Komposition von Web-Services mit Hilfe von BPEL4WS
-
Teleseminar Web-Services

Seminararbeit
von

Holger Rössig
aus
Heidelberg

vorgelegt am
Lehrstuhl für Praktische Informatik IV
Prof. Dr.-Ing. Wolfgang Effelsberg
Fakultät für Mathematik und Informatik
Universität Mannheim

Juli 2004

Betreuer: Dipl. Inform. Jochen Dinger

1. Einleitung

Wie man BPEL4WS[1] sinnvoll einsetzt, und warum die bereits bestehenden Standards wie WSDL, SOAP oder UDDI nicht ausreichen, um diesen Zweck zu erfüllen, soll im Folgenden geklärt werden.
Mit WSDL[2] (**W**eb **S**ervice **D**escription **L**anguage) werden die Schnittstellen von Web-Services beschrieben, mit SOAP[3] (**S**imple **O**bject **A**ccess **P**rotokoll) werden Nachrichten zwischen Web-Services ausgetauscht, und UDDI[4] (…) dient zum Auffinden von Web-Services für eine gewünschte Aufgabe.
Um Web-Services zu größeren Geschäftsprozessen zusammenzufassen, reichen die erwähnten Standards nicht aus, und das ist genau der Ansatzpunkt für BPEL4WS.
Folgendes einfaches Beispiel beschreibt ein mögliches Szenario:

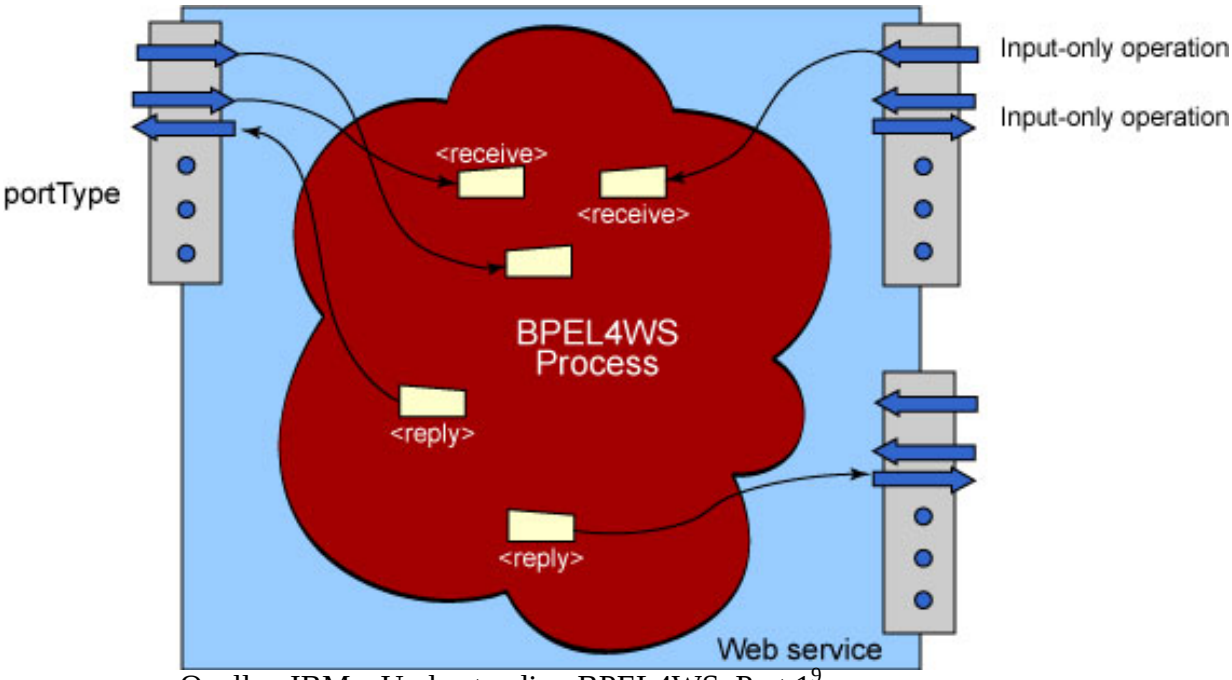
Man stelle sich vor eine Fluggesellschaft bietet einen Web-Service an, mit dem es möglich ist, Füge zu buchen. Eine Hotelkette wiederum bietet einen Web-Service zum Buchen von Hotelzimmern. Wenn man nun eine Reise buchen möchte, muss man bei der Fluggesellschaft den Flug, bei der Hotelkette das entsprechende Zimmer buchen und man muss selbst dafür sorgen, dass diese zeitlich auch aufeinander abgestimmt sind.
Ein Reisebüro könnte nun einen Web-Service anbieten mit dem es möglich ist, Flug und Hotel auf einmal zu buchen, mit automatischer Überprüfung, ob auch beides zur gewünschten Zeit verfügbar ist.
Dieser Web-Service soll nun eine Kombination der beiden anderen darstellen.

Vor BPEL4WS gab es nur Spezifikationen, wie WSFL[5] von IBM und XLANG[6] von Microsoft. Diese sind allerdings nicht kompatibel zueinander und haben beide ihre Vor- und Nachteile. WSFL ist mehr Graphen-orientiert und XLANG bietet bessere Konstrukte zur Strukturierung von Prozessen.
Die „**B**usiness **P**rocess **E**xecution **L**anguage **for W**eb **S**ervices“ (BPEL4WS) ist aus beiden Ansätzen entstanden und kombiniert deren Vorteile in einer Sprache. In BPEL4WS ist es nun möglich, diesen im Beispiel beschriebenen kombinierten Prozess zu erstellen.
BPEL4WS Version 1.0 ist im August 2002 erschienen, momentan aktuell ist die Version 1.1 vom März 2003. Es wurde von IBM, Microsoft und BEA initiiert und liegt momentan OASIS[7] zur Standardisierung vor.
Eine formale Definition eines Geschäftsprozesses gibt Jian Yang : „*A Process is viewed as a series of activities, where an activity represents a well-defined business function …*“[8] (“ *Ein Geschäftsprozess wird als Serie von Aktivitäten angesehen, wobei jede Aktivität eine wohl-definierte Geschäftsfunktion darstellt …*“)

[1] [BPEL] **B**usiness **P**rocess **E**xecution **L**anguage **for W**eb **S**ervices
[2] [WSDL] Web Service Definition Language
[3] [SOAP] Simple Object Access Protokoll
[4] [UDDI]
[5] [WSFL] Web Service Flow Language
[6] [XLANG]
[7] [OASIS]
[8] [YANG] Yang, Jian : Web Service Componentizatzion

2. Komposition von Web-Services

Wie schon in der Einleitung erwähnt, bieten vorhandene Web-Services bzw. die von ihnen angebotenen Dienste oft nur eingeschränkte Funktionalität oder sind nur für ganz spezielle Gebiete einsetzbar. Um eine komplexere Transaktion, wie im einführenden Beispiel, durchführen zu können, muss man die vorhandenen Web-Services zu einem Geschäftsprozess kombinieren.
BPEL4WS bietet genau hierfür die Sprachkonstrukte um Web-Services zu Geschäftsprozessen zusammenzufassen.
Diese hierdurch neu entstehenden Web-Services werden als BPEL4WS Prozesse bezeichnet.
Folgende Abbildung zeigt den schematischen Aufbau eines solchen Prozesses:



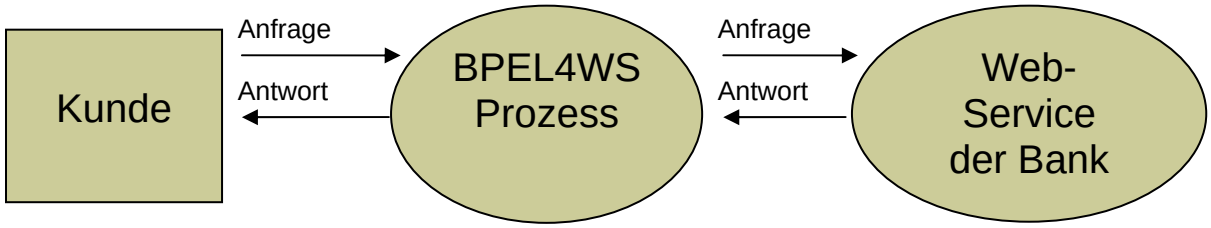
Quelle : IBM, „Understanding BPEL4WS, Part 1[9]

Das Interface ist eine Kollektion von WSDL `portType`s, wie das auch bei herkömmlichen Web-Services der Fall ist. Dieser Umstand bietet dem Anwender den großen Vorteil, dass es keinen Unterschied macht, ob er nun mit einem Web-Service im herkömmlichen Sinne oder mit einem BPEL4WS Prozess interagiert.
Mit dem oben erwähnten WSDL können also die Interfaces mit den über die `portType`s angebotenen `operation`s und auch die ausgetauschten Nachrichten genau beschrieben werden.
Wie dies im Einzelnen funktioniert zeigen die folgenden Kapitel an einem einfachen Beispiel.

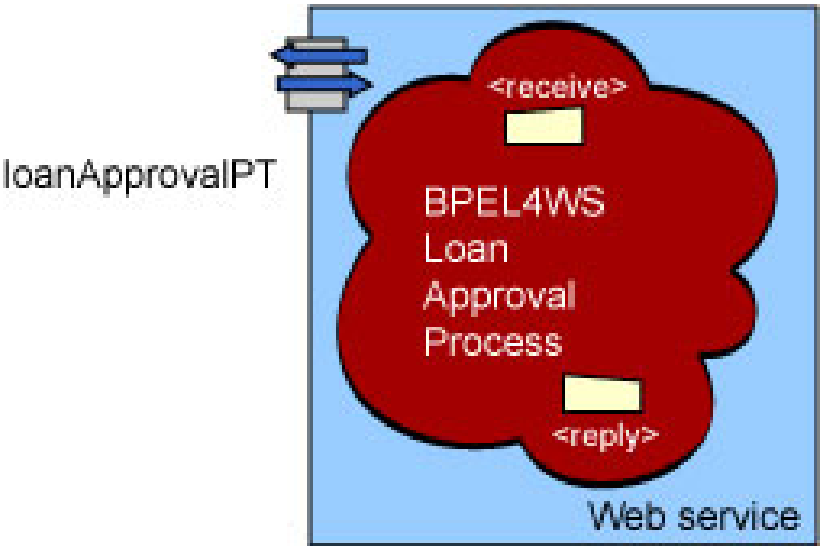
[9] [IBM1] IBM, „Understanding BPEL4WS, Part 1“

3. BPEL4WS im Detail

Anhand eines einfachen Beispiels, des „Loan Approval Process“ [10], sollen nun die Schritte zum Erstellen eines BPEL4WS Prozesses und dessen einzelne Elemente erklärt werden.
Dem Beispiel liegt folgendes Szenario zu Grunde:
Es soll ein BPEL4WS Prozess erstellt werden, der es dem Kunden einer Bank ermöglicht, die Bewilligung eines gewünschten Kredits zu erfragen. Es wird davon ausgegangen, dass es bei der Bank schon einen Web-Service gibt, der genau diese Aufgabe erledigen kann. Im Folgenden wird nun ein BPEL4WS Prozess erstellt, der die Anfrage des Kunden entgegen nimmt, diese an den Web-Service der Bank weiterleitet, die Antwort von der Bank erhält und diese dann wieder zum Kunden zurücksendet.
Folgende Grafik dient zur Veranschaulichung der Situation:



Dies macht zwar wenig Sinn, da hier ja nur die Komposition von einem Service stattfindet, was man ja eigentlich nicht als solche bezeichnen kann, jedoch wird der entstehende BPEL4WS Prozess deutlich übersichtlicher werden, woran man dann besser die Grundlagen erklären kann. Das grundsätzliche Funktionsprinzip von BPEL4WS wird hierdurch nicht beeinträchtigt.
Hier sieht man den schematischen Aufbau des „Loan Approval Process“:



Quelle: IBM, Learning BPEL4WS, Part 2[11]

Nach außen hat der Prozess als Interface einen WSDL `portType` mit dem Namen „loanApprovalPT“. Genaueres hierzu im nächsten Abschnitt.3.1 WSDL Descriptions.

[10] [IBM2] IBM, Learning BPEL4WS, Part 2
[11] [IBM2] IBM, Learning BPEL4WS, Part 2

Als erstes werden genaue Beschreibungen der Interfaces und der auszutauschenden Nachrichten erstellt. Diese werden in WSDL geschrieben, welches wie auch BPEL4WS selbst auf XML[12] basiert. Folgender Code zeigt die WSDL Datei, die definiert wie die Nachricht, die der Benutzer an den BPEL4WS Prozess schickt, aussieht:

loandefinition.wsdl[13]

```
<definitions   targetNamespace="http://tempuri.org/services/loandefinitions"
                  xmlns:tns="http://tempuri.org/services/loandefinitions"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns="http://schemas.xmlsoap.org/wsdl/">
            <message name="creditInformationMessage">
                  <part name="firstName" type="xsd:string"/>
                  <part name="name" type="xsd:string"/>
                  <part name="amount" type="xsd:integer"/>
            </message>
            <message name="loanRequestErrorMessage">
                  <part name="errorCode" type="xsd:integer"/>
            </message>
</definitions>
```

Hier sieht man wie eine solche WSDL Beschreibung aufgebaut ist. Es gibt einen alles umschließenden `<definitions>` Tag der kenntlich macht dass es sich hier um eine Definition im Sinne von WSDL handelt. Nach einigen Namespace Bindungen, die die korrekte Zuordnung von Attributen ermöglichen, findet man hier zwei `<message>` Tags, die zwei verschiedene Nachrichten definieren. Diese sind durch das Attribut `name` zu unterscheiden. Die „creditInformationMessage“ besteht z.B. aus drei Teilen die mit dem `<part>` Tag gekennzeichnet sind, jeder dieser Teile hat einen Namen, durch das Attribut `name` gekennzeichnet, und einen bestimmten Typ, mit dem Attribut `type` gekennzeichnet. In diesem Fall ist der Teil „firstName“ vom Typ „`string`“ im Namespace „`xsd`“. Dies ist nun die Definition der Nachricht die der Benutzer an den BPEL4WS Prozess sendet. Der zweite `<message>` Tag definiert eine Fehlermeldung, die der Benutzer im Fehlerfall zurückerhält.
Als nächstes wird die WSDL Beschreibung des Web-Services vorgestellt, den die Bank anbietet:

loanapprover.wsdl[14]

```
<definitions      targetNamespace="http://tempuri.org/services/loanapprover"
                  xmlns:tns="http://tempuri.org/services/loanapprover"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:loandef="http://tempuri.org/services
                              /loandefinitions"
                  xmlns="http://schemas.xmlsoap.org/wsdl/">
            <import  namespace="http://tempuri.org/services/loandefinitions"
                  location="http://localhost:8080/bpws-
                  samples/loanapproval/loandefinitions.wsdl"/>

            <message name="approvalMessage">
                  <part name="accept" type="xsd:string"/>
            </message>
```

[12] [XML] E**x**tensible **M**arkup **L**anguage
[13] [IBM2] IBM, Learning BPEL4WS, Part 2
[14] [IBM2] IBM, Learning BPEL4WS, Part 2

```
      <portType name="loanApprovalPT">
            <operation name="approve">
                  <input message="loandef:creditInformationMessage"/>
                  <output message="tns:approvalMessage"/>
                  <fault name="loanProcessFault"
                              message="loandef:loanRequestErrorMessage"/>
            </operation>
      </portType>
      <binding ...> ... </binding>
      <service name="LoanApprover">....</service>
</definitions>
```

Hier ist nun interessant, dass mit `<import>` die WSDL Datei *loandefinitons.wsdl* eingebunden wird, so dass auch hier die zwei eben besprochenen Nachrichten definiert sind. Zudem wird zur Zuordnung der Attribute der Namespace mit dem link zur *loandefintions* Datei an das Namespace Präfix `loandef` gebunden, damit später klar ist, wo z.B. `creditInformationMessage` definiert ist.
Ganz entscheidend ist hier die Definition des `portType`s „loanApprovalPT“, der auch von dem zu erstellenden BPEL4WS Prozess verwendet werden wird, wie schon eingangs auf der schematischen Darstellung zu erkennen war. Da dieser Prozess ja dem Benutzer genau die gleichen Dienste anbieten wird, wie der Web-Service der Bank.
In diesem `<portType>` Tag befindet sich der `<operation>` Tag, der angibt, dass eine Operation mit dem Namen „`approve`“ angeboten wird, die als `<input>` eine Nachricht vom Typ `creditInformationMessage` erwartet, als `<output>` eine `approvalMessage` oder im Fehlerfall (`<fault>`) eine `loanRequestErrorMessage` zurückgibt.
Diese `approvalMessage` wird hier auch zusätzlich wieder mit dem `<message>` Tag definiert, da sie in der importierten Datei noch nicht enthalten war.
Der nächste Code-Block zeigt wie die WSDL Beschreibung des zu erstellenden BPEL4WS Prozesses aussieht:

loan-approval.wsdl[15]

```
<definitions targetNamespace="http://loans.org/wsdl/loan-approval"
      xmlns="http://schemas.xmlsoap.org/wsdl/"
      xmlns:slnk="http://schemas.xmlsoap.org/ws/2002/06/service-link/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:lns="http://loans.org/wsdl/loan-approval"
      xmlns:apns="http://tempuri.org/services/loanapprover">

      <import namespace="http://tempuri.org/services/loanapprover"
      location="http://localhost:8080/bpws-
      samples/loanapproval/loanapprover.wsdl"/>

      <import namespace="http://tempuri.org/services/loandefinitions"
      location="http://localhost:8080/bpws-
      samples/loanapproval/loandefinitions.wsdl"/>
```

[15] [IBM2] IBM, Learning BPEL4WS, Part 2

```
      <slnk:serviceLinkType name="loanApprovalLinkType">
            <slnk:role name="approver">
                  <portType name="apns:loanApprovalPT"/>
            </slnk:role>
      </slnk:serviceLinkType>

      <service name="loanapprovalServiceBP"/>
</definitions>
```

Hier werden beide Dateien von zuvor importiert. Damit ist auch hier der gleiche `portType` bereitgestellt und es sind auch die Definitionen der Nachrichten vorhanden. Was hier neu ist sind die `serviceLinkTypes` mit denen die Verbindung zwischen zwei Services hergestellt werden kann. In diesem Fall soll der BPEL4WS Prozess mit dem Web-Service der Bank verbunden werden. Mit `<role>` wird angegeben, welche Rolle der Service annimmt der über diesen Link erreicht wird, und mit `<portType>` wird der `portType` spezifiziert, der benutzt werden soll.

Mit den vorgestellten WSDL Dateien ist nun alles definiert und man kann beginnen den BPEL4WS Prozess „*loanApprovalPorcess*“ zu schreiben.
Der Prozess beginnt mit dem `<process>` Tag der den gesamten Prozess umschließen wird:

3.2 Der BPEL4WS Prozess

```
<process    name="loanApprovalProcess"
            targetNamespace="http://acme.com/simpleloanprocessing"
      xmlns="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
      xmlns:lns="http://loans.org/wsdl/loan-approval"
      xmlns:loandef="http://tempuri.org/services/loandefinitions"
      xmlns:apns="http://tempuri.org/services/loanapprover">[16]
```

Die soeben vorgestellten WSDL Dateien werden an die Namespace Präfixe `lns`, `loandef` und `apns` gebunden, damit man nachvollziehen kann wie entsprechend verwendete Attribute definiert sind.

3.2.1 Partner

Als nächstes muss definiert werden mit welchen Partnern der BPEL4WS Prozess interagiert. In diesem Fall sind dies der Kunde (customer) und der Web-Service der Bank (approver):

```
      <partners>
            <partner    name="customer"
                        serviceLinkType="lns:loanApprovalLinkType"
                        myRole="approver"/>
            <partner    name="approver"
                        serviceLinkType="lns:loanApprovalLinkType"
                        partnerRole="approver"/>
      </partners>[17]
```

[16] [IBM2] IBM, Learning BPEL4WS, Part 2
[17] [IBM2] IBM, Learning BPEL4WS, Part 2

Dafür wird der `<partners>` Tag benutzt, in dem alle Partner mit dem jeweiligen Tag `<partner>` zusammengefasst werden. Jeder Partner bekommt einen Namen und es wird der `serviceLinkType` angegeben, über den man ihn erreicht. Mit `myRole` wird die Rolle angegeben, die der Prozess selbst in Verbindung mit dem Partner spielt, mit `partnerRole` entsprechend die des Partners.

3.2.2 Container

Um in BPEL4WS Daten schreiben und lesen zu können, braucht man Container. In diese kann man dann Daten ablegen und sie auch wieder herausnehmen. Dazu wird der `<containers>` Tag verwendet:

```
<containers>
      <container  name="request"
                  messageType="loandef:creditInformationMessage"/>
      <container  name="approvalInfo„
                  messageType="apns:approvalMessage"/>
</containers>
```
[18]

In diesen werden dann so viele `<container>` Tags geschachtelt, wie man benötigt. Es wird hier ein Container benötigt, um die vom Kunden erhaltene `creditInformationMessage` zu speichern, und ein zweiter, um die vom Web-Service der Bank zurückerhaltene `approvalMessage` zu speichern. Mit `messageType` wir der genaue Typ der Nachricht, die gespeichert werden soll, angegeben.

3.2.3 Activities

Mit Activities kann man nun den Prozess dazu bringen Dinge zu tun, wie Nachrichten zu empfangen oder diese weiterzuleiten. Ein BPEL4WS Prozess kann aus nur einer oder aus mehreren Activities bestehen. Sie können grundsätzlich in zwei Klassen eingeteilt werden:

- **Structure activities**
  - `<sequence>`
  - `<flow>`
  - `<switch>`
  - `<while>`
  - `<pick>`
  - `<scope>`
  - …
- **primitive activities**
  - `<receive>`
  - `<invoke>`
  - `<reply>`
  - `<assign>`
  - `<empty>`
  - …

Mit *structure activities* kann man den Ablauf und die Struktur des Prozesses steuern, was in Kapitel 3.3 genauer erklärt wird.

[18] [IBM2] IBM, Learning BPEL4WS, Part 2

Mit *primitive activities* kann man konkrete Operationen ausführen, wie z.B. Nachrichten empfangen oder versenden.

Für diesen Prozess wird die *structure activity* `<sequence>` als Haupt-Activity verwendet, welche dafür sorgt dass alle in sie hineingeschachtelten activities (egal ob structure oder primitive) sequentiell in der Reihenfolge wie sie dastehen ausgeführt werden.

```
<sequence>
      <receive    name="receive1"
                  partner="customer"
                  portType="apns:loanApprovalPT"
                  operation="approve"
                  container="request"
                  createInstance="yes">
      </receive> [19]
```

`<receive>` ist die erste *primitive activity* die ausgeführt wird. Über Attribute wird ihr ein Name gegeben, der Partner, der zu verwendende `portType` und die auszuführende `operation` angegeben. Zusätzlich wird noch der Name des `containers` spezifiziert in den die zu empfangende Nachricht gespeichert werden soll, hier „`request`“ wie oben definiert.
Mit diesem `<receive>` wird also eine `loanApprovalMessage` des Kunden `customer` im `container` „`request`“ gespeichert.

```
      <invoke     name="invokeapprover"
                  partner="approver"
                  portType="apns:loanApprovalPT"
                  operation="approve"
                  inputContainer="request"
                  outputContainer="approvalInfo">
      </invoke> [20]
```

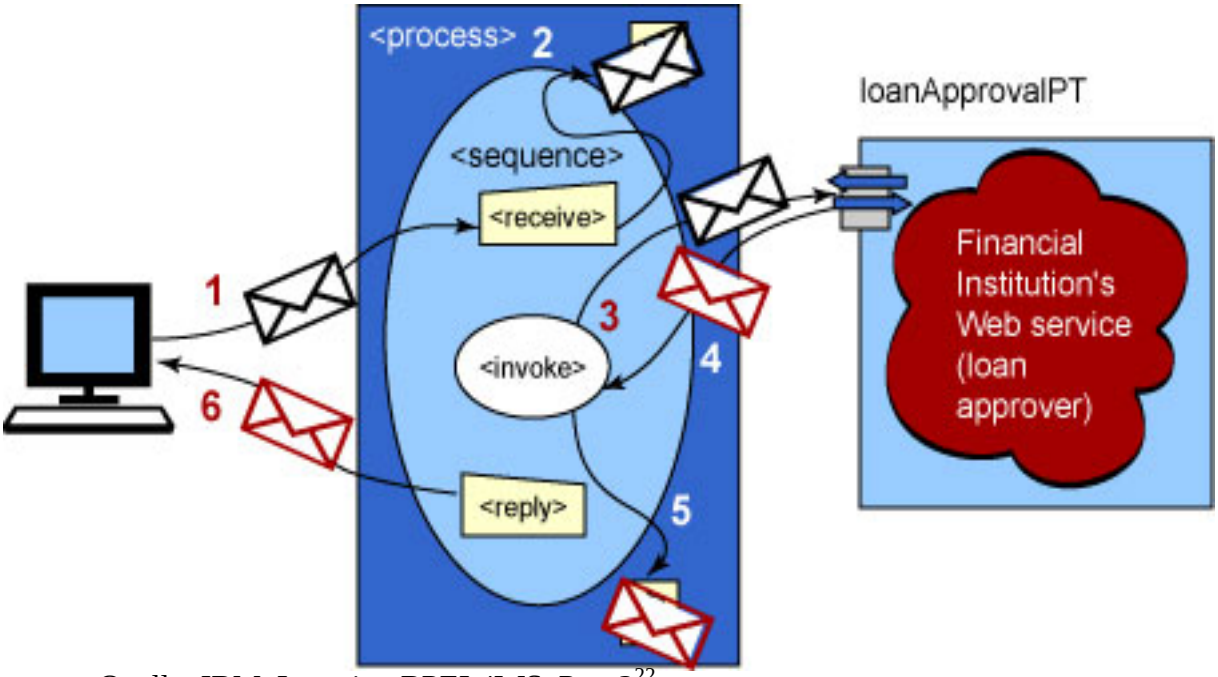
Als nächstes wird `<invoke>` ausgeführt. Wie auch bei `<receive>` wird über Attribute Name, Partner, etc. angegeben. Interessant ist hier dass zwei `container` angegeben sind, nämlich `inputConatiner` und `outputContainer`. `<invoke>` wird jetzt die von `<receive>` empfangene und im `container` „`request`“ abgelegte `loanApprovalMessage` des Kunden aus eben diesem `conatiner` nehmen und an den Web-Service der Bank schicken. Das Ergebnis wird dann im `container` „`approvalInfo`“ gespeichert.

```
      <reply      name="reply"
                  partner="customer"
                  portType="apns:loanApprovalPT"
                  operation="approve"
                  container="approvalInfo">
      </reply>
</sequence>
</process> [21]
```

[19] [IBM2] IBM, Learning BPEL4WS, Part 2
[20] [IBM2] IBM, Learning BPEL4WS, Part 2
[21] [IBM2] IBM, Learning BPEL4WS, Part 2

Um die Transaktion zu beenden wird dann am Ende noch `<reply>` aufgerufen, was dafür sorgt, dass die vom Web-Service der Bank über `<invoke>` erhaltene Nachricht aus dem `conatiner` „`approvalInfo`“ wieder herausgenommen, und an den Kunden zurück geschickt wird. Damit ist dieser kleine Beispielprozess schon vollständig und es muss nur noch der `<sequence>` Tag der die drei primitive activities umschließt und dann der `<process>` Tag der den ganzen Prozess umschließt geschlossen werden.

Der ganze Ablauf der Transaktion wir hier noch einmal graphisch dargestellt:



Quelle: IBM, Learning BPEL4WS, Part 2[22]

[22] [IBM2] IBM, Learning BPEL4WS, Part 2

3.3 Ablaufsteuerung

In diesem Kapitel wird, losgelöst von dem Bank-Beispiel, genauer auf die structure activities und die control links eingegangen, die man zu Steuerung des Prozess Ablaufes einsetzten kann.
Structure activities sind aus Microsofts XLANG[23] übernommen worden, control links hingegen stammen aus IBMs WSFL[24]. Beide Konzepte können in BPEL4WS teilweise alternativ oder auch in Kombination eingesetzt werden, um bestimmte Abläufe des Prozesses zu erreichen.
Um dies zu verdeutlichen, werden jetzt verschiedene Beispiele für Soll-Abläufe und mögliche Lösungen in BPEL4WS vorgestellt.

3.3.1 sequentielle Ausführung

Gegeben seien zwei abstakte activities (activityA, activityB), die für beliebige konkrete activities stehen können, egal ob structure oder primitive.
Angenommen, diese sollen nacheinander ausgeführt werden, also activityA → activityB.
Dieses Problem kann man in BPEL4WS nun auf zwei Arten lösen, entweder mit structure activities oder mit control links.

1. Möglichkeit (mit der structure activity `<sequence>`)

```
<sequence>
            activityA
            activityB
</sequence>[25]
```

Durch den `<sequence>` Tag wird erst `activityA`, dann `activityB` ausgeführt.

2. Möglichkeit (mit control links)

```
<flow>
      <links>
            <link name="L"/>
      </links>
      activityA
            <source linkName="L"/> ...
      activityB
            <target linkName="L"/> ...
 </flow>[26]
```

Hier wird als Haupt-Activity ein `<flow>` gewählt, da diese im Gegensatz zu `<sequence>` keine Reihenfolge einhält, und `actvityA` und `activityB` gleichzeitig laufen lässt. Um nun zu gewährleisen, dass `activityB` erst dann ausgeführt wird, wenn `activityA` beendet ist, erstellen wir einen control link „L“, der beide activities verbindet. Er wird mit `<link name=“L“/>` erzeugt im Tag `<links>`. Jetzt definieren wir `activityA` als

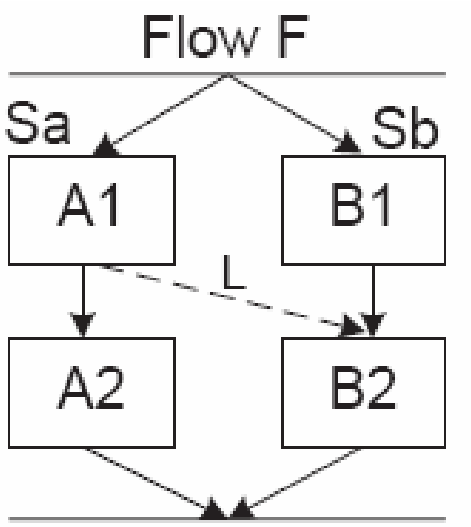
[23] [XLANG]
[24] [WSFL]
[25] [WADH]
[26] [WADH]

Quelle (`source`) des Links und `activityB` als Ziel (`target`). Activites die als Ziel eines Links definiert sind, werden erst dann ausgeführt, wenn der Kontrollfluss über den Link bei ihnen ankommt. Hier startet jetzt zuerst `activityA`, da sie kein Ziel eines Links ist, und `activityB` wartet. Erst wenn `activityA` beendet ist, läuft der Kontrollfluss
über den Link zu `activityB`, und `activityB` startet.

3.3.2 Synchronisation

Gegeben seien 4 *activities*: A1, A2, B1 und B2.
A1 und A2 seien in einer `<sequence>` Sa, B1und B2 in einer `<sequence>` Sb und diese beiden *sequences* in einem `<flow>` F.
`Sa` und `Sb` starten also parallel und `A2` wird nach `A1` und `B2` nach `B1` ausgeführt.
Es soll nun erreicht werden, dass `B2` erst dann startet wenn `B1` UND `A1` beendet sind. Dies lässt sich wieder über einen *control link* „L“, wie in der Graphik zu sehen, bewerkstelligen.

```
<flow name="F">
      <links>
            <link name="L"/>
      </links>
      <sequence name="Sa">
            activityA1
            <source linkName="L"/>
            activityA2
      </sequence>
      <sequence name="Sb">
            activityB1
            activityB2
            <target linkName="L"/>
      </sequence>
 </flow>[27]
```



Wie beim ersten Beispiel, wartet hier die *`activityB2`* , weil sie Ziel des Links „L“ ist. Allerdings nur im Fall, wenn `B1` schneller beendet ist als A1. Ist A1 schneller fertig als B1, dann ist B2 durch die <*sequence*> Sb noch gar nicht an der Reihe, startet dann aber nach Beendigung von B1 sofort, weil der Link „L“ „schon angekommen“ ist. Das bedeutet dass für den Start von B2 müssen 2 Bedingungen erfüllt sein:

- B2 muss innerhalb des normalen Kontrollflusses „an der Reihe“ sein
- Der Link „L“ muss schon „angekommen“ sein

Durch einen *control link* kann also die Ausführung zweier parallel laufender *sequences* synchronisiert werden.

[27] [WADH]

3.3.3 Switch

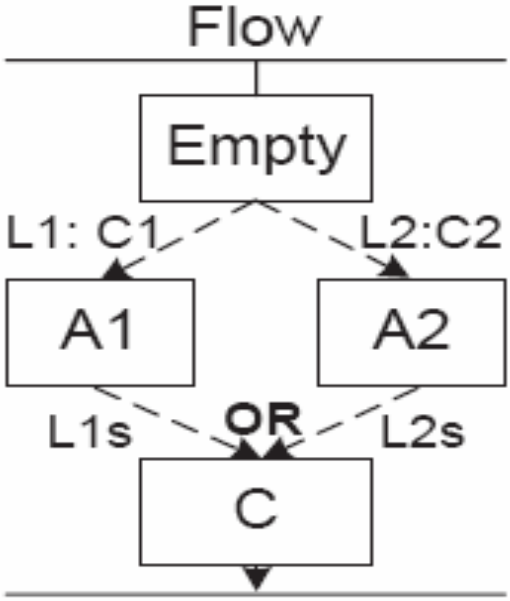
Gegeben seien 3 activities: `A1`, `A2` und `C`.
Der Prozess ist in einem Wartezustand und soll, wenn Bedingung `C1` eintritt, `activityA1` ausführen, anschließend `activityC` ausführen und dann beenden. Wenn Bedingung `C2` eintritt soll er `activityA2` ausführen, dann `activityC` und dann beenden.
In diesem Fall gibt es wieder zwei Möglichkeiten, einmal nur mit structure activities aus XLANG oder in Kombination mit control links aus WSFL.

1. Möglichkeit (structure activitiy `<switch>`)

```
<switch>
      <case condition="C1">
            activityA1
      </case>
      <case condition="C2">
            activityA2
      </case>
</switch>
activityC [28]
```

2. Möglichkeit (structure activities mit control links)

```
<flow>
      <links>
            <link name="L1"/>
            <link name="L2"/>
            <link name="L1s"/>
            <link name="L2s"/>
      </links>
      <empty>
            <source linkName="L1"
            transitionCondition="C1"/>
            <source linkName="L2"
            transitionCondition="C2"/>
      </empty>
      activityA1
            <target linkName="L1">
            <source linkName="L1s">
      activityA2
            <target linkName="L2">
            <source linkName="L2s">
      activityC
            joinCondition="L1s OR L2s"
            <target linkName="L1s">
            <target linkName="L2s"> ...
 </flow>[29]
```



Durch den `<flow>` würden eigentlich alle dort hineigeschachtelten Elemente gleichzeitig ausgeführt werden, also `<links>`, `<empty>` und die *activities* `A1`, `A2` und `C`.
Die *activities* sind aber alle Ziel eines *control links*, und warten daher bis der Kontrollfluss über den Link bei ihnen ankommt. Außer der *activity* `<empty>`. Diese wird ausgeführt, tut nichts und beinhaltet die Quelle von Link „L1“ und „L2“. Diese Links starten aber auch nicht

[28] [WADH]
[29] [WADH]

bis die über das Attribut `transitionCondition` erwartete Bedingung eintritt. Tritt nun z.B. die Bedingung „`C1`“ ein, dann geht der Kontrollfluss über Link „`L1`“ zu `A1` und `A1` als Ziel dieses Links wird ausgeführt. Danach startet der Link „`L1s`“ dessen Quelle `A1` ist und läuft zu `C`. `C` ist aber Ziel von zwei Links, nämlich „`L1s`“ und „`L2s`“ und hat daher ein Attribut `joinCondition` , welches auf „`L1s OR L2s`“ gesetzt ist. Daher wird nun `C` ausgeführt, da die `joinCondition` erfüllt ist. Der Ablauf bei Eintritt der Bedingung „`C2`“ gestaltet sich analog über die Links „`L2`“ und „`L2s`“.

4. Scopes, Fehlerbehandlung & Kompensation

Bei der Ausführung BPEL4WS Prozessen können auch Fehler auftreten, u.a. da ja auch immer „fremde“ Web-Services beteiligt sind, und diese dann entweder selbst Fehler verursachen können, oder auch zeitweise nicht verfügbar sein können, was ebenfalls zu einem Fehler im Prozess führt. Um dies in den Griff zu bekommen, kann man Teile des Prozesses in Scopes zusammenfassen, die dort auftretenden Fehler mit `<faultHandler>`*n* abfangen und dann noch eventuell vorhandene `<compensationHandler>` aufrufen, um schon ausgeführte Vorgänge rückgängig zu machen.

4.1 Scopes

Mit Hilfe von Scopes kann man mehrere activities, also Teile eines Prozesses, in einen gemeinsamen Kontext bringen. Das ist sinnvoll, wenn diese logisch zusammenhängen, wie z.B. im anfangs erwähnten Beispiel mit der Reisebuchung. Man könnte dort die Buchung des Fluges und die Buchung des Hotelzimmers in einen Scope zusammenfassen, falls der ganze Prozess noch mehr Aktivitäten ausführt.
Einen Scope erzeugt man mit dem `<scope>` Tag, in das man einfach die gewünschten activities schachtelt.
Bsp.:

```
<scope>
      <flow> …. </flow>
</scope>[30]
```

Innerhalb des hier verwendeten `<flow>` könnte nun z.B. die Flugbuchung und die Hotelbuchung gleichzeitig ausgeführt werden.

4.2 Fehlerbehandlung

In BPEL4WS gibt es 3 Ursachen für Fehler die auftreten können:

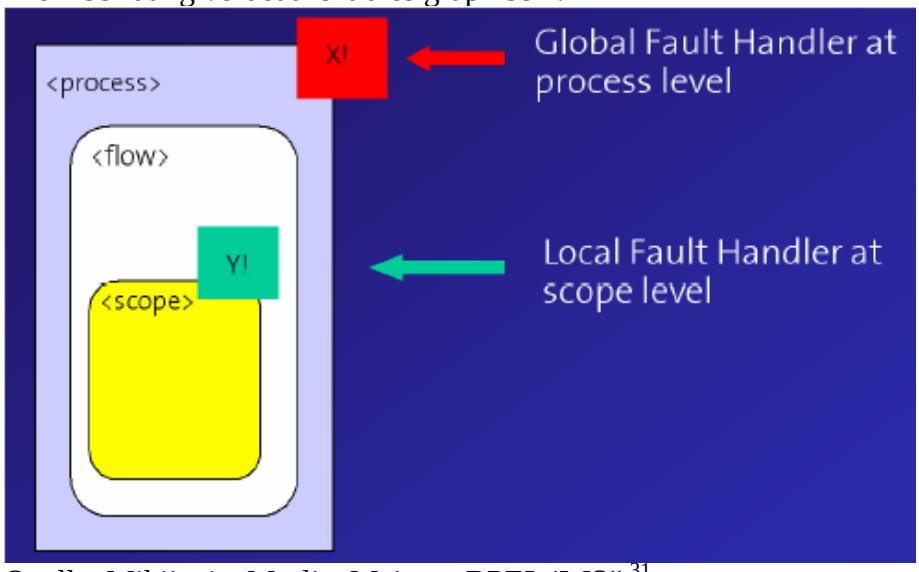
- ein aufgerufener Web-Service verursacht einen Fehler
- eine `<throw>` activity wurde aufgerufen (mit dieser kann man selbst definierte Fehler auswerfen)
- ein interner Fehler tritt auf, z.B. typemissmatch zu Laufzeit

[30] [MMM] Mikijevic, Maslic, Maier : „BPEL4WS“

Um jetzt einen Fehler der in einem Scope auftritt abzufangen, muss man einen `<faultHandler>` für diesen Scope definieren. Es ist aber auch möglich global Fehler für den ganzen Prozess abzufangen. Man kann also `<faultHandler>` definieren auf

- **Scope-Ebene** oder auf
- **Prozess-Ebene**

Die Abbildung verdeutlicht dies graphisch :



Quelle: Mikijevic, Maslic, Maier : „BPEL4WS“ [31]

Beispielcode für die Definition eines lokalen FaultHandlers auf Scope-Ebene:

```
<scope>
      <faultHandlers>
            <catch      faultName="lns:loanProcessFault"
                        faultVariable="error">
            <sequence   name="fault-sequence">
                  <reply      partner="customer"
                              portType="lns:loanApprovalPT"
                              operation="obtain"
                              variable="approvalInfo"
                              faultName="lns:loanProcessFault"/>
            </sequence>
            </catch>
      </faultHandlers>
      <flow>
      -> insert some activities here
      </flow>
</scope>
```

Mit diesem `<faultHandler>` kann man einen Fehler mit dem Namen „`loanProcessFault`“ abfangen und dann mit der `<reply>` activity den Kunden benachrichtigen dass ein Fehler aufgetreten ist.

[31] [MMM] Mikijevic, Maslic, Maier : „BPEL4WS“

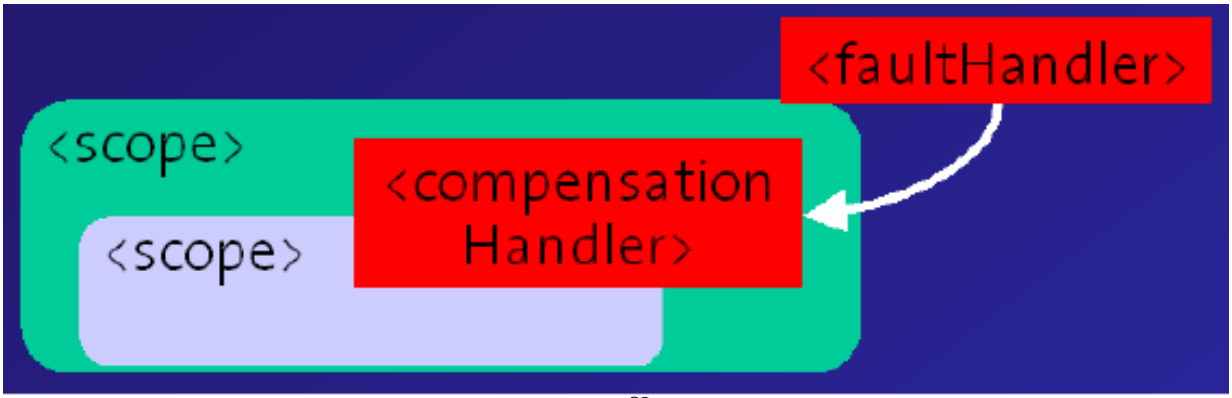
4.3 Kompensation

Kompensation heißt, bestimmte Vorgänge die schon ausgeführt wurden im Fehlerfall wieder rückgängig zu machen, wenn dies von Nöten ist. Beispielsweise bei dem Reisebuchungsbeispiel könnte ja die Flugbuchung und die Hotelbuchung in einen Scope zusammengefasst werden, da diese logisch zusammengehören. Werden nun diese beiden Aktivitäten gleichzeitig ausgeführt, und es tritt bei der Flugbuchung ein Fehler auf, sei es weil kein Flug mehr frei ist für den gewünschten Termin, oder weil der Web-Service einen anderweitigen Fehler erzeugt, dann muss die eventuell schon erfolgreich durchgeführte Buchung des Hotels wieder rückgängig gemacht werden, da man ja ohne Flug dort nicht ankommen kann.
Diese Aufgabe kann nun ein `<compensationHandler>` übernehmen, in dem man die für die Kompensation nötigen Aktivitäten zusammenfasst. Verständlicherweise müssen die beteiligten Web-Services auch solche kompensierenden Operationen, wie das Rückgängigmachen einer Buchung, unterstützen.

Bei Kompensation kann man unterscheiden zwischen:

- **expliziter** Kompensation und
- **impliziter** Kompensation

Von **expliziter** Kompensation spricht man, wenn der CompensationHandler explizit z.B. von einem FaultHandler aufgerufen wird, wie folgende Abbildung verdeutlicht:



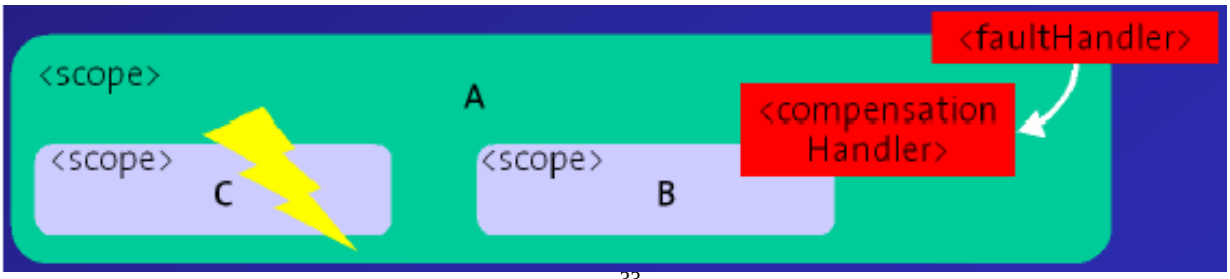
Quelle: Mikijevic, Maslic, Maier : „BPEL4WS“ [32]

Hier ruft der `<faultHandler>` des umschließenden Scopes den `<compesnationHandler>` des inneren Scopes auf, da dessen Aktionen rückgängig gemacht werden müssen. Dies geschieht explizit über die `compensate` activity.

[32] [MMM] Mikijevic, Maslic, Maier : „BPEL4WS“

Implizite Kompensation im Gegensatz bedeutet, dass der `<compensationHandler>` nicht explizit über die `compensate` activity aufgerufen wird, da z.B. der Scope, dessen Aktionen zu kompensieren sind, gar keinen Fehler erzeugt hat, aber aufgrund des Kontextes doch kompensiert werden muss.

Die Graphik zeigt die Situation:



Quelle: Mikijevic, Maslic, Maier : „BPEL4WS“[33]

Hier laufen `Scope B` und `Scope C` parallel innerhalb von `Scope A`. `Scope B` hat seine Arbeit schon erfolgreich beendet, als `Scope C` einen Fehler erzeugt. Da `B` und `C` keinen eigenen `<faultHandler>` haben, fängt der `<faultHandler>` des umschleißenden `Scope A` den Fehler ab und startet nun implizit den `<compensationHandler>` von `B`, da dessen Aktionen ja auch zu `Scope A` im übergeordneten Sinne gehören, und `A` ja wegen des Fehlers in `C` auch nicht korrekt beendet wird.

[33] [MMM] Mikijevic, Maslic, Maier : „BPEL4WS“

5. Ausblick

Als Konkurrenz zu BPEL4WS das von OASIS momentan standardisiert wird, gibt es noch WSCI[34] von W3C[35]. WSCI steht für „**W**eb **S**ervice **C**horeogrphy **I**nterface“, wobei man hier noch den Unterschied zwischen Choreography und Orchestration herausstellen muss:
„If by orchestration you mean there is a central controller and the business process is run by that controller, and by choreography you mean there is no controller and the business process is run in a distributed way by each of the participants, then I think I am right in saying that BPEL4WS does not cover choreography. In this case, the WS-Choreography working group will be working on a separate but related problem space. “[36]
(Martin Chapman)
Dieses Zitat gibt einen Ansatz zur Unterscheidung zwischen „Orchestration“ und „Choreography“ bezüglich Web-Services. Bei „Orchestration“ hat man eine zentrale Stelle an der der Prozess ausgeführt wird, wie dies bei BPEL4WS der Fall ist. Der BPEL4WS Prozess ist die zentrale Stelle die auf andere Web-Services zugreift, wobei der komplette Ablauf aber dort stattfindet. Bei „Choreography“ ist dies nicht der Fall, sondern jeder Teilnehmer an dem Prozess ist gleichgestellt und hat einen Teil der Prozesslogik in sich. Diese Möglichkeit bietet BPEL4WS nicht, und daher ist Martin Chapman auch der Meinung: *"The two specifications may not be rivals. They could address different (but related) problem spaces."* [37]
Man sieht also dass BPEL4WS nicht alles kann, aber die Aussichten, dass es zum Standard wird stehen sehr gut, da große Unternehmen wie Microsoft, IBM, BEA und auch SAP inzwischen BPEL4WS unterstützen. Vielleicht wird es auch in Zukunft eine Fusion zwischen BPEL4WS und WSCI geben, so dass dann in einer Sprache sowohl Orchestration als auch Choreography unterstützt werden.

Unterstützt wird BPEL4WS momentan u.a. vom

- Microsoft Biztalk Server 2004 [38]und von
- IBM WebSphere Business Integration[39].

Man findet auch schon Implementationen wie

- BPWS4J[40] von IBM (eine BPEL4WS Java Runtime)
- Orchestration Server [41]von Collaxa
- ChoreoServer[42] von OpenStorm Software

[34] [WSCI] Web Service Choreography Interface
[35] [W3C]
[36] [MC] Martin Chapman
[37] [MC] Martin Chapman
[38] [BIZ]
[39] [WSBI]
[40] [BPW]
[41] [OS]
[42] [CS]

Literatur

[BIZ] Microsoft Biztalk Server 2004
http://www.microsoft.com/biztalk/

[BPEL] BPEL4WS Business Process Execution Language for Web Services
www-106.ibm.com/developerworks/library/ws-bpel/

[BPW] Business Process Execution Language for Web Services Java™ Run Time
http://www.alphaworks.ibm.com/tech/bpws4j

[CS] ChoreoServer von OpenStorm Software
http://www.openstorm.com

[IBM1] IBM, Understanding BPEL4WS, Part 1
http://www-106.ibm.com/developerworks/webservices/library/ws-bpelcol1/

[IBM2] IBM, Learning BPEL4WS, Part 2
http://www-106.ibm.com/developerworks/webservices/library/ws-bpelcol2/

[MC] Zitat von Martin Chapman aus dem Artikel „Smith on WS-C“
http://otn.oracle.com/oramag/webcolumns/2003/techarticles/smith_wsc.html

[MMM] Mikijevic, Maslic, Maier : „BPEL4WS“
http://n.ethz.ch/student/jodaniel/37-310/reports/BPEL4WS_Micijevic&Maier&Maslic.pdf

[OASIS] Organization for the Advancement of Structured Information Standards
http://www.oasis-open.org

[OS] Orchestration Server von Collaxa
http://www.collaxa.com

[SOAP] Simple Object Access Protocol
http://www.w3.org/TR/soap/

[UDDI] Universal Description, Discovery and Integration
http://www.uddi.org

[W3C] World Wide Web Consortium
http://www.w3.org

[WADH] Wohed, Aalst, Dumas, Hofstede: „Pattern Based Analysis of BPEL4WS
http://xml.coverpages.org/AalstBPEL4WS.pdf

[WSBI] WebSphere Business Integration
http://www-306.ibm.com/software/integration/

[WSCI] Web Service Choreogrphy Interface
http://www.w3.org/TR/wsci/

[WSDL] Web Service Description Language
http://www.w3.org/TR/wsdl

[WSFL] Web Services Flow Language von IBM
http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[XLANG] Microsoft XLANG, Erweiterung von WSDL
http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

[XML] Extensible Markup Language
http://www.w3.org/TR/REC-xml/

[YANG] Jian Yang, Web Service Componentization
October 2003/Vol. 46, No. 10 COMMUNICATIONS OF THE ACM