst. Über den Extension Point `org.eclipse.ui.newWizards` lädt Eclipse nun beim Starten der Anwendung eine neue BPEL^{light}-Wizard-Klasse, mit deren Hilfe der Modellierer ein neues BPEL^{light}-Projekt erstellen kann.

6.2.2.2 Erstellen eines BPEL^{light}-Prozesses

Für die Erstellung neuer Prozesse existiert im BPEL-Editor ein Wizard. Dieser Wizard verwendet Templates, die das grundlegende Gerüst für einen Prozess vorgegeben. Der Benutzer wählt zu Beginn der Erstellung eines neuen Prozesses aus, welches Template er für seinen neuen Prozess verwenden möchte. Die noch offenen Informationen innerhalb des Templates, wie beispielsweise der Prozessname, werden durch den Wizard vom Modellierer erfragt und direkt in das Template eingesetzt. Dieses Vorgehen wurde für das Erstellen von BPEL^{light}-Prozessen übernommen.

Es existieren zwei Arten von BPEL^{light}-Prozessen, ausführbare und abstrakte Prozesse. Für beide BPEL^{light}-Prozessarten wurde ein Wizard erstellt, um einen neuen Prozess über ein Template zu erstellen. Der Unterschied für den Aufbau der Templates liegt im Namensraum, der für die Prozesse eingebunden werden muss. Im Falle eines abstrakten Prozesses muss zusätzlich auf das entsprechende Abstract Process Profile verwiesen werden. Bei beiden ist der BPEL^{light}-Namensraum definiert, ein passendes `<bpel:extension>`-Element eingefügt sowie die Elemente `<bl:partners>`, `<bl:conversations>` und `<bl:expressions>` bereits vordefiniert. Ansonsten enthalten die Templates keine weiteren Informationen, die für die Geschäftslogik relevant sind. Sie werden daher „Empty BPEL^{light} Process“ bzw. „Empty BPEL^{light} abstract Process“ genannt.

6.2.2.3 Erweiterung des Datenmodells

Wie bereits erwähnt, basiert der BPEL-Designer auf einem EMF-Modell, das (indirekt) durch das XSD von BPEL mit Informationen gespeist wird. Um das Modell mit BPEL^{light} zu erweitern, wurde das XSD (bzw. Model), in dem die BPEL^{light}-Erweiterungen definiert sind, ebenfalls als Ressource für das EMF-Modell des BPEL-Designers angegeben, sodass der Java-Quellcode über eben dieses Modell generiert werden kann.

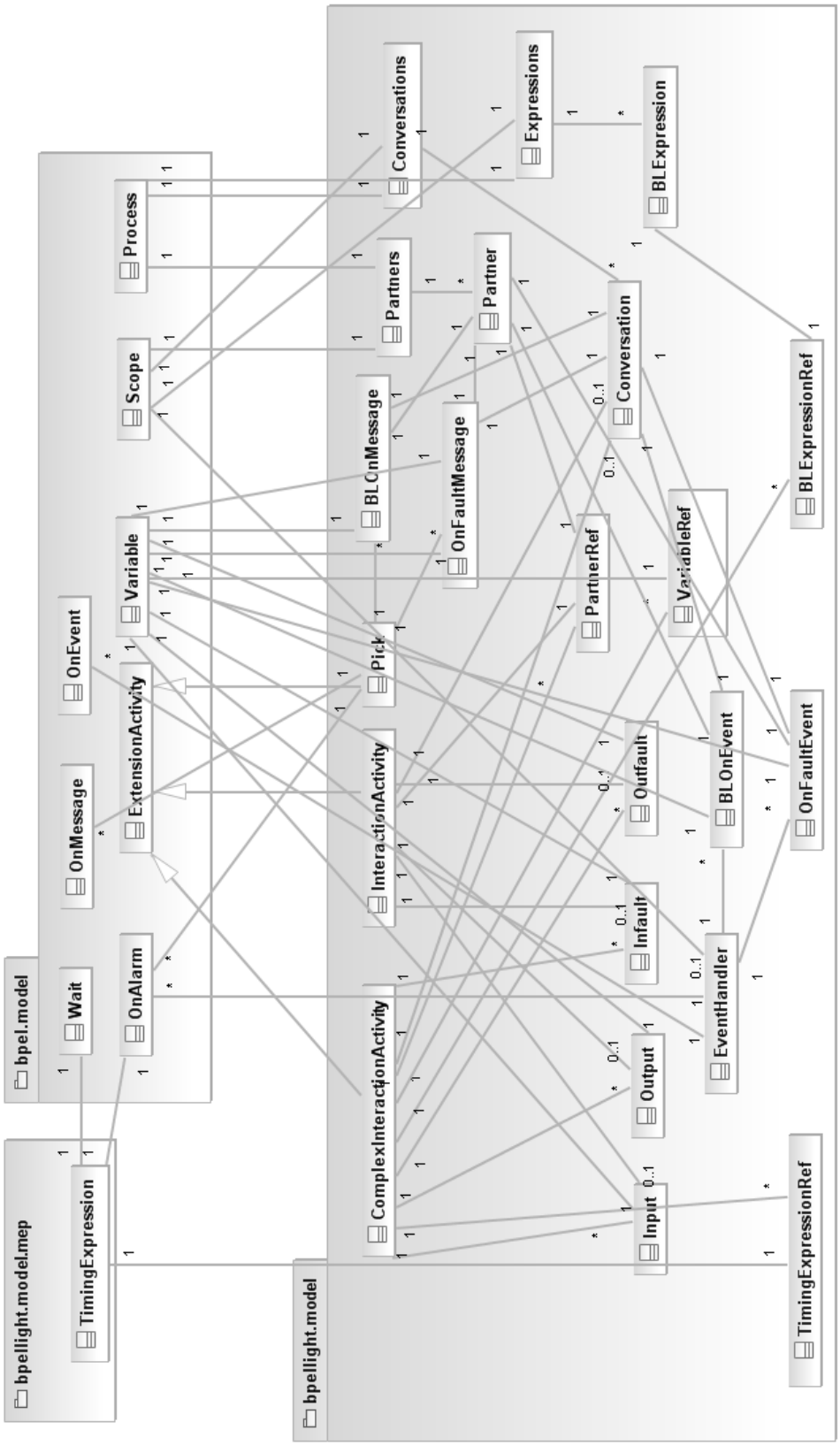


Abbildung 67: Klassenmodell des BPEL^{light}-Editors [Ho08]

Abbildung 67 zeigt die Klassen, die generiert wurden und ihre Relationen zueinander. Jede Klasse erbt direkt oder indirekt von der Klasse `org.eclipse.bpel.model.ExtensibleElement`. Bei der Darstellung wird von Attributen und Methoden abstrahiert. Aus dem BPEL-Modell sind nur die für die Erweiterung relevanten Klassen abgebildet.

Für jedes BPEL^{light}-Sprachkonstrukt werden zwei Javaklasse generiert. Eine Interface-Klasse und eine Implementierungsklasse. Programmierlogik enthält nur die Implementierungsklasse. Abweichungen des Datenmodells gegenüber den in Kapitel 3 vorgestellten Sprachkonstrukten von BPEL^{light} werden im Folgenden direkt in den entsprechenden Umsetzungen der Sprachkonstrukte beschrieben.

6.2.2.4 Einlesen und Schreiben einer BPEL-Datei

Wie bereits erwähnt, wird zum Einlesen bzw. Schreiben einer `.bpel`-Datei die Klasse `BPELReader` bzw. `BPELWriter` verwendet. Diese sind jedoch nur in der Lage, WS-BPEL-Sprachkonstrukte einzulesen bzw. zu schreiben. Um der Erweiterbarkeit von WS-BPEL Rechnung zu tragen, existiert im BPEL-Editor ein Erweiterungsmechanismus für das Verarbeiten unbekannter Aktivitäten. Diesem Erweiterungsmechanismus folgend wurden für die Aktivitäten `<bl:interactionActivity>` und `<bl:pick>` Klassen zum Einlesen bzw. Schreiben implementiert und in der Klasse `ExtensionRegistry` registriert. Kommt der `BPELReader` bzw. `BPELWriter` nun bei der Verarbeitung zu einer `<bpel:extensionActivity>`, holt er sich bei der `ExtensionRegistry` eine Instanz der Klasse, die für das Einlesen bzw. Schreiben verantwortlich ist. Abbildung 68 zeigt das Vorgehen zum Einlesen einer `<bpel:extensionActivity>`. Das Schreiben einer `<bpel:extensionActivity>` erfolgt analog.

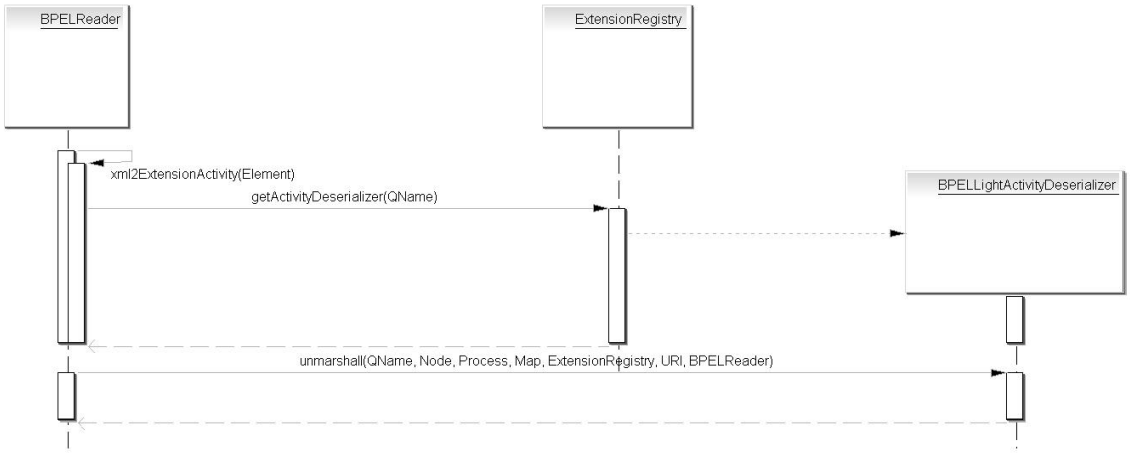


Abbildung 68: Ablauf der Deserialisierung einer `<bpel:extensionActivity>` [Hoeh08]

Für alle anderen BPEL^{light}-Konstrukte, für die kein Erweiterungmechanismus genutzt werden konnte, musste sowohl die Klasse `BPELReader` als auch `BPELWriter` um entsprechende Methoden zum Einlesen bzw. Schreiben der neuen Konstrukte erweitert werden.

6.2.2.5 Erweiterung der Implementierung um `<bl:partner>`, `<bl:conversation>` und `<bl:expression>`

Die Besonderheit der BPEL^{light}-Sprachkonstrukte `<bl:partner>`, `<bl:conversation>` und `<bl:expression>` liegt darin, dass sie nicht Teil des Kontrollflusses sind. Sie werden für den Prozess selber oder nur innerhalb eines gewissen Geltungsbereichs definiert und können dann von den darin definierten Aktivitäten referenziert werden. In dieser Hinsicht sind sie vergleichbar mit den BPEL-Sprachkonstrukten Partnerlink, Correlation Set und Variable.

Grafische Darstellung

Da die genannten Konstrukte nicht direkt im Kontrollfluss vorkommen, werden sie nicht im Editor-Canvas sondern im Process Tray angezeigt. Es werden immer die für den Prozess definierten Konstrukte angezeigt, außer die aktuelle Selektion ist ein `<bpel:scope>`. In diesem Fall zeigt der Process Tray die für den `<bpel:scope>` definierten Konstrukte an. Um BPEL^{light} zu

unterstützen, wurde der Process Tray erweitert, so dass zusätzlich zu den BPEL-Konstrukten die Konstrukte `<bl:partner>`, `<bl:conversation>` und `<bl:expression>` dargestellt werden können.

Abbildung 69 zeigt ein Beispiel für die Definition von BPEL^{light}-Konstrukten, die für den Prozess „buyProcess“ (vergleiche Listing 24) definiert wurden.

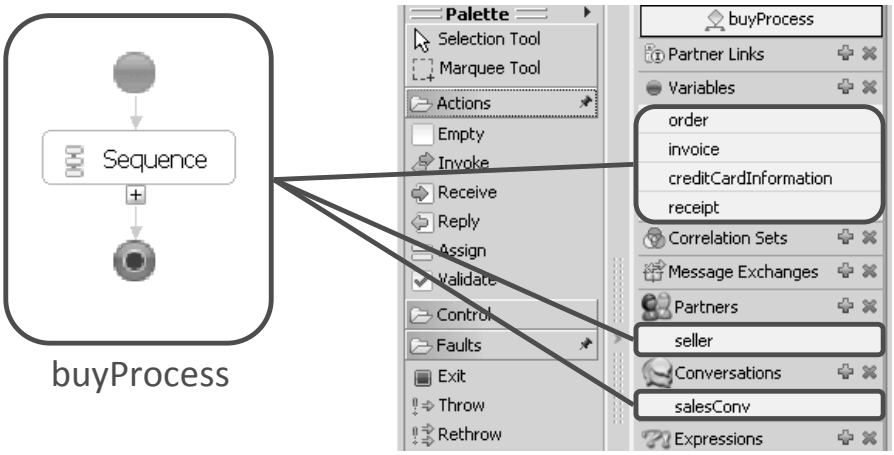


Abbildung 69: Grafische Darstellung der für den Prozess „buyProcess“ definierten BPEL^{light}-Konstrukte

Umsetzung des Konzepts im Modell

Für alle drei Konstrukte wurde eine eigene Klasse eingeführt. Eine Änderung gegenüber den im Konzept beschriebenen Konstrukten war nicht notwendig.

Erzeugen neuer Objekte

Das Erzeugen neuer Objekte für die Sprachkonstrukte ist direkt über den Process Tray möglich. Für jede Kategorie (repräsentiert durch die Behälterobjekte Partners, Conversations und Expressions) existiert ein mit einem Pluszeichen versehener Button, über den ein entsprechendes Konstrukt erzeugt werden kann. Ein Klick bei Partners erzeugt beispielsweise ein dem `<bl:partner>`-Konstrukt entsprechendes Objekt. Je nachdem welcher Geltungsbereich bei der Erstellung des neuen Objektes aktiv ist, wird es entweder für den Prozess oder den aktuell selektierten Geltungsbereich erzeugt.

Bearbeiten vorhandener Objekte

Das Setzen des Attributs `name` bei Partner, Conversation und Expression ist direkt im Process Tray möglich. Die Attribute `expressionLanguage` und `value` der `<bl:expression>` können in der Properties Page nach Selektion der Expression im Process Tray bearbeitet werden. Handelt es sich bei vorliegendem Prozess um einen abstrakten Prozess zur Definition eines Nachrichtenaustauschmusters so können die beiden Attribute auf `##opaque` gesetzt werden.

Löschen der Objekte

Das Entfernen von Objekten ist nur für Partner-, Conversation- und Expression-Objekte möglich. Die Behälterobjekte Partners, Conversations und Expressions bleiben existent. Analog zum Anlegen der Objekte existiert im Process Tray für jede Kategorie ein Button, der als graues „X“ dargestellt wird. Dieser entfernt die aktuelle Selektion der Objekte in seiner Kategorie. Alternativ ist es möglich, die selektierten Objekte durch den Tastaturbefehl „Entf“ oder über den „Delete“-Eintrag im Kontextmenü zu entfernen.

6.2.2.6 Erweiterung der Implementierung um `<bl:interactionActivity>` und `<bl:pick>`

Sowohl die `<bl:interactionActivity>` als auch die `<bl:pick>`-Aktivität sind `<bpel:extensionActivity>`s, also bezüglich ihrer Verwendung vergleichbar mit anderen BPEL-Aktivitäten wie beispielsweise `<bpel:invoke>` oder `<bpel:empty>`. Im Gegensatz zu den bereits beschriebenen BPEL^{light} Sprachkonstrukten sind die `<bl:interactionActivity>` und die `<bl:pick>`-Aktivität also Teil des Kontrollflusses.

Grafische Darstellung

Die `<bl:interactionActivity>` und das `<bl:pick>` werden im Editor-Canvas analog zu den WS-BPEL-Aktivitäten dargestellt. Die Darstellung entspricht der Notation für Aktivitäten in der BPMN 1.1-Spezifikation (vgl. [OMG08]), wonach Aktivitäten als Rechteck mit abgerundeten Ecken dargestellt werden.

Eine `<bl:pick>`-Aktivität muss zusätzlich noch Kindelemente anzeigen können. Dazu wird die Darstellung der `<bpel:pick>`-Aktivität aus dem BPEL-Editor übernommen (siehe Abbildung 70). Diese zeigt die Kindelemente in einem weiteren Rechteck mit abgerundeten Ecken an, im Folgenden Container genannt. Der Container kann „geöffnet“ (b) oder „geschlossen“ (a) werden, je nachdem ob der Inhalt angezeigt werden soll oder nicht.

Abbildung 71 zeigt die grafische Darstellung der `<bl:interactionActivity>`. Ist die darzustellende `<bl:interactionActivity>` eine komplexe Aktivität, die ein Nachrichtenaustauschmuster referenziert, so wird sie analog zur `<bl:pick>`-Aktivität mit einem Container unter der eigentlichen Aktivität dargestellt (b). Ist der Container geöffnet, so wird in ihm der Kontrollfluss des Nachrichtenaustauschmusters angezeigt (c) (in dieser Abbildung handelt es sich um das Nachrichtenaustauschmuster „Out-Optional-In“). Handelt es sich um eine einfache `<bl:interactionActivity>`, so wird sie ohne Container dargestellt (a).

a) `<bl:pick>` mit geschlossenem Container

b) `<bl:pick>` mit geöffnetem Container

Abbildung 70: Grafische Darstellung der `<bl:pick>`-Aktivität

a) einfache `<bl:interactionActivity>`

b) komplexe `<bl:interactionActivity>` mit geschlossenem Container

c) komplexe `<bl:interactionActivity>` mit geöffnetem Container

Abbildung 71: Grafische Darstellung der `<bl:interactionActivity>`

Umsetzung des Konzepts im Modell

Sowohl die `<bl:interactionActivity>` als auch die `<bl:pick>`-Aktivität werden dem Konzept entsprechend als `<bpel:extensionActivity>`s realisiert, d.h. beide Aktivitätsklassen erben von der Klasse `BPELExtensionActivity`.

Die Umsetzung der `<bl:interactionActivity>` erfolgt durch zwei Objektrepräsentationen, eine zur Modellierung von einfachen `<bl:interactionActivity>`s und eine zur Modellierung von komplexen `<bl:interactionActivity>`s. Die Objektrepräsentation der komplexen `<bl:interactionActivity>` ermöglicht das Referenzieren einer `<bl:conversation>` und eines Nachrichtenaustauschmusters. Dem

Nachrichtenaustauschmuster entsprechend können zudem mehrere `<bl:partner>` referenziert und mehrere eingehende bzw. ausgehende Nachrichten sowie Ausdrücke definiert werden. Die Objektrepräsentation der einfachen `<bl:interactionActivity>` ermöglicht das Referenzieren einer `<bl:conversation>` und eines `<bl:partner>`s sowie die Definition einer eingehenden bzw. ausgehenden Nachricht. Zudem ist es möglich, über das Attribut `mep` ein Nachrichtenaustauschmuster zu referenzieren. Wird diese Möglichkeit genutzt, so wird die Aktivität beim nächsten Einlesen als komplexe `<bl:interactionActivity>` behandelt.

Ein `<bl:pick>` wird über die Klasse `Pick` realisiert. Eine Abweichung von der Beschreibung gegenüber dem Konzept findet für die Umsetzung nicht statt.

Erzeugen neuer Aktivitäten

Zum Erstellen der neuen Aktivitäten wurden die Möglichkeiten des Erzeugens per Kontextmenü und Palette genutzt. Die Einträge im Kontextmenü werden durch die Einträge in der Palette vorgegeben. Es war also nur eine Erweiterung der Palette notwendig. Diese konnte über eine vorhandene Erweiterungsschnittstelle vorgenommen werden. Beim Aufbau der Palette werden nun zusätzlich zu den WS-BPEL-Aktivitäten BPEL^{light}-Aktivitäten in die Palette geladen.

Bearbeiten von Aktivitäten

Das Setzen des Attributs `name` der Aktivitäten sowie der Kindelemente `<bl:onMessage>`, `<bl:onFaultMessage>` und `<bl:onAlarm>` ist direkt auf dem Editor Canvas möglich. Alle weiteren Kindelemente und Attribute der Aktivitäten sowie der eben genannten Kindelemente können in der Properties Page nach Selektion des entsprechenden Objekts im Editor Canvas bearbeitet werden. Handelt es sich bei vorliegendem Prozess um einen abstrakten Prozess zur Definition eines Nachrichtenaustauschmusters, so können dem Profil für abstrakte BPEL^{light}-Prozesse folgend manche Attribute auf `##opaque` gesetzt werden.

Löschen von Aktivitäten

Das Löschen von Aktivitäten ist über zwei Arten möglich. Entweder über einen Kontextmenüeintrag oder den Tastaturbefehl „Entf“. Um Aktivitäten aus dem aktuell geöffneten Prozess zu entfernen, werden diese im Editor-Canvas selektiert.

6.2.3 Editor zur Abbildung auf WSDL 1.1

Soll BPEL^{light} in einer Web-Service-Umgebung eingesetzt werden, muss die Modellierungsumgebung neben einem BPEL^{light} Editor eine Deployment Komponente und geeignete Editoren zur Verfügung stellen, um WSDL-Schnittstellen zu spezifizieren und die Abbildung der BPEL^{light}-Prozesslogik auf WSDL vornehmen zu können. Die Architektur einer solchen Modellierungsumgebung ist in Abbildung 72 dargestellt.

Die Erstellung der WSDL-Beschreibungen ist u.a. mit dem WSDL-Editor von Eclipse[25] möglich. Der Deployment Deskriptor, der die Abbildung der Prozesslogik auf die erstellten WSDL-Beschreibungen definiert, kann u.a. mit dem XML Editor von Eclipse geschrieben werden. Eine graphische Benutzeroberfläche steht hierzu derzeit nicht zur Verfügung.

Abbildung 72: Modellierungsumgebung zum Einsatz von BPEL^{light}-Prozessen in Web-Service-Umgebungen

[25] http://www.eclipse.org/webtools/

6.2.4 BPEL4SWS-Editor – Abbildung von BPEL^{light} auf Semantische Web-Services

Soll BPEL^{light} in Ausführungsumgebungen für Semantische Web-Services eingesetzt werden, muss die Modellierungsumgebung neben den im vorherigen Abschnitt gezeigten Komponenten zusätzlich über einen Editor zur Definition von semantischen Beschreibungen verfügen. Die Architektur einer solchen Modellierungsumgebung ist in Abbildung 73 dargestellt.

Zur Definition der semantischen Beschreibungen kann unter anderem auf WSMO Studio[26] zurückgegriffen werden. WSMO Studio ist ein auf Eclipse basierendes semantisches Web-Service-Modellierungswerkzeug und umfasst u.a. einen Ontologieeditor, einen Editor zur Erstellung von WSMO-Web-Service- und Goal-Beschreibungen und einen SAWSDL-Editor zur Annotation von WSDL-Dateien mit Ontologiekonzepten. Es können also alle SWS-spezifischen Artefakte mit WSMO Studio erstellt werden.

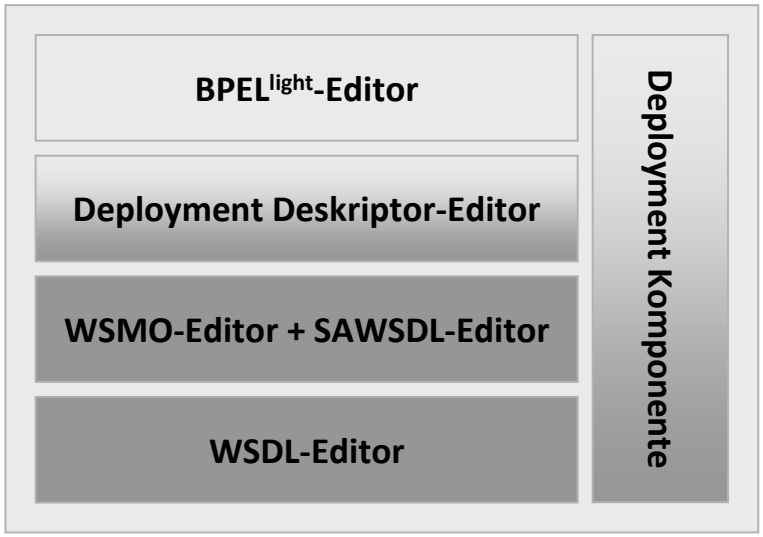


Abbildung 73: Modellierungsumgebung zum Einsatz von BPEL^{light}-Prozessen in Semantischen Web-Service-Ausführungsumgebungen

[26] http://www.wsmostudio.org/

6.3 Zusammenfassung

In diesem Kapitel wurde die Architektur und Implementierung einer Infrastruktur präsentiert, die es ermöglicht, BPEL^{light}-Prozesse zu modellieren und auszuführen.

Die vorgestellte Modellierungsumgebung umfasst einen BPEL^{light} Editor, eine Deployment Komponente und Editoren zur Erstellung von Deployment Deskriptoren bzw. zur Spezifikation von Schnittstellen, die jeweils von der verwendeten Servicetechnologie abhängen. Der BPEL^{light} Editor ermöglicht sowohl die Definition von Nachrichtenaustauschmustern in Form von abstrakten BPEL^{light}-Prozessen als auch die Definition ausführbarer BPEL^{light}-Prozesse. Die Verwendung von komplexen `<bl:interactionActivity>`s, deren Ausführungssemantik über Nachrichtenaustauschmuster definiert sind, wird vom BPEL^{light} Editor zur Definition beider Prozessarten unterstützt.

Eine ähnliche Aufteilung wie bei der Modellierungsumgebung findet sich auch in der präsentierten Ausführungsumgebung wieder: Neben der BPEL^{light} spezifischen Engine gibt es Komponenten, die von der jeweiligen Servicetechnologie abhängen. Dies sind die Discovery Komponente, das Repository, die Vermittlungs- (bzw. Mediation-) Komponente, die fachliche Funktionalität realisierenden zur Verfügung stehenden Dienste sowie die Integrationschicht der BPEL^{light}-Engine, welche die Abbildung von BPEL^{light} auf die Servicetechnologie übernimmt und somit die BPEL^{light}-Engine in die Ausführungsinfrastruktur einbettet. Die BPEL^{light}-Engine unterstützt nicht nur die Ausführung von Prozessen, die lediglich von einfachen `<bl:interactionActivity>`s Gebrauch machen. Die abstrakten Nachrichtenaustauschmuster, welche die Semantik von komplexen `<bl:interactionActivity>`s beschreiben, werden durch Parametrisierung in ausführbare MEP-Prozesse überführt, die dann zur Laufzeit von den entsprechenden komplexen `<bl:interactionActivity>`s aufgerufen werden.

Bei der Umsetzung der Architektur konnte auf existierende Komponenten zurückgegriffen werden. Die BPEL^{light} spezifischen Komponenten, d.h. der BPEL^{light} Editor und die BPEL^{light}-Engine, wurden durch Erweiterung

entsprechender BPEL-Implementierungen – des Eclipse BPEL-Editors und ODE – realisiert. Das zeigt, dass der Aufbau einer BPEL^{light} Infrastruktur mit vertretbarem Aufwand möglich ist und keine vollständige Neuimplementierung erfordert.

Die präsentierten Ergebnisse finden sich auch in den bereits veröffentlichten Publikationen [NHL+08, KWL+07, KLL+08b, LDN+07, NLKL07, NLKL07c, NLKL09].

KAPITEL 7

ZUSAMMENFASSUNG UND AUSBLICK

Zusammenfassung

Die Business Process Execution Language (BPEL) ist der erste allgemein anerkannte Standard zur Spezifikation ausführbarer Geschäftsprozesse. BPEL bietet Konzepte zur Spezifikation des Kontroll- und Datenflusses zwischen einzelnen Aktivitäten sowie zur Spezifikation der Anforderung an die Implementierung von Aktivitäten. Ein BPEL-Prozess beantwortet somit die beiden Fragen „*Wie* wird die vom Prozess zu erbringende Funktionalität realisiert bzw. durch die Abfolge welcher Aktivitäten?“ und „*Womit* wird die Funktionalität einer einzelnen Aktivität realisiert?“. Die Anforderungen an Aktivitätsimplementierungen werden in BPEL mittels der Web Service Description Language (WSDL) formuliert und direkt in die Prozessmodelle kodiert. Allerdings verweisen die Aktivitäten lediglich auf die Operation, d.h. auf die abstrakte Schnittstellenbeschreibung des zu verwendenden Web-Services. Sie werden in der Prozessbeschreibung nicht gegen die tatsächliche Implementierung der Operation gebunden. Es ist also möglich, zur Laufzeit frei zwischen unterschiedlichen, die angegebene Operation implementierenden Diensten zu wählen. Auf den ersten Blick ermöglicht es BPEL dadurch einem Unternehmen seine Geschäftspartner jederzeit zu wechseln, ohne die eigenen Prozesse neu implementieren zu müssen. Angesichts des wachsenden Preisdrucks, der durch die Globalisierung auf weltweit agierende Unternehmen herrscht, ist diese Möglichkeit, sich zur Laufzeit die kostengünstigsten Dienste auswählen zu können, nicht hoch genug einzuschätzen.

Allerdings bieten unterschiedliche Organisationen bzw. Unternehmen die gleiche Funktionalität in der Regel auf unterschiedliche Art und Weise an, d.h. unter Verwendung unterschiedlicher Porttypen und Operationen. Dass

sich dieser Umstand kurz- oder mittelfristig ändert, dass es also standardisierte domänenspezifische Porttypen und Operationen gibt, wird von der Industrie als unrealistisch eingeschätzt. Folglich müssen BPEL-Prozessmodelle in der Regel jedes Mal geändert werden, wenn Dienste eines anderen, Anbieters in Anspruch genommen werden sollen. Dies ist aufwendig und schränkt die Agilität von Unternehmen in nicht unbedeutendem Maße ein.

Um die Flexibilität zu erhöhen, wurde in dieser Arbeit zunächst das Konzept einer von WSDL unabhängigen BPEL-basierten Workflowsprache eingeführt. Anschließend wurden alternative Ansätze zur Definition einer solchen Sprache vorgestellt, analysiert und bewertet. Der Erfolg versprechendste Ansatz wurde dann in dieser Arbeit im Detail ausgearbeitet und trägt den Namen BPEL^{light}. Er spezifiziert ein von WSDL-Beschreibungen unabhängiges Interaktionsmodell als Erweiterung zu BPEL. Die Verwendung des neuen Interaktionsmodells ermöglicht es, zu jedem Zeitpunkt des Prozesslebenszyklus eine geeignete Aktivitätsimplementierung auszuwählen und dies unabhängig von der zum Auffinden und Aufrufen der Implementierung notwendigen Servicetechnologie.

Die Unabhängigkeit von WSDL ist jedoch nicht die einzige positive Eigenschaft des neuen Interaktionsmodells. Darüber hinaus besitzt es eine höhere Mächtigkeit als das WSDL-basierte Modell von BPEL: Es ermöglicht den logischen Zusammenhang mehrerer bilateraler Interaktionen zu erfassen. Damit ist es nicht wie BPEL beschränkt auf die Definition einer Ansammlung bilateraler Interaktionen, sondern ermöglicht die explizite Definition multilateraler Interaktionen. Außerdem definiert BPEL^{light} nicht wie herkömmliches BPEL einen den WSDL 1.1-Operationstypen entsprechenden fixen Satz von Interaktionsaktivitäten, sondern erlaubt es, die Semantik von Interaktionsaktivitäten mittels so genannter Nachrichtenaustauschmuster zu spezifizieren. Um dies zu ermöglichen, wurde in dieser Arbeit ein Profil für abstraktes BPEL^{light} entwickelt, welches es ermöglicht, wieder verwendbare Aktivitätsbeschreibungen zu erstellen.

Auch wenn in dieser Arbeit ein von WSDL unabhängiges BPEL definiert wurde, hat sie die Popularität der Web-Service-Technologie dennoch nicht außer Acht gelassen. Es wurden Konzepte vorgestellt, die es ermöglichen, BPEL^{light} in einer Web-Service-Umgebung einzusetzen, d.h. von Web-Services als Aktivitätsimplementierungen Gebrauch zu machen und die Funktionalität des BPEL^{light}-Prozesses als Web-Service zur Verfügung zu stellen. Um zu unterstreichen, dass BPEL^{light} jedoch auch in anderen Implementierungen der SOA eingesetzt werden kann, die nicht auf Web-Services basieren, wurden Konzepte vorgestellt, die den Einsatz von BPEL^{light} in Semantischen Web-Service-Umgebungen ermöglichen (BPEL4SWS). In diesem Zuge wurde das SWS-Framework WSMO sowie die Architektur der zugehörigen Referenzimplementierung erweitert, um für Geschäftsprozesse typische, lang laufende, asynchrone Kommunikation mit Semantischen Web-Service unterstützen zu können. Die Art und Weise der Anbindung der Web-Service- und Semantischen Web-Service-Technologie an BPEL^{light} ermöglicht eine für den Nutzer des vom Prozess angebotenen Dienstes transparente Migration eines BPEL-Prozesses zu einem BPEL^{light}-Prozess. Für einen BPEL4SWS-Prozess bedeutet das auch, dass er jederzeit sowohl über eine semantisch beschriebene als auch über eine klassische WSDL-Schnittstelle aufgerufen werden kann.

Die erarbeiteten Konzepte wurden abschließend durch eine Architektur und prototypische Implementierung einer Entwicklungsumgebung sowie zugehöriger Laufzeitumgebungen validiert.

Ausblick

Diese Arbeit hat Abbildungsvorschriften von BPEL^{light} Nachrichten bzw. Konversationen auf die Web-Service-Technologie bzw. auf die Semantische Web-Service-Technologie (WSMO) definiert und eine Umsetzung der Abbildung mittels Deployment Deskriptoren vorgestellt. Durch die Entkopplung von Dienstbeschreibung und der Beschreibung der Anforderungen eines Klienten in WSMO kann dadurch bei Verwendung semantischer Web-Services zur Laufzeit jeder funktional geeigneter Dienst ausgewählt werden.

Allerdings unterstützt die in dieser Arbeit verwendete Implementierung des semantischen Service Busses (SSB) lediglich den blockierenden, synchronen Aufruf und nicht die konzeptionell mögliche asynchrone Kommunikation oder einen Nachrichtenaustausch, der mehr als zwei aufeinander folgende Nachrichten umfasst. Die Implementierung eines SSB mit voller Unterstützung aller in dieser Arbeit präsentierten Kommunikationsszenarien ist Teil unserer weiteren Arbeit. Ebenfalls weiteren Arbeiten überlassen bleibt die Implementierung der Komponente der BPEL4SWS-Engine, die es ermöglicht semantisch definierten Kontrollfluss auszuwerten.

Bei Verwendung der Web-Service-Technologie zum Auffinden einer geeigneten Aktivitätsimplementierung ist die Flexibilität zur Laufzeit eingeschränkt, da die Prozessbeschreibungen und die abstrakten Schnittstellenbeschreibungen nach dem Deployment miteinander gekoppelt sind. Um auch bei Verwendung der Web-Service-Technologie die Flexibilität zur Laufzeit gewährleisten zu können, muss es möglich sein, die Abbildung der BPEL^{light} Nachrichten auf WSDL-Operationen zur Laufzeit anpassen zu können. Eine Umsetzung dieser Flexibilität zur Laufzeit beispielsweise unter Verwendung von WS-PolicyAttachment [BHH+07] ist Teil unserer künftigen Arbeit.

Ebenfalls der weiteren Forschung überlassen bleibt die Validierung von Prozessfragmenten, welche die Semantik eines bestimmten Nachrichten-austauschmusters implementieren sollen, d.h. die Überprüfung der Korrektheit des Prozessfragments.

LITERATURVERZEICHNIS

URLs wurden zuletzt am 01.04.2011 geprüft.

[Ag86] Agha, G., „Actors: A Model of Concurrent Computation in Distributed Systems“, Doctoral Dissertation, MIT Press, 1986, online abrufbar unter: http://dspace.mit.edu/handle/1721.1/6952.

[AWBH00] van der Aalst, W. M. P., Barros, A. P., ter Hofstede, A. H. M. and Kiepuszewski. B., „Advanced Workflow Patterns”, Proc. Of Conference on Cooperative Information Systems (CoopIS2000), Eilat, Israel, Springer, LNCS Volume 19, September 2000.

[AAD+07a] Agrawal, A., Amend, M., Das, M., Ford, M., Keller, C., Kloppmann, M., König, D., Leymann, F., Müller, R., Pfau, G., Plösser, K., Rangaswamy, R., Rickayzen, A., Rowley, M., Schmidt, P., Trickovic, I., Yiu, A., Zeller, M., „WS-BPEL Extension for People (BPEL4People), Version 1.0“, OASIS Specification, June 2007, online abrufbar unter: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/BPEL4People_v1.pdf.

[AAD+07b] Agrawal, A., Amend, M., Das, M., Ford, M., Keller, C., Kloppmann, M., König, D., Leymann, F., Müller, R., Pfau, G., Plösser, K., Rangaswamy, R., Rickayzen, A., Rowley, M., Schmidt, P., Trickovic, I., Yiu, A., Zeller, M., „Web Services Human Task (WS-HumanTask), Version 1.0”, OASIS Specification, June 2007, online abrufbar unter: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/WS-HumanTask_v1.pdf.

[AFM+05] Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M.T., Sheth, A., Verma, K., „Web Service Semantics – WSDL-S“, W3C Member Submission, 2005, online abrufbar unter: http://www.w3.org/Submission/WSDL-S/.

[AAA+07] Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guízar, A., Kartha, N., Liu, C. K., Khalaf, R., König, D., Marin, M., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P. and Yiu A. (Eds.), „Web Services Business Process Execution Language, Version 2.0”, OASIS Specification, April 2007, online abrufbar unter: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

[BEF+06] Ballinger, K., Ehnebuske, D., Ferris, C., Gudgin, M., Liu, C., Nottingham, M. and Yendluri, P., „Basic Profile Version 1.1. WS-I specification”, 2006, online abrufbar unter: http://www.ws-i.org/Profiles/BasicProfile-1.1.html.

[BDH05] Barros, A., Dumas, M. and ter Hofstede, A., „Service Interaction Patterns”, Proc. Of the International Conference on Business Process Management (BPM 2005), Nancy, France, Springer Verlag, LNCS, September 2005.

[BDO05] Barros, A., Dumas, M. and Oaks, P., „A Critical Overview of the Web Services Choreography Description Language (WS-CDL)”, BPTrends Newsletter, Volume 3, Number 3, March 2005.

[BeFi99] Berners-Lee, T. and Fischetti, M., „Weaving the Web: the Original Design and Ultimative Destiny of the World Wide Web by its Inventors“, 1. Auflage, San Francisco, HarperCollins, September 1999, ISBN 0-06-251586-1.

[BiOr06] Biswas, R. and Ort, E., „The Java Persistence API – A Simpler Programming Model for Entity Persistence", May 2006, online abrufbar unter: http://java.sun.com/developer/technicalArticles/J2EE/jpa/.

[BFM98] Berners-Lee, T., Fielding, R. and Masinter, L., „Uniform Resource Identifiers (URI): Generic Syntax", RFC Editor , United States, 1998, online abrufbar unter: http://www.ietf.org/rfc/rfc2396.txt.

[BrGu04] Brickley, D. and Guha, R.V., „RDF Vocabulary Description Language 1.0: RDF Schema", W3C Recommendation, February 2004, online abrufbar unter: http://www.w3.org/TR/rdf-schema/.

[Br08] de Bruin, J., „WSML Abstract Syntax and Semantics", WSML Working Group Final Draft, August 2008, online abrufbar unter: http://www.wsmo.org/TR/d16/d16.3/v1.0/.

[Bu00] Burbeck, S., „The Tao of e-Business Services." IBM Corporation, 2000, online abrufbar unter: http://www.ibm.com/developerworks/webservices/library/ws-tao/.

[BHH+07] Boubez, T., Hirsch, F., Hondo, M. Orchard, D., Vedamuthu, A.s., Yalcinalp, Ü. And Yendluri, P., "Web Services Policy 1.5 – Attachment (WS-PolicyAttachment)", W3C Recommendation, September 2007, online abrufbar unter: http://www.w3.org/TR/ws-policy-attach/.

[CaSe06] Cardoso, J. and Seth, A., „Semantic Web Services, Processes and Applications (Semantic Web and Beyond: Computing for Human Experience)", Vol. 3, Springer, 2006, ISBN 978-0-387-30239-3.

[CCH+08] Carenini, A. , Cimpian, E., Haines, M., Kerrigan, M., Norton, B.,Mocan, A., Scicluna, J. and Zaremba, Michal, „Reference Ontology for Semantic Service Oriented Architectures", OASIS Committee Draft, November 2008, online abrufbar unter: http://www.oasis-open.org/committees/document.php?document_id=25381&wg_abbrev=semantic-ex.

[Ch04] Chappell, D. A., „Enterprise Service Bus", First Edition, O'Reilly, June 2004, ISBN 978-0-596-00675-4.

[CHL+07] Chinnici, R., Haas, H., Lewis, A., Moreau, J.-J., Orchard, D. and Weerawarana, S. „Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts". W3C Recommendation, June 2007, online abrufbar unter: http://www.w3.org/TR/wsdl20-adjuncts/.

[CMRW07] Chinnici, R., Moreau, J.-J., Ryman, A and Weerawarana, S., „Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language", W3C Recommendation, June 2007, online abrufbar unter: http://www.w3.org/TR/wsdl20/.

[CGKL03] Curbera, F., and Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S. and Weerawarana, S., „Business Process Execution Language for Web Service v1.1", May 2003, online abrufbar unter: http://www.ibm.com/developerworks/library/ws-bpel.

[CCMW01] Christensen, E., Curbera, F., Meredith, G. and Weerawarana, S., „Web Services Description Language (WSDL) 1.1", W3C Note, March 2001, online abrufbar unter: http://www.w3.org/TR/wsdl.

[CGK+02] Curbera, F., Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S. and Weerawarana, S., „Business Process Execution Language for Web Services, Version 1.0", July 2002, online abrufbar unter: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel1.pdf.

[CLK08] Cekov, L., van Lessen, T. and Konstantinov, M., „Deliverable 6.2: Process Execution Engine – Final Implementation", SUPER Deliverable, September 2008, online abrufbar unter: http://www.ip-super.org/res/Deliverables/M30/D6.6.pdf.

[Cl81] Clinger, W., „Foundations of Actor Semantics", Doctoral Dissertation, MIT, June 1981, online abrufbar unter: https://dspace.mit.edu/handle/1721.1/6935.

[DOS03] Dacota, M., Obrst, L. and Smith K, „The Semantic Web: A Guide to the Future of XML, Web Services and Knowledge Management", John Wiley & Sons, June 2003, ISBN 0-471-43257-1.

[DKLW07] Decker, G., Kopp, O., Leymann, F., and Weske, M., „BPEL4Chor: Extending BPEL for Modeling Choreographies", Proc. Of the International Conference on Web Services (ICWS 2007), Salt Lake City, Utah, USA, IEEE Computer Society Press, July 2007.

[DCH+04] Domingue, J., Cabral, L., Hakimpour, F., Sell, D. and Motta, E., „IRS-III: A Platform and Infrastructure for Creating WSMO-based Semantic Web Services", Proc. Of the Workshop on WSMO Implementations (WIW 2004), 2004, online abrufbar unter: http://iswc2004.semanticweb.org/demos/45/paper.pdf.

[FaLa07] Farrell, J. and Lausen, H., „Semantic Annotations for WSDL and XML Schema", W3C Recommendation, August 2007, online abrufbar unter: http://www.w3.org/TR/sawsdl/.

[Fe01] Fensel, D., „Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce", 1st Edition, Springer, August 2001, ISBN 978-3540416029.

[FeBu02] Fensel, D. and Bussler, C., „The Web Service Modeling Framework WSMF, Electronic Commerce Research and Applications", Elsevier, 2002.

[Fe07] Fernandez, J. V., „BPEL with Explicit Data Flow: Model, Editor, and Partitioning Tool", Diplomarbeit Nr. 2616, Institut für Architektur von Anwendungssystemen, Universität Stuttgart, Juli 2007.

[GHJV95] Gamma, E., Helm, R., Johnson, R and Vlissides, J., „Entwurfsmuster: Elemente wieder verwendbarer objektorientierter Software", Addison Wesley Verlag, Juli 2004, ISBN 978-3-8273-2199-2.

[GHVD03] Grosof, B.N., Horrocks, I., Volz, R. and Decker S., „Description Logic Programs: Combining Logic Programs with Description Logic", Proc. Of the Twelfth International World Wide Web Conference (WWW 2003), Budapest, ACM, May 2003.

[Gr93] Gruber, T. R., „A Translation Approach to Portable Ontologies", Knowledge Acquisition, Band 5, Nummer 2, Seite 199–220, 1993

[GHR06] Gudgin, M., Hadley, M. and Rogers, T., „Web Services Addressing 1.0 — Core", W3C Recommendation, May 2006, online abrufbar unter: http://www.w3.org/TR/ws-addr-core/.

[HBS73] Hewitt, C., Bishop, P. and Steiger, R., „A Universal Modular Actor Formalism for Artificial Intelligence", IJCAI, 1973, online abrufbar unter: http://dli.iiit.ac.in/ijcai/IJCAI-73/PDF/027B.pdf.

[HCM+05] Haller, A., Cimpian, E., Mocan, A., Oren, E. and Bussler, C., „WSMX – A Semantic Service-Oriented Architecture", Proc. International Conference on Web Services (ICWS 2005), Orlando, Florida, IEEE Computer Society Press, July 2005.

[HPH03] Horrocks, I., Patel-Schneider, P.F. and Van Harmelen, F., „From SHIQ and RDF to OWL: The Making of a Web Ontology Language", Journal of Web Semantics: Science, Services and Agents on the World Wide Web, Elsevier, 2003.

[Ho08] Höhensteiger, B., "Konzipierung und Implementierung eines BPEL^{light} Editors mit Unterstützung für Message Exchange Patterns", Diplomarbeit, Institut für Architektur von Anwendungssystemen, Universität Stuttgart, 2008, online abrufbar unter: http://elib.uni-stuttgart.de/opus/volltexte/2008/3829/.

[HoWo03] Hohpe, G. and Woolf, B., „Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions", Amsterdam, Addison-Wesley Longman, Oktober 2003, ISBN 978-0321200686.

[Ka06] Karastoyanova, D., „Enhancing Flexibility and Reusability of Web Service Flows through Parameterization", Doctoral Dissertation, TU Darmstadt, Shaker Verlag, 2006, ISBN 3-8322-5709-8.

[KaLe09] Karastoyanova, D., and Leymann, F., „BPEL'n'Aspects: Adapting Service Orchestration Logic", Proc. Of 7th International Conference on Web Services (ICWS 2009), Los Angeles, California, USA, IEEE Computer Society Press, July 2009.

[KLF+07] Konstantinov, M., van Lessen. T., Facca, F. M., Gaaloul, W., Vulcu G. and Schreder, B., „Deliverable 6.2: Process Execution Engine First Prototype", SUPER Deliverable, September, 2007, online abrufbar unter: http://www.ip-super.org/res/Deliverables/M18/D6.2.pdf.

[KLL+08] Karastoyanova, D., van Lessen, T., Leymann, F., Nitzsche, J. and Wutke, D., „WS-BPEL Extension for Semantic Web Services (BPEL4SWS), Version 1.0", Technischer Bericht Nr. 2008/03, Institut für Architektur von Anwendungssystemen, Universität Stuttgart, 2008, online abrufbar unter: ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-2008-03/TR-2008-03.pdf.

[KLL+08b] Karastoyanova, D., van Lessen, T., Leymann, F., Ma, Z., Nitzsche, J., Wetzstein, B., Bhiri, S., Hauswirth, M., and Zaremba, M., „A Reference Architecture for Semantic Business Process Management Systems", Multikonferenz Wirtschaftsinformatik (MKWI 2008), Munich, Germany, IEEE Computer Society Press, February 2008.

[KaLe09] Karastoyanova, D., and Leymann, F., „BPEL'n'Aspects: Adapting Service Orchestration Logic". In Proceedings of 7th International Conference on Web Services (ICWS 2009), IEEE Computer Society Press, July, 2009.

[KLNW06] Karastoyanova, D., Leymann, F., Nitzsche, J., Wetzstein, B. and Wutke, D., „Parameterized BPEL Processes: Concepts and Implementation", Proc. Of the International Conference on Business Process Management (BPM 2006), Short Papers, Vienna, Austria, Sep. 2006.

[KWL+07] Karastoyanova, D., Wetzstein, B., van Lessen, T., Wutke, D., Nitzsche, J. and Leymann, F., „Semantic Service Bus: Architecture and Implementation of a Next Generation Middleware" Proc. Of the Second International ICDE Workshop on Service Engineering (SEIW 2007), Istanbul, Turkey, 2007.

[KLW95] Kifer, M., Lausen, H. and Wu, J., „Logical Foundations of Object-oriented and Frame-based Languages", Journal of the ACM, 42:741-843, July 1995.

[KlCa04] Kline, G. and Caroll J. J., „Resource Description Framework (RDF): Concepts and Abstract Syntax", W3C Recommendation, February 2004, online abrufbar unter: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[KKL+05] Kloppmann, M., Koenig, D., Leymann, F., Pfau, G., Rickayzen, A., von Riegen, C., Schmidt, P. and Trickovic, I. „WS-BPEL Extension for Sub-processes – BPEL-SPE", White Paper, September 2005, online abrufbar unter: http://xml.coverpages.org/BPEL-SPE-Subprocesses.pdf

[KMV+07] Kopecký, J., Moran, M., Vitvar, T. and Roman, D., „WSMO Grounding", WSMO Working Group Working Draft, Version 0.1, April 2007, online abrufbar unter: http://www.wsmo.org/TR/d24/d24.2/v0.1/.

[LPR05] Lausen, H., Polleres, A. and Roman, D., „Web Service Modeling Ontology (WSMO)" W3C Member Submission, June 2005.

[LLN11] van Lessen, T., Lübke, D. and Nitzsche, J., „Geschäftsprozesse automatisieren mit BPEL", dPunkt.Verlag, Januar 2011.

[LDN+07] van Lessen, T., Nitzsche, J., Dimitrov, M., Konstantinov, M., Karastoyanova, D., Cekov, L. and Leymann, F., „An Execution Engine for Semantic Business Processes", Proc. Of the 2nd International ICSOC Workshop on Business Oriented Aspects concerning Semantics and Methodologies in Service-oriented Computing (SemSoc), September 2007.

[LNL08] van Lessen, T., Nitzsche, J. and Leymann, F., „Formalising Message Exchange Patterns using BPEL^{light}", Proc. Of the 5th International Conference on Services Computing (SCC'08) Research Track, Honululu, Hawaii, IEEE Computer Society Press, July 2008.

[LNP+08] van Lessen, T., Nitzsche, J., Pedrinaci, C., Wetzstein, B. and Zaremba, M., „Deliverable: 7.6: Semantic Service Bus", SUPER Deliverable, September 2008, online abrufbar unter: http://www.ip-super.org/res/Deliverables/M30/D7.6.pdf.

[Le01] Leymann, F., „Web Services Flow Language (WSFL) 1.0", IBM Corp., May 2001.

[LeRo00] Leymann, F. and Roller, D., „Production Workflow", Prentice Hall, September 2000, ISBN 978-0130217530.

[MBH+04] Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N. and Sycara, K.. „OWL-S: Semantic Markup for Web Services", W3C Member Submission, November 2004, online abrufbar unter: http://www.w3.org/Submission/OWL-S/.

[McHa04] McGuinness, D.L. and Van Harmelen, F., „OWL Web Ontology Language Overview", W3C Recommendation, February 2004, online abrufbar unter: http://www.w3.org/TR/owl-features/.

[MAMC03] Mandell, D. and McIlraith, S., „Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation", Proc. Of the International Semantic Web Conference (ISWC2003), Sanibel Island, Florida, 2003.

[NHL+08] Nitzsche, J., Höhersteiger, B., Leymann, F., Sonntag, M. and Tost, M.: „Defining the Behaviour of BPEL^{light} Interaction Activities Using Message Exchange Patterns", Proc. Of the Service Wave 2008, Madrid, Spain, Springer Verlag, LNCS, December 2008.

[NLKL07] Nitzsche, J., van Lessen, T., Karastoyanova, D. and Leymann, F., „WSMO/X in the Context of Business Processes: Improvement Recommendations" International Journal of Web Information Systems, Vol. 3(1/2), Emerald, 2007.

[NLKL07b] Nitzsche, J., van Lessen, T., Karastoyanova, D. and Leymann, F., „BPEL^{light}", Proc. Of the 5th International Conference on Business Process Management (BPM 2007), Brisbane, Australia, Springer Verlag, LNCS, September 2007.

[NLKL07c] Nitzsche, J., van Lessen, T., Karastoyanova, D. and Leymann, F., „BPEL for Semantic Web Services (BPEL4SWS)", Proc. Of the 3rd International (OTM) Workshop on Agents and Web Services in Distributed Environments (AweSome'07), Vilamoura, Portugal, 2007.

[NLKL09] Nitzsche, J., van Lessen, T., Karastoyanova, D. and Leymann, F., „Composing Services on the Grid Using BPEL4SWS", Journal of Multiagent and Grid Systems, Vol. 5, Number. 3, IOS Press, 2009, DOI 10.3233/MGS-2009-0130.

[NLL08] Nitzsche, J, van Lessen, T. and Leymann, F., „WSDL 2.0 Message Exchange Patterns: Limitations and Opportunities", Proc. Of the 3rd International Conference on Internet and Web Applications and Services (ICIW 2008), Athens, Greece, IEEE Computer Society Press, June 2008.

[NLL08b] Nitzsche, J., van Lessen, T. and Leymann, F. „Extending BPEL^{light} for Expressing Multi-Partner Message Exchange Patterns", Proc. Of the 12th International EDOC Conference (EDOC 2008), Munich, Germany, IEEE Computer Society Press, September 2008.

[NiNo08] Nitzsche, J. and Norton, B., „Ontology Based Data Mediation in BPEL (for Semantic Web Services)", Proc. Of Business Process Management Workshops, Milano, Italy 2008.

[Pa07] Papazoglou, M., „Web Services: Principles and Technology", 1st Edition, Prentice Hall, July 2007, ISBN 978-0321155559.

[StSt04] Staab, S. und Studer, R (Hrsg.): „Handbook on Ontologies in Information Systems", Berlin, Springer Verlag, November 2003, ISBN 978-3540408345.

[St09] Stein, S., „Modelling Method Extension for Service-Oriented Business Process Management“, Doctoral Dissertation, Christian-Albrecht-Universität zu Kiel, 2008, online abrufbar unter: http://sebstein.hpfsc.de/publications/stein2009phd.pdf

[StBr08] Steinmetz, N. and de Bruijn, J., „WSML/OWL Mapping”, WSML WG Working Draft, Version 0.1, May 2008, Online abrufbar unter: http://www.wsmo.org/TR/d37/v0.1/.

[St08] Studer, R., „Ontologien“ In: Kurbel, K., Becker, J., Gronau, N., Sinz, E., Suhl, L. (Hrsg.): Enzyklopädie der Wirtschaftsinformatik – Online-Lexikon, Zweite Auflage, München, Oldenbourg, 24.9.2008, Online abrufbar unter: http://www.enzyklopaedie-der-wirtschaftsinformatik.de (Abruf: 22.07.2009).

[So08] Sonntag, M., „Conceptual Design and Implementation of a BPEL^{light} Workflow Engine with Support for Message Exchange Pattern”, Diplomarbeit, Institut für Architektur von Anwendungssystemen, Universität Stuttgart, 2008, online abrufbar unter: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2822&mod=0&engl=0&inst=IAAS

[TaSt02] Tanenbaum, A. S. and van Steen, M., „Distributed Systems: Principles and Paradigms”, 2nd Edition, Prentice Hall, April 2008, ISBN 978-0136135531.

[TeWa05] Ten-Hove, R. and Walker, P., „Java Business Integration (JBI) 1.0”, Sun Microsystems Inc., August, 2005.

[Th01] Thatte, S., „XLANG”, Mirosoft Corp., 2001.

[VGS+05] Verma, K., Gomadam, K., Seth, A.P., Miller, J.A., and Wu, Z., „The The METEOR-S approach for configuring and executing dynamic web processes“, online abrufbar unter: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.5528&rep=rep1&type=pdf.

[Wa1726] Walch, J. G. „Philosophisches Lexicon“, 1726.

[WCL+05] Weerawarana, S., Curbera, F., Leymann, F., Storey, T. and Ferguson, D. (Ed.), „The Web Services Platform Architecture”, Addison Wesley. April 2005, ISBN 978-0131488748.

[We07] Weske, M, „Business Process Management: Concepts, Languages, Architectures”, Berlin, Springer, September 2007, ISBN 978-3540735212.

[OMG08] Object Management Group, „Business Process Modeling Notation (BPMN) 1.1”, Document Number: formal/2008-01-17, January 2008, online abrufbar unter: http://www.omg.org/spec/BPMN/1.1/.

[OMG09] Object Management Group, „OMG Unified Modeling Language (OMG UML) Superstructure”, Document Number: formal/2009-02-02, February 2009, online abrufbar unter: http://www.omg.org/spec/UML/2.2/Superstructure.

[ZaOr05] Zaremba, M. and Oren, E., „WSMX Execution Semantics”, WSMX Working Group Working Draft, Version 0.2, October 2005, online abrufbar unter: http://www.wsmo.org/TR/d13/d13.2/.

ANHANG

A. (SA)WSDL-Beschreibung Personalwesen

```
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://www.example.org/hrm/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="hrm"
                  targetNamespace="http://www.example.org/hrm/">
  <wsdl:message name="getPerformanceReviewRequest">
    <wsdl:part element="tns:EmployeeDetails" name="parameters"/>
  </wsdl:message>
  <wsdl:message name="getPerformanceReviewResponse">
    <wsdl:part element="tns:PerformanceReview" name="parameters"/>
  </wsdl:message>

  <wsdl:portType name="hrmPT">
    <wsdl:operation name="getPerformanceReview">
      <wsdl:input
        message="tns:getPerformanceReviewRequest"
        sawsdl:liftingSchemaMapping="http://.../tEmployeeDetails2Ont.xslt"
        sawsdl:loweringSchemaMapping="http://.../Ont2tEmployeeDetails.xslt"
        sawsdl:modelReference="http://.../hrm#employee"/>
      <wsdl:output
        message="tns:getPerformanceReviewResponse"
        sawsdl:liftingSchemaMapping="http://.../tPerformanceReview2Ont.xslt"
        sawsdl:loweringSchemaMapping="http://.../Ont2tPerformanceReview.xslt"
        sawsdl:modelReference="http://.../hrm#performanceReview"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="hrmSOAP" type="tns:hrmPT">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getPerformanceReview">
      <soap:operation
        soapAction="http://www.example.org/hrm/getPerformanceReview"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="hrm">
    <wsdl:port binding="tns:hrmSOAP" name="hrmSOAP">
      <soap:address location="http://www.example.org/"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

B. (SA)WSDL-Beschreibung Reisebüro

```
<wsdl:definitions xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
                  xmlns:con="http://www.example.org/travelAgentConsumer/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://www.example.org/travelAgent/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="travelAgent"
                  targetNamespace="http://www.example.org/travelAgent/">

  <wsdl:message name="bookTripRequest">
    <wsdl:part element="tns:tripDetails" name="parameters"/>
  </wsdl:message>

  <wsdl:portType name="travelAgentPT">
    <wsdl:operation name="bookTrip">
      <wsdl:input
        message="tns:bookTripRequest"
        sawsdl:liftingSchemaMapping="http://.../tTripDetails2Ont.xslt"
        sawsdl:loweringSchemaMapping="http://.../Ont2tTripDetails.xslt"
        sawsdl:modelReference="http://.../travelAgent#trip"/>
    </wsdl:operation>
  </wsdl:portType>

  <plnk:partnerLinkType name="travelAgentPLT">
    <plnk:role name="travelAgent" portType="tns:travelAgentPT"/>
    <plnk:role name="customer" portType="con:travelAgentConsumerPT"/>
  </plnk:partnerLinkType>

  <wsdl:binding name="travelAgentSOAP" type="tns:travelAgentPT">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="bookTrip">
      <soap:operation soapAction="http://www.example.org/travelAgent/bookTrip"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="travelAgent">
    <wsdl:port binding="tns:travelAgentSOAP" name="travelAgentSOAP">
      <soap:address location="http://www.example.org/"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

C. (SA)WSDL-Beschreibung Reisebürokunde

```
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://www.example.org/travelAgentConsumer/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  name="travelAgentConsumer"
                  targetNamespace="http://www.example.org/travelAgentConsumer/">

  <wsdl:message name="bookTripResponse">
    <wsdl:part element="tns:booking" name="parameters"/>
  </wsdl:message>

  <wsdl:portType name="travelAgentConsumerPT">
    <wsdl:operation name="receiveBooking">
      <wsdl:input
        message="tns:bookTripResponse"
        sawsdl:liftingSchemaMapping="http://.../tBooking2Ont.xslt"
        sawsdl:loweringSchemaMapping="http://.../Ont2tBooking.xslt"
        sawsdl:modelReference="http://.../travelAgent#booking"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="travelAgentConsumerSOAP" type="tns:travelAgentConsumerPT">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="receiveBooking">
      <soap:operation
        soapAction="http://www.example.org/travelAgentConsumer/receiveBooking"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="travelAgentConsumer">
    <wsdl:port binding="tns:travelAgentConsumerSOAP"
               name="travelAgentConsumerSOAP">
      <soap:address location="http://www.example.org/"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

D. (SA)WSDL-Beschreibung Onlineversandhandel

```
<wsdl:definitions xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
                  xmlns:con="http://www.example.org/onlineShopConsumer/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://www.example.org/onlineShop/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:sawsdl="http://www.w3.org/ns/sawsdl"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="onlineShop"
                  targetNamespace="http://www.example.org/onlineShop/">

  <wsdl:types>
    <xsd:schema>
      <xsd:element name="Order" type="tns:tOrder" />

      <xsd:complexType name="tOrder">
        <xsd:sequence>
          <xsd:element name="Item" type="tns:tItem" maxOccurs="unbounded"
                       minOccurs="1" />
        </xsd:sequence>
        <xsd:attribute name="OrderId" type="xsd:string" />
      </xsd:complexType>

      <xsd:complexType name="tItem">
        <xsd:sequence>
          <xsd:element name="Number" type="xsd:string" maxOccurs="1"
                       minOccurs="1" />
          <xsd:element name="Quantity" type="xsd:integer" maxOccurs="1"
                       minOccurs="1" />
        </xsd:sequence>
      </xsd:complexType>

      <xsd:element name="Invoice" type="tns:tInvoice" />

      <xsd:complexType name="tInvoice">
        <xsd:sequence>
          <xsd:element name="AccountNumber" type="xsd:string" maxOccurs="1"
                       minOccurs="1" />
          <xsd:element name="AccountHolder" type="xsd:string" maxOccurs="1"
                       minOccurs="1" />
          <xsd:element name="BankIdentificationCode" type="xsd:string"
                       maxOccurs="1" minOccurs="1" />
        </xsd:sequence>
        <xsd:attribute name="Number" type="xsd:string" />
        <xsd:attribute name="OrderId" type="xsd:string" />
        <xsd:attribute name="Amount" type="xsd:string" />
        <xsd:attribute name="Currency" type="xsd:string" />
      </xsd:complexType>

      <xsd:element name="Payment" type="tns:tPayment" />
```

```
    <xsd:complexType name="tPayment">
      <xsd:sequence>
        <xsd:element name="CreditCardNumber" type="xsd:string" maxOccurs="1"
                     minOccurs="0" />
        <xsd:element name="CreditCardHolder" type="xsd:string" maxOccurs="1"
                     minOccurs="0" />
        <xsd:element name="AccountNumber" type="xsd:string" maxOccurs="1"
                     minOccurs="0" />
        <xsd:element name="AccountHolder" type="xsd:string" maxOccurs="1"
                     minOccurs="0" />
        <xsd:element name="BankIdentificationCode" type="xsd:string"
                     maxOccurs="1" minOccurs="0" />
      </xsd:sequence>
      <xsd:attribute name="PaymentType" type="xsd:string" />
      <xsd:attribute name="InvoiceNumber" type="xsd:string" />
    </xsd:complexType>

    <xsd:element name="Receipt" type="tns:tReceipt" />

    <xsd:complexType name="tReceipt">
      <xsd:sequence>
        <xsd:element name="InvoiceNumber" type="xsd:string" maxOccurs="1"
                     minOccurs="1" />
      </xsd:sequence>
    </xsd:complexType>

  </xsd:schema>
</wsdl:types>

<wsdl:message name="order">
  <wsdl:part element="tns:Order" name="parameters"/>
</wsdl:message>
<wsdl:message name="invoice">
  <wsdl:part element="tns:Invoice" name="parameters"/>
</wsdl:message>
<wsdl:message name="payment">
     <wsdl:part name="parameters" element="tns:Payment"/>
</wsdl:message>
<wsdl:message name="receipt">
     <wsdl:part name="parameters" element="tns:Receipt"/>
</wsdl:message>

<wsdl:portType name="sellerPT">
  <wsdl:operation name="orderItem">
    <wsdl:input message="tns:order"
                sawsdl:liftingSchemaMapping="http://.../tItem2Ont.xslt"
                sawsdl:loweringSchemaMapping="http://.../Ont2tItem.xslt"
                sawsdl:modelReference="http://.../onlineshop#order" />
    <wsdl:output message="tns:invoice"
                 sawsdl:liftingSchemaMapping="http://.../tInvoice2Ont.xslt"
                 sawsdl:loweringSchemaMapping="http://.../Ont2tInvoice.xslt"
                 sawsdl:modelReference="http://.../onlineshop#invoice"/>
  </wsdl:operation>
```

```
    <wsdl:operation name="payBill">
      <wsdl:input message="tns:payment"
                  sawsdl:liftingSchemaMapping="http://.../tPayment2Ont.xslt"
                  sawsdl:loweringSchemaMapping="http://.../Ont2tPayment.xslt"
                  sawsdl:modelReference="http://.../onlineshop#payment"/>
      <wsdl:output message="tns:receipt"
                   sawsdl:liftingSchemaMapping="http://.../tReceipt2Ont.xslt"
                   sawsdl:loweringSchemaMapping="http://.../Ont2tReceipt.xslt"
                   sawsdl:modelReference="http://.../onlineshop#receipt"/>
    </wsdl:operation>
  </wsdl:portType>

  <plnk:partnerLinkType name="salesPLT">
    <plnk:role name="seller" portType="tns:sellerPT"/>
    <plnk:role name="buyer" portType="con:buyerPT"/>
  </plnk:partnerLinkType>

  <wsdl:binding name="onlineShopSOAP" type="tns:sellerPT">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="orderItem">
      <soap:operation soapAction="http://www.example.org/onlineShop/orderItem"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="payBill">
      <soap:operation soapAction="http://www.example.org/onlineShop/payBill"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="onlineShop">
    <wsdl:port binding="tns:onlineShopSOAP" name="onlineShopSOAP">
      <soap:address location="http://www.example.org/"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

E. (SA)WSDL-Beschreibung Onlineversandhandelkunde

```
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://www.example.org/onlineShopConsumer/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  name="onlineShopConsumer"
                  targetNamespace="http://www.example.org/onlineShopConsumer/">

  <wsdl:types>
    <xsd:schema>
      <xsd:element name="ShippingDetails" type="tns:tShippingDetails" />

      <xsd:complexType name="tShippingDetails">
       <xsd:sequence>
         <xsd:element name="DeliveryDate" type="xsd:string" maxOccurs="1"
                      minOccurs="1" />
       </xsd:sequence>
       <xsd:attribute name="InvoiceNumber" type="xsd:string" />
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>

  <wsdl:message name="shippingDetails">
    <wsdl:part element="tns:ShippingDetails" name="parameters"/>
  </wsdl:message>
    <wsdl:portType name="buyerPT">
    <wsdl:operation name="receiveShippingDetails">
      <wsdl:input message="tns:shippingDetails"
                  sawsdl:liftingSchemaMapping="http://.../tShipping2Ont.xslt"
                  sawsdl:loweringSchemaMapping="http://.../Ont2tShipping.xslt"
                  sawsdl:modelReference="http://.../onlineshop#shippingDetails"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="onlineShopConsumerSOAP" type="tns:buyerPT">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="receiveShippingDetails">
      <soap:operation soapAction="http://www.example.org/onlineShopConsumer/
                                  receiveShippingDetails"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="onlineShopConsumer">
    <wsdl:port binding="tns:onlineShopConsumerSOAP" name="onlineShopConsumerSOAP">
      <soap:address location="http://www.example.org/"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

F. WSML Ontologie Onlineversandhandel

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"
namespace { _"http://www.example.org/ontologies/onlineshop#",
         dc _"http://purl.org/dc/elements/1.1#" }

ontology onlineshop

     concept order

     concept shoppingCart subConceptOf order
        hasItems ofType (1 *) item

     concept invoice
        hasNumber ofType _integer
        belongsToShoppingCart ofType shoppingCart
        hasAmount ofType (1) _float
        hasCurrency ofType (1) currency
        hasInvoicingParty ofType legallyCompetentPerson
        hasAccountNumber ofType _string
        hasBankIdentificationCode ofType _string

     concept legallyCompetentPerson
        hasSurname ofType _string
        hasPrename ofType _string

     concept currency

     instance euro memberOf currency

     concept payment
        belongsToInvoice ofType (1) invoice

     concept creditcard subConceptOf payment
        hasNumber ofType (1) _integer
        hasOwner ofType (1) person

     concept receipt
        belongsToPayment ofType (1) payment

     concept shippingDetails
        hasDeliveryDate ofType (1) _string
        belongsToOrder ofType (1) order

     concept item
        hasId ofType (1) _string

     concept book subConceptOf item
        hasTitle ofType (1) _string
        hasAuthor ofType (1 *) author
```

G. WSMO-Capability-Beschreibung Onlineversandhandel

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-dl"
namespace {os _"http://www.example.org/ontologies/onlineshop#"}

capability os#capability
  sharedVariables {?request}
  precondition
    definedBy
      ?request memberOf os#order or
      ?request memberOf os#creditCard.

  postcondition
    definedBy
      (?request memberOf os#order implies
        exists (?response memberOf os#invoice)) and
      (?request memberOf os#creditCard implies
        exists ((?1stResponse memberOf os#receipt) and
                (?2ndResponse memberOf os#shippingDetails))).
```

H. Syntax <bl:interactionActivity>

```
<bpel:extensionActivity>
  <bl:interactionActivity conversation="NCName"?
                          createInstance="yes|no"?
                          mep="anyURI"?
                          standard-attributes>
    standard-elements
    <bl:input messageLabel="NCName"
              messageRef="NCName"?
              variable="NCName"?/>*
      <bpel:correlations>?
        <bpel:correlation set="NCName" initiate="yes|join|no"? />+
      </bpel:correlations>
      <bpel:from/>?
    </bl:input>
    <bl:output messageLabel="NCName"
               messageRef="NCName"?
               variable="NCName"?/>*
      <bpel:correlations>?
        <bpel:correlation set="NCName" initiate="yes|join|no"? />+
      </bpel:correlations>
      <bpel:to/>?
    </bl:output>
    <bl:inFault messageLabel="NCName"
                messageRef="NCName"?
                faultName="NCName"
                variable="NCName"?/>*
      <bpel:correlations>?
        <bpel:correlation set="NCName" initiate="yes|join|no"? />+
      </bpel:correlations>
      <bpel:from/>?
    </bl:inFault/>
    <bl:outFault messageLabel="NCName"
                 messageRef="NCName"?
                 faultName="QName-List"/>*
      <bpel:correlations>?
        <bpel:correlation set="NCName" initiate="yes|join|no"? />+
      </bpel:correlations>
    </bl:outFault>
    <bl:partner partnerRef="NCName"?
                name="NCName"/>+
    <bl:expression expressionRef="NCName"?
                   name="NCName"/>*
    <bl:timingExpression timingExpressionLabel="NCName"?
                         timingExpressionRef="NCName"?
                         expressionLanguage="anyURI"?
                         type="for|until|repeatEvery"?
                         value="expr"?/>*
  </bl:interactionActivity>
</bpel:extensionActivity>
```

I. Nachrichtenaustauschmuster „out-only“ in BPEL^{light}

```
<bpel:process xmlns:bpel="http://.../wsbpel/2.0/process/abstract"
              xmlns:bl="http://.../BPELlight/abstract/mep/2008"
              suppressJoinFailure="yes"
              targetNamespace="http://.../mepInBpel"
              name="out-only">...
  <bl:conversations>
    <bl:conversation name="out-onlyConv"/>
  </bl:conversations>
  <bl:partners>
    <bl:partner name="receiver"/>
  </bl:partners>
  <bpel:extensionActivity>
    <bl:interactionActivity bpel:name="sendMessage"
                            conversation="out-onlyConv">
        <bl:input messageLabel="out-message"/>
        <bl:partner name="receiver"/>
    </bl:interactionActivity>
  </bpel:extensionActivity>
</bpel:process>
```

J. Nachrichtenaustauschmuster „out-optional-in“ in BPEL^{light}

```
<bpel:process xmlns:bpel="http://.../wsbpel/2.0/process/abstract"
              xmlns:bl="http://.../BPELlight/abstract/mep/2008"
              suppressJoinFailure="yes"
              targetNamespace="http://.../mepInBpel"
              name="out-optional-in">...
  <bl:conversations>
    <bl:conversation name="out-optional-inConv"/>
  </bl:conversations>
  <bl:partners>
    <bl:partner name="partner"/>
  </bl:partners>
  <bpel:flow>
    <bpel:links>
      <bpel:link name="sendToReceive"/>
    </bpel:links>
    <bpel:extensionActivity>
      <bl:interactionActivity mep="out-only"
                              bpel:name="sendRequest"
                              conversation="out-optional-inConv">
        <bl:input messageLabel="outOptionalIn!out"
                  messageRef="out-only!out"/>
        <bl:partner name="partner"/>
        <bpel:sources>
          <bpel:source linkName="sendToReceive "/>
        </bpel:sources>
      </bl:interactionActivity>
    </bpel:extensionActivity>
    <bpel:extensionActivity>
      <bl:pick bpel:name="receiveResponse">
        <bpel:targets>
          <bpel:target linkName="sendToReceive "/>
        </bpel:targets>
        <bl:onMessage messageLabel="outOptionalIn!in"
                      conversation="out-optional-inConv"
                      partner="partner">
          <bpel:empty/>
        </bl:onMessage>
        <bl:onFaultMessage messageLabel="fault"
                           conversation="out-optional-inConv"
                           partner="partner">
        </bl:onFaultMessage>
        <bl:onAlarm timingExpressionLabel="outOptionalIn!timeOut">
          <bpel:empty/>
        </bl:onAlarm>
      </bl:pick>
    </bpel:extensionActivity>
  </bpel:flow>
</bpel:process>
```

K. Nachrichtenaustauschmuster „request-for-bid“ in BPEL^{light}

```
<bpel:process xmlns:bpel="http://.../wsbpel/2.0/process/abstract"
              xmlns:bl="http://.../BPELlight/abstract/mep/2008"
              suppressJoinFailure="yes"
              targetNamespace="http://.../mepInBpel"
              name="request-for-bid">
  <bpel:variables>
    <bpel:variable name="counter" type="xsd:long"/>
    <bpel:variable name="result" type="Array"/>
  </bpel:variables>
  <bl:conversations>
    <bl:conversation name="request-for-bidConv"/>
  </bl:conversations>
  <bl:partners>
    <bl:partner name="bidder"/>
  </bl:partners>
  <bpel:forEach parallel="yes"
                counterName="counter">
    <bpel:startCounterValue>1</bpel:startCounterValue>
    <bpel:finalCounterValue>
      count($bidder/list/partnerInstance)
    </bpel:finalCounterValue>
    <bpel:scope isolated="yes">
      <bl:partners>
        <bl:partner name="rfbBidder"/>
      </bl:partners>
      <bpel:sequence>
        <bpel:assign>
          <bpel:copy>
            <bpel:from>$bidder/list/partnerInstance[$counter]</bpel:from>
            <bpel:to>$rfbBidder/current</bpel:to>
          </bpel:copy>
        </bpel:assign>
        <bpel:extensionActivity>
          <bl:interactionActivity mep="out-optional-in"
                                  bpel:name="getBid"
                                  conversation="request-for-bidConv">
            <bl:input messageLabel="request-for-bid!request"
                      messageRef="out-optional-in!out"/>
            <bl:output variable="out-optional-inResponse"
                       messageRef="out-optional-in!in"/>
            <bl:timingExpression timingExpressionLabel="requestForBid!timeOut"
                                 timingExpressionRef="outOptionalIn!timeOut" />
            <bl:partner name="rfbBidder"/>
          </bl:interactionActivity>
        </bpel:extensionActivity>
        <bpel:assign>
          <bpel:copy>
            <bpel:from>$out-optional-inResponse</bpel:from>
            <bpel:to>$result[$counter]</bpel:to>
          </bpel:copy>
        </bpel:assign>
      </bpel:sequence>
    </bpel:scope>
  </bpel:forEach>
</bpel:process>
```

Danksagungen

An dieser Stelle möchte ich all jenen danken, die mich während meiner Arbeit an dieser Dissertation in den letzten Jahren begleitet haben. Mein Dank gilt zunächst meinem Doktorvater Prof. Dr. Frank Leymann am Institut für Architektur von Anwendungssystemen für das mir entgegen gebrachte Vertrauen und die Unterstützung während der letzten Jahre. Mein besonderer Dank gilt auch Jun.-Prof. Dr. Dimka Karastoyanova für die vielen wertvolle Ratschläge und die vertrauensvolle Zusammenarbeit während meiner Zeit am IAAS.

Während eben dieser Zeit hatte ich auch das Vergnügen mit einigen großartigen Menschen zusammenzuarbeiten bei denen ich mich für die interessanten Diskussionen, gemeinsamen Reisen und Feiern bedanke: Vor allem Tammo van Lessen, Daniel Wutke, Ralph Mietzner, Daniel Martin, Oliver Kopp, Tobias Unger, Thorsten Scheibler, Branimir Wetzstein und Olaf Zimmermann.

Nicht zuletzt danke ich meiner Frau Sibille und meinen Söhnen Philipp und Felix für ihre Liebe und Geduld und meinen Eltern Heike und Werner ohne deren Unterstützung während des Studiums und der Promotion diese Arbeit nicht möglich gewesen wäre.