Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 2428

# BPEL-Webservice-Koordination auf Basis einer Konfigurationsdatei

Sven Knödler

| | |
|---|---|
| **Studiengang:** | Informatik |
| **Prüfer:** | Prof. Dr. Frank Leymann |
| **Betreuer:** | Dipl.-Inf. Stefan Pottinger,<br>Dipl.-Inf. Oliver Kopp |
| **begonnen am:** | 01. 03. 2006 |
| **beendet am:** | 31. 08. 2006 |
| **CR-Klassifikation:** | C. 2. 4, E. 1, H. 3. 4,<br>H. 3. 5 |

# Inhaltsverzeichnis


**1 Einleitung** **5**

**2 Koordination von Webservices** **7**
2.1 WS-Coordination . . . . . 7
2.1.1 Activation Service . . . . . 8
2.1.2 Registration Service . . . . . 9
2.1.3 Protocol Service . . . . . 10
2.2 WS-CDL . . . . . 10

**3 Modell der zentral beschriebenen Koordination** **12**
3.1 Coordination Configuration Format - CCF . . . . . 13
3.1.1 Partners . . . . . 13
3.1.2 AlignmentPoints . . . . . 14
3.1.3 Conditions . . . . . 15
3.1.4 Coordinate . . . . . 16
3.2 Ausführungsmodell . . . . . 17
3.2.1 Beginn der Koordination (Activation und Registration) . . . . 20
3.2.2 Context-Forwarding . . . . . 22
3.2.3 Alignment . . . . . 24
3.2.4 Ende der Koordination . . . . . 26

**4 Transformation der BPEL-Prozesse** **27**
4.1 Startaktivitäten . . . . . 27
4.2 Koordinationsinterne Aufrufe . . . . . 34
4.3 Alignment . . . . . 37
4.4 Auswertung der Bedingungen . . . . . 41
4.5 Reaktionen . . . . . 44
4.6 Ende der Koordination . . . . . 46
4.7 Correlation . . . . . 47

**5 Weitere implementierte Services** **51**
5.1 Protocol Service . . . . . 51
5.2 Zwischenservices für Activation und Registration . . . . . 56
5.3 Service für den Datenbankzugriff . . . . . 57

**6 Erweiterungsmöglichkeiten** **59**
6.1 Verhalten bei Fehlern . . . . . 59
6.2 Privacy . . . . . 61
6.3 AlignmentPoints . . . . . 63
6.4 Reaktionen . . . . . 64

**7 Zusammenfassung und Ausblick** **67**

**A Transformation und Deployment** **68**

**B Plattform zur Ausführung** **70**

**C Anforderungen an BPEL-Quellen** **74**
C.1 Anforderungen an BPEL-Dateien . . . . . 74
C.2 Anforderungen an WSDL-Dateien . . . . . 74

**D XML-Schema des CCF-Formats** **76**

**Literaturverzeichnis** **79**

**Abbildungsverzeichnis** **82**

# 1 Einleitung

Mit der Verbreitung der Serviceorientierten Architektur (SOA) geht auch die Entwicklung und der Einsatz der damit verbundenen Systemplattformen einher. Obwohl es inzwischen eine Vielzahl von Standards in diesem Bereich gibt, haben nur wenige eine weite Verbreitung gefunden oder sich sogar als Eckpfeiler der SOA etabliert. Für die Schnittstellenbeschreibung von Webservices ist dies die Web Services Description Language (WSDL) [CCMW01], für den Transport das Hypertext Transfer Protocol (HTTP) [FMF+99]. Nachrichtenaustausche werden neben dem oftmals als REST-Architektur bezeichneten Zugriff mittels HTTP-Methoden, vor allem im Zusammenhang mit WSDL durch SOAP [GHM+03] realisiert. Im Bereich der Beschreibung von Geschäftsprozessen hat sich die von namhaften IT-Unternehmen entwickelte Business Process Execution Language (WS-BPEL) [ACD+03] zum De-facto-Standard entwickelt.

Die so entstandenen verteilten, lose gekoppelten Anwendungen verlangen nach weiteren Standards, welche das Zusammenspiel der teilnehmenden Services regelt. WS-Coordination [CCF+05] stellt die Grundlage für solche Koordinationen zur Verfügung auf deren Basis dann weitere, auf die jeweiligen Anforderungen spezialisierte Protokolle, definiert werden. Sprachen wie WS-CDL [KBR+05] sollen dagegen die so genannte Choreographie von Services, d. h. die Sequenzen und Bedingungen der Nachrichtenaustausche aus globaler Sicht beschreiben. WS-CDL hat jedoch den entscheidenden Nachteil, dass es bisher keine verfügbare Engine zur Ausführung der erzeugten Choreographien gibt.

An dieser Stelle setzt die vorliegende Arbeit an. Ähnlich wie bei WS-CDL soll das Zusammenspiel von teilnehmenden Services in einer zentralen Datei festgelegt werden können. Aus dieser zentralen Datei und den vorliegenden in BPEL beschriebenen Services wird dann ein lauffähiges System erzeugt, das sich den Vorgaben entsprechend verhält.

Die Aufgabenstellung lässt sich dabei in drei Teile gliedern:

- Der Entwurf des Formats der zentralen Datei, welches ermöglichen soll, koordinationsspezifische Zusammenhänge auszudrücken.
- Die Entwicklung eines Koordinationsprotokolls, auf welches die Vorgaben aus der zentralen Datei abgebildet werden können, sowie die Implementierung eines entsprechenden Protocol Service.
- Die Implementierung eines Transformators, der die gegebenen BPEL-Prozesse so modifiziert, dass sie zusammen mit einem Koordinator die beschriebene Koordination ausführen. Der Koordinator stammt aus der Diplomarbeit von Thorsten Vetter [Vet06] und wird um den neuen Protocol Service ergänzt.

Im Folgenden wird zunächst der Standard WS-Coordination noch einmal zusammenfassend beschrieben, da er die Grundlage für die Koordinationen darstellt. Zur Abgrenzung der Arbeit gegenüber WS-CDL, wird auch der Choreographie-Standard in

wenigen Worten umrissen. Dann kann das entwickelte Modell der Koordination mit seinen verschiedenen Bestandteilen beschrieben werden und schließlich auch die Transformation der zugrunde liegenden BPEL-Prozesse.

# 2 Koordination von Webservices

Ein verteiltes System, wie es im Bereich der SOA anzutreffen ist, besteht aus mehreren grundsätzlich unabhängigen Services, die miteinander kommunizieren. Man spricht dabei auch von einer losen Kopplung [Tan95]. Dennoch existieren in der Regel ein oder mehrere gemeinsame Ziele, welche die eigentliche Aufgabe des Systems darstellen. Um diese Ziele erreichen zu können ist eine Koordination der Teilsysteme unverzichtbar [NLJ96].

Für Webservices wurde deshalb der WS-Coordination Standard ins Leben gerufen. Da die Koordination in dieser Arbeit auf WS-Coordination aufbaut, wird diese in den folgenden Abschnitten zusammengefasst.

Ein weiterer Standard, der das Zusammenspiel von Services betrifft, ist WS-CDL. Hierbei stehen die Nachrichtenaustausche zwischen den Services im Mittelpunkt. Der entscheidende Aspekt von WS-CDL für diese Arbeit ist die zentrale Beschreibung des Ablaufs, da diese Vorgehensweise auch im zu entwickelnden Format verfolgt werden soll. Zur Verdeutlichung wird auch dieser Standard anhand gezielt gewählter Ausschnitte in aller Kürze beleuchtet.

## 2.1 WS-Coordination

Der Standard WS-Coordination [CCF+05] wurde von BEA Systems, IBM und Microsoft entwickelt und stellt ein erweiterbares Framework für die Koordination von Webservices zur Verfügung. Es handelt sich dabei also nicht um eine Menge festgelegter Protokolle zur Koordination, sondern um einen grundsätzlichen Mechanismus, der um die jeweiligen Protokolle erweitert wird. Der Ablauf stellt sich, in wenigen Worten zusammengefasst, immer gleich dar:

- Ein Teilnehmer der Koordination erzeugt beim Koordinator einen Kontext.
- Der Kontext wird an die anderen Teilnehmer weitergeleitet.
- Mit Hilfe des Kontexts registrieren sich die Teilnehmer für ein bestimmtes Koordinationsprotokoll.
- Die Teilnehmer tauschen mit dem Koordinator protokollspezifische Nachrichten aus.

Zur Unterstützung dieses Ablaufs ist der Koordinator in drei Services unterteilt, den Activation Service, den Registration Service sowie den jeweiligen Protocol Service. Der Ablauf ist zur Verdeutlichung noch einmal in Abbildung 1 zusammengefasst.

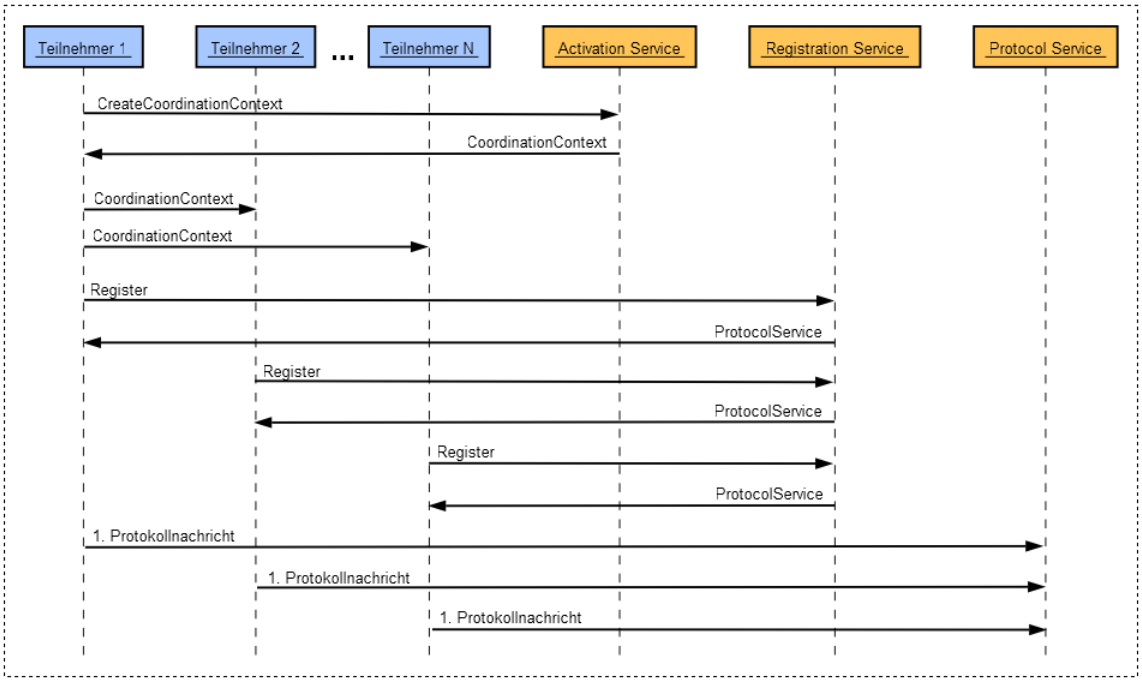


Abbildung 1: *Leicht vereinfachter Ablauf der Anfangsphase einer Koordination*

### 2.1.1 Activation Service

Dies ist der Ausgangspunkt jeder Koordination. Durch einen Aufruf der Operation `CreateCoordinationContext` des Activation Service wird ein `CoordinationContext` für einen der bekannten `CoordinationTypes` erstellt. Als Pseudoschema dargestellt, hat die Nachricht für die `CreateCoordinationContext`-Operation folgende Form:

```
<CreateCoordinationContext ...>
  <Expires> ... </Expires>?
  <CurrentContext> ... </CurrentContext>?
  <CoordinationType> ... </CoordinationType>
  ...
</CreateCoordinationContext>
```

Auf die beiden optionalen Elemente soll nicht näher eingegangen werden. Sie dienen zur Festlegung eines Verfallsdatums für den Kontext bzw. zur Erzeugung von untergeordneten Aktivitäten. Wichtig ist das Element `CoordinationType`. Mit seiner Hilfe kann einer der dem Koordinator bekannten Koordinationstypen festgelegt werden. Für die Koordinationen im Rahmen dieser Arbeit wird ein neuer Koordinationstyp definiert werden.

Bei erfolgreicher Erzeugung eines Kontexts, wird mit einer Nachricht der folgenden Form geantwortet:

```
<CreateCoordinationContextResponse ...>
  <CoordinationContext> ... </CoordinationContext>
  ...
</CreateCoordinationContextResponse>
```

Das einzige vordefinierte Element ist der `CoordinationContext`, welcher vereinfacht folgende Form hat:

```
<CoordinationContext>
  <Identifier> ... </Identifier>
  <CoordinationType> ... </CoordinationType>
  <RegistrationService>
    <wsa:Address> ... </wsa:Address>
  </RegistrationService>
</CoordinationContext>
```

Das Element `Identifier` legt dabei einen eindeutigen Bezeichner für den Kontext fest und unter `CoordinationType` findet sich wieder der Bezeichner für den Koordinationstyp, der angefordert wurde. Sehr wichtig für die Teilnehmer der Koordination ist das `Address`-Element, das die aufzurufende URL für den Registration Service enthält. Der Kontext wird nach seinem Empfang in der Regel an die Teilnehmer der Koordination weitergeleitet, damit diese sich registrieren können. Wie der Kontext übermittelt wird ist anwendungsabhängig und nicht Teil der Spezifikation.

### 2.1.2 Registration Service

Mithilfe des Registration Service können sich alle Teilnehmer für ein Koordinationsprotokoll registrieren. Die Nachricht der aufzurufenden `Register`-Operation hat folgende Form:

```
<Register ...>
  <ProtocolIdentifier> ... </ProtocolIdentifier>
  <ParticipantProtocolService>...</ParticipantProtocolService>
  ...
</Register>
```

Durch `ProtocolIdentifier` wird das Koordinationsprotokoll festgelegt, mithilfe des `ParticipantProtocolService`-Elements die Adresse des protokollverarbeitenden Teils des Teilnehmers.

Die Antwort bei erfolgreicher Registrierung ist wieder auf das wesentliche reduziert und enthält lediglich die Endpunktinformationen des Protocol Services mit dem die Kommunikation zur Koordinierung durchgeführt wird:

```
<RegisterResponse ...>
  <CoordinatorProtocolService>...</CoordinatorProtocolService>
  ...
</RegisterResponse>
```

### 2.1.3 Protocol Service

Nach der Registrierung steht dem Teilnehmer die Adresse des Protocol Service zur Verfügung. Ab diesem Punkt wird die gesamte Koordination über diesen Service abgewickelt. Welche Nachrichten dabei ausgetauscht werden ist gänzlich protokollabhängig und nicht Teil der WS-Coordination-Spezifikation. Auch ist nicht festgelegt, wie die registrierten Teilnehmer gespeichert werden und wie der Zugriff auf diese Information erfolgt. Für die Koordinationen in dieser Arbeit wird der bestehende Koordinator um einen neuen Protocol Service erweitert. Die Registrierungsdaten werden bei der vorliegenden Implementierung zentral in einer Datenbank gespeichert und der Protocol Service kann auf die benötigten Tabellen zugreifen.

## 2.2 WS-CDL

Die Web Services Choreography Description Language (WS-CDL) [KBR+05] wurde von der Web Services Choreography Working Group des W3C ins Leben gerufen, der auch wieder namhafte Unternehmen angehören (Vgl. [CHO]). Aufgabe der Sprache ist es, das untersuchbare Verhalten der Zusammenarbeit von Webservices aus einer globalen Sicht zu beschreiben. Bisher konnte WS-CDL noch keine breite Anwendung finden und es bleibt abzuwarten, ob sich dies mittelfristig ändern wird oder ob andere Standards, die im selben Problemfeld angesiedelt sind, erfolgreicher sein werden (Vgl. dazu [HWS05] und [Kip06]). Die Sprache ist gegenüber WS-Coordination relativ umfangreich, weshalb an dieser Stelle nur ein Überblick über die für diese Arbeit relevanten Aspekte gegeben werden soll.

Die Teilnehmer und ihre Beziehung zueinander werden über `RoleTypes`, `RelationshipTypes` und `ParticipantTypes` beschrieben. Ein `ParticipantType` gruppiert dabei das beobachtbare Verhalten eines einzelnen Teilnehmers, ein `RoleType` legt einen Teil des Verhaltens des Teilnehmers fest und über den `RelationshipType` werden die gegenseitigen Verbindlichkeiten zwischen den Teilnehmern bestimmt.

Durch `InformationTypes`, `Variables` und `Tokens` wird der Datenteil der Choreographie beschrieben. Durch sie können Zustände und andere untersuchbare Informationen überprüft werden. Bei den Variablen wird dabei unterschieden zwischen:

- **Information Exchange Capturing Variables:** Diese dienen der Speicherung von Informationen eingehender oder der Festlegung ausgehender Nachrichten.

- **State Capturing Variables:** Um erkennbare Änderungen an `RoleTypes` aufgrund von Informationsaustauschen zu speichern, wird dieser Variablentyp verwendet. Also zur Speicherung des Zustandes einer Interaktion wie z. B. der Zustand „Bestellung abgeschickt “ bei einem Bestellvorgang.
- **Channel Capturing Variables:** Variablen dieser Art sind dazu gedacht, um Eigenschaften wie die URL, zu der die Nachricht geschickt werden kann, sowie andere Channel-Eigenschaften wie z. B. Security- oder Reliability-Vorkehrungen, zu speichern.
- **Exception Capturing Variables:** Der letzte Typ von Variablen dient der Speicherung von Informationen über verursachte Fehler.

Die eigentliche Festlegung des Informationsaustauschs geschieht über `ChannelTypes`. Hier kann u. a. die Art der auszutauschenden Informationen, sowie die Anzahl der Austausche, berücksichtigt werden.

Zusammengefasst erfolgt die Verbindung der Teilnehmer dann über `Channels`, wobei Variablen dazu benutzt werden um Informationsaustausche, Zustände von Interaktionen sowie Channel-Eigenschaften und Fehlerinformationen zu speichern. Auf dieser Basis wird die Choreographie durchgeführt und den Teilnehmern ermöglicht ihre Informationen abzugleichen.

Dieser Ausschnitt aus der Sprache reicht bereits aus, um im folgenden Kapitel die Unterschiede zum Ansatz dieser Arbeit zu erkennen. WS-CDL bietet noch einige hier nicht genannten Möglichkeiten. Die Beschreibung aller anderen Sprachteile und Zusammenhänge entnehmen Sie deshalb bitte der Spezifikation [KBR+05].

# 3 Modell der zentral beschriebenen Koordination

Dieses Kapitel widmet sich dem im Rahmen dieser Arbeit entwickelten Modell einer Koordination. Dazu gehört neben der Beschreibung des zugrunde liegenden Formats auch das Ausführungsmodell. Der Arbeitsablauf für den Einsatz des Konzepts ist folgender (vgl. Abbildung 2):

- Erstellung der einzelnen BPEL-Prozesse und zugehörigen WSDL-Dateien ohne Rücksichtnahme auf Koordination. Nur wenige zusätzliche Bedingungen zur Schema-Validität müssen durch diese BPEL-Prozesse erfüllt sein (Vgl. Anhang C).
- Erstellung einer zentralen Datei zur Beschreibung der Koordination (siehe Abschnitt 3.1).
- Transformation der bestehenden Prozesse mithilfe des entwickelten CCF-Transformers.
- Deployment der modifizierten Prozesse und WSDL-Dateien auf der ActiveBPEL-Engine.

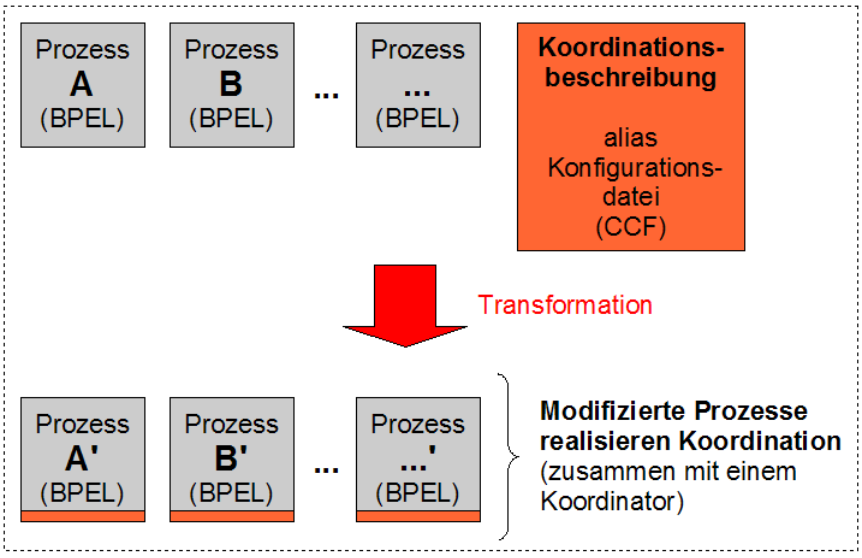


Abbildung 2: *Hinzufügen transformationsspezifischer Teile*

Die Grundlage für die Koordination bilden, ähnlich wie bei WS-CDL, Variablen. Da das neue Modell allerdings im Gegensatz zu WS-CDL ausschließlich die Koordination von BPEL-Prozessen betrifft, wurde kein neues Variablen-Konstrukt eingeführt. Stattdessen werden durch den Benutzer ausgewählte Variablen aus den teilnehmenden BPEL-Prozessen verwendet.

## 3.1 Coordination Configuration Format - CCF

Das XML-Format für die zentrale Datei zur Beschreibung der Koordination, das Coordination Configuration Format (CCF), wurde gezielt so einfach wie möglich gehalten, so dass ein Einsatz ohne längere Einarbeitung möglich sein sollte. Eine CCF-Datei lässt sich als Pseudo-Schema folgendermaßen zusammenfassen:

```
<coordinations activationService="...">

  <partners>
  </partners>
    <partner name="..." bpel="..." wsdl="..."/>+

  <conditions>
    <condition name="..." expression="..."/>+
  </conditions>

  <alignmentPoints>
    <alignmentPoint name="...">+
      <align partner="..." alignment="..."/>+
    </alignmentPoint>
  </alignmentPoints>

  <coordinate conditions="..." alignmentPoints="..."
                                    do="..." else="..."/>+

</coordinations>
```

Das einzige Attribut des Wurzelelements dient dabei zur Festlegung eines Activation Service. Als Wert wird dementsprechend die URL zum Activation Service erwartet. Im Folgenden werden nun die Elemente einer CCF-Datei im Detail besprochen.

### 3.1.1 Partners

Mithilfe dieses Elements werden die Partner festgelegt, welche in der Koordination teilnehmen können. Jedes Partner-Element repräsentiert dabei einen BPEL-Prozess. Natürlich können später bei der Ausführung mehrere Instanzen eines Prozesses existieren. Das Partner-Element hat folgende Attribute:

- `name`: Hiermit wird ein eindeutiger Name für den Prozess festgelegt. Diese Namen sind notwendig, um die Zusammenhänge in den weiter unten beschriebenen Elementen auszudrücken.
- `bpel`: Dieses Attribut stellt die Adresse zum ursprünglichen Prozess-Quellcode in BPEL zur Verfügung, der durch den CCF-Transformer modifiziert werden

soll. Die Angabe muss als URL erfolgen. Für lokale Dateien ist demnach der „`file://`“-Präfix zu verwenden.

- `wsdl`: Dient zur Festlegung der zum Prozess gehörigen WSDL-Datei. Auch für dieses Attribut muss eine URL angegeben werden.

Ein Beispiel für einen `Partners`-Abschnitt mit zwei Prozessen könnte folgendermaßen aussehen:

```
<partners>
  <partner name="A" bpel="file:///C:/PartnerA.bpel"
                    wsdl="file:///C:/PartnerA.wsdl" />
  <partner name="B" bpel="file:///C:/PartnerB.bpel"
                    wsdl="file:///C:/PartnerB.wsdl" />
</partners>
```

Hierbei wird auf Dateien verwiesen, die sich im Wurzelverzeichnis von Laufwerk `C` befinden.

### 3.1.2 AlignmentPoints

AlignmentPoints sind Stellen im Interaktionsablauf, an denen ein Abgleich der Teilnehmer erfolgen soll. Dabei wird gewartet bis jeder gelistete Prozess eine anzugebende Aktivität erreicht hat. Für mehr Informationen über den Ablauf siehe Abschnitt 3.2.

Ein AlignmentPoint muss zur späteren Referenzierung durch ein `Name`-Attribut benannt werden.

Es kann eine beliebige Anzahl von AlignmentPoints definiert werden. Jeder AlignmentPoint enthält wiederum pro Partner, der an diesem Abgleich beteiligt sein soll, ein `Align`-Element.

Das `Align`-Element schließlich hat die folgenden Attribute:

- `partner`: Hiermit wird angegeben, für welchen Partner das `Align`-Element gelten soll. Zu Verwenden ist dabei der eindeutige Name, der dem jeweiligen Partner mittels den `Name`-Attributs im `Partner`-Element gegeben wurde.
- `alignment`: Durch diese Angabe wird festgelegt, vor welcher Aktivität bzw. welchen Aktivitäten im Prozess der Abgleich erfolgen soll. Zwei Angaben sind dabei erlaubt. Mit `onActivity(Aktivität)` kann der Name einer einzelnen Aktivität festgelegt werden, mittels `onConcurrentActivities(Aktivität1, Aktivität2, ...)` können mehrere parallel auftretende Aktivitäten genannt werden. Genaueres dazu in Abschnitt 3.2. Es ist noch anzumerken, dass Namen von Aktivitäten in BPEL normalerweise nicht eindeutig sein müssen. Die für diesen Zweck fungierenden Aktivitäten müssen dieses Kriterium jedoch erfüllen (Vgl. Anhang C).

Es folgt noch ein Beispiel für einen möglichen AlignmentPoints-Abschnitt:

```
<alignmentPoints>
  <alignmentPoint name="AP1">
    <align partner="A" alignment="onActivity(AP1Act)" />
    <align partner="B"
      alignment="onConcurrentActivities(AP1Act1,AP1Act2)" />
  </alignmentPoint>
</alignmentPoints>
```

### 3.1.3 Conditions

Mithilfe von `Conditions` können Bedingungen festgelegt werden, welche später mit einem AlignmentPoint in Verbindung gebracht werden und beim Auftreten dieses AlignmentPoints dann überprüft werden. Es kann eine beliebige Zahl von `Conditions` festgelegt werden. Jede hat dabei folgende Attribute:

- `name`: Dies ist wieder ein eindeutiger Name zur Referenzierung der `Condition`.
- `expression`: Die eigentliche Bedingung, ein XPath-Ausdruck, der sich entweder zu `true()` oder zu `false()` auswerten lassen muss. Um Zugriff auf die Prozessvariablen zu erhalten, kann eine zusätzliche Funktion `getPartnerVariable` verwendet werden. Diese gibt es in zwei Ausführungen:

  `getPartnerVariable(Partnername, Variablenname)`

  und

  `getPartnerVariable(Partnername, Variablenname, Part-Name, Query)`.

  Letztere ist dabei für den Zugriff auf die Elemente einer Message-Variablen gedacht und entspricht bis auf die Angabe des Partnernamens der BPEL-Funktion `getVariableData()` (Vgl. [ACD+03]). Unter `Partnername` ist wieder der im Partner-Element vergebene, eindeutige Name des Prozesses zu verstehen, dessen Variable referenziert wird. Zu beachten ist, dass die angegebenen Variablen im Prozess bereits existieren müssen und im Scope der Aktivitäten, zu denen die `Condition` später ausgewertet wird, sichtbar sein müssen. In der Regel wird es sich dabei um globale Variablen handeln. Die Variablenfunktionen dürfen nicht geschachtelt werden.

Ein `Conditions`-Element, zum Vergleich von Preisen könnte wie folgt aussehen:

```
<conditions>
  <condition name="C1" expression=
            "getPartnerVariable(B,price) &lt;=
             getPartnerVariable(A,maxPrice)" />
</conditions>
```

Die `Expression` ist dabei folgendermaßen zu deuten: Der Wert der Variablen `price` des Partners `B` muss kleiner oder gleich des Wertes der Variablen `maxPrice` des Partners `A` sein. Für das Kleiner-Zeichen muss eine Escape-Sequenz verwendet werden, da dieses in XML nicht in Attributen erlaubt ist.

### 3.1.4 Coordinate

Das `Coordinate`-Element ist das Herzstück der CCF-Datei. Hier werden nun AlignmentPoints mit `Conditions` und den zu erfolgenden Reaktionen zusammengebracht. Es verfügt über vier Attribute:

- `conditions:` Eine kommaseparierte Liste von `Conditions`, die überprüft werden sollen.
- `alignmentPoints:` Eine kommaseparierte Liste von AlignmentPoints, bei deren Eintreten die Bedingungen überprüft werden sollen.
- `do:` Die auszuführende Reaktion, wenn die Bedingung für alle Instanzen der betroffenen Prozesse wahr ist.
- `else:` Die auszuführende Reaktion, wenn die Bedingung nicht für alle Instanzen der betroffenen Prozesse erfüllt ist.

Die Reaktion besteht in beiden Fällen aus einer Kette von Anweisungen. Die Anweisungen werden jeweils durch ein Semikolon getrennt. Zwei Anweisungen wurden im Rahmen der Arbeit implementiert.

Zum einen `terminate`, welches eine `Terminate`-Aktivität in BPEL auslöst, zum anderen `throwFault`, die einen Fehler im BPEL-Prozess wirft. Hier besteht noch starkes Erweiterungspotential (Vgl. Abschnitt 6). Alle Anweisungen existieren in mehreren Versionen. Der eigentliche Anweisungsname hat folgenden Aufbau:

`<Anweisungsname><Quantifizierer><Konformität>(<Parameter-Liste>);`

- **Anweisungsname:** Basisname der Anweisung. Also `terminate` oder `throwFault`.
- **Quantifizierer:** `All`, falls die Anweisung für alle registrierten Teilnehmer ausgeführt werden soll. Ansonsten wird der Quantifizierer ausgelassen und die betroffenen Partner werden als Parameter angegeben.
- **Konformität:** Dieser Teil ist nur im `Else`-Attribut zulässig. Im `Do`-Attribut entfällt er. Mögliche Werte sind `Conforming` und `NonConforming`. `Conforming` bedeutet, dass die Anweisung nur für Instanzen der Partner, für welche sich die `Condition` zu `true()` auswertet, ausgeführt wird. `NonConforming` steht entsprechend für die Ausführung bei Instanzen, für die die `Condition` `false()` ergibt.

- **Parameter-Liste:** Die Parameterliste hängt von der gewählten Anweisung und dem Quantifizierer ab. Bei `terminate` unter Verwendung des Quantifizierers `All` sind keine Parameter notwendig. Für `throwFault` ist in jedem Fall als erster Parameter der Fehlername vorgesehen. Wird der Quantifizierer ausgelassen, folgt für beide Anweisungen eine kommaseparierte Liste von Partnernamen.

Anhand einiger Beispiele soll der Gesamtzusammenhang noch einmal verdeutlicht werden:

```
<coordinate conditions="C1" alignmentPoints="AP1"
            do="" else="terminateNonConforming(A,B)" />
```

Eine solche Angabe hätte zur Folge, dass alle Instanzen der Prozesse `A` und `B`, für die die Bedingung `C1` nicht erfüllt ist, beendet werden.

```
<coordinate conditions="C1" alignmentPoints="AP1"
            do="terminateAll()" else="" />
```

Dies bewirkt, dass alle Prozesse genau dann terminiert werden, wenn die Bedingung `C1` für alle Instanzen aller Partner wahr ist.

```
<coordinate conditions="C1,C2" alignmentPoints="AP1"
            do="terminateAll()"
            else="terminateAllConforming()" />
```

Hier werden alle Instanzen, für welche die Bedingungen `C1` und `C2` wahr sind, beendet.

Die Einführung eines `Do`- und `Else`-Zweiges, gleichzeitig mit der Unterscheidung von `Conforming` und `NonConforming`, beruht auf dem Gedanken, größtmöglichste Unterscheidbarkeit zu unterstützen. Deshalb ist der Fall, dass eine Bedingung für alle Instanzen der Partner erfüllt ist, gesondert zu behandeln. Der Fall, dass eine Bedingung für keine der Instanzen erfüllt ist, kann durch Negation der `Expression` abgefangen werden. In Abbildung 3 ist die Unterscheidung noch einmal schematisch dargestellt.

## 3.2 Ausführungsmodell

Für die Ausführung der in einer CCF-Datei beschriebenen Koordination wird ein zu diesem Zweck entwickeltes Protokoll verwendet. Der Ablauf einer CCF-Koordination ist dabei folgender:

- Ein Teilnehmer wird durch eine Nachricht instanziiert. Die Nachricht kommt dabei von einer Quelle außerhalb des zu koordinierenden Prozess-Gefüges.
- Der Teilnehmer erzeugt mittels des Activation Service einen `Coordination-Context`.

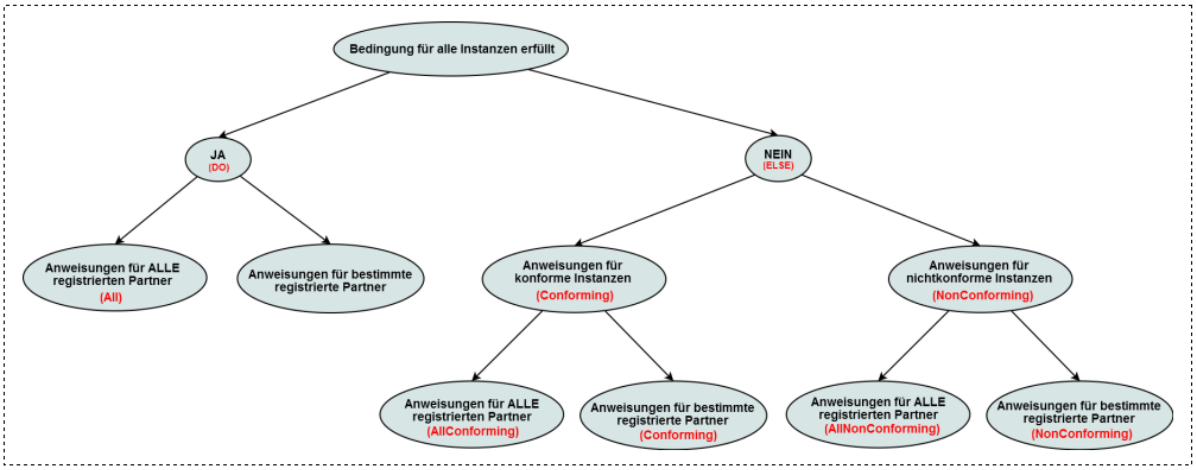


Abbildung 3: *Baum zur Unterscheidung der Fälle und Instanzenmengen*

- Er registriert sich selbst beim Registration Service für die Teilnahme an der Koordination.
- Der Teilnehmer erzeugt eine Instanz des in BPEL modellierten Protocol Service.
- Bei allen Aufrufen an Prozesse, die in der CCF-Datei genannt sind und die möglicherweise eine neue Instanz erzeugen, wird der `CoordinationContext` mitgeschickt. Wurde tatsächlich eine neue Instanz erzeugt, führt der jeweilige Prozess dann nur noch die Registrierung durch.
- Alle Instanzen aller teilnehmenden Prozesse schicken beim Erreichen eines AlignmentPoint eine entsprechende Nachricht an den Protocol Service und warten auf eine Rückmeldung.
- Die Rückmeldung wird vom Protocol Service dann generiert, wenn er von allen für den AlignmentPoint maßgeblichen Prozess-Instanzen eine Nachricht empfangen hat.
- Die Prozesse führen je nach dem Auswertungsergebnis der Bedingung aus der CCF-Datei entsprechende Aktionen aus.
- Vor der Beendigung jeder Prozess-Instanz wird eine Ende-Benachrichtigung an den Protocol Service gesendet.
- Hat der Protocol Service von allen registrierten Teilnehmern eine solche Nachricht erhalten, beendet er sich selbst und damit die Koordination.

Der Ablauf ist am Beispiel von zwei Prozessen in Abbildung 4 zusammengefasst. In den folgenden Abschnitten wird nun auf die Details der einzelnen Phasen einer Koordination eingegangen.

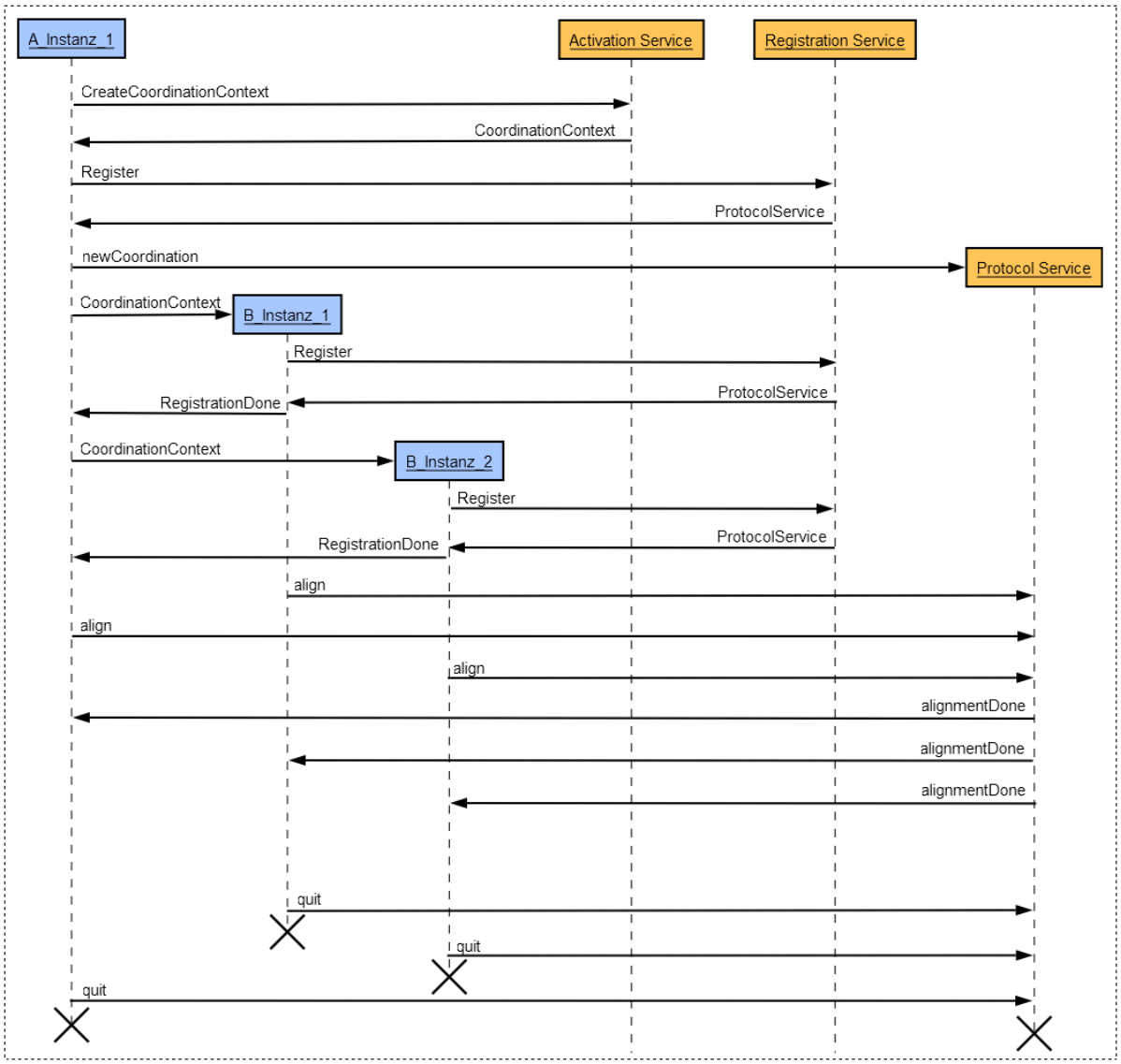


Abbildung 4: *Beispiel für den Ablauf einer CCF-Koordination mit zwei Partnern*

### 3.2.1 Beginn der Koordination (Activation und Registration)

Angestoßen wird die Koordination, wie bereits erwähnt, durch eine instanzerzeugende Nachricht von einer Quelle außerhalb der Menge der Teilnehmer. Die empfangene Nachricht ist dabei nur durch den ursprünglichen BPEL-Prozess festgelegt. Zu jeder möglichen instanzerzeugenden Nachricht wurde vom Transformator eine zugehörige erweiterte Nachricht zu einer erweiterten Operation (siehe 4.1) eingefügt. Diese wird für instanzerzeugende Aufrufe, die innerhalb der zu koordinierenden Prozessmenge stattfinden, verwendet. Ein neuer `CoordinationContext` wird nur erzeugt, wenn es sich um eine ursprüngliche Nachricht handelt. Dieser wird durch das Senden der in 2.1.1 beschriebenen `CreateCoordinationContext`-Nachricht an den Activation Service erstellt und durch die Rückmeldung an den Prozess empfangen.

Eine typische Nachricht an den Activation Service wäre z. B. :

```
<CreateCoordinationContext>
  <CoordinationType>
    http://vortexport.net/CCFCoordination
  </CoordinationType>
  <InstanceIdentifier>
    http://vortexport.net/instanceId/1
  </InstanceIdentifier>
</CreateCoordinationContext>
```

Dabei steht der `CoordinationType` für alle CCF-Koordinationen fest. Der `InstanceIdentifier` ist eine URI, welche die sendende Prozess-Instanz eindeutig identifiziert und für die Korrelation in BPEL verwendet wird. Dieses Element ist in der Activation Service Implementierung von Thorsten Vetter im Header vorgesehen. Da Headerinformationen allerdings nicht für die Korrelation in BPEL verwendet werden können, wurde ein eigener Activation Service erstellt, der diese Angabe im Body der Nachricht erlaubt (Siehe Kapitel 5.2). Dieselbe Erweiterung wurde auch in der Antwort eingeführt:

```
<CreateCoordinationContextResponse>
  <CoordinationContext>
    ...
  </CoordinationContext>
  <InstanceIdentifier>
    http://vortexport.net/instanceId/1
  </InstanceIdentifier>
</CreateCoordinationContextResponse>
```

Der Inhalt des `CoordinationContext`-Elements wurde in der Nachricht zu Gunsten der Übersicht ausgelassen.

Ist die Aktivierung erfolgt oder wurde der `CoordinationContext` durch eine erweiterte Nachricht erhalten, steht durch den Kontext die Adresse des Registration Service zur Verfügung. Jeder teilnehmende Prozess registriert sich dabei für das `CCFCoordination`-

Protokoll, sowie zusätzlich für alle Koordinationen an denen er beteiligt ist. Dabei entspricht jede Koordination, bildlich also jedes `Coordinate`-Tag in der CCF-Datei, genau einem Protokoll. Der Name des Protokolls wird durch den CCF-Transformer aus den Namen der `Conditions` und weiteren Informationen generiert. Durch die Unterscheidung zwischen verschiedenen Protokollen sind gezielte Antworten des Protocol Service an die jeweils relevanten Teilnehmer möglich. Die Registrierung für das `CCFCoordination`-Protokoll wird z. B. für das Feststellen des Endes der Koordination benötigt (Siehe 3.2.4).

Auch für den Registration Service mussten die bereits beim Activation Service besprochenen Erweiterungen zur Ermöglichung der Korrelation vorgenommen werden. Hier aber zusätzlich auch noch bezüglich dem `ContextIdentifier`, welcher ursprünglich ebenfalls im Header übertragen wurde:

```
<Register>
  <ProtocolIdentifier>
    http://vortexport.net/CCFProtocol_C1_AP1_27130053
  </ProtocolIdentifier>
  <ContextIdentifier>
    http://www.iaas.uni-stuttgart.de/wscoor/contextID/25
  </ContextIdentifier>
  <ParticipantProtocolService>
    ...
  </ParticipantProtocolService>
</Register>
```

Eine mögliche Antwort, mit der Erweiterung:

```
<RegisterResponse>
  <CoordinatorProtocolService>
    ...
  </CoordinatorProtocolService>
  <ParticipantIdentifier>
    http://www.iaas.uni-stuttgart.de/wscoor/participantID/14
  </ParticipantIdentifier>
  <ParticipantinstanceIdentifier>
    http://vortexport.net/instanceId/1
  <ParticipantinstanceIdentifier>
</RegisterResponse>
```

Das Feld `ParticipantIdentifier` wurde durch die Implementierung von Thorsten Vetter eingeführt, erfüllt in diese Arbeit jedoch keine Aufgabe, da es nicht zur Korrelation in BPEL geeignet ist.

### 3.2.2 Context-Forwarding

Nach der Registrierung fährt der Prozess mit seiner eigentlichen Aufgabe fort. Möglicherweise erfolgen dabei instanzerzeugende Aufrufe an andere teilnehmende Prozesse. Diese Aufrufe wurden durch den CCF-Transformator durch erweiterte Aufrufe ersetzt, welche folgende Änderungen enthalten:

- **Neue Operation:** Der Name der neuen Operation wird aus dem ursprünglichen durch Erweiterung mittels eines Suffix erstellt.
- **Erweiterte Nachricht:** Die Nachricht wird um drei Elemente erweitert.
- **Synchronität:** Der Aufruf wird, falls dies noch nicht der Fall ist, zu einem synchronen Aufruf umfunktioniert.

Angenommen der ursprüngliche Aufruf besitzt folgende ausgehende Nachricht:

```
<message name="getPriceMsg">
  <part name="articleId" type="xsd:long" />
</message>
```

Dann hätte die zugehörige erweiterte Nachricht folgende Form:

```
<message name="getPriceMsg_ccf_context">
  <part name="articleId" type="xsd:long" />
  <part name="ccf_invokeIndex" type="xsd:long" />
  <part name="ccf_instanceId" type="xsd:anyURI" />
  <part name="ccf_context" type="xsd:anyType" />
</message>
```

Die `articleId` wurde dabei aus der Originalnachricht übernommen. `ccf_invokeIndex` und `ccf_instanceId` werden benutzt, um eigentlich asynchrone Aufrufe in eine synchrone Form zu wandeln. Dazu wird eine zusätzliche Rückantwort, auf welche der Absender wartet, durch den Transformator eingeführt (Siehe auch Abschnitt 4.2). Unter `ccf_context` wird der `CoordinationContext` übertragen, so dass dieser dem aufgerufenen Teilnehmer zur Verfügung steht und dieser bei Bedarf eine Registrierung vornehmen kann.

Die Umwandlung in synchrone Aufrufe ist wichtig, um fehlerhaftes Verhalten beim Ablauf der Koordination zu vermeiden. Nehmen wir z. B. an, es gibt zwei Partner, `A` und `B`. Partner `A` erzeugt durch seine Aufrufe zwei Instanzen von `B`. Wenn diese Aufrufe nun nicht synchron sind, kann es sein, dass Partner `A` einen AlignmentPoint erreicht, bevor die letzte Instanz von `B` sich registrieren konnte. Ist für die Auswertung am AlignmentPoint eine Berücksichtigung aller Instanzen von `B` notwendig, so wird die letzte, noch nicht registrierte Instanz, fälschlicherweise ausgelassen (Vgl. Abbildung 5). Sind die Aufrufe synchron, ist gewährleistet, dass die erzeugte Instanz bereits registriert ist, da erst dann eine Rückmeldung an den Aufrufer erfolgt. Für die genaue Umsetzung der Rückmeldung siehe Abschnitt 4.2.

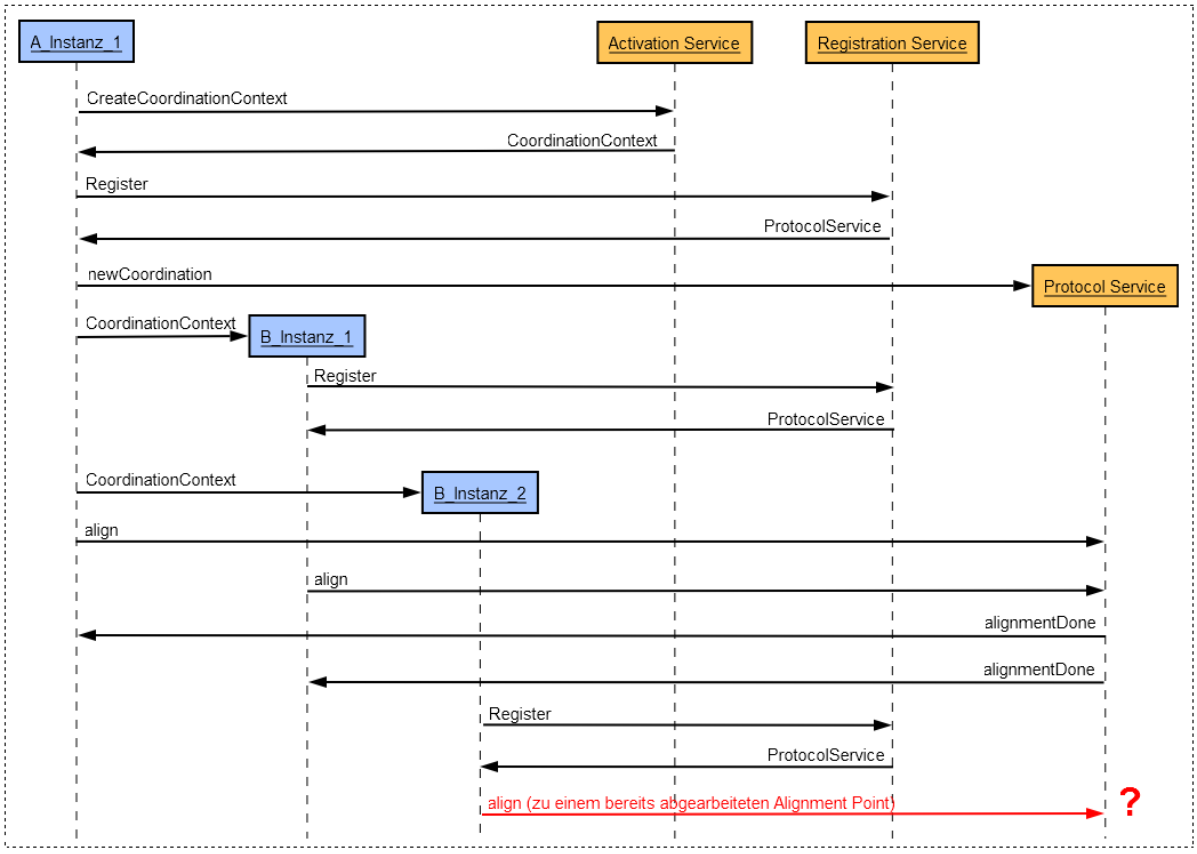


Abbildung 5: *Möglicher fehlerhafter Ablauf bei asynchronen Aufrufen*

### 3.2.3 Alignment

Kommen wir nun zur entscheidenden Phase bei der Koordination, dem Abgleich. In der CCF-Datei wird der Punkt des Abgleichs in den `Align`-Elementen durch die Angabe von Aktivitäten, vor welchen dieser durchgeführt werden soll, festgelegt. Im Falle von zwei oder mehr parallelen Aktivitäten wird gewartet, bis jede der noch erreichbaren Aktivitäten im Originalprozess vor der Ausführung steht (Vgl. 4.3). Ist dieser Punkt erreicht oder handelt es sich um eine einzelne Aktivität, werden nacheinander folgende Schritte ausgeführt:

- Senden einer `Align`-Nachricht an den Protocol Service.
- Warten auf eine Antwort vom Protocol Service. Diese erfolgt erst nach dem Eingang der `Align`-Nachrichten aller beteiligten Prozess-Instanzen.
- Auswertung der in der CCF-Datei angegebenen Bedingung anhand der vom Protocol Service übermittelten Variablen.
- Ausführung der dem Auswertungsergebnis entsprechenden Anweisungen.
- Ausführung der Aktivität welche in der CCF-Datei als Abgleichungspunkt angegeben wurde.
- Fortführung des normalen Prozessablaufs.

Zur Ermöglichung der Auswertung in den Teilnehmern, sind in der Regel Variablenwerte anderer Teilnehmer notwendig. Diese werden dadurch zugänglich, dass jeder Prozess mit seiner `Align`-Nachricht genau die eigenen Variablen mitschickt, welche für die Auswertung relevant sind. Diese Entscheidung muss allerdings nicht dynamisch zur Laufzeit getroffen werden, sondern kann vom CCF-Transformer statisch bei der Modifizierung erfolgen, da sie für jeden AlignmentPoint und Partner fix ist. Zu den einzelnen Prozessen gelangt die Gesamtinformation dann über die Rückmeldung des Protocol Service.

Die `Align`-Nachricht hat als WSDL-Definition folgende Form:

```
<message name="alignMsg">
  <part name="alignment" type="tns:alignmentType"/>
  <part name="contextId" type="xsd:anyURI"/>
  <part name="participantInstanceId" type="xsd:anyURI"/>
</message>
```

Die `contextId` ist der eindeutige Bezeichner, der für den `CoordinationContext` erzeugt und vom Activation Service zur Verfügung gestellt wurde. Sie dient zur Korrelation mit dem Protocol Service, da für jeden Kontext eine Instanz des selbigen existiert. Die `participantInstanceId` ist wiederum der eindeutige Bezeichner für die Prozess-Instanz des Teilnehmers und dient zur Korrelation der Rückmeldungen an diesen. Der Teil `alignment` enthält schließlich die zum Abgleich relevanten Informationen. Als Pseudoschema stellt er sich wie folgt dar:

```
<alignment>
  <contextId>...</contextId>
  <alignmentPoint>...</alignmentPoint>
  <protocolId>...</protocolId>
  <partner>...</partner>
  <variablesList>
    <variable>...</variable>+
  </variablesList>
  <variables>
    <variable>...</variable>+
  </variables>
</alignment>
```

Das `contextId`-Element entspricht wie zuvor, dem eindeutigen Bezeichner für den `CoordinationContext`. Unter `alignmentPoint` wird der String zur Benennung des AlignmentPoints gespeichert, welcher in der CCF-Datei vergeben wurde. Da pro Koordination ein eigenes Protokoll verwaltet wird, alle Protokolle zu einer CCF-Datei aber durch eine einzige Protocol Service Instanz verarbeitet werden, wird über das `protocolId`-Element das zugehörige Protokoll übermittelt. Der Name des Partners, von dem die `Align`-Nachricht stammt, findet sich im String unter dem `Partner`-Element wieder.

Die `variable`-Elemente unterhalb der `variablesList` sind Strings, welche die Variablen, die für den AlignmentPoint benötigt werden, benennen. Die Namen werden vom CCF-Transformer aus den ursprünglichen Namen der BPEL-Variablen und weiteren Eigenschaften wie dem Partnernamen generiert. Dagegen tragen die `variable`-Elemente innerhalb des `variables`-Elements die für die Auswertung relevanten Informationen. Ein solches `variable`-Element besitzt dabei folgende Struktur:

```
<variable>
  <name>...</name>
  <value>...</value>
  <participantInstanceId>...</participantInstanceId>
</variable>
```

Der unter `name` genannte Name entspricht dabei dem Namen aus der `variablesList`. Unter dem `value`-Element wird schließlich der Wert der Variablen zu der durch `participantInstanceId` festgelegten Instanz gespeichert. Dabei kann es sich um skalare Werte, wie XML-Schema-Typen oder um komplexere Strukturen handeln.

Auf Basis dieser Informationen und der Liste der registrierten Teilnehmer aus der Datenbank, kann der Protocol Service die benötigten Informationen sammeln und entscheiden, wann diese vollständig sind, um dann eine Rückmeldung zu generieren.

Die Rückmeldung wird an alle an der speziellen Koordination teilnehmenden Prozess-Instanzen geschickt. Das heißt an die Prozesse, welche im jeweiligen AlignmentPoint oder in den `Do`- und `Else`-Attributen des `coordinate`-Elements genannt werden. Der

Aufbau der Rückmeldungsnachricht ist in folgender WSDL-Definition dargestellt:

```
<message name="alignmentDoneMsg">
  <part name="contextId" type="xsd:anyURI" />
  <part name="alignmentPoint" type="xsd:string" />
  <part name="variables" type="tns:variablesType" />
  <part name="participantInstanceId" type="xsd:anyURI" />
</message>
```

Die Elemente `participantId` und `contextId` sowie `alignmentPoint` dienen, wie auch in der `Align`-Nachricht, zur Identifikation des Teilnehmers, des Kontexts sowie des AlignmentPoints zu dem die Rückmeldung erfolgt. Unter dem `variables`-Element befinden sich wie schon in der `Align`-Nachricht die eben aufgeführten `variable`-Elemente, welche die für die Auswertung nötigen Informationen liefern. Hier finden sich aber im Gegensatz zur `Align`-Nachricht, alle die durch den Protocol Service gesammelten Variablen der Partner wieder.

Alle benötigten Informationen für eine Auswertung stehen nach dem Empfang der Antwort vom Protocol Service zur Verfügung, so dass diese im direkten Anschluss erfolgen kann (Siehe Abschnitt 5.1). Steht das Ergebnis der Auswertung fest, werden die jeweiligen in der CCF-Datei festgelegten Schritte ausgeführt. Das heißt, die Instanz wird entweder terminiert, ein Fehler wird geworfen, für den im Fault-Handler weitere Aktionen erfolgen können, oder es wird ohne Reaktion weitergearbeitet (Siehe auch 6.4).

### 3.2.4 Ende der Koordination

Ist ein Prozess am Ende seiner Ausführung angelangt, sei es durch Erreichen der letzten Aktivität oder z. B. durch eine in der CCF-Datei festgelegte Terminierung, teilt er dies dem Protocol Service durch eine Nachricht mit:

```
<message name="quitMsg">
  <part name="contextId" type="xsd:anyURI" />
  <part name="participantInstanceId" type="xsd:anyURI" />
</message>
```

Die `contextId` dient hierbei wieder der Korrelation mit der korrekten Protocol Service Instanz. Über die `participantInstanceId` wird der eindeutige Bezeichner der vor der Beendigung stehenden Instanz mitgeteilt. Der Protocol Service sammelt diese Bezeichner und kann aufgrund der für das CCF-Protokoll registrierten Teilnehmer überprüfen, ob bereits alle Teilnehmer ihre Arbeit beendet haben. Ist dies der Fall, sind die Koordinationen aus der CCF-Datei abgelaufen und auch die jeweilige Instanz des Protocol Service kann beendet werden.

# 4 Transformation der BPEL-Prozesse

Wie bereits angesprochen, sind die Ausgangsbasis für die Koordination, normale, ohne Rücksicht auf Koordination modellierte BPEL-Prozesse mit zugehörigen WSDL-Dateien sowie eine CCF-Datei. Der Transformator formt diese Prozesse anhand der Vorgaben aus der CCF-Datei um, so dass ein ausführbares koordiniertes Gefüge entsteht. Dieses Kapitel beschäftigt sich mit dieser Remodellierung der Prozesse und geht auf die Schwierigkeiten dabei ein.

Um möglichst beliebige Prozesse zu koordinieren sind, wie Sie sehen werden, erhebliche Codezusätze und der Einsatz von z. T. komplizierten Konstruktionen notwendig. Das liegt u. a. daran, dass in BPEL 1.1 keine Unterprogramme im eigentlichen Sinne möglich sind, die Aufgaben abhängig von ihren Parametern durchführen. So bleibt die Wahl zwischen der Kapselung solcher Aufgaben in eigenständige Services oder einer Ersatzkonstruktion, welche z. B. bei der Modellierung der AlignmentPoints Verwendung finden wird. Auch zwei weitere Spracheigenschaften von BPEL 1.1 erschweren die Umsetzung. Zum einen die Tatsache, dass keine Möglichkeit besteht, bestehende Datenstrukturen dynamisch durch BPEL-Aktivitäten zu erweitern, zum anderen steht im Standard auch keine Möglichkeit zur Verfügung, Instanzen von Prozessen eindeutig zu identifizieren. Beide Probleme wurden aber durch die Verwendung plattformspezifischer Eigenschaften der verwendeten ActiveBPEL-Engine gelöst (Siehe auch Anhang B).

## 4.1 Startaktivitäten

Um eine Koordination nach WS-Coordination zu beginnen, ist die Erzeugung eines `CoordinationContext` durch Aufruf des Activation Service notwendig, sowie anschließend eine Registrierung der Teilnehmer. Es stellt sich nun die Frage, wann eine solche Activation durchgeführt werden soll und wann lediglich eine Registrierung. Grundsätzlich ist zu erwarten, dass der Prozess, welcher als erster aus der Menge der Teilnehmer instanziiert wird, die Aufgabe der Activation übernimmt. Nun wurde im CCF-Format aber keinerlei Kennzeichnung eines Startprozesses vorgenommen. Je nach Anwendung wäre diese Kennzeichnung auch gar nicht durchführbar, da nicht festgelegt sein muss, welcher Prozess als erster aufgerufen wird.

Deshalb wird allen teilnehmenden Prozessen die Funktionalität für beides, Activation sowie Registration mitgegeben. Umgesetzt wird dies durch die Einführung erweiterter Operationen wie bereits in Abschnitt 3.2.2 kurz angesprochen. Zu jeder ursprünglichen instanzerzeugenden Operation wird also eine neue erweiterte Operation generiert. Dazu müssen folgende Schritte durchgeführt werden:

- Einfügen einer erweiterten Nachricht in die zugehörige WSDL-Datei
- Erweiterung des `PortType` um eine neue erweiterte Operation
- Erweiterung des `Binding` um die neue erweiterte Operation

- Einfügen einer Startaktivität für die neue Operation

Die Ergänzungen von Nachricht, `PortType` und `Binding` sind dabei sehr einfach. Nehmen wir z. B. folgende Elemente in der WSDL-Datei als gegeben an (einige Inhalte wurden aus Platzgründen durch drei Punkte abgekürzt):

```
<message name="getPriceMsg">
  <part name="articleId" type="xsd:long" />
</message>

<portType name="Example_PT">
  <operation name="getPrice">
    <input message="tns:getPriceMsg" />
    <output message="tns:returnPriceMsg">
  </operation>
</portType>

<binding name="Example_SOAPBinding" type="tns:Example_PT">
  <soap:binding style="rpc" transport="..." />
  <operation name="getPrice">
    <soap:operation soapAction="getPrice" style="rpc" />
    <input>
      <soap:body encodingStyle="..."
                 namespace="urn:example" use="encoded" />
    </input>
  </operation>
</binding>
```

Dann sehen sie nach dem Hinzufügen der Erweiterungen wie folgt aus:

```
<message name="getPriceMsg">
  <part name="articleId" type="xsd:long" />
</message>
<message name="getPriceMsg_ccf_context">
  <part name="articleId" type="xsd:long" />
  <part name="ccf_invokeIndex" type="xsd:long" />
  <part name="ccf_instanceId" type="xsd:anyURI" />
  <part name="ccf_context" type="xsd:anyType" />
</message>

<portType name="Example_PT">
  <operation name="getPrice">
    <input message="tns:getPriceMsg" />
    <output message="tns:returnPriceMsg">
  </operation>
  <operation name="getPrice_ccf_context">
    <input message="tns:getPriceMsg_ccf_context" />
    <output message="tns:returnPriceMsg">
  </operation>
</portType>

<binding name="Example_SOAPBinding" type="tns:Example_PT">
  <soap:binding style="rpc" transport="..." />
  <operation name="getPrice">
    <soap:operation soapAction="getPrice" style="rpc" />
    <input>
      <soap:body encodingStyle="..."
                 namespace="urn:example" use="encoded" />
    </input>
  </operation>
  <operation name="getPrice_ccf_context">
    <soap:operation soapAction="getPrice_ccf_context"
                    style="rpc" />
    <input>
      <soap:body encodingStyle="..."
                 namespace="urn:example" use="encoded" />
    </input>
  </operation>
</binding>
```

Die Hinzunahme einer neuen Startaktivität für die erweiterte Operation gestaltet sich nicht so einfach. Sie wird im Folgenden im Zusammenhang mit weiteren Problematiken erläutert.

Ist die Erweiterung erfolgt, stehen jeweils Paare von ursprünglichen Operationen und erweiterten Operationen zur Verfügung. Innerhalb der Teilnehmer wird immer die erweiterte Operation aufgerufen, bei welcher der Kontext bereits mitgeliefert wird. Der aufgerufene Prozess muss somit nur eine Registrierung durchführen. Wird an einem Prozess eine ursprüngliche Operation aufgerufen, ist klar, dass diese von außerhalb der Menge der Teilnehmer stammen muss und eine neue Koordination beginnt. In diesem Fall wird vor der Registrierung eine Aktivierung durchgeführt. Abbildung 6 verdeutlicht diesen Zusammenhang.

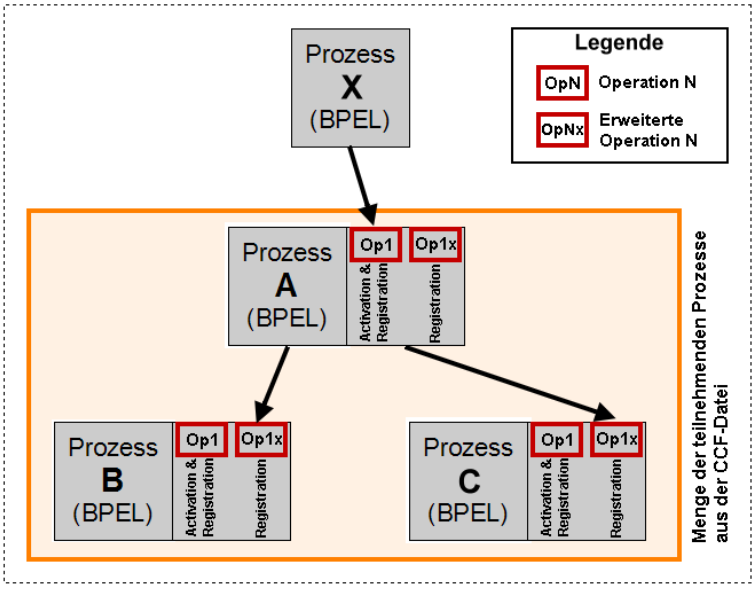


Abbildung 6: *Unterscheidung von internen und externen Aufrufen*

Kommen wir nun zur Umsetzung in den Prozessen. In BPEL kann ein Prozess mehrere Startaktivitäten haben. Diejenige, welche als erste aufgerufen wird, erzeugt dabei eine Instanz des Prozesses. Der spätere Aufruf der weiteren Startaktivitäten dagegen nicht, sofern anhand der Korrelation die Nachrichten an dieselbe Instanz weitergeleitet werden.

Ein erster Ansatz wäre es, jede Startaktivität mit einer Sequenz für Activation und Registration auszustatten. Dann würden weitere Startaktivitäten für die erweiterten Operationen hinzugefügt, welche nur in eine Sequenz zur Registrierung münden. Das Problem dabei ist, dass dann beim Aufruf mehrerer dieser Operationen jedesmal eine Registrierung durchgeführt wird, diese jedoch nur einmal erfolgen sollte. Naheliegend wäre eine Variable, die wie in klassischen Programmiersprachen oft verwendet, als Flag dient und speichert, ob bereits eine Registrierung erfolgt ist. Die Abfrage der Variablen vor einer expliziten Initialisierung ist allerdings nicht zulässig. Ein `Assign`-Element zur Initialisierung darf aber erst nach einer Startaktivität (`Receive` oder `Pick`) folgen. Das heißt, dieses Verfahren ist nicht durchführbar (Vgl. Abbildung 7).

Da keine Möglichkeit gefunden wurde, den Vorgang mit einfachen Mitteln zu modellieren, musste ein komplexerer Ablauf entworfen werden. Dazu werden alle Startaktivitäten als `onMessage`-Elemente unter einer `Pick`-Aktivität zusammengefasst. Dann

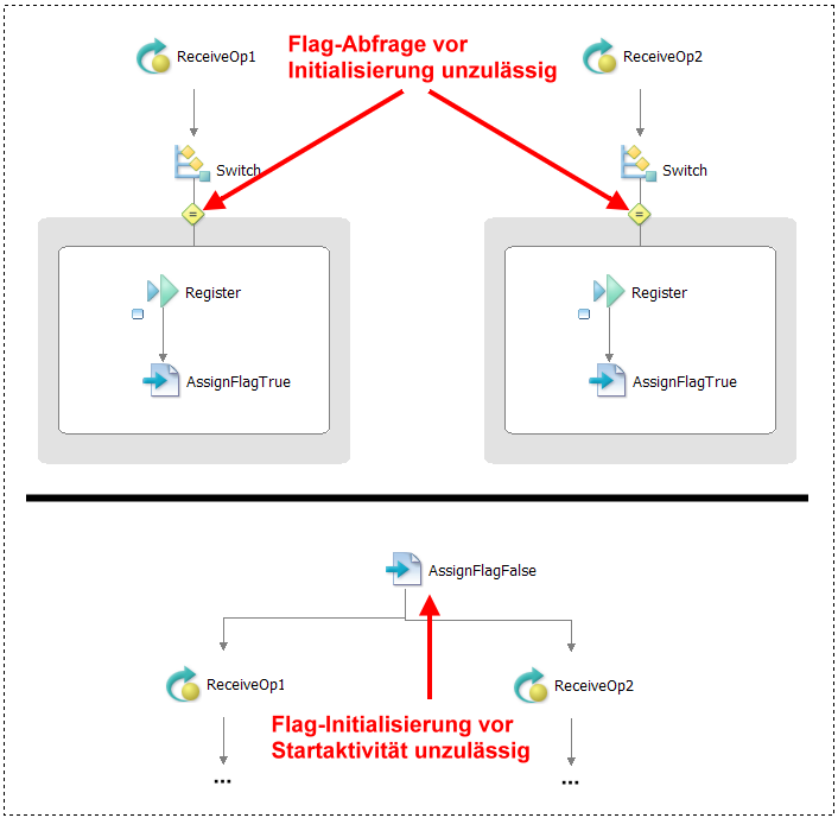


Abbildung 7: *Problem der Flag-Initialisierung*

werden die zugehörigen erweiterten Operationen ebenfalls als `onMessage`-Elemente hinzugefügt. Nach dieser Umformung existiert dann also nur noch eine Startaktivität, das neu konstruierte `Pick`. Die Aktivitäten in möglichen ursprünglichen `onMessage`-Elementen wurden bei der Umformung an dieser Stelle entfernt und werden an späterer Stelle wieder eingesetzt.

In den neu geformten `onMessage`-Zweigen befindet sich dagegen nur noch eine `Assign`-Aktivität, welche im Wesentlichen folgende Zuweisungen enthält:

- Festlegung des eindeutigen Bezeichners der Prozess-Instanz mithilfe der ActiveBPEL-Funktion `getProcessId()`.
- Setzen eines Index, anhand dessen festgestellt werden kann, welcher `onMessage`-Zweig aktiviert wurde.
- Setzen eines Flags zur Kennzeichnung, dass soeben eine neue Instanz erzeugt wurde.
- Speicherung der Tatsache, ob es sich um eine Einweg-Nachricht oder eine Nachricht mit Rückantwort handelt. Diese Information ist für die Umwandlung in einen synchronen Aufruf notwendig (Vgl. Abschnitt 3.2.2).
- Im Falle einer erweiterten Operation: Umkopieren der Nachrichtenteile aus der erweiterten Message-Variable in eine Variable des ursprünglichen Nachrichtentyps.

Nach der neuen `Pick`-Startaktivität folgt ein `Switch` in dessen `Case`-Elementen anhand des eben erwähnten Index entschieden wird, ob nur eine Registrierung oder Activation und Registration durchgeführt werden müssen. Für erweiterte Operationen ist das wie gehabt nur Registration, für ursprüngliche Operationen beides. Im Falle einer Activation wird auch die benötigte neue Instanz des Protocol Service erzeugt.

Ist die Registrierung beendet, folgt eine weitere `Switch`-Aktivität mit einem `Case`-Element. Der Inhalt wird dabei nur ausgeführt, wenn es sich ursprünglich um eine One-Way-Operation handelte. Eine Besprechung dieses Blocks erfolgt bei der Erläuterung der Remodellierung von Aufrufen in Abschnitt 4.2.

Zuvor wurde erwähnt, dass mehrere Startaktivitäten hintereinander aufgerufen werden können. In unserer bisherigen Konstruktion ist dies nicht mehr möglich, da die Pick-Aktivität nur einmal ausgeführt wird. Das Verhalten des Prozesses wäre somit grundlegend verändert, was unerwünscht ist.

Deshalb folgt nun eine `Flow`-Aktivität, die dazu dient, das ursprüngliche Verhalten wiederherzustellen. Bevor wir uns dem Innenleben dieses Block widmen, soll die Gesamtkonstruktion noch einmal in Abbildung 8 zusammengefasst werden. Der Inhalt der Sequenzen für Activation und Registration wird nicht genauer untersucht. Die nachfolgende `Sequence`-Aktivität zur Beendigung wird im Abschnitt 4.6 besprochen.

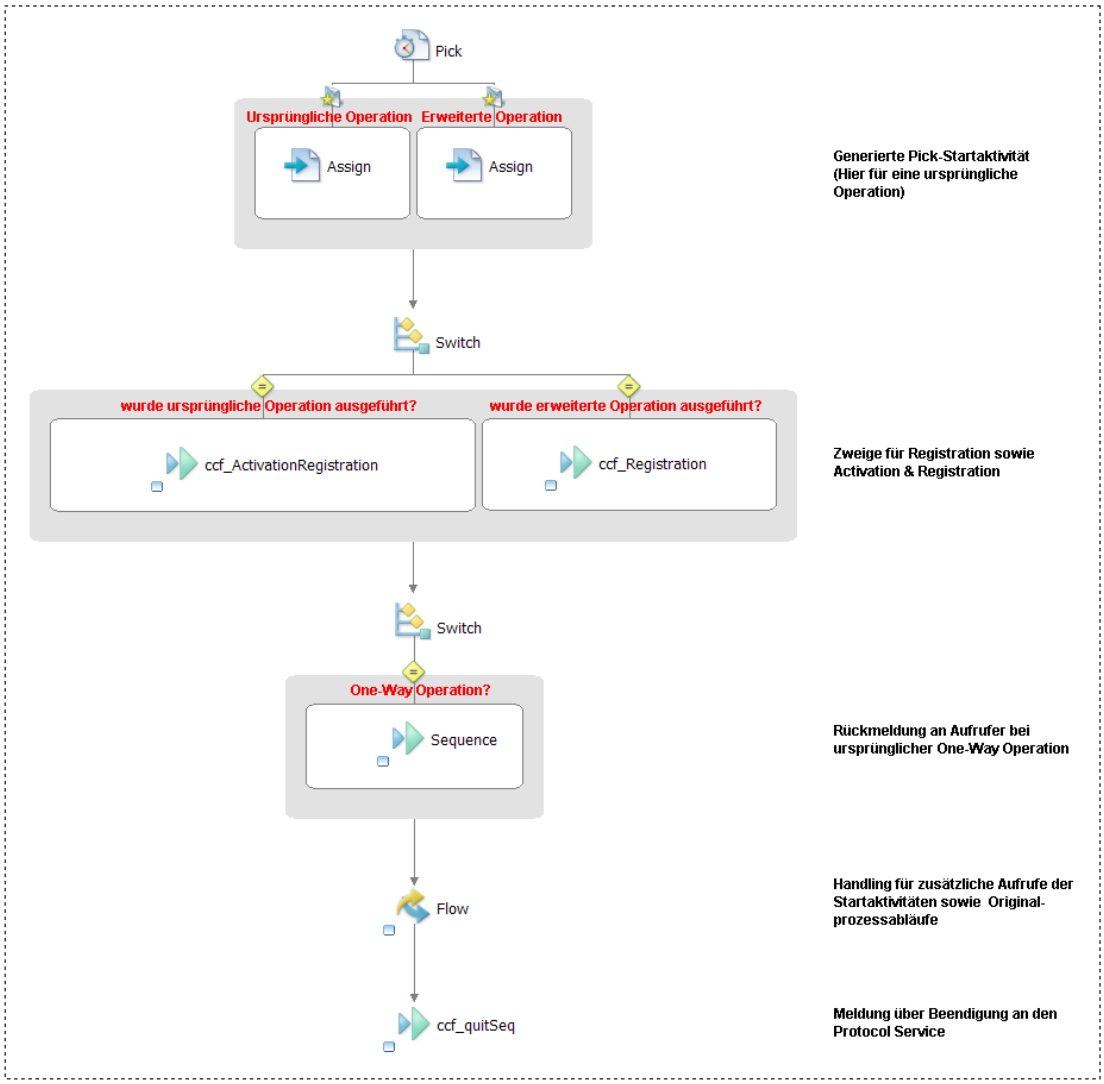


Abbildung 8: *Remodellierter Prozess im Überblick (ohne Event-Handler)*

Betrachten wir die Umsetzung des Ablaufs zur Wiederherstellung des ursprünglichen Verhaltens auch beim mehrfachen Aufruf von Startaktivitäten. Dazu enthält der `Flow`-Container eine Reihe von parallelen `Switch`-Aktivitäten. Für jede ursprüngliche Startaktivität genau eine. Die `Switches` enthalten wiederum zwei `Case`-Zweige. Der eine wird zur Ausführung gebracht, wenn soeben eine neue Instanz von der zu diesem `Switch` gehörigen Operation oder erweiterten Operation erzeugt wurde. Dies kann anhand des oben erwähnten Index sowie dem Flag zur Kennzeichnung einer neuen Instanz festgestellt werden. Tritt dieser Fall ein, so wird das Flag auf `false()` gesetzt. Durch diesen Aufbau werden nach dem Erzeugen einer neuen Instanz genau die zusätzlichen früheren Startaktivitäten aktiviert, welche in der `Pick`-Startaktivität nicht ausgeführt wurden. Auch hier muss im Falle einer ursprünglichen One-Way-Operation eine Rückmeldung an den Aufrufer erfolgen.

Der dann folgende Block besteht aus einem umgebenden `Switch`, der in seinen `Cases` das ursprüngliche Verhalten des Prozesses nach den einzelnen Start-`Receives` bzw. das Verhalten innerhalb der `onMessage`-Elemente der ursprünglichen Start-`Picks` enthält.

Ein Sonderfall muss an dieser Stelle noch angesprochen werden, da durch ihn eine zusätzliche Komplikation des Zusammenhanges erfolgt. Nehmen wir an, die Startaktivitäten im ursprünglichen Prozess befinden sich alle in serialisierbaren `Scopes` (Siehe „Serializable Scopes“ in der Spezifikation [ACD+03]). Nehmen wir weiter an, mindestens zwei dieser Startaktivitäten empfangen in dieselbe Variable. Durch die Kapselung in einen serialisierbaren `Scope` ist sichergestellt, dass die Zugriffe auf die Variable welche innerhalb der jeweiligen `Scopes` stehen, nacheinander erfolgen. Der Empfang im vorgestellten Ablauf erfolgt allerdings nicht innerhalb von serialisierbaren `Scopes`. Deshalb wird für jede Message-Variablen pro empfangender Aktivität eine weitere Variable generiert. In diese Variablen wird dann empfangen. Direkt vor der Ausführung der ursprünglichen nachfolgenden Aktivitäten, in diesem speziellen Fall also innerhalb des serialisierbaren `Scopes`, werden die Daten dann in die ursprüngliche Variable umkopiert. Somit ist der serielle Zugriff wieder hergestellt. Abbildung 9 verdeutlicht diese Lösung nochmals.

## 4.2 Koordinationsinterne Aufrufe

Während des Ablaufs der Koordination, wird es in der Regel vorkommen, dass ein Teilnehmer den anderen durch eine `Invoke`-Aktivität aufruft. Möglicherweise erzeugt dieser Aufruf eine neue Instanz des aufgerufenen Prozesses. Ob dies der Fall ist, lässt sich allerdings nicht statisch entscheiden, d. h. der Transformator kann nicht feststellen, welche Aufrufe tatsächlich instanzerzeugend sein werden. Zwar wird eine Liste aller instanzerzeugenden Operationen aus den Prozessen gewonnen, jedoch kann, z. B. beim zweiten Aufruf der Operation, die Nachricht auch an eine bereits bestehende Instanz geleitet werden.

In Abschnitt 3.2.2 wurde bereits die Notwendigkeit der Synchronität instanzerzeugender Aufrufe innerhalb der Teilnehmer erläutert. Ist ein Aufruf bereits im ursprünglichen Prozess synchron, d. h. handelt es sich um eine Two-Way-Operation, dann sind

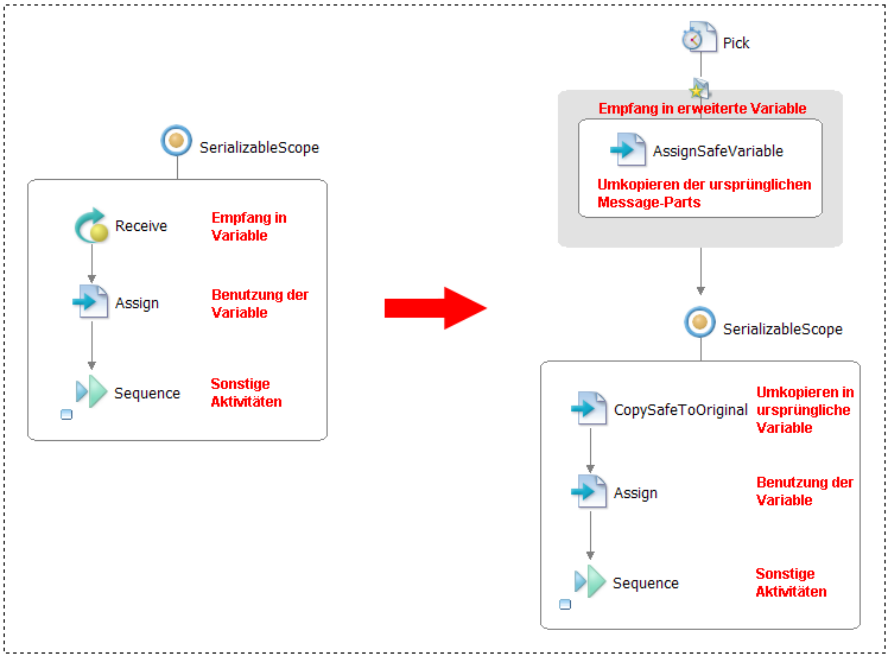


Abbildung 9: *Beibehaltung der Semantik von serialisierbaren Scopes*

für einen solchen Aufruf keine Änderungen notwendig. Bei Einweg-Operationen, die instanzerzeugend sein können, wird dagegen eine zusätzliche Rückmeldung vom CCF-Transformer eingefügt.

Auf der Empfängerseite sieht der Ablauf nach der Transformation wie folgt aus:

- Empfang der Nachricht in der `Pick`-Startaktivität oder in der ersetzten parallelen `Pick`- bzw. `Receive`-Aktivität (siehe Abschnitt 4.1).
- Die Nachricht enthält ein Element `ccf_invokeIndex` (vgl. 3.2.2), welches in die Rückmeldung kopiert wird.
- Die durch den CCF-Transformer hinzugefügte Operation `ccf_regDoneOp` des aufrufenden Prozesses wird aufgerufen.

Der Änderungen am aufrufenden Prozess sind etwas umfangreicher. Betrachten wir zunächst den Ablauf beim Aufrufer:

- Kopieren des `ccf_invokeIndex` in die erweiterte Nachricht. Dieser Index dient zur eindeutigen Identifikation der aufrufenden `Invoke`-Aktivität. Jedes möglicherweise instanzerzeugende `Invoke` erhält also eine solche Nummer.
- Aufruf der erweiterten Operation des anderen Teilnehmers.
- Empfangen der Rückmeldung innerhalb eines Event-Handlers.

- Selbstaufruf einer eigens für die Rückmeldung für dieses `Invoke` erzeugten Operation. Eine `Receive`-Aktivität wartet auf diesen Aufruf und blockiert bis dahin die Fortführung des Ablaufs an dieser Stelle.
- Fortführung des normalen Ablaufs.

Da wegen der Möglichkeit von Nebenläufigkeit, etwa durch den `Flow`-Container, mehrere Aufrufe an Partner zur selben Zeit stattfinden können und somit auch Rückmeldungen zur selben Zeit eingehen können, reicht es nicht aus eine einzige Operation für die Rückmeldung zur Verfügung zu stellen und mittels `Receive`-Aktivität auf sie zu warten. Eine Zuordnung wäre leider nicht möglich. Deshalb wurde vom Transformator für jedes möglicherweise instanzerzeugende `Invoke` eine eigene Operation für die Rückmeldung eingefügt.

Wünschenswert wäre es, wenn nun der antwortende Service direkt die jeweilige Operation aufrufen könnte. Leider ist es in BPEL 1.1 aber nicht möglich die Operation eines Aufrufs dynamisch festzulegen. Ein möglicher Ansatz wäre es gewesen, einen Zwischenservice zu implementieren, der eine Nachricht entgegennimmt, in der die ursprüngliche Nachricht sowie zusätzlich auch die aufzurufende Operation festgelegt wird. Dieser Service könnte dann die somit dynamisch gewählte Operation mit den übergebenen Daten aufrufen und auch etwaige Antworten weiterleiten. Abbildung 10 veranschaulicht diese Art des Aufrufs.

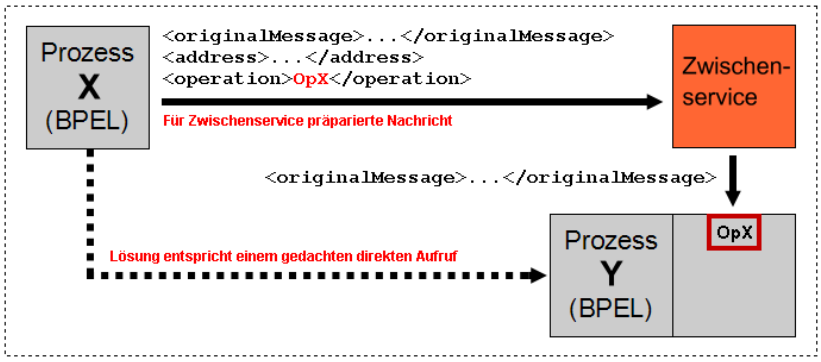


Abbildung 10: *Dynamischer Operationsaufruf durch einen Zwischenservice*

Eine weitere Möglichkeit stellt eine Erweiterung der ActiveBPEL-Engine dar, die in der Diplomarbeit von Daniel Wutke entstanden ist [Wut06]. Diese Erweiterung erlaubt es u. a. die Operation von Aufrufen dynamisch festzulegen. Leider liegt die Erweiterung zur Zeit noch nicht für die verwendete Version 2 der Engine vor.

Da bereits drei weitere Zwischenservices für die Implementierung benötigt wurden (siehe Kapitel 5) und die Verwendung der Erweiterung durch den Versionsunterschied nicht möglich war, wurde eine andere Lösung gefunden.

Der Prozess erhält eine feste Operation, welche für das Absetzen der Rückmeldung aufgerufen werden kann. Diese wird mittels eines Event-Handlers empfangen. Für jedes möglicherweise instanzerzeugende `Invoke` wird eine weitere Operation zum Empfang der Rückmeldung und Entblockieren des Prozessablaufs hinzugefügt. Innerhalb des

Event-Handlers kann nun anhand des bereits erwähnten `ccf_invokeIndex` entschieden werden, welche der Operationen zu einer Entblockierung der korrekten `Receive`-Aktivität führt. Auf diese Operation führt der Prozess einen Selbstaufruf aus. Der genaue Ablauf ist anhand eines Beispiels Abbildung 11 entnehmbar.

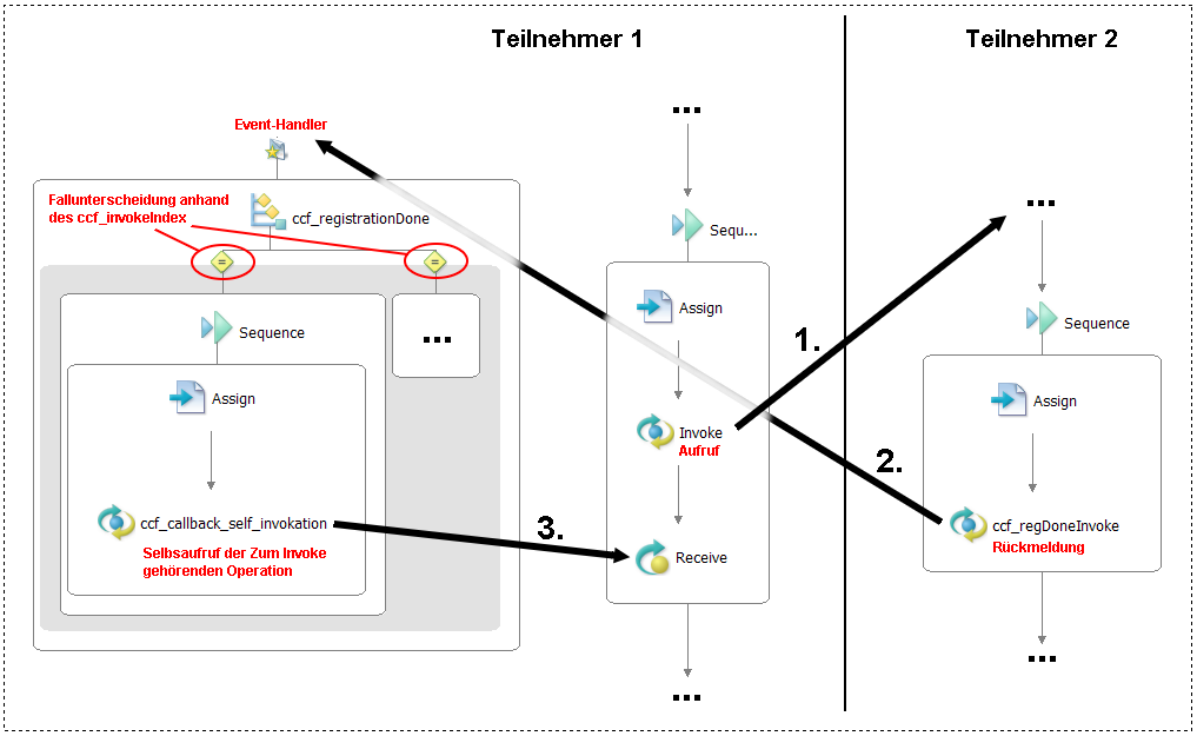


Abbildung 11: *Beispiel - Teilnehmer 1 ruft Teilnehmer 2 auf*

## 4.3 Alignment

Ein Abgleich erfolgt bei Erreichen eines in der CCF-Datei angegebenen Alignment-Point im Prozess. Dabei stehen dem Benutzer zwei Möglichkeiten zur Verfügung. Der Abgleich vor Erreichen einer einzelnen Aktivität (`onActivity`) oder vor dem Erreichen mehrerer paralleler Aktivitäten (`onConcurrentActivities` - siehe Abschnitt 3.1.2).

Im Falle einer einzelnen Aktivität ist die Transformation sehr einfach, im Falle mehrerer Aktivitäten stellt sich aber die Frage, wann genau der Abgleich erfolgen soll. Folgende Möglichkeiten wären naheliegend:

- Abgleich bei jedem Erreichen einer der parallelen Aktivitäten.
- Abgleich beim ersten Erreichen einer der parallelen Aktivitäten.
- Abgleich bei Erreichen beider Aktivitäten.

Der erstgenannte Ansatz ist sicherlich auch der einfachste. Unklar ist aber, was z. B. beim Erreichen der zweiten von mehreren parallelen Aktivitäten geschehen soll. Findet tatsächlich jedesmal ein Abgleich statt, dann hängt die Anzahl der Abgleiche davon ab, wieviele der parallelen Aktivitäten erreicht werden. Dieses Verhalten ist in den meisten Fällen sicherlich nicht wünschenswert.

Die zweite Möglichkeit erscheint auf den ersten Blick sinnvoller. Bei Erreichen einer der Aktivitäten könnte dies durch Setzen einer Variable festgehalten werden, die dann beim Erreichen der weiteren Aktivitäten überprüft wird, so dass kein erneuter Abgleich vorgenommen wird. Nun hat aber der Benutzer bei der Angabe von Aktivitäten für AlignmentPoints sehr wahrscheinlich die Intention, dass ein Abgleich genau beim Erreichen dieser stattfinden soll und dann entsprechende Reaktionen erfolgen. Was genau geschieht, z. B. welche Aktivitäten noch ausgeführt werden hinge bei dieser Implementierung aber von der jeweiligen Ausführung ab und wäre nicht deterministisch. Nehmen wir z. B. das in Abbildung 12 dargestellte Szenario an.

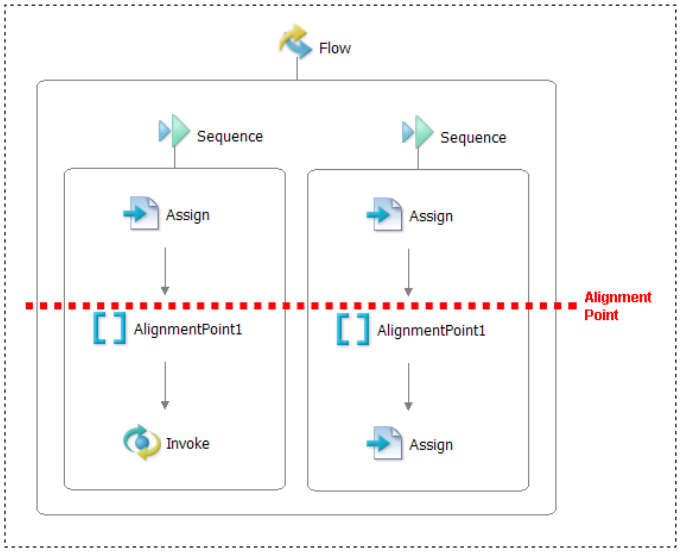


Abbildung 12: *AlignmentPoint mit zwei parallelen Aktivitäten*

Der Übersicht wegen wird hier der ursprüngliche Prozess und nicht die transformierte Form gezeigt. Erreicht der Prozess zuerst die linke Aktivität des AlignmentPoint und wäre die nach der Auswertung auszuführende Reaktion ein Terminieren des Prozesses, so würde die `Invoke`-Aktivität nicht ausgeführt. Wird aber zuerst die rechte Aktivität erreicht und kurz darauf die linke, so kann es sein, dass je nach den zeitlichen Gegebenheiten noch während der Prozess auf eine Antwort vom Protocol Service wartet, das `Invoke` ausgeführt wird. Dieses Verhalten ist sehr unangenehm, da es viel zusätzlicher Denkarbeit bei der Modellierung des Prozesses zur Vermeidung unerwünschter Effekte bedarf.

Der dritte genannte Ansatz birgt dieses Problem nicht. Es wird erst abgeglichen, wenn beide Aktivitäten erreicht sind. Sollte einer der beiden Aktivitäten allerdings nie erreicht werden, z. B. weil eine nicht erfüllte Bedingung den Zweig nicht aktiviert hat, so wird endlos gewartet.

Dies führt zur im CCF-Transformer umgesetzten Lösung. Abgeglichen wird genau dann, wenn alle noch erreichbaren Aktivitäten des AlignmentPoint erreicht sind. Sehr hilfreich ist dabei das von BPEL unterstützte Verfahren der Dead-Path-Elimination (Siehe auch [Muc97]). Die Funktionsweise soll anhand von Abbildung 13 in wenigen Worten beschrieben werden.

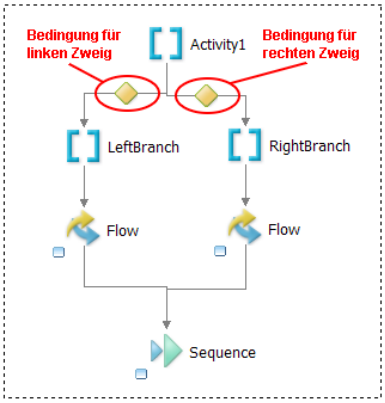


Abbildung 13: *Ablauf zur Erläuterung der Dead-Path-Elimination*

Die `Sequence`-Aktivität besitzt zwei eingehende `Links`. Das heißt, sie wird eigentlich nur dann ausgeführt, wenn beide Zweige abgearbeitet wurden. Wird die Bedingung z. B. des linken Zweiges zu `false()` ausgewertet, erkennt die BPEL-Engine, dass die Aktivitäten dieses Zweiges nicht mehr ausgeführt werden können und markiert diesen Pfad entsprechend als „tot“. Wird im rechten Zweig schließlich die Verknüpfung zur Sequenz erreicht, wird diese dann trotzdem ausgeführt, da alle eingehenden Verknüpfungen entweder aktiviert oder aber „tot“ sind. Voraussetzung dafür ist, dass der `joinFailure` für diese Aktivität unterdrückt wird (z. B. durch Setzen von `suppressJoinFailure='yes'`). Das bedeutet also, dass die Sequenz sowohl ausgeführt wird, wenn beide Zweige abgearbeitet wurden, als auch wenn nur einer der Zweige überhaupt aktiviert und dann abgearbeitet wurde.

Diese Eigenschaft wird nun für die Umsetzung des Alignments bei mehreren parallelen Aktivitäten verwendet. Der Block zum Ausgleich wird von allen Zweigen in denen sich eine in der CCF-Datei genannte Aktivität befindet, verlinkt und erhält schließlich auch wieder eine Verknüpfung zurück zur jeweiligen Aktivität. So wird die zu Beginn dieses Abschnitts erwähnte Semantik erreicht. Abbildung 14 zeigt die Transformation anhand von zwei parallelen Aktivitäten.

Die zentrale Alignment-Sequenz, wie sie auch in der Abbildung zu sehen ist, wird auch für AlignmentPoints mit einzelnen Aktivitäten verwendet, dann aber ohne die hier gezeigte Einbettung und Verknüpfung.

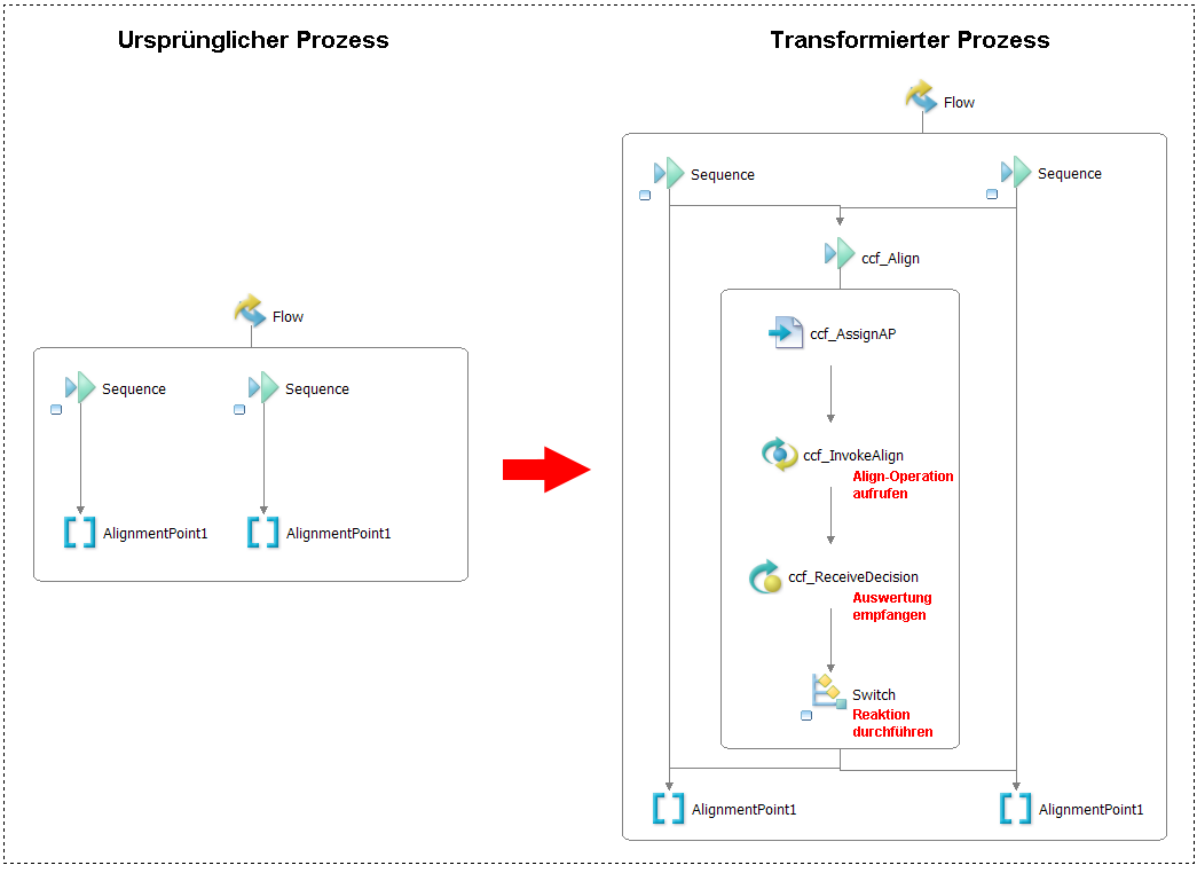


Abbildung 14: *Transformation eines AlignmentPoint bei parallelen Aktivitäten*

Der Abgleich erfolgt dann in mehreren Schritten. Zuerst wird die `Align`-Nachricht (siehe Abschnitt 3.2.3) vorbereitet um dann zum Protocol Service geschickt zu werden. Die Antwort wird in einem Event-Handler empfangen, der in Abschnitt 4.4 beschrieben wird und die Auswertung vornimmt, um diese via Selbstaufruf an das in der Abbildung sichtbare `Receive` zu senden. Danach folgt je nach Ergebnis der Auswertung eine entsprechende Reaktion, die innerhalb der `Switch`-Aktivität festgehalten ist und im Abschnitt 4.5 besprochen wird.

Für die Rückmeldung vom Protocol Service wird jedem Prozess in seiner WSDL-Datei eine zusätzliche Operation mit zugehörigem `Binding` hinzugefügt:

```
<portType name="Example_PT">
  <operation name="ccf_alignment_done_operation">
    <input message="ccf_protocol_ns:alignmentDoneMsg" />
  </operation>
</portType>

<binding name="Example_SOAPBinding" type="tns:Example_PT">
  <soap:binding style="rpc" transport="..." />
  <operation name="ccf_alignment_done_operation">
    <soap:operation soapAction="ccf_alignment_done_operation"
                    style="rpc" />
    <input>
      <soap:body encodingStyle="..."
                 namespace="urn:example" use="encoded" />
    </input>
  </operation>
</binding>
```

Das Format der Rückmeldung wurde bereits in Abschnitt 3.2.3 besprochen.

Es muss noch angemerkt werden, dass die hier vorgestellte Umsetzung von Alignment-Points mit parallelen Aktivitäten nur funktioniert, wenn diese Aktivitäten nicht in unterschiedlichen `While`- oder `Serializable-Scope`-Aktivitäten liegen, da die Verlinkung über die Grenzen dieser Aktivitäten nicht gestattet ist. Siehe hierzu auch Anhang C.

## 4.4 Auswertung der Bedingungen

Wie bereits angesprochen wurde, fügt der CCF-Transformer den Prozessen einen Event-Handler hinzu, der zum Empfang der Rückmeldung des Protocol Service dient. Der Code führt nacheinander folgende Aufgaben durch:

- Empfang der Rückmeldung des Protocol Service.

- Aktivierung des korrekten zum AlignmentPoint der Rückmeldung gehörenden Zweigs.
- Überprüfung der Bedingungen aus der CCF-Datei für die Variablen aller Instanzen.
- Auswertung, ob für diese Instanz eine Reaktion notwendig ist.
- Selbstaufruf zur Entblockierung des wartenden `Receives` und zur Ausführung der Reaktion.

Der Event-Handler besteht, wie in Abbildung 15 zu sehen, aus einer umgebenden `Switch`-Aktivität, die für jeden AlignmentPoint, für den eine Rückmeldung empfangen wird, ein `Case`-Element enthält. Die Aktivierung des korrekten Zweigs wird durch Überprüfung auf den jeweiligen Namen des AlignmentPoint über das Element `alignmentPoint` aus der Nachricht (siehe Abschnitt 3.2.3) durchgeführt.

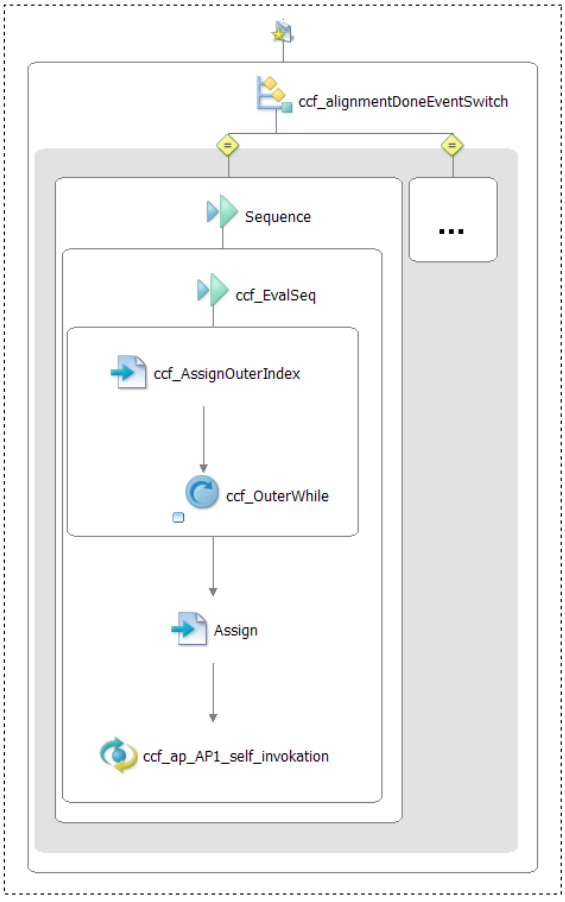


Abbildung 15: *Event-Handler mit je einem Case-Element pro AlignmentPoint*

In einer Schleife werden die Variablen der Nachricht dann nach einer Instanz-Variablen der ersten in der Bedingung auftauchenden Variable durchsucht. Ist diese gefunden, werden in zusätzlichen Schleifen alle weiteren Variablen derselben Prozess-Instanz festgelegt. Danach wird nach der nächsten Variable eines anderen Prozesses gesucht und

für diesen Prozess dann wiederum alle Variablen die zur Instanz gehören. Dieses Verfahren wird wiederholt, so dass alle Variablen berücksichtigt sind. Genaugenommen setzen die geschachtelten Schleifen Index-Variablen auf die jeweiligen Variablen aus der eingehenden Nachricht.

Die Auswertung der Bedingung wird durch eine Umsetzung der `getPartnerVariable`-Methoden in `getVariableData`-Methoden auf die durch die Indices referenzierten Nachrichtenteile generiert (vgl. Abbildung 16). Alle sonstigen Teile des Ausdrucks aus der CCF-Datei können direkt übernommen werden.

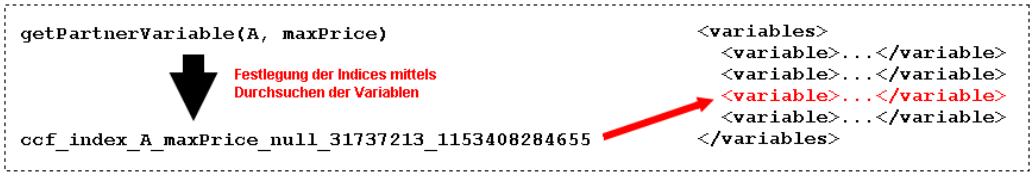


Abbildung 16: *Umwandlung von CCF-Methoden*

Die nunmehr in XPath vorliegende Bedingung muss für alle Instanzenkombinationen einzeln ausgewertet werden. Gibt es vom Prozess, in dem gerade ausgewertet wird, eine Anzahl $n$ von Instanzen und von einem zweiten Prozess, dessen Variable für die Auswertung relevant ist $m$ Instanzen, so ergibt sich die Anzahl der Auswertungen als Produkt der Anzahl der Instanzen. Für jeden weiteren mit einer Variablen beteiligten Prozess multipliziert sich das Ergebnis mit der Anzahl der Prozess-Instanzen. Allgemein ergibt sich die Gesamtanzahl $N$ von Vergleichen also aus der Formel:

$$N = \prod_{i=1}^{n} I(P_i) \tag{1}$$

Dabei ist $n$ die Anzahl der unterschiedlichen Prozesse und $I(P_i)$ die Anzahl der Instanzen eines Prozesses $P_i$.

Das ermittelte Ergebnis wird durch den bereits erwähnten Selbstaufruf an den Reaktionsteil des Prozesses weitergeleitet (siehe 4.5). Für jeden AlignmentPoint muss der Prozess also über eine Operation für den Selbstaufruf verfügen. Diese wird der WSDL-Datei des Prozesses durch den CCF-Transformer hinzugefügt und hat für einen AlignmentPoint mit dem Namen `AP1` z. B. folgende Form:

```
<portType name="Example_PT">
  <operation name="ccf_alignmentDecisionOp_AP1">
    <input message="ccf_protocol_ns:alignmentDecisionMsg" />
  </operation>
</portType>

<binding name="Example_SOAPBinding" type="tns:Example_PT">
  <soap:binding style="rpc" transport="..." />
  <operation name="ccf_alignmentDecisionOp_AP1">
    <soap:operation soapAction="ccf_alignmentDecisionOp_AP1"
```

```
                    style="rpc" />
    <input>
      <soap:body encodingStyle="..."
                 namespace="urn:example" use="encoded" />
    </input>
  </operation>
</binding>
```

Ein Sonderfall tritt ein, wenn der Prozess, in dem ausgewertet wird, gar keinen AlignmentPoint zu dieser Koordination besitzt, er aber an der Reaktion beteiligt ist. In diesem Fall findet die Reaktion direkt im Event-Handler statt und die `Invoke`-Aktivität zum Selbstaufruf wird nicht eingefügt.

## 4.5 Reaktionen

In Abschnitt 3.1.4 wurde bereits erläutert, welche Anweisungen und Anweisungsvarianten implementiert wurden. Hier soll nun die Umsetzung im CCF-Transformer besprochen werden.

Die Bedingungen für die zu erfolgenden Reaktionen, sowie die Reaktionen selbst sind für jeden Prozess unterschiedlich. Dies ergibt sich aus dem Aufbau der CCF-Datei. Maßgeblich ist das `Coordinate`-Element.

Der `Do`-Zweig, der ausgeführt wird, wenn die Bedingung für alle Instanzen gilt, kann zwei Arten von Anweisungen enthalten. Zum einen Anweisungen, welche für alle Prozesse der Koordination gelten (`All` im Schlüsselwort), zum anderen Anweisungen, die für eine aufgelistete Menge von Prozessen ausgeführt werden (Kein `All` im Schlüsselwort). Abhängig von der Zugehörigkeit zur betroffenen Menge, werden also entsprechende Aktivitäten in den BPEL-Prozess eingefügt oder nicht.

Im `Else`-Zweig wird es geringfügig komplizierter. Zur Übersicht sei nochmals auf Abbildung 3 aus Abschnitt 3.1.4 verwiesen. Es wird zusätzlich nach konformen und nicht konformen Instanzen unterschieden. Dies schlägt sich natürlich in der Bedingung für eine Ausführung von Aktivitäten nieder. Ist der Prozess also Teil der Menge der betroffenen Prozesse, gilt es zusätzlich während der Laufzeit zu überprüfen, ob die Bedingung für die jeweilige Instanz zutrifft oder nicht.

All dies fließt in eine Nachricht ein, die benutzt wird, um einen Selbstaufruf durchzuführen, der das wartende `Receive` entblockiert. Die Notwendigkeit eines Selbstaufrufs lässt sich wie schon bei der Synchronisierung von Aufrufen zwischen den Teilnehmern erklären (Vgl. Abschnitt 4.2): Der Protocol Service kann nicht dynamisch die Operation eines Aufrufes festlegen. Mehrere Operationen sind aber notwendig, da auch mehrere `Receive`-Aktivitäten gleichzeitig auf eine Rückmeldung warten können. Deshalb wird der dynamische Teil wie besprochen im Event-Handler ausgeführt, wobei die Operationen der Aufrufe statisch in die `Case`-Elemente einer `Switch`-Aktivität eingefügt werden können.

Die WSDL-Definition der Nachricht zum Selbstaufruf hat folgende Form:

```
<message name="alignmentDecisionMsg">
  <part name="evaluationResult" type="xsd:boolean"/>
  <part name="thisInstanceEvaluationResult"
        type="xsd:boolean"/>
  <part name="participantInstanceId" type="xsd:anyURI"/>
</message>
```

Der Teil `evaluationResult` gibt an, ob die Auswertung für alle Instanzen `true()` ergab oder nicht. Anhand dieses Feldes wird entschieden ob der `Do`- oder der `Else`-Zweig abgearbeitet werden soll. Der zweite Teil gibt entsprechend wieder, ob die Bedingung für diese Instanz erfüllt ist oder nicht und dient zur Unterscheidung von konformen und nicht konformen Prozess-Instanzen. Die `participantInstanceId` dient lediglich zur Korrelation und enthält einen eindeutigen Bezeichner für diese Instanz.

Die Nachricht wird nun zur Unterscheidung der Fälle und Aktivierung der gewünschten Aktivitäten benutzt. Dazu werden zwei geschachtelte `Switch`-Aktivitäten eingefügt, die die jeweiligen Zweige sowie die Aktivitäten zur Ausführung der jeweiligen Reaktion enthalten.

Abbildung 17 zeigt ein Beispiel eines Reaktionsteils. Wie man leicht sehen kann, enthält es eine Sequenz von einer `Quit`-Nachricht und einem `Terminate`, die CCF-Datei hat also möglicherweise ein `Else`-Attribut der Form:

```
<coordinate ... else="terminateAllNonConforming();" />
```

Für eine Reaktion, die einen Fehler werfen soll, wird analog dazu eine `Throw`-Aktivität im jeweiligen Zweig generiert.

Es sei noch einmal darauf hingewiesen, dass der Reaktionsteil nur für Prozesse, die für eine Koordination selbst einen AlignmentPoint besitzen, direkt nach der Sequenz des Abgleichs eingefügt wird. Bei Prozessen, die lediglich an der Reaktion beteiligt sind, wird dieser Block im Event-Handler eingefügt.

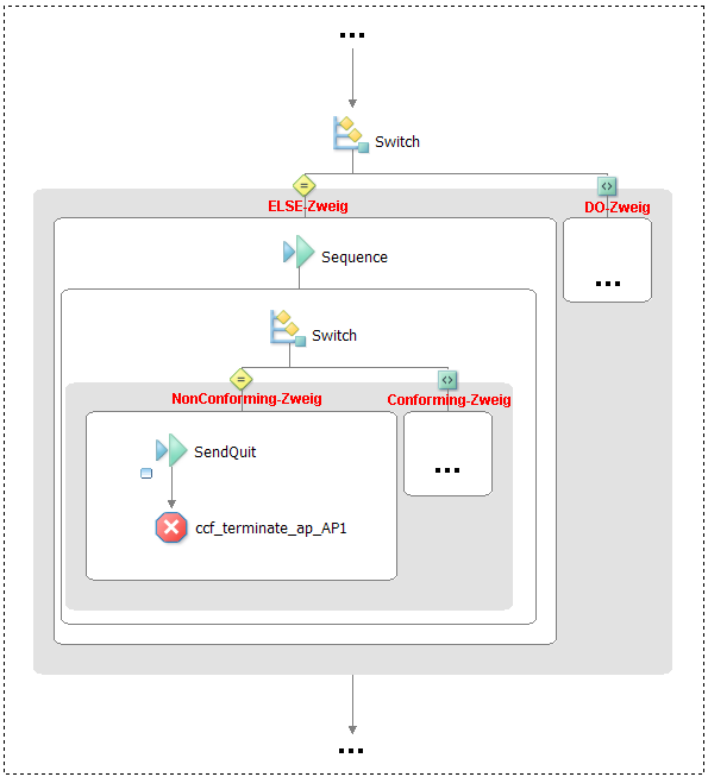


Abbildung 17: *Reaktionsteil in einem modifizierten BPEL-Prozess*

## 4.6 Ende der Koordination

Das Ende der Koordinationen wurde bereits in Abschnitt 3.2.4 im Kontext der Ausführung besprochen. Zusammengefasst wird die gesamte Koordination und damit auch der Protocol Service dann beendet, wenn von allen registrierten Teilnehmern eine `Quit`-Nachricht beim Protocol Service eingegangen ist. Diese Nachrichten sollen direkt vor der Beendigung einer Prozess-Instanz geschickt werden.

Daraus resultiert, dass Sequenzen zum Verschicken der Nachricht an folgenden Stellen durch den CCF-Transformer eingefügt werden müssen:

- Als eine neue Aktivität, die nach einer normalen Ausführung des Ursprungsprozesses aktiviert wird.
- Vor jedem expliziten `terminate`, also vor jeder `Terminate`-Aktivität im Originalprozess.
- Vor jeder `Terminate`-Aktivität, die als Reaktion auf die Auswertung eingefügt wurde.

Zur Umsetzung des ersten Punktes, wird wie wir bereits in Abbildung 8 gesehen haben, der generierten Gesamtablauf-Sequenz ein Teil für die Beendigung hinzugefügt. Um alle ursprünglich vorhandenen `Terminate`-Aktivitäten mit einer Beendigungs-Sequenz

auszustatten, durchsucht der CCF-Transformator die BPEL-Dateien und fügt an den entsprechenden Stellen die notwendigen Aktivitäten ein. Die Aktivitäten aus dem letzten Punkt wurden bereits bei der Generierung des Reaktionsteils berücksichtigt.

Der jeweils hinzugefügte Code ist sehr einfach, wie wir in Abbildung 18 in der grafischen Darstellung sehen können.

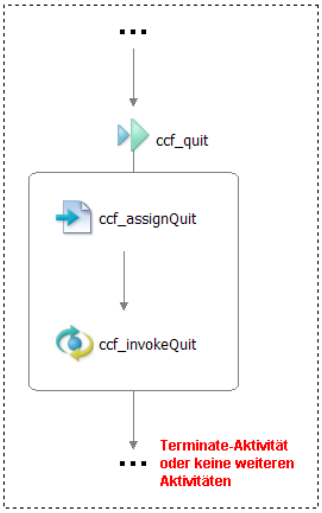


Abbildung 18: *Beendigungs-Sequenz*

## 4.7 Correlation

An vielen Stellen in den teilnehmenden Prozessen ist Korrelation im Sinne von BPEL notwendig (Siehe [ACD+03]). Grundsätzlich versteht man darunter die korrekte Zuordnung von Nachrichten zu Prozess-Instanzen. Die Korrelation der Nachrichten des Ursprungsprozesses ist Aufgabe des Benutzers. Da durch die Transformation aber einige neue Nachrichtenaustausche hinzukommen, müssen auch die bestehenden `CorrelationSets` erweitert, bzw. neue hinzugefügt werden.

Änderungen treten dabei an folgenden Stellen auf:

- Erweiterte Operationen für die Startaktivitäten
- Rückmeldungen zur Synchronisierung von koordinationsinternen Aufrufen
- Selbstaufruf zur Entblockierung bei koordinationsinternen Aufrufen
- Rückmeldungen des Protocol Service
- Selbstaufruf zur Entblockierung bei Erreichen eines AlignmentPoints

Die komplizierteste Änderung betrifft den erstgenannten Punkt, die Korrelation der erweiterten Startaktivitäten. Können mehrere Instanzen des Prozesses parallel vorhanden sein, so muss der Benutzer bereits im Ursprungsprozess eine Korrelation der

ursprünglichen Nachrichten vorgesehen haben. Der Transformator muss dann folgende Schritte durchführen:

- Suchen des für die ursprüngliche Operation verwendeten `CorrelationSets` in der BPEL-Quelldatei
- Suchen der zugehörigen `Properties` in den zugehörigen WSDL-Dateien
- Suchen des `PropertyAlias` zur Ursprungsnachricht
- Generierung und Hinzufügen einer Kopie des `PeropertyAlias` mit geringfügiger Anpassung für die erweiterte Nachricht

Nach der Erweiterung könnte eine Beispiel-`Property` wie folgt aussehen:

```
<bpws:property name="startCorrel" type="xsd:anyURI" />
<bpws:propertyAlias messageType="tns:startMsg"
                    part="procId"
                    propertyName="tns:startCorrel" />
<bpws:propertyAlias messageType="tns:startMsg_ccf_context"
                    part="procId"
                    propertyName="tns:startCorrel" />
```

Der zweite `PropertyAlias` ist dabei der durch den Transformator hinzugefügte.

Alle anderen durch den CCF-Transformer zu generierenden Korrelationen finden über einen eindeutigen Bezeichner für die Prozessinstanz statt. Der Bezeichner setzt sich dabei aus einem festen Präfix, sowie der durch die proprietäre ActiveBPEL-Funktion `getProcessId()` ermittelte Instanzen-Identifikation zusammen und hat für die Instanz mit der Id `24` beispielsweise folgende Form:

```
http://vortexport.net/instanceId/24
```

Zu jeder Nachricht, die einer Korrelation bedarf, wird dieser Bezeichner in einem eigenen Teil der Nachricht untergebracht wie z. B. als `participantInstanceId` in der Rückmeldung für den Protocol Service zu einem AlignmentPoint:

```
<message name="alignmentDoneMsg">
  <part name="contextId" type="xsd:string"/>
  <part name="alignmentPoint" type="xsd:string"/>
  <part name="variables" type="tns:variablesType"/>
  <part name="participantInstanceId" type="xsd:anyURI"/>
</message>
```

Weitere Beispiele dazu finden sich in Abschnitt 3.2.

Für alle diese Nachrichten wird jedem teilnehmenden BPEL-Prozess vom Transformator ein neues `CorrelationSet` mit der zugehörigen `Property` hinzugefügt:

```
<correlationSets>
  <correlationSet ... />
  ...
  <correlationSet name="ccf_correlationSet"
                  properties="ccf_correlation_ns:ccf_instanceId"/>
</correlationSets>
```

Für jede zu korrelierende Nachricht wird dann in einer zentralen WSDL-Beschreibung noch ein `PropertyAlias` eingefügt. In der Übersicht sind das die folgenden:

```
<bpws:property name="ccf_instanceId" type="xsd:anyURI"/>
<bpws:propertyAlias
  messageType="ccf_activation_ns:CreateContextMsg"
  part="InstanceIdentifier"
  propertyName="tns:ccf_instanceId"/>
<bpws:propertyAlias
  messageType="ccf_activation_ns:CreateContextReplyMsg"
  part="CreateCoordinationContextResponse"
  propertyName="tns:ccf_instanceId"
  query="/CreateCoordinationContextResponse/
         InstanceIdentifier"/>
<bpws:propertyAlias
  messageType="ccf_protocol_ns:alignMsg"
  part="participantInstanceId"
  propertyName="tns:ccf_instanceId"/>
<bpws:propertyAlias
  messageType="ccf_protocol_ns:alignmentDecisionMsg"
  part="participantInstanceId"
  propertyName="tns:ccf_instanceId"/>
<bpws:propertyAlias
  messageType="ccf_protocol_ns:alignmentDoneMsg"
  part="participantInstanceId"
  propertyName="tns:ccf_instanceId"/>
<bpws:propertyAlias
  messageType="ccf_protocol_ns:newCoordinationMsg"
  part="participantInstanceId"
  propertyName="tns:ccf_instanceId"/>
<bpws:propertyAlias
  messageType="ccf_protocol_ns:newCoordinationReplyMsg"
  part="participantInstanceId"
  propertyName="tns:ccf_instanceId"/>
<bpws:propertyAlias
  messageType="ccf_protocol_ns:quitMsg"
  part="participantInstanceId"
  propertyName="tns:ccf_instanceId"/>
```

```
<bpws:propertyAlias
   messageType="ccf_protocol_ns:regDoneMsg"
   part="participantInstanceId"
   propertyName="tns:ccf_instanceId"/>
<bpws:propertyAlias
   messageType="ccf_registration_ns:RegisterReplyMsg"
   part="RegisterReply"
   propertyName="tns:ccf_instanceId"
   query="/RegisterReply/ParticipantInstanceIdentifier"/>
<bpws:propertyAlias
   messageType="ccf_registration_ns:RegisterRequestMsg"
   part="ParticipantProtocolService"
   propertyName="tns:ccf_instanceId"
   query="/ParticipantProtocolService/ReferenceProperties/
          ParticipantInstanceIdentifier"/>
```

Alle Operationen, die eine Korrelation benötigen, sind so durch die zentrale WSDL-Datei sowie die Änderungen an der zum Prozess gehörenden WSDL-Beschreibung abgedeckt und können auf das jeweilige `CorrelationSet` verweisen.

# 5 Weitere implementierte Services

Zur Umsetzung des Ablaufs der Koordination war es notwendig, neben den Modifikationen an den Ursprungsprozessen, weitere Services zu implementieren:

- **Den Protocol Service:** Dieser ist zentraler Bestandteil jeder Koordination (Siehe Abschnitt 2.1).
- **Die Zwischenservices für die Durchführung von Activation und Registration:** Der Activation Service und Registration Service des Koordinators von Thorsten Vetter [Vet06] waren für die Zusammenarbeit mit BPEL-Prozessen leider nicht direkt einsetzbar.
- **Den Service zum Zugriff auf die Datenbank der Registrierung:** Der Protocol Service benötigt die Liste der registrierten Teilnehmer um Entscheidungen zu treffen und Nachrichten an Teilnehmer schicken zu können.

Zur Umsetzung der Services kamen die Sprachen Java sowie BPEL zum Einsatz. Jeder der Services soll in den Abschnitten dieses Kapitels genauer besprochen werden.

## 5.1 Protocol Service

Jede Koordination nach WS-Coordination [CCF+05] benötigt einen Protocol Service, der die eigentliche Kommunikation mit den Teilnehmern nach der Activation und Registration übernimmt. Der für das CCF-Protokoll benötigte Service, der im Rahmen dieser Arbeit entwickelt wurde, wurde vollständig in BPEL implementiert.

Aus dieser Tatsache resultieren zwei unmittelbare Probleme. Zum einen ist ein spezifikationskonformer BPEL-Prozess nicht in der Lage, Datenstrukturen zu erweitern, was aber dringend notwendig ist um z. B. eingehende Informationen zu sammeln. Abhilfe schafft hier eine Option der verwendeten ActiveBPEL-Enginge, die es ermöglicht, den nach der Spezifikation zu werfenden Fehler bei der Zuweisung von Daten an ein nicht vorhandenes Element zu unterdrücken und stattdessen das neue Element in diese Struktur einzufügen. In Anhang B wird die Aktivierung dieser Funktion erläutert.

Das zweite Problem besteht im Zugriff auf die Registrierungsdaten der Teilnehmer. Diese werden vom Registration Service in einer Datenbank abgelegt. Leider haben wir jedoch aus BPEL heraus keine Möglichkeit auf Datenbanken zuzugreifen. Deshalb wurde ein zusätzlicher Service in Java implementiert, der diese Aufgabe übernimmt und die Daten an den Protocol Service liefert (Siehe 5.3).

Kommen wir nun zum Aufbau des Protocol Service. Grundsätzlich ist von den Teilnehmern eine von drei möglichen Aufrufen zu erwarten, die sich im `PortType` für den Protocol Service widerspiegeln:

```
<portType name="ccfProcessPT">
  <operation name="newCoordination">
    <input message="tns:newCoordinationMsg"/>
    <output message="tns:newCoordinationReplyMsg"/>
  </operation>
  <operation name="align">
    <input message="tns:alignMsg"/>
  </operation>
  <operation name="quit">
    <input message="tns:quitMsg"/>
  </operation>
</portType>
```

Die Operation `newCoordination` wird einmalig nach der Activation aufgerufen und erzeugt eine neue Instanz des Protocol Service, welche dann für alle Koordinationen aus der jeweiligen CCF-Datei zuständig ist. Die `Align`-Operation dient zum Empfang von Abgleich-Nachrichten der Teilnehmer, die einen AlignmentPoint erreicht haben. Die dritte Möglichkeit besteht in der Ausführung einer `Quit`-Operation, welche von jedem Teilnehmer, der im Begriff ist beendet zu werden, aufgerufen wird.

Im Überblick sieht der Protocol Service wie in Abbildung 19 dargestellt aus.

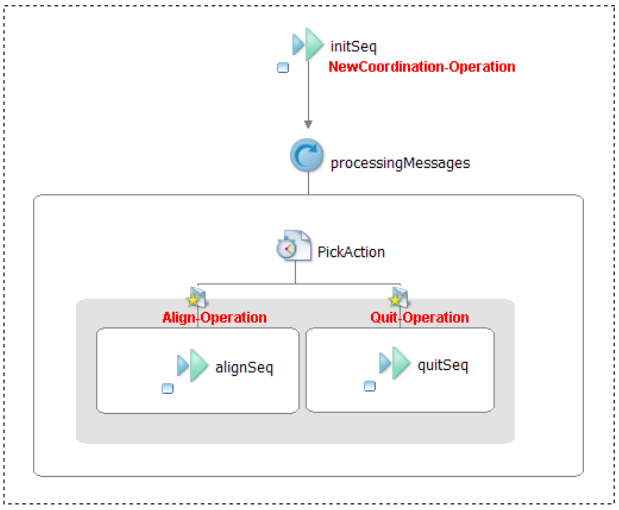


Abbildung 19: *Protocol Service im Überblick*

Der Prozess beginnt mit einer Sequenz (`initSeq`) zur Initialisierung des Services, welcher wir uns zuerst widmen werden. Dann folgt in einer `While`-Schleife geschachtelt eine `Pick`-Aktivität, welche die `onMessage`-Elemente zu den beiden verbleibenden Operationen enthalten.

Der Initialisierungsteil ist sehr einfach geartet und wird in Abbildung 20 dargestellt. Durch den Empfang der `newCoordination`-Nachricht durch die `Receive`-Startaktivität wird eine neue Instanz des Prozesses erzeugt. Alle Zähler für AlignmentPoints und Teil-

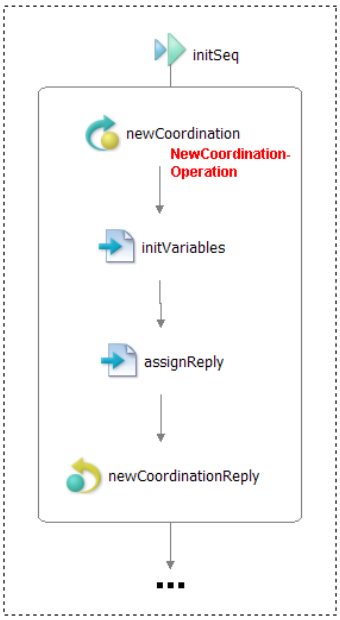


Abbildung 20: *Initialisierungsteil des ProtocolService*

nehmer werden daraufhin initialisiert, die Rückmeldung wird vorbereitet und schließlich durch die `Reply`-Aktivität abgeschickt.

Ab diesem Zeitpunkt ist der Service zum Empfang von `Align`- sowie `Quit`-Nachrichten bereit. Trifft eine `Align`-Nachricht ein, so wird der in Abbildung 21 dargestellte Zweig aktiviert. Einige Teile des Ablaufs sind aus Platzgründen nur durch ihr Container-Element dargestellt, enthalten in Wirklichkeit jedoch umfangreichen Code. Mittels einer `While`-Schleife wird überprüft, ob in der Liste der bereits empfangenen AlignmentPoints der soeben neu empfangene enthalten ist. Ist dies nicht der Fall, wird der neue AlignmentPoint mitsamt den übermittelten Variablen hinzugefügt (linker Zweig). Ist er aber bereits vorhanden, wird die bereits gespeicherte Struktur lediglich durch die übermittelten Variablen innerhalb einer `While`-Schleife erweitert (rechter Zweig).

Ist die Erweiterung der AlignmentPoints bzw. Variablen erfolgt, wird überprüft, ob für diesen AlignmentPoint bereits alle Variablen vorliegen. Wenn nicht, ist der Durchlauf der `While`-Schleife beendet und es wird auf die nächste Nachricht gewartet. Sollten aber bereits alle Variablen zum AlignmentPoint verfügbar sein, wird die Sequenz aus Abbildung 22 ausgeführt.

Durch Aufruf des Service für den Datenbankzugriff (siehe 5.3), werden die registrierten Teilnehmer ermittelt, die dann als XML-Struktur in der Antwort zur Verfügung stehen. Die Liste der Teilnehmer wird nun mittels einer `While`-Schleife durchlaufen und für jeden Teilnehmer, der am aktuellen AlignmentPoint beteiligt ist, wird eine `alignmentDone`-Nachricht an selbigen gesendet. Die Überprüfung, welche Teilnehmer relevant sind, erfolgt auf Basis der Protokolle, für die sich der jeweilige Prozess registriert hat. Nach der Rückmeldung ist dieser Durchlauf der `While`-Schleife beendet.

Die letzte mögliche Operation ist `Quit`. Auf eine grafische Darstellung des Ablaufs beim Eintreffen einer solchen Nachricht wird an dieser Stelle verzichtet, da die Abläufe

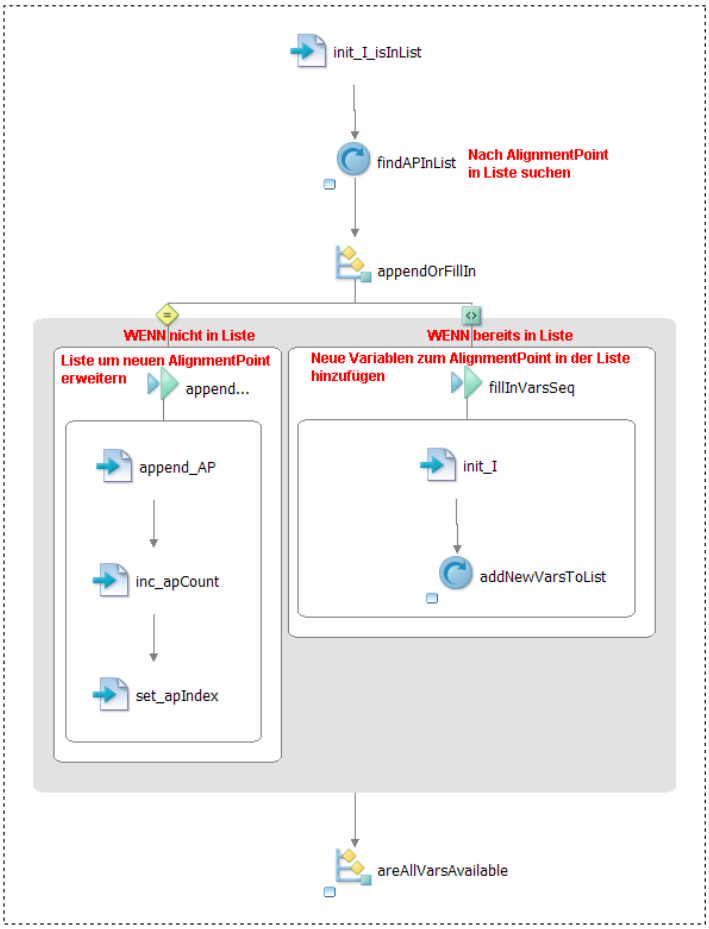


Abbildung 21: *Align-Operation des Protocol Service*

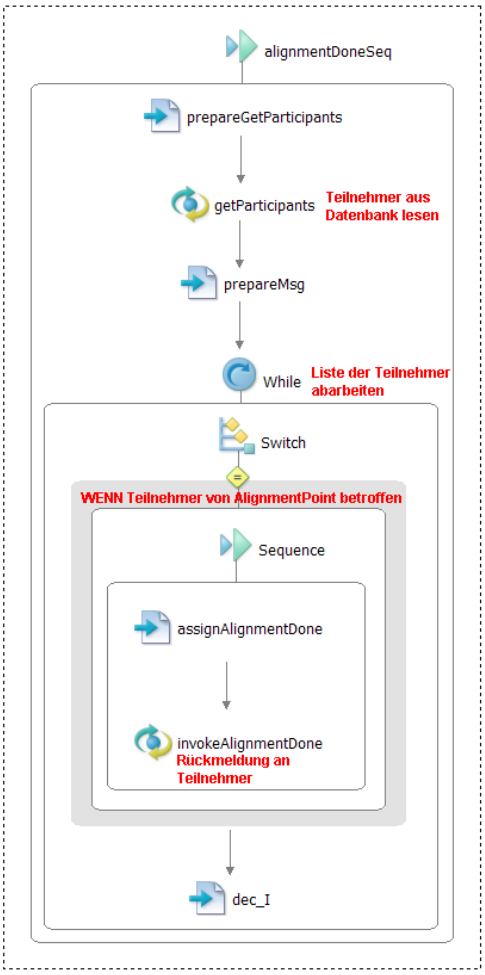


Abbildung 22: *Rückmeldung an betroffene Teilnehmer*

denen beim Alignment sehr ähneln. Der Prozess sammelt in einer Variablen alle Teilnehmer, von denen bereits eine `Quit`-Nachricht eingegangen ist. Durch das Abrufen der Teilnehmer aus der Datenbank lässt sich feststellen, ob von allen Teilnehmern bereits eine solche Nachricht vorliegt. Überprüft werden dabei die Registrierungen für das CCF-Protokoll, für das sich jede Instanz registrieren muss. Sind von allen Prozess-Instanzen `Quit`-Nachrichten eingegangen, ist die Koordination beendet, der Protocol Service verlässt die äußere `While`-Schleife und wird ebenfalls beendet.

## 5.2 Zwischenservices für Activation und Registration

In der vorliegenden Implementierung des Koordinators von Thorsten Vetter [Vet06] sind bereits ein Activation Service sowie ein Registration Service enthalten. Leider wurde das Nachrichtenformat aber so gewählt, dass einige Informationen im Header der SOAP-Nachricht ausgetauscht werden. Leider auch insbesondere solche, welche der Korrelation dienen könnten.

Da BPEL 1.1 keine Korrelation mit Header-Elementen zulässt (vgl. [ADO+05]), wurden für die Registration und Activation zwei neue Services eingeführt, welche alle Informationen im Body der Nachricht austauschen, diese also ohne weiteres in BPEL zur Verfügung stehen und darüberhinaus zur Korrelation dienen können.

Abbildung 23: *Zwischenservices für Activation und Registration*

Die neuen Services wurden in Java implementiert, wobei das Grundgerüst mittels des Werkzeugs WSDL2Java [AXI] generiert wurde. Die eigentlichen Arbeiten für die Activation bzw. Registration übernehmen weiterhin die ursprünglichen Services, die neuen mit vereinfachter Nachrichtenstruktur dienen vielmehr als Vermittler zwischen dem Aufrufer und dem Registrator bzw. Activator.

Abbildung 23 stellt die Vorgehensweise grafisch dar. Der jeweilige Service empfängt eine Nachricht, verschiebt die notwendigen Elemente in den Header und ruft dann den ursprünglichen Service auf. Bei der Antwort wird der umgekehrte Weg gewählt und die Header Elemente werden in den Body der Nachricht zurück an den Aufrufer verlagert.

Das Resultat sind ein Activation Service sowie ein Registration Service, die beide von BPEL-Prozessen aus verwendet werden können und darüberhinaus auch die Anforderungen für die dringend notwendige Korrelation mit den entsprechenden Prozess-Instanzen erfüllen.

## 5.3 Service für den Datenbankzugriff

Wie bereits zu Anfang dieses Kapitels beschrieben, bietet BPEL keine Möglichkeit direkt auf Daten aus einer Datenbank zuzugreifen. Deshalb wurde ein weiterer Service in Java erstellt, der den Datenbankzugriff übernimmt und die benötigten Daten in einer Nachricht zurückliefert.

Die Einstellungen, um welche Datenbank es sich handelt, welcher Datenbanktreiber verwendet werden soll, sowie Benutzername und Passwort zum Zugriff werden in der Implementierung von Thorsten Vetter in Property-Dateien gespeichert, die sich in einem festen Verzeichnis auf dem Laufwerk `C` befinden (Siehe [Vet06]). Um Verzeichnisse und Dateien nicht wiederholt hart codieren zu müssen, wird statt einem direkten Zugriff auf die Dateien die `Repository`-Klasse von Thorsten Vetter benutzt, welche diese Aufgabe übernimmt.

Die für die Koordination benötigten Daten sind auf zwei Tabellen verteilt:

- `Participant`: Diese Tabelle enthält für jeden Teilnehmer die Adresse, den Instanzenbezeichner sowie das Protokoll für welches die Registrierung gilt. Zusätzlich gibt es noch ein Feld für die Kontext-Identifikation sowie eine Coordinator-Identifikation welche hier nur für die Verknüpfung der Tabellen notwendig ist.
- `Coordinator`: In dieser Tabelle sind für unsere Zwecke nur die Kontext-Identifikation sowie wiederum das Feld `coordinator_id` wichtig.

Der Zugriff auf die Daten erfolgt mittels SQL-Anfragen (Siehe [War06]). Im Speziellen ist das für den Datenbankzugriffs-Service die Query:

```
SELECT participant.* FROM participant INNER JOIN coordinator
  ON participant.coordinator_id = coordinator.coordinator_id
  WHERE context_id='...'
```

Dabei wird für die hier eingefügten drei Punkte der eindeutige Bezeichner für den Kontext eingesetzt, für welchen die Teilnehmer zu ermitteln sind.

Die Rückmeldung des Service enthält dann die Tabelleninhalte der Datenbankabfrage als XML-Struktur mit der folgenden Form (als Pseudoschema):

```
<participant>*
  <participantId>...</participantId>
  <coordinatorId>...</coordinatorId>
  <participantService>...</participantService>
  <protocolId>...</protocolId>
  <participantInstanceId>...</participantInstanceId>
  <state>...</state>
<participant>
```

Wichtig für die Verwendung im Rahmen des Protocol Service sind allerdings nur die Felder `protocolId`, `participantInstanceId` sowie `participantService` in welchem die Adresse des Teilnehmers gespeichert wird.

# 6 Erweiterungsmöglichkeiten

Der beschriebene Ansatz zur Koordination, sowie die Implementierung bieten an vielen Stellen Potential für Erweiterungen, welche den Umfang der Arbeit gesprengt hätten. Einige der Möglichkeiten sollen in diesem Kapitel betrachtet werden. Im Einzelnen handelt es sich dabei um Erweiterungen in folgenden Bereichen:

- **Fehlerverhalten:** In der Implementierung ist kein Verhalten festgelegt, falls z. B. ein Teilnehmer ausfällt.
- **Privacy:** Beispielsweise werden Variablen der Geschäftsprozesse im Klartext über das Netzwerk übertragen.
- **AlignmentPoints:** Weitere Möglichkeiten, den Zeitpunkt eines Abgleichs festzulegen sind denkbar.
- **Reaktionen:** Neben der Terminierung und dem Werfen eines Fehlers, sind weitere Anweisungen als Reaktion auf eine Auswertung sinnvoll. Einige sollen hier besprochen werden.

Die nun folgenden Abschnitte gehen näher auf das Erweiterungspotential der vier genannten Bereiche ein.

## 6.1 Verhalten bei Fehlern

In einem verteilten System gibt es eine Vielzahl möglicher Fehlerquellen. Oftmals können Teile des Systems ausfallen, während andere fehlerfrei weiterarbeiten. Man spricht dann von einem partiellen Fehler (Vgl. [Tan03]). Das Spektrum möglichen Fehlverhaltens lässt sich in folgende Kategorien unterteilen (Siehe [RK03] sowie [LSP82]):

- **Absturzfehler (Crash Failure):** Ein Teilnehmer stürzt ab, aber arbeitet bis zu seinem Absturz korrekt.
- **Auslassungsfehler (Omission Failure):** Eine Nachricht wird nicht gesendet bzw. empfangen oder geht verloren.
- **Timing-Fehler (Timing Failure):** Eine Antwort erfolgt nicht innerhalb einer festgelegten Zeitspanne.
- **Antwortfehler (Response Failure):** Das Format oder der Inhalt einer Antwort ist nicht korrekt.
- **Byzantinische Fehler (Arbitrary Failure):** Zu beliebigen Zeitpunkten können beliebige Verfälschungen in Nachrichten oder Kontrollfluss vorkommen.

Für den Umgang mit den aufgelisteten Fehlerarten wurde eine Vielzahl von Algorithmen und Vorgehensweisen unterschiedlichen Schwierigkeitsgrades entwickelt. In der vorliegenden Implementierung wurde keine Rücksicht auf möglicherweise auftretende Fehler genommen. An dieser Stelle soll ein Teilansatz vorgestellt werden, der Absturzfehler, Auslassungsfehler sowie Timing-Fehler berücksichtigt und sich in die bestehende Implementierung einbetten lässt.

Alle drei berücksichtigten Fehlerarten lassen sich in dem vorgestellten Gefüge von Teilnehmern und Koordinator durch das Ausbleiben einer Nachricht erkennen. Dabei sind zwei wesentliche Fälle zu unterscheiden:

- **Das Ausbleiben einer Antwort-Nachricht auf der Transportschicht (Vgl. [Zim80]):** Dieser Fall tritt auf, wenn das Ziel nicht verfügbar ist, die Antwort nicht empfangen wurde, keine Antwort gesendet wurde oder die Antwort verloren ging.
- **Das Ausbleiben einer Nachricht auf der Anwendungsschicht:** Dieser Fehler kann dann auftreten, wenn beispielsweise der Protocol Service vor dem Senden der Rückmeldungen abstürzt.

Tritt der erste Fall ein, so wird in der jeweiligen BPEL-Engine ein entsprechender Fehler geworfen. Dieser kann abgefangen werden. Bei Auftreten des zweiten Falls tritt jedoch kein Fehler auf. Vielmehr äußert sich das Fehlverhalten dadurch, dass z. B. endlos auf eine Rückmeldung vom Protocol Service gewartet wird.

Für eine Behandlung dieser Situation wäre es sinnvoll eine Zeitspanne festzulegen, innerhalb der eine Antwort vom Protocol Service erfolgen muss. Da je nach Art der Koordination früher oder später mit einer Antwort zu rechnen ist, sollte dieser Wert durch den Benutzer in der CCF-Datei festzulegen sein.

Denkbar wäre z. B. die Erweiterung des `Coordinations`-Elements um ein `Timeout`-Attribut. Für eine Zeitspanne von 2 Sekunden ergibt sich dann folende Zeile:

```
<coordinations activationService="..." timeout="2000ms">
```

Alternativ könnte das `Timeout`-Attribut auch in den jeweiligen `Partner`-Elementen eingebettet werden. So könnte für jeden Partner eine eigene Zeitspanne festgelegt werden. Die Kombination beider Möglichkeiten böte dann maximale Flexibilität in der Angabe.

Bei der Transformation der BPEL-Prozesse kann das gewünschte Verhalten durch den in Abbildung 24 dargestellten Einsatz einer `Pick`-Aktivität mit `onAlarm`-Element anstatt eines einfachen `Receives` hergestellt werden. Im `onAlarm`-Handler könnte dann wiederum ein Fehler geworfen werden.

Innerhalb des Protocol Service gilt es dann auch, die aufgetretenen Fehler abzufangen und eine entsprechende Reaktion auszuführen. Nicht berücksichtigt wäre dennoch der Fall, dass alle Teilnehmer nach der Registrierung ausfallen, etwa weil sie alle auf derselben Maschine ausgeführt werden und diese abstürzt. Der Protocol Service erhält dann

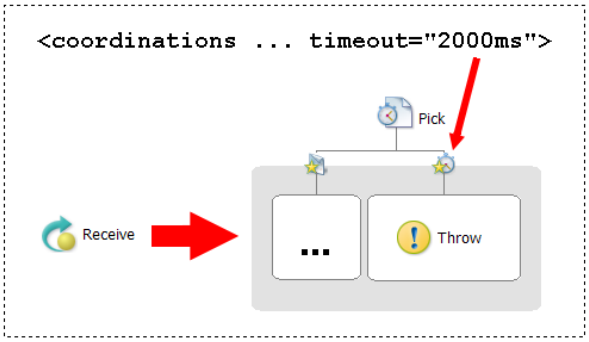


Abbildung 24: *Einsatz von Pick und onAlarm zur Unterstützung von Timeouts*

weder `Align`-Nachrichten noch Benachrichtigungen zur Beendigung. Ein möglicher Ansatz zur Lösung dieses Problems wäre das periodische Senden von Verfügbarkeits-Überprüfungsnachrichten an die Teilnehmer (Vgl. [Dab06]). Diese könnten in den Prozessen durch einen Event-Handler abgearbeitet werden.

Nun gilt es noch, die Art der Reaktion bei einem Fehler festzulegen. Die einfachste Reaktion wäre sicherlich eine Terminierung der Prozesse, die einen Fehler feststellen. Da BPEL jedoch das weit hochwertigere Konzept der Kompensation kennt, könnte auch der jeweilige `CompensationHandler`, sofern vorhanden, aufgerufen werden. Sehr flexibel wäre eine Lösung durch die Erweiterung der `Partner`-Elemente des CCF-Formats. Zusätzliche Attribute könnten es dem Benutzer gestatten genaue Angaben über das Verhalten bei einem Fehler zu jedem einzelnen Prozess festzulegen. Auf die Vielfalt der Möglichkeiten an dieser Stelle soll nicht weiter eingegangen werden.

## 6.2 Privacy

Beim Austausch von XML-Nachrichten ist zu bedenken, dass die übertragenen Daten oftmals im Klartext gesendet werden und möglicherweise von Dritten mitgelesen werden können. Enthalten die Nachrichten etwa Firmengeheimnisse, so müssen zusätzliche Vorkehrungen zum Schutz der Information getroffen werden. In diesem Zusammenhang spricht man auch von der Vertraulichkeit der Daten (Vgl. [Man03]). Im Rahmen dieser Arbeit ist dabei zu unterscheiden zwischen:

- dem Schutz der Daten gegenüber Dritten
- dem Schutz der Daten gegenüber anderen Teilnehmern

Hinzu kommt, dass die gesendeten Nachrichten möglicherweise kompromitiert, d. h. bewusst verfälscht werden. Auch wenn diese Tatsache nicht Gegenstand dieser Arbeit sein soll, so verlangen alle soeben genannten Probleme nach einem fundamentalen Konzept: Dem sicheren Nachrichtenaustausch.

Zur Realisierung dieser grundlegenden Komponente existiert eine Vielzahl von Standards wie z. B. WS-Security [NKMHB06], WS-Trust [ABB+05] oder Secure Socket

Layer [FKK96], welche die unterschiedlichen Aspekte abdecken. Je nach verwendeter BPEL-Engine können unterschiedliche Standards für die Sicherung der Übertragung eingesetzt werden. An dieser Stelle soll nicht weiter auf die Herstellung von Sicherheit und Vertraulichkeit einer Verbindung eingegangen werden, vielmehr wird angenommen, dass es solche Mechanismen gibt und diese eingesetzt werden. Weiterführende Informationen finden sich in der Literatur zum Thema (Siehe [Sch05] und [Cor06]).

Als eine sichere Verbindung wird im Folgenden ein Nachrichtenaustausch verstanden, der sowohl die Integrität der Nachrichten sowie auch deren Vertraulichkeit durch den Einsatz eines entsprechenden kryptographischen Verfahrens berücksichtigt.

Stehen für die Kommunikation solche sicheren Verbindungen zur Verfügung, so wurde das erstgenannte Problem, der Schutz der Daten gegenüber Dritten, gelöst.

Die Lösung des zweiten Punktes gestaltet sich schwieriger, da das vorgestellte Verfahren zur Koordination die für die Auswertung der Bedingungen notwendigen Variablen an alle betroffenen Teilnehmer verteilt. Angenommen die Teilnehmer werden jedoch auf den Servern unterschiedlicher Unternehmen ausgeführt und der genaue Wert der Variablen soll den anderen beteiligten Unternehmen nicht mitgeteilt werden, so ist das implementierte Verfahren nicht geeignet.

Da für eine Auswertung der Bedingungen und die Herstellung einen Konsenses über das Ergebnis der Auswertung die enthaltenen Variablen zur Verfügung stehen müssen, gibt es in jedem Falle einen Service, welcher über die Gesamtinformation verfügt. Diese kann für den Zweck der Vertraulichkeit aber ausgelagert werden und autonom agieren. Voraussetzung dafür ist, dass alle teilnehmenden Parteien dieser Instanz trauen. Da auch der Protocol Service über die Gesamtinformation verfügt, wäre es naheliegend, die Entscheidungsfindung in einen neuen komplexeren Protocol Service zu verlagern, welcher durch eine unabhängige Institution betrieben wird. Abbildung 25 fasst diesen Zusammenhang noch einmal zusammen.

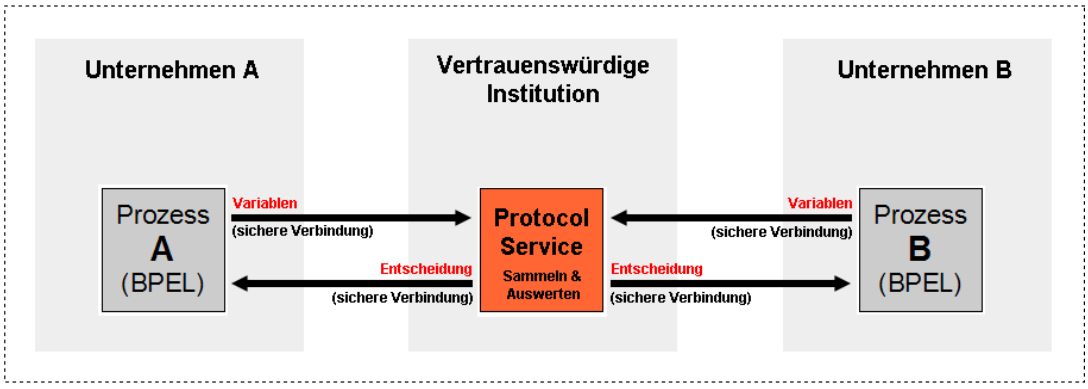


Abbildung 25: *Ausgelagerter Protocol Service mit integrierter Entscheidungsfindung*

Wird der Protocol Service ausgelagert, so könnte jeder Teilnehmer über eine eigene gesicherte Verbindung zu diesem verfügen. Über diese können eigene Variablen übertragen und Rückmeldungen empfangen werden. Den Voraussetzungen für die Vertraulichkeit von Daten innerhalb der Menge der Teilnehmer wäre somit ebenfalls genüge getan.

Eine Implementierung des eben vorgestellten Ansatzes ist sicherlich mit einem erheblichen Aufwand verbunden und unter Umständen nicht ohne weiteres mit der verwendeten ActiveBPEL-Engine durchführbar, da diese z. B. keine Unterstützung für WS-Security bietet.

## 6.3 AlignmentPoints

Für die Festlegung des Zeitpunktes zu dem ein Abgleich erfolgen soll wurden zwei Möglichkeiten erläutert, die im Rahmen dieser Arbeit implementiert wurden: Der Abgleich direkt vor Ausführung einer Aktivität und der Abgleich vor mehreren parallel auftretenden Aktivitäten (Vgl. Abschnitt 3.1.2). Im Folgenden sollen nun weitere sinnvolle Angaben vorgestellt werden, die nicht mehr in die Implementierung eingeflossen sind. Im Einzelnen sind das:

- `onVariableChange()`
- `onEachIncomingMessage()`
- `onFault()`
- `onEvent()`
- `onStartup()`
- `onEnd()`

Die Funktion `onVariableChange()` soll angeben, dass bei jeder möglichen Veränderung der als Parameter übergebenen Variablen ein Abgleich stattfindet. Im Prozess müssten dazu alle `Receives`, alle `onMessage`-Aktivitäten, alle Two-Way-`Invokes`, sowie `Assigns` durch den CCF-Transformer einen Zusatz für den Abgleich erhalten. Da Abgleiche dieser Art in den verschiedenen Teilnehmern in der Regel in unterschiedlicher Anzahl vorkommen werden, ist auch das Konzept der Abgleichung zu überarbeiten. Eine Möglichkeit wäre es einen solchen Abgleich immer auf Basis der zuletzt bekannten Werte durchzuführen. Durch Verwendung dieser Funktion können z. B. Werte der Teilnehmer fortlaufend überwacht werden um bei einem Verlassen des zulässigen Bereichs Aktionen auszulösen.

Da ein Entwickler die Vorgänge im eigenen Prozess kennt, die Daten anderer Prozesse u. U. jedoch nicht in seinem Einflussbereich liegen, könnte durch die Verwendung von `onEachIncomingMessage()` bei jeder eingehenden Nachricht eine Abstimmung der empfangenen Daten erfolgen. Die Transformation erfolgt wie schon zuvor durch das Modifizieren aller Two-Way-`Invokes`, `onMessage`-Aktivitäten sowie `Receives`. Auch hierfür wäre wiederum eine Erweiterung des Konzepts für das Alignment notwendig.

Die Angabe `onFault()` mit dem Bezeichner eines Fehlers als Parameter könnte einen Abgleich bei Auftreten fehlerhaften Verhaltens auslösen. Der CCF-Transformer würde

dazu den jeweiligen Prozessen neue Fault-Handler mit Aktivitäten für den Abgleich hinzufügen. Ein solcher Abgleich wäre sogar ohne eine Erweiterung des Modells möglich.

Für den Abgleich bei Auftreten eines Events, könnte eine Funktion `onEvent()` eingeführt werden. Dies erscheint sinnvoll, da ein Event nur durch bestimmte Nachrichten ausgelöst wird, also die Art der eingehenden Nachricht gegenüber zu `onEachIncoming-Message()` ein Kriterium ist. Kann man sicherstellen, dass ein Event nur ein einziges Mal auftritt, könnte die Funktion ohne Änderungen am Modell implementiert werden. Kann diese Einschränkung nicht gemacht werden, so sind Erweiterungen, wie auch schon bei den zuvor erwähnten Möglichkeiten durchzuführen.

Die Angaben `onStartup()` sowie `onEnd()` erfordern beide ebenfalls keine Änderungen am Konzept des Abgleichs. Sie lösen ein Alignment zu Beginn bzw. zum Ende des Prozesses aus. Zu klären wäre noch, ob auch ein erzwungenes Ende durch eine `Terminate`-Aktivität berücksichtigt werden sollte oder nicht. Die Änderungen am Prozess dafür gestalten sich sehr einfach. Für `onStart()` könnte direkt nach der Registrierung der Code für den Abgleich hinzugefügt werden. Für `onEnd()` gäbe es die Möglichkeit, den gesamten Prozess in ein `Scope`-Element zu schachteln und der `Scope`-Aktivität als Nachfolger eine Sequenz für den Abgleich anzuhängen.

## 6.4 Reaktionen

Wie schon bei den möglichen Angaben zur Festlegung von AlignmentPoints, sind auch für die Anweisungen, welche bei Erfüllung bzw. Nichterfüllung der Bedingungen ausgeführt werden, viele weitere Angaben denkbar. Einige ausgewählte sollen an dieser Stelle in aller Kürze besprochen werden:

- `compensate()`
- `syncVariablesToMax()`
- `syncVariablesToMin()`
- `syncVariablesToAverage()`
- `linkTo()`

Die Anweisung `compensate()` soll, wie der Name schon vermuten lässt, die Ausführung des jeweiligen Compensation-Handlers bewirken. Durch die bereits implementierte Angabe `throwFault()` ist dies ebenfalls möglich, jedoch muss der Compensation-Handler vom Benutzer selbst durch Einfügen einer `Compensate`-Aktivität in den Fault-Handler aktiviert werden. Bei der Angabe von `compensate()` würde der CCF-Transformer einen neuen Fehler kreieren und zu diesem einen Fault-Handler installieren, der durch eine `Throw`-Aktivität ausgelöst wird. Im Fault-Handler wird dann noch eine `Compensate`-Aktivität erzeugt, so dass das gewünschte Verhalten entsteht. Abbildung 26 zeigt die für die Modifikation notwendigen Zusätze.

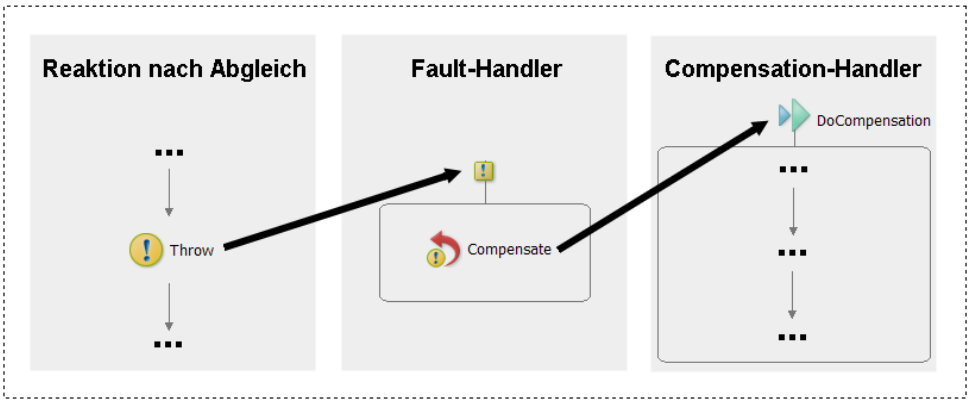


Abbildung 26: *Hinzugefügte Aktivitäten zur Compensation als mögliche Reaktion*

Die Anweisungen `syncVariablesToAverage()`, `syncVariablesToMax()` sowie `syncVariablesToMin()` könnten es ermöglichen, Variablen von Prozess-Instanzen auf einen gemeinsamen Wert zu setzen. Je nach Variante auf das Minimum, Maximum oder den Durchschnitt der bisherigen Werte. Abbildung 27 stellt einen solchen Abgleich von Variablen dar. Flexibler wäre es, für die Bestimmung des Abgleichs eine eigene Sprache einzusetzen in der ausgedrückt werden kann, wie sich der neue Wert aus den bisherigen ermitteln lässt.

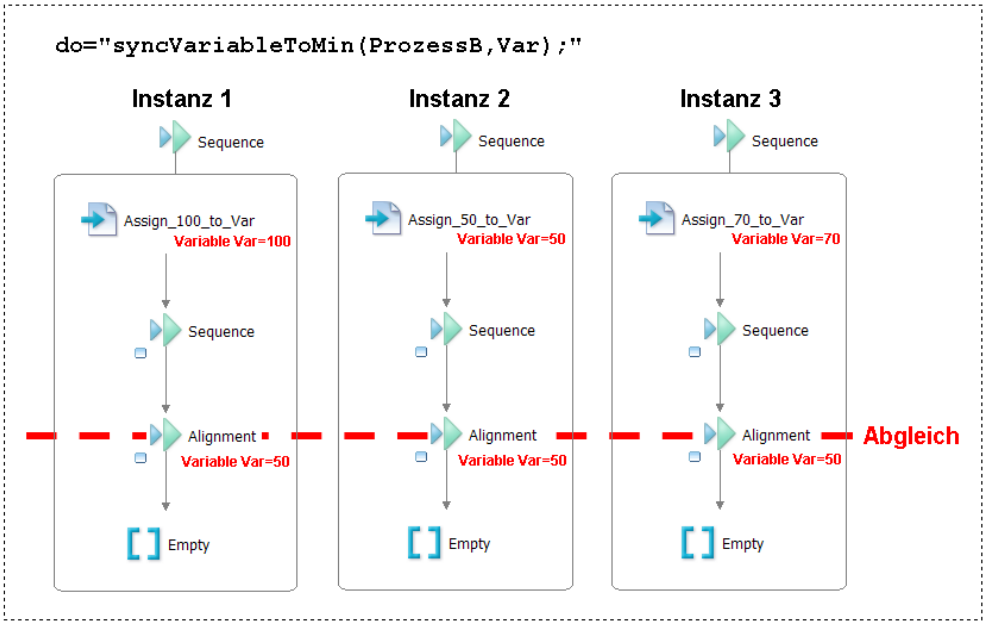


Abbildung 27: *Abgleich von Instanz-Variablen auf das Minimum*

Mittels `linkTo()` können, ähnlich wie bei dem aus imperativen Programmiersprachen bekannten Befehl `goto`, bestimmte Aktivitäten angesprungen werden. Umgesetzt wird das Verhalten durch Hinzufügen eines Links zwischen dem Block für den Abgleich und dem gewünschten Ziel. Zu beachten ist dabei aber, dass Links nicht aus oder in eine `While`-, `Serializable Scope` Aktivität sowie in oder aus Event-Handlern oder

Compensation-Handlern führen dürfen. Ist der zu verknüpfende Block bisher ohne Links, ähnelt die Ausführung sehr der von Unterprogrammen. Ein solcher Quellprozess ist allerdings nicht standardkonformes BPEL, da diese Aktivitäten dann möglicherweise frei stehen. Durch eine Folge dieser Befehle könnten dann mehrere Sequenzen durch die entsprechende Verlinkung nacheinander ausgeführt werden. Abbildung 28 zeigt ein Beispiel für einen Prozess mit abgleichsbedingter Verknüpfung.

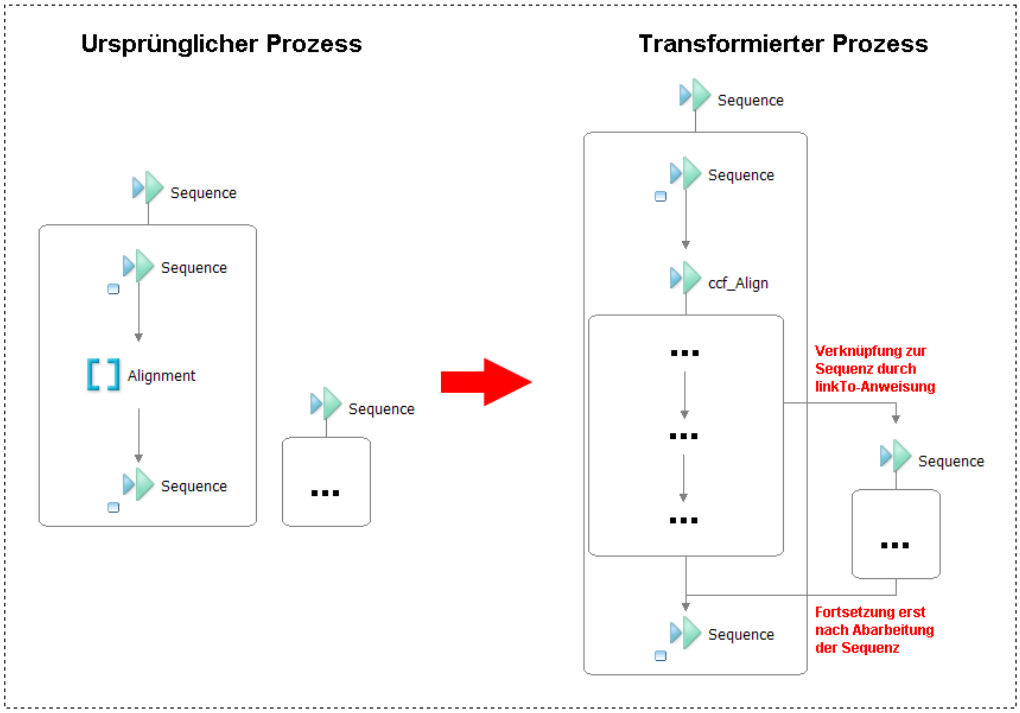


Abbildung 28: *Verknüpfung einer Sequenz durch linkTo-Anweisung*

# 7 Zusammenfassung und Ausblick

In dieser Arbeit wurde gezeigt, dass eine Koordination von BPEL-Prozessen auf Grundlage von WS-Coordination, sowie die Implementierung eines Protocol Service in BPEL möglich sind. Darüberhinaus konnten durch eine zentrale Beschreibung charakterisierte Koordinationen automatisiert in eine ausführbare Form gebracht werden. Die Umwandlung von BPEL-Quellen und CCF-Datei in die ausführbare Koordination erfolgte dabei durch einen in Java implementierten Transformator.

Das entwickelte Modell gestattet eine Vielzahl möglicher Erweiterungen. Einige davon, welche die Mächtigkeit des CCF-Formats an verschiedenen Stellen erhöhen, wurden in Kapitel 6 besprochen. Andere Erweiterungen wie ein Konzept zur persistenten Speicherung des Koordinationszustands waren dagegen nicht Teil dieser Arbeit.

Für einen direkten Praxiseinsatz ist die zugrundeliegende Vorgehensweise kaum geeignet. Die Modifikation von BPEL-Prozessen stellt den Benutzer vor Probleme. Zum Einen muss bei jeder Änderung an den Quellprozessen der Transformationsvorgang neu angestoßen werden. Außerdem ist es für die Transformation erforderlich, dass die Prozess-Dateien aller Teilnehmer lokal verfügbar sind, was sich insbesondere bei Koordinationen, die Unternehmensgrenzen überschneiden, schwierig darstellt.

Vielmehr konnte die Arbeit zeigen, dass eine zentral beschriebene Koordination ohne größeren Zusatzaufwand seitens der Prozess-Entwickler in ausführbarer Weise umsetzbar ist. Wünschenswert für den Praxiseinsatz wäre eine Integration hinreichend mächtiger Möglichkeiten zur Unterstützung von Koordination in die Sprache BPEL, oder alternativ, die nahtlose Einbettung einer eigenständigen Koordinationssprache und die Unterstützung der neuen Konstrukte durch die ausführenden BPEL-Engines.

Die zukünftigen Entwicklungen im Bereich der Webservice-Choreographie, insbesondere der Web Services Choreography Working Group [CHO] des W3C, werden sicherlich in Richtung einer besseren Integration in die vorhandenen Sprachen und Laufzeitumgebungen gehen.

# A Transformation und Deployment

Die Benutzung des Transformators gestaltet sich sehr einfach. Der einzige Parameter für den Aufruf ist die Quelldatei zu der eine ausführbare Koordination erstellt werden soll. Der Aufruf erfolgt also wie folgt:

```
java -cp .;lib/jdom.jar main.Main <CCF-Quelldatei>
```

Der Transformator liest dann die Quellen ein und erzeugt modifizierte Versionen von ihnen, welche den Suffix `_MOD` erhalten. Sollten beim Einlesen oder der Transformation Probleme auftreten, wird eine entsprechende Exception geworfen. Es wurde darauf geachtet, möglichst aussagekräftige Meldungen zu generieren. Sollte die Ursache eines Problems jedoch nicht sofort ersichtlich sein, so prüfen Sie Prozesse und WSDL-Dateien bitte noch einmal nach den in Anhang C genannten Kriterien.

Das Deployment unter ActiveBPEL 2.0 ist ebenfalls sehr einfach. Verwenden Sie zur Erzeugung der Deployment Deskriptoren den ActiveBPEL Designer und führen Sie folgende Schritte durch:

- Aktivieren Sie im Hauptmenü `File/New/Deployment Descriptor`.
- Wählen Sie in der Verzeichnisstruktur den Prozess aus für welchen der Deskriptor erzeugt werden soll. Achten Sie darauf, dass als Deployment Platform der Eintrag `ActiveBPEL-Engine` ausgewählt wurde. Klicken Sie dann auf die Schaltfläche `Next`.
- Hier sind keine Einstellungen vorzunehmen. Klicken Sie also abermals auf `Next`.
- Jetzt erhalten Sie eine Liste von `PartnerLinks` zu denen noch die Endpunktinformation fehlt.
  - Für die `PartnerLinks` für Activation, Registration und Protocol Service wählen Sie als `Invoke Handler` den Eintrag `address`, und als `Endpoint type` den Eintrag `dynamic`.
  - Bei den `PartnerLinks` für die Synchronisierung von koordinationsinternen Aufrufen (`ccf_regDone_...`) wählen Sie ebenfalls `address` und `dynamic`.
  - Für den `PartnerLink` zum Aufruf des Prozesses wählen Sie `System Default / static` aus und geben die statischen Endpunktdaten in das Eingabefeld ein. Für das `Binding` wählen Sie bitte `RPC Encoded` und geben dann im Eingabefeld `Service` den Namen unter welchem der Service deployed werden soll ein.
  - Für die `PartnerLinks` zum Aufruf der anderen Teilnehmer machen Sie Angaben je nach Anforderung.
- Klicken Sie dann auf `Next` und im Folgedialog ebenfalls auf `Next`.

- Wählen Sie dann das Zielverzeichnis für den Deskriptor aus und klicken die Schaltfläche `Finish`.
- Öffnen Sie den Deskriptor in einem Texteditor und überpfüfen Sie, ob im Abschnitt `wsdlReferences` die korrekten WSDL-Dateien eingebunden werden. Überprüfen Sie insbesondere, dass die jeweiligen modifizierten und nicht die ursprünglichen Versionen für die zu den Prozessen gehörigen WSDL-Dateien verwendet werden.
- Wiederholen Sie diesen Vorgang für alle teilnehmenden Prozesse.

Über die Export-Funktion des Designers können die Prozesse nun mithilfe der erstellten Deskriptoren deployed werden. Details über das Deployment entnehmen Sie bitte der ActiveBPEL Designer Dokumentation.

# B Plattform zur Ausführung

Zur Ausführung des CCF-Transformers und der Koordinationen wurden folgende Komponenten verwendet:

- ActiveBPEL 2.0 unter Apache Tomcat 5.0.28 auf Port 8083
- Apache Axis 1.2.1 unter Apache Tomcat 5.5.15 auf Port 8080
- Axis TCPMonitor mit Weiterleitungen von Port 8084 auf Port 8083 und von Port 8081 auf Port 8080
- MySQL Server 5.0
- MySQL Connector/J 3.1
- Java 2 Runtime Environment 1.5.0, Standard Edition
- WS-Coordination Implementierung auf Port 8080 von Thorsten Vetter

Da einige Teile der Implementierung als Java-Klassen vorliegen, andere wiederum in BPEL modelliert wurden (vgl. Abbildung 29), ist es notwendig, sowohl die BPEL-Engine als auch Apache Axis zur Ausführung der Java-Klassen gleichzeitig auszuführen.

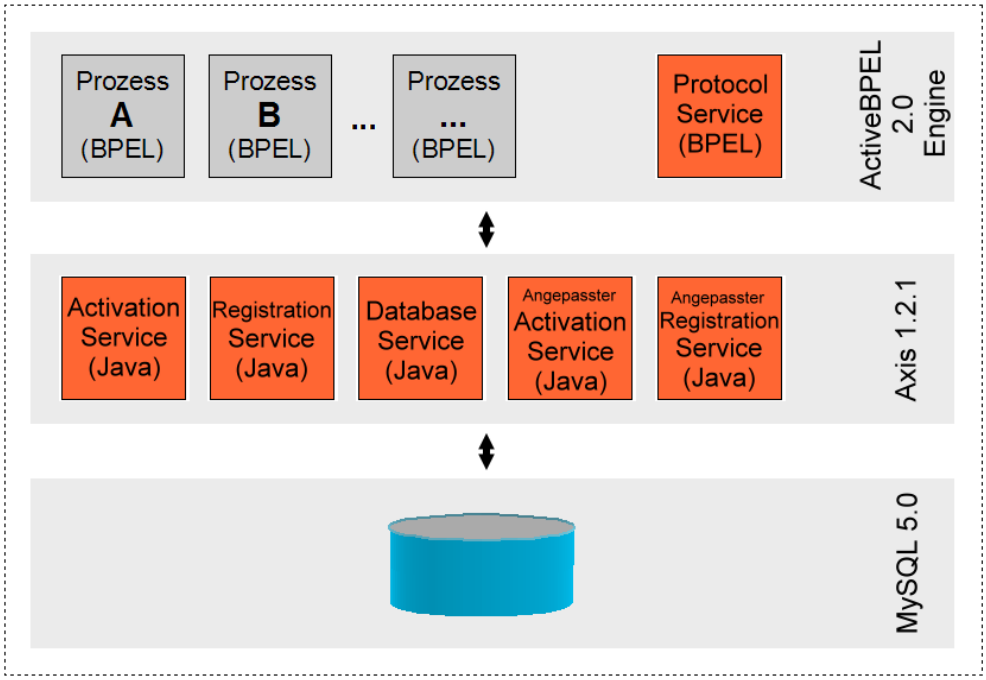


Abbildung 29: *Systemplattform: Axis, ActiveBPEL und MySQL*

Beide basieren auf dem Apache Tomcat Server. Da es aber nicht gelungen ist die beiden Systeme auf einem einzigen Tomcat-Server auszuführen, wurden zwei parallele

Tomcat-Installationen verwendet. Diejenige zur Ausführung von Axis und den Java-Teilen der Implementierung wird dabei auf dem voreingestellten Port 8080 betrieben, die andere zur Ausführung der ActiveBPEL-Engine auf Port 8083.

Zur Anpassung des Ports der ActiveBPEL-Installation sind folgende Schritte durchzuführen:

- Wechseln Sie in das Verzeichnis des Tomcat-Servers Ihrer ActiveBPEL Installation (Hier: C:\Programme\Active Endpoints\ActiveBPEL Designer\Server \ActiveBPEL_Tomcat)
- Wechseln Sie weiter in das Unterverzeichnis `conf`.
- In diesem Verzeichnis befindet sich eine Datei `server.xml`, die Sie mit einem beliebigen Text-Editor bearbeiten können.
- Suchen Sie in der Datei nach folgenden Zeilen:

```
<Connector port="8080"
  maxThreads="150" minSpareThreads="25"
  maxSpareThreads="75" enableLookups="false"
  redirectPort="8443" acceptCount="100"
  debug="0" connectionTimeout="20000"
  disableUploadTimeout="true" />
```

- Ändern Sie das Attribut Port auf den Wert `8083`:

```
<Connector port="8083"
  maxThreads="150" minSpareThreads="25"
  maxSpareThreads="75" enableLookups="false"
  redirectPort="8443" acceptCount="100"
  debug="0" connectionTimeout="20000"
  disableUploadTimeout="true" />
```

- Damit beide Server parallel lauffähig sind müssen noch weitere Ports geändert werden. Suchen Sie deshalb nach folgenden Zeilen:

```
<Connector port="8009"
  enableLookups="false" redirectPort="8443" debug="0"
  protocol="AJP/1.3" />
```

- Ändern Sie den Wert für das `Port`-Attribut auf `8010`:

```
<Connector port="8010"
  enableLookups="false" redirectPort="8443" debug="0"
  protocol="AJP/1.3" />
```

- Genauso ändern Sie die Zeile

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">
```

in

```
<Server port="8006" shutdown="SHUTDOWN" debug="0">
```

Für die Ausführung der Koordinationen ist der Koordinator von Thorsten Vetter zu installieren und konfigurieren. Dies umfasst auch die Einrichtung einer Datenbank. Details entnehmen Sie den der Implementierung des Koordinators beigelegten Hilfedateien.

Alle Zugriffe auf Webservices wurden so eingestellt, dass sie nicht über die tatsächlichen Ports der Services, sondern über den nächsthöheren Port durchgeführt werden, also die Ports 8081 und 8084. Zur Weiterleitung an die tatsächlichen Ports wird das Werkzeug TCPMonitor verwendet, welches Axis beiliegt und ein Betrachten der Nachrichtenaustausche ermöglicht. Wie TCPMonitor verwendet wird, entnehmen Sie der Dokumentation von Axis [AXI]. Abbildung 30 zeigt die Oberfläche des TCPMonitor-Tools nach der erfolgreichen Einrichtung beider Weiterleitungen.

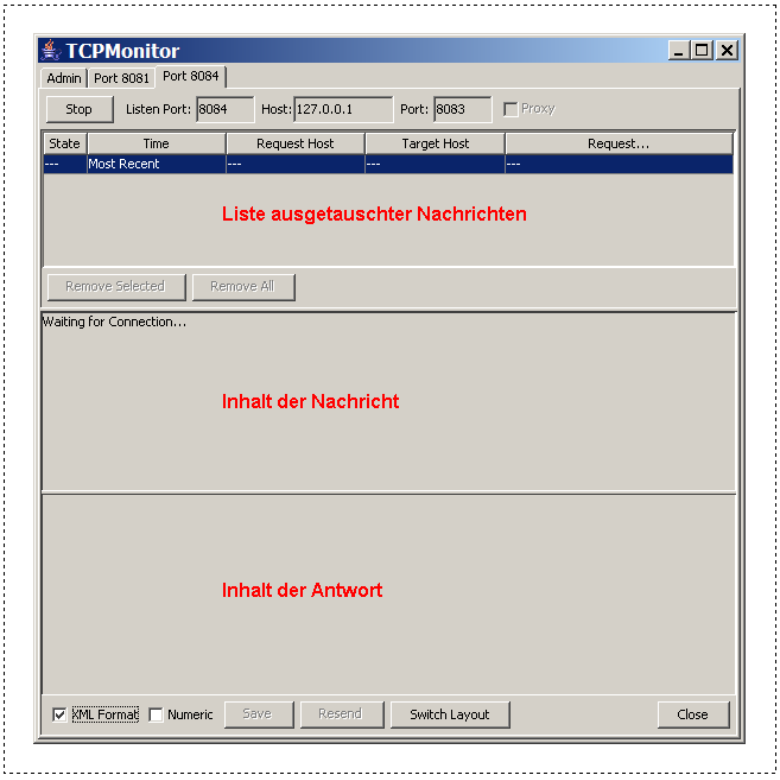


Abbildung 30: *Axis TCPMonitor mit Weiterleitung der Ports 8081 und 8084*

Das Deployment des Protocol Service kann vom ActiveBPEL-Designer aus mit dem beiliegenden Deskriptor erfolgen. Ebenso liegen Deployment-Deskriptoren für die Zwischenservices für Activation und Registration sowie den Service für den Datenbankzugriff bei, mit welchen alle Services einzubinden sind.

Für eine fehlerfreie Ausführung der BPEL-Prozesse ist noch die Änderung einer Einstellung am ActiveBPEL-Server notwendig. Führen Sie dazu folgende Schritte durch:

- Vergewissern Sie sich, dass die ActiveBPEL-Engine auf Port 8083 ausgeführt wird.
- Rufen Sie in einem Webbrowser Ihrer Wahl die ActiveBPEL-Administration durch Eingabe der URL `http://localhost:8083/BpelAdmin` auf.
- Klicken Sie auf den Link `Configuration` um zum Menü für die Serverkonfiguration zu gelangen.
- Aktivieren Sie das Kontrollkästchen für die Option `Auto create target path for Copy/To`.
- Speichern Sie die Änderung durch Klick auf die Schaltfläche Update.

Diese Einstellung ermöglicht es in BPEL bestehende Datenstrukturen zu erweitern wie es z. B. im Protocol Service dringend erforderlich ist. Nach diesen Schritten ist das System auf die Ausführung von Koordinationen vorbereitet. Das Deployment der modifizierten Prozesse wird in Anhang A beschrieben.

# C Anforderungen an BPEL-Quellen

Das Koordinationsmodell sowie der CCF-Transformer wurden so gestaltet, dass der Benutzer bei der Modellierung der Geschäftsprozesse möglichst wenig Rücksicht auf die Belange der Koordination nehmen muss. Dennoch gibt es einige Punkte bei der Erstellung der BPEL- und WSDL-Dateien zu beachten. Sie werden an dieser Stelle, getrennt nach Anforderungen an BPEL-Dateien und WSDL-Dateien, in Listenform zusammengefasst.

## C.1 Anforderungen an BPEL-Dateien

- **Validität:** Die Quelldateien müssen wohlgeformten WS-BPEL 1.1 Quellcode enthalten.
- **Verfügbarkeit der BPEL-Quelle:** Die Quelldatei für den Geschäftsprozess muss lesend zugreifbar sein.
- **Schreibrechte:** Für alle Verzeichnisse, die BPEL-Quelldateien enthalten, müssen Schreibrechte vorhanden sein, um die modifizierten Versionen erstellen zu können.
- **Eindeutige Bezeichner:** Für alle durch ihren Namen referenzierten Aktivitäten muss ein eindeutiger Bezeichner als Name vergeben werden.
- **Gültigkeitsbereich von Variablen:** Alle für einen Abgleich notwendigen Variablen müssen im `Scope` der Aktivität, vor der der Abgleich erfolgt, sichtbar sein. Bei parallelen Aktivitäten gilt Entsprechendes für den gemeinsamen Eltern-`Scope`.
- **Ablauflogik und AlignmentPoints:** Die Wahl der Aktivitäten für AlignmentPoints ist sinnvoll zu treffen, so dass ein Abgleich auch tatsächlich durchgeführt wird. Nicht sinnvoll ist beispielsweise die Angabe zweier aufeinanderfolgenden Aktivitäten als parallelen AlignmentPoint.
- **Mehrfaches Erreichen eines AlignmentPoint:** Es ist darauf zu achten, dass, falls ein AlignmentPoint während eines Prozessdurchlaufs mehrfach erreicht werden kann, dies auch in den anderen, am Alignment beteiligten Prozessen stattfindet.

## C.2 Anforderungen an WSDL-Dateien

- **Validität:** Die Quelldateien müssen wohlgeformten WSDL 1.1 Quellcode enthalten.
- **Verfügbarkeit der WSDL-Quelle:** Die Quelldatei für die zum Geschäftsprozess gehörenden Definitionen muss lesend zugreifbar sein.

- **Schreibrechte:** Für alle Verzeichnisse, die WSDL-Quelldateien enthalten, müssen Schreibrechte vorhanden sein, um die modifizierten Versionen erstellen zu können.
- **Namespaces:** Der Ziel-Namespace der WSDL-Quelle muss mit dem in der zugehörigen BPEL-Quelldatei für Nachrichtendefinitionen und andere verwendeten WSDL-Elemente übereinstimmen. Die Namespace-Präfixe dürfen dennoch unterschiedlich gewählt werden.
- **WSDL-Einzeldateien:** Es wird empfohlen, alle für einen Geschäftsprozess notwendigen Definitionen in einer einzelnen WSDL-Datei unterzubringen. Dringend erforderlich ist, dass alle zu erweiternden Nachrichten, Operationen und Bindings in der im WSDL-Attribut der CCF-Datei angegebenen Datei vorhanden sind. Etwaige importierte Quellen werden bei der Erweiterung nicht berücksichtigt.
- **Pro Prozess eine Datei:** Es wird empfohlen, zu jeder BPEL-Quelldatei eine WSDL-Quelldatei zu erstellen und diese nicht mehrfach zu verwenden.

# D XML-Schema des CCF-Formats

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://vortexport.net/CCFSchema"
           xmlns="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">

  <xs:element name="coordinations"
              type="coordinationsType"/>

  <xs:complexType name="coordinationsType">
    <xs:choice>
      <xs:element name="partners" type="partnersType"
                  minOccurs="1" maxOccurs="1"/>
      <xs:element name="conditions" type="conditionsType"
                  minOccurs="1" maxOccurs="1"/>
      <xs:element name="alignmentPoints"
                  type="alignmentPointsType"
                  minOccurs="1" maxOccurs="1"/>
      <xs:element name="coordinate" type="coordinateType"
                  minOccurs="1" maxOccurs="unbounded"/>
    </xs:choice>
    <xs:attribute name="activationService" type="xs:anyURI"
                  use="optional"/>
  </xs:complexType>

  <xs:complexType name="partnersType">
    <choice>
      <xs:element name="partner" type="partnerType"
                  minOccurs="1" maxOccurs="unbounded"/>
    </choice>
  </xs:complexType>

  <xs:complexType name="conditionsType">
    <choice>
      <xs:element name="condition" type="conditionType"
                  minOccurs="1" maxOccurs="unbounded"/>
    </choice>
  </xs:complexType>
```

```
<xs:complexType name="alignmentPointsType">
  <choice>
    <xs:element name="alignmentPoint"
                type="alignmentPointType"
                minOccurs="0" maxOccurs="unbounded"/>
  </choice>
</xs:complexType>

<xs:complexType name="coordinateType">
  <xs:attribute name="conditions" type="xs:string"
                use="required"/>
  <xs:attribute name="alignmentPoints" type="xs:string"
                use="optional"/>
  <xs:attribute name="do" type="xs:string"
                use="optional"/>
  <xs:attribute name="else" type="xs:string"
                use="optional"/>
</xs:complexType>

<xs:complexType name="partnerType">
  <xs:attribute name="name" type="xs:NCName"
                use="required"/>
  <xs:attribute name="bpel" type="xs:anyURI"
                use="required"/>
  <xs:attribute name="wsdl" type="xs:anyURI"
                use="optional"/>
</xs:complexType>

<xs:complexType name="conditionType">
  <xs:attribute name="name" type="xs:NCName"
                use="required" />
  <xs:attribute name="expression" type="xs:string"
                use="required"/>
</xs:complexType>

<xs:complexType name="alignmentPointType">
  <choice>
    <xs:element name="align" type="alignType"
                minOccurs="0" maxOccurs="unbounded"/>
  </choice>
  <xs:attribute name="name" type="xs:NCName"
                use="required"/>
</xs:complexType>
```

```
  <xs:complexType name="alignType">
    <xs:attribute name="partner" type="xs:string"
                  use="required"/>
    <xs:attribute name="alignment" type="xs:string"
                  use="required"/>
  </xs:complexType>

</xs:schema>
```

# Literatur


[ABB+05] Anderson, Steve ; Bohren, Jeff ; Boubez, Toufic ; Chanliau, Marc ; Della-Libera, Giovanni ; Dixon, Brendan ; Garg, Praerit ; Gudgin, Martin ; Hallam-Baker, Phillip ; Hondo, Maryann ; Kaler, Chris ; Lockhart, Hal ; Martherus, Robin ; Maruyama, Hiroshi ; Nadalin, Anthony ; Nagaratnam, Nataraj ; Nash, Andrew ; Philpott, Rob ; Platt, Darren ; Prafullchandra, Hemma ; Sahu, Maneesh ; Shewchuk, John ; Simon, Dan ; Srinivas, Davanum ; Waingold, Elliot ; Waite, David ; Walter, Doug ; Zolfonoon, Riaz: *Web Services Trust Language (WS-Trust)*. Version: 2005. ftp://www6.software.ibm.com/software/developer/library/ws-trust.pdf, Abruf: 16.08.2006

[ACD+03] Andrews, Tony ; Curbera, Francisco ; Dholakia, Hitesh ; Goland, Yaron ; Klein, Johannes ; Leymann, Frank ; Liu, Kevin ; Roller, Dieter ; Smith, Doug ; Thatte, Satish ; Trickovic, Ivana ; Weerawarana, Sanjiva: *Business Process Execution Language for Web Services Version 1.1*. Version: Mai 2003. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf, Abruf: 25.07.2006

[ADO+05] Aalst, W.M.P. van d. ; Dumas, M. ; Ouyang, C. ; Rozinat, A. ; Verbeek, H.M.W.: *Choreography Conformance Checking: An Approach based on BPEL and Petri Nets*. Department of Technology Management, Eindhoven University of Technology and Faculty of Information Technology, Queensland University of Technology. http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2005/BPM-05-25.pdf. Version: 2005

[AXI] *Axis User's Guide*. : *Axis User's Guide*, http://ws.apache.org/axis/java/user-guide.html

[CCF+05] Cabrera, Luis F. ; Copeland, George ; Feingold, Max ; Freund, Robert W. ; Freund, Tom ; Johnson, Jim ; Joyce, Sean ; Kaler, Chris ; Klein, Johannes ; Langworthy, David ; Little, Mark ; Nadalin, Anthony ; Newcomer, Eric ; Orchard, David ; Robinson, Ian ; Shewchuk, John ; Storey, Tony: *Web Services Coordination (WS-Coordination)*. Version: August 2005. ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf, Abruf: 25.07.2006

[CCMW01] Christensen, Erik ; Curbera, Francisco ; Meredith, Greg ; Weerawarana, Sanjiva: *Web Services Description Language (WSDL) 1.1*. Version: März 2001. http://www.w3.org/TR/wsdl, Abruf: 25.07.2006

[CHO] *Web Services Choreography Working Group*. http://www.w3.org/2002/ws/chor/

[Cor06] Corporation, Microsoft: *Web Service Security: Scenarios, Patterns, and Implementation Guidance for Web Services Enhancements (Wse) 3.0*. Microsoft Press, 2006

[Dab06] Dabbert, Julian: *Fault tolerance and recovery in middleware*. Fakultät für Elektrotechnik - Informatik - Mathematik, Arbeitsgruppen Datentechnik und Softwaretechnik, Universität Paderborn. `http://wwwcs.uni-paderborn.de/cs/ag-schaefer/Lehre/Lehrveranstaltungen/Seminare/FTSHS/Ausarbeitung_Dabbert.pdf`. Version: 2006

[FKK96] Freier, Alan O. ; Karlton, Philip ; Kocher, Paul C.: *The SSL Protocol Version 3.0*. Version: 1996. `http://wp.netscape.com/eng/ssl3/`, Abruf: 16.08.2006

[FMF+99] Fielding, R. ; Mogul, J. C. ; Frystyk, H. ; Masinter, L. ; Leach, P. ; Berners-Lee, T.: *Hypertext Transfer Protocol - HTTP/1.1*. Version: Juni 1999. `http://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf`, Abruf: 25.07.2006

[GHM+03] Gudgin, Martin ; Hadley, Marc ; Mendelsohn, Noah ; Moreau, Jean-Jacques ; Nielsen, Henrik F.: *SOAP Version 1.2 Specification*. Version: Juni 2003. `http://www.w3.org/TR/soap12`, Abruf: 25.07.2006

[HWS05] Hildebrandt, Ingo ; Wagner, Marc ; Stützner, Carsten: *Evaluierung verschiedener Szenarien zur Anwendung von BPEL und WS-CDL*. Institut für Architektur von Anwendungssystemen (IAAS) - Universität Stuttgart, 2005

[KBR+05] Kavantzas, Nickolas ; Burdett, David ; Ritzinger, Gregory ; Fletcher, Tony ; Lafon, Yves ; Barreto, Charlton: *Web Services Choreography Description Language Version 1.0*. Version: November 2005. `http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/`, Abruf: 25.07.2006

[Kip06] Kipp, Alexander: *Ablösung von WS-CDL durch BPEL und WSFL Global Model*, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Diplomarbeit, 2006

[LSP82] Lamport, Leslie ; Shostak, Robert ; Pease, Marshall: The Byzantine Generals Problem. Version: 1982. `http://dx.doi.org/http://doi.acm.org/10.1145/357172.357176`. New York, NY, USA : ACM Press, 1982 (3). – Forschungsbericht. – 382–401 S.. – ISSN 0164–0925

[Man03] Manhart, Klaus: *Sicherheit bei Webservices*. tecCHANNEL.de Website. `http://www.tecchannel.de/sicherheit/grundlagen/402212/`. Version: 2003

[Muc97] Muchnick, Steven S.: *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997

[NKMHB06] Nadalin, Anthony ; Kaler, Chris ; Monzillo, Ronald ; Hallam-Baker, Phillip: *Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)*. Version: 2006. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss, Abruf: 16.08.2006

[NLJ96] Nwana, H. S. ; Lee, L. ; Jennings, N. R.: *Co-ordination in software agent systems*. Version: Oktober 1996. http://www.ecs.soton.ac.uk/~nrj/download-files/bttj96.pdf, Abruf: 16.08.2006

[RK03] Reiser, Hans P. ; Kapitza, Rüdiger: *Verteilte Algorithmen (VA) Vorlesungsskript, Wintersemester 2003/04*. Lehrstuhl für Informatik 4, Universität Erlangen-Nürnberg. http://www4.informatik.uni-erlangen.de/Lehre/WS03/V_VA/Skript/. Version: 2003

[Sch05] Schwenk, Jörg: *Sicherheit und Kryptographie im Internet. Von sicherer E-Mail bis zu IP-Verschlüsselung*. Vieweg, 2005

[Tan95] Tannenbaum, Andrew S.: *Distributed Operating Systems*. Prentice Hall International Editions, 1995

[Tan03] Tanenbaum, Andrew S.: *Verteilte Systeme. Grundlagen und Paradigmen*. Pearson Studium, 2003

[Vet06] Vetter, Thorsten: *Anpassung und Implementierung verschiedener Transaktionsprotokolle auf WS-Coordination*, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Diplomarbeit, 2006. ftp://ftp.informatik.uni-stuttgart.de/pub/library/medoc.ustuttgart_fi/DIP-2386/DIP-2386.pdf

[War06] Warner, Daniel: *SQL in Theorie und Praxis. Nach dem ISO/ANSI-Standard SQL-99 und SQL 2003*. Addison-Wesley, München, 2006

[Wut06] Wutke, Daniel: *Erweiterung einer Workflow-Engine zur Unterstützung von parametrisierten Web Service Flows*, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Diplomarbeit, 2006. ftp://ftp.informatik.uni-stuttgart.de/pub/library/medoc.ustuttgart_fi/DIP-2401/DIP-2401.pdf

[Zim80] Zimmermann, Hubert: OSI Reference Model-The ISO Model of Architecture for Open Systems Interconnection. In: *IEEE Transactions on Communications* 28 (1980). http://www.comsoc.org/livepubs/50_journals/pdf/RightsManagement_eid=136833.pdf

# Abbildungsverzeichnis


1 *Leicht vereinfachter Ablauf der Anfangsphase einer Koordination* . . . . 8
2 *Hinzufügen transformationsspezifischer Teile* . . . . . 12
3 *Baum zur Unterscheidung der Fälle und Instanzenmengen* . . . . . 18
4 *Beispiel für den Ablauf einer CCF-Koordination mit zwei Partnern* . . 19
5 *Möglicher fehlerhafter Ablauf bei asynchronen Aufrufen* . . . . . 23
6 *Unterscheidung von internen und externen Aufrufen* . . . . . 30
7 *Problem der Flag-Initialisierung* . . . . . 31
8 *Remodellierter Prozess im Überblick (ohne Event-Handler)* . . . . . 33
9 *Beibehaltung der Semantik von serialisierbaren Scopes* . . . . . 35
10 *Dynamischer Operationsaufruf durch einen Zwischenservice* . . . . . 36
11 *Beispiel - Teilnehmer 1 ruft Teilnehmer 2 auf* . . . . . 37
12 *AlignmentPoint mit zwei parallelen Aktivitäten* . . . . . 38
13 *Ablauf zur Erläuterung der Dead-Path-Elimination* . . . . . 39
14 *Transformation eines AlignmentPoint bei parallelen Aktivitäten* . . . . 40
15 *Event-Handler mit je einem Case-Element pro AlignmentPoint* . . . . . 42
16 *Umwandlung von CCF-Methoden* . . . . . 43
17 *Reaktionsteil in einem modifizierten BPEL-Prozess* . . . . . 46
18 *Beendigungs-Sequenz* . . . . . 47
19 *Protocol Service im Überblick* . . . . . 52
20 *Initialisierungsteil des ProtocolService* . . . . . 53
21 *Align-Operation des Protocol Service* . . . . . 54
22 *Rückmeldung an betroffene Teilnehmer* . . . . . 55
23 *Zwischenservices für Activation und Registration* . . . . . 56
24 *Einsatz von Pick und onAlarm zur Unterstützung von Timeouts* . . . . 61
25 *Ausgelagerter Protocol Service mit integrierter Entscheidungsfindung* . 62
26 *Hinzugefügte Aktivitäten zur Compensation als mögliche Reaktion* . . . 65
27 *Abgleich von Instanz-Variablen auf das Minimum* . . . . . 65
28 *Verknüpfung einer Sequenz durch linkTo-Anweisung* . . . . . 66
29 *Systemplattform: Axis, ActiveBPEL und MySQL* . . . . . 70
30 *Axis TCPMonitor mit Weiterleitung der Ports 8081 und 8084* . . . . . 72

**Erklärung**

Hiermit versichere ich, diese Arbeit
selbständig verfasst und nur die
angegebenen Quellen benutzt zu haben.

---

(Sven Knödler)