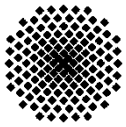

Institut für Architektur von Anwendungssystemen (IAAS)

Universität Stuttgart
Universitätsstraße 38
D – 70569 Stuttgart

Diplomarbeit Nr. 3138

Sprachübergreifende Überwachung von Geschäftsprozessen

Eike Klenk

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. David Schumm
begonnen am:	02.02.2011
beendet am:	04.08.2011
CR-Klassifikation:	C.2.4, D.2.2, H.4.1, H.5.2, H.5.3

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In der heutigen Zeit sind Geschäftsprozesse wichtige Bestandsteile der Unternehmensführung. Damit können ganze Geschäftsbereiche oder nur Unterbereiche, inklusive aller Umwelteinflüsse und Abhängigkeiten, modelliert werden. Das erleichtert die Planung und die Organisation des Unternehmens und gibt Rückschlüsse auf etwaige Verbesserungsmöglichkeiten. In der Praxis ist die Business Process Execution Language, kurz BPEL [1], ein Standard um Geschäftsprozesse zu modellieren, auszuführen und anschließend zu Überwachen. BPEL ist sehr technisch gehalten und daher für Manager oder Geschäftsführer oft schwer zu lesen und zu verstehen. Für diese Anwendergruppen ist eine weniger technische, visuell modellierbare, Sprache zur Erstellung und Überwachung von Geschäftsprozessen geeigneter.

Die Überwachung von Geschäftsmodellen wird ein immer wichtigerer Aspekt in der Unternehmensführung. Da die erstellten Geschäftsmodelle häufig über Details verfügen, die nicht für jeden Betrachter von Interesse sind, geht die Entwicklung in Richtung von Werkzeugen, mit denen es möglich ist, die Komplexität zu verringern oder uninteressante Details für den aktuellen Betrachter auszublenden. Der aktuelle Stand ist, dass Geschäftsprozesse in der Sprache überwacht werden, in denen sie erstellt wurden. Die Überwachung eines Geschäftsprozesses in einer anderen, vielleicht für den Betrachter verständlicheren, Sprache, soll in Zukunft auch möglich sein. Es gibt zwar viele Ansätze um verschiedene Sprachen aufeinander abzubilden, aber es gibt bisher nur wenige um den aktuellen Ablauf zu projizieren.

In [2] wurde der Business Process Illustrator, kurz BPI [3], konzipiert und entwickelt. Dabei handelt es sich um ein Werkzeug zur Überwachung von Geschäftsprozessen, die mit BPEL erstellt wurden. Mit dem Werkzeug kann ein Graph eines Prozessmodells, der mit Statusinformationen der Prozessinstanz erweitert wurde, überwacht werden. Zudem ist es möglich den Detailgrad der Ansicht auf den Geschäftsprozess zu variieren. In der weiteren Entwicklung des BPI soll es möglich sein einen Geschäftsprozess in einer anderen Prozessmodellierungssprache zu überwachen, als mit der Sprache, mit der der Geschäftsprozess ausgeführt wird. Als Vorbereitung für diese Erweiterung wird in dieser Arbeit ein Konzept erstellt, das es ermöglicht, die Zustände aus BPEL nach BPMN zu übertragen. Hierfür wird beschrieben, wie aus der BPEL-Datei eine BPMN-Datei generiert wird und anschließend, die Zustände mit zuvor definierten Regeln übertragen werden.

1.2 Abgrenzung des Themas

Die Vielfalt an Sprachen macht das Thema sehr mächtig. Um das Thema einzugrenzen werden in dieser Arbeit die beiden Sprachen BPEL und BPMN, Business Process Model and Notation [4], betrachtet. Das heißt es soll eine Überwachung eines Geschäftsprozesses mit BPMN ermöglicht werden, während der Geschäftsprozess selbst mit BPEL ausgeführt wird.

1.3 Aufgabenstellung

Ein Geschäftsprozess, der mit BPEL erstellt wurde und ausgeführt wird, soll als BPMN Prozess überwacht werden. In dieser Arbeit sollen Konzepte erstellt werden, um die Projektion

des Ausführungszustands von BPEL nach BPMN darzustellen. Weiter sollen Techniken zur Übertragung von BPEL Variablen zu BPMN Informationen spezifiziert werden.

1.4 Aufbau

In Kapitel 2 werden Beispiele zur Verbesserung des Verständnisses des Lesers vorgestellt. In Kapitel 3 folgen die Grundlagen der Arbeit. Hier werden die beiden Sprachen BPEL und BPMN vorgestellt und deren Konstrukte erläutert. In Kapitel 4 ist die Transformation von BPEL zu BPMN das Thema. Die Abbildungen aller BPEL Aktivitäten in BPMN werden erörtert und durch Beispiele gestützt. Die Projektion der Zustände wird in Kapitel 5 behandelt. Zuerst wird der theoretische Ansatz zur Projektion beschrieben, gefolgt von der Definition der Zustandsräume für den Prozess und die Aktivitäten. Zusätzlich werden die benötigten XML-Erweiterungen für BPEL und BPMN, sowie die Generierung einer unterstützenden Mappings-Datei beschrieben. Die Erläuterung der Probleme bei der Projektion der Zustände schließt das Kapitel ab. Abgeschlossen wird die Arbeit von der Zusammenfassung der Erkenntnisse und einem Ausblick für die Zukunft.

2 Beispiele für BPEL und BPMN

Als Einführung und zur Förderung des Verständnisses beginnt diese Arbeit mit einem Anwendungsbeispiel. Dargestellt wird eine Produktanfrage in einem Onlineshop. Dabei wird eine Anfrage empfangen und die Verfügbarkeit des Produkts abgefragt. Anschließend werden je nach Verfügbarkeit verschiedene Aktivitäten ausgeführt. Ist das Produkt verfügbar, werden keine weiteren Aktivitäten ausgeführt. Ist das Produkt nicht verfügbar, wird überprüft, ob es nachbestellt werden kann und ist dies der Fall, werden alle Lieferangebote erfasst und ausgewertet. Kann das Produkt nicht nachbestellt werden, wird nach ähnlichen Produkten gesucht und anschließend werden die Ergebnisse der Suche zusammengefasst. Abgeschlossen wird der Prozess, indem ein Produktangebot als Antwort zurückgegeben wird. In Abbildung 1 wird der BPEL-Prozess graphisch mit Hilfe des BPEL Designer [5] dargestellt und in Listing 1 ist der passende, schon mit Zuständen erweiterte, BPEL-Code zu sehen. In Abbildung 2 ist das Beispiel in BPMN dargestellt, ebenfalls bereits zustandsbehaftet. Um die Lesbarkeit zu gewährleisten wird in dem Beispiel nur die Variable *Product* als Data Object mit zwei Message Flow Verbindungen abgebildet. In Abbildung 3 ist die BPEL zu BPMN Zuordnungen graphisch dargestellt. Die roten Pfeile kennzeichnen die Zuordnungen der Zustände der BPEL Aktivitäten zu ihren Darstellungen in BPMN.

Um die bereits zustandsbehafteten Diagramme verstehen zu können, ist in Tabelle 1 eine Legende der Zustände angegeben.

Name	**Symbol**
Inactive	
Ready	
Skipped	
Executing	
Completed	
Iteration Completed	
Compensated	
Faulted	
Terminated	

Tabelle 1: Legende der Zustände

2.1 BPEL

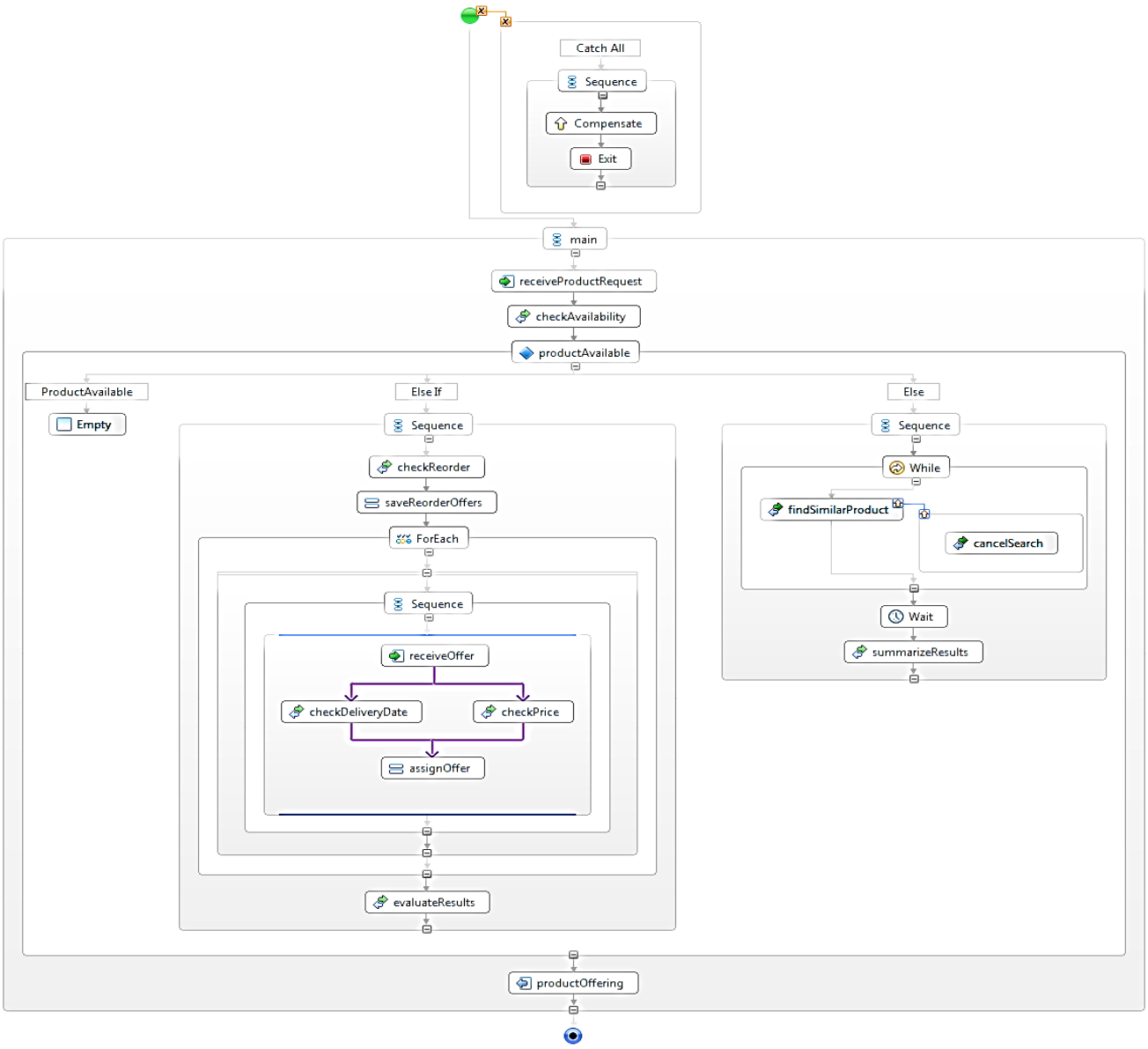

Abbildung 1: Graphisch dargestellter BPEL-Prozess

Der <process> ist das Hauptkonstrukt eines BPEL Prozesses. Innerhalb des <process> Konstrukts werden globale Einstellungen für den BPEL Prozess vorgenommen. Dazu zählen <extensions>, <import>s, <partnerLinks>, <variables>, <correlationSets>, <message-Exchanges>, <eventHandlers> und <faultHandlers>. Im <process> Konstrukt darf nur eine Aktivität enthalten sein. In diesem Fall ist das eine <sequence>. Weiterführende Informationen zu BPEL folgen in Kapitel 3.

```
<bpel:process name="Beispiel"
  targetNamespace="http://sample.bpel.org/bpel/sample"
  suppressJoinFailure="yes"
  xmlns:tns="http://sample.bpel.org/bpel/sample"
  xmlns:bpel=
    "http://docs.oasis-open.org/wsbpel/2.0/process/executable"
  executableProcessProfile=
```

```
    "http://docs.oasis-open.org/wsbpel/2.0/process/executable
    /simple-template/2006/08">
   <ext:processState>Executing</ext:processState>
   <bpel:import location="Beispiel.wsdl"
     namespace=http://sample.bpel.org/bpel/sample
     importType="http://schemas.xmlsoap.org/wsdl/" />
   <bpel:variables>
      <bpel:variable name="product" messageType="tns:productType">
         <ext:activityId>variable 1</ext:activityId>
         <ext:variableValue></ext:variableValue>
      </bpel:variable>
      <bpel:variable name="availabilityInfo"
        messageType="tns:availabilityType">
         <ext:activityId>variable 2</ext:activityId>
         <ext:variableValue></ext:variableValue>
      </bpel:variable>
      <bpel:variable name="productOffers" messageType="tns:productType">
         <ext:activityId>variable 3</ext:activityId>
         <ext:variableValue></ext:variableValue>
      </bpel:variable>
      <bpel:variable name="productOffer" messageType="tns:productType">
         <ext:activityId>variable 4</ext:activityId>
         <ext:variableValue></ext:variableValue>
      </bpel:variable>
      <bpel:variable name="productDeliveryDate" messageType="tns:date">
         <ext:activityId>variable 5</ext:activityId>
         <ext:variableValue></ext:variableValue>
      </bpel:variable>
      <bpel:variable name="productPrice" messageType="tns:float">
         <ext:activityId>variable 6</ext:activityId>
         <ext:variableValue></ext:variableValue>
      </bpel:variable>
      <bpel:variable name="evaluatedProductOffers"
        messageType="tns:productArrayType">
         <ext:activityId>variable 7</ext:activityId>
         <ext:variableValue></ext:variableValue>
      </bpel:variable>
      <bpel:variable name="similarProductRequest"
        messageType="tns:productType">
         <ext:activityId>variable 8</ext:activityId>
         <ext:variableValue></ext:variableValue>
      </bpel:variable>
   </bpel:variables>
   <bpel:faultHandlers>
      <ext:activityId>faulthandlers 1</ext:activityId>
      <ext:activityState>Ready</ext:activityState>
      <ext:projectionType>Casual</ext:projectionType>
      <bpel:catchAll>
         <ext:activityId>catchall 1</ext:activityId>
         <bpel:sequence>
         <ext:activityId>sequence 1</ext:activityId>
         <bpel:compensate>
            <ext:activityId>compensate 1</ext:activityId>
            <ext:activityState>Inactive</ext:activityState>
            <ext:projectionType>Casual</ext:projectionType>
         </bpel:compensate>
         <bpel:exit name="Exit">
            <ext:activityId>exit 1</ext:activityId>
            <ext:activityState>Inactive</ext:activityState>
            <ext:projectionType>Casual</ext:projectionType>
         </bpel:exit>
         </bpel:sequence>
```

```
      </bpel:catchAll>
   </bpel:faultHandlers>
   <bpel:sequence name="main">
      <ext:activityId>sequence 2</ext:activityId>
      <bpel:receive name="receiveProductRequest" variable="product"
        createInstance="yes" operation="sendProductRequest">
         <ext:activityId>receive 1</ext:activityId>
         <ext:activityState>Completed</ext:activityState>
         <ext:projectionType>Casual</ext:projectionType>
      </bpel:receive>
      <bpel:invoke name="checkAvailability" inputVariable="product"
        outputVariable="availabilityInfo"
        operation="requestAvailability">
         <ext:activityId>invoke 1</ext:activityId>
         <ext:activityState>Completed</ext:activityState>
         <ext:projectionType>Casual</ext:projectionType>
      </bpel:invoke>
      <bpel:if name="ProductAvailable">
         <ext:activityId>if 1</ext:activityId>
         <ext:activityState>Executing</ext:activityState>
         <ext:projectionType>Casual</ext:projectionType>
         <bpel:condition>
           bpel:getVariableProperty('availabilityInfo',
           'inventory:level') > 0
         </bpel:condition>
         <bpel:sequence>
            <ext:activityId>sequence 3</ext:activityId>
            <bpel:empty name="Empty">
               <ext:activityId>empty 1</ext:activityId>
               <ext:activityState>Skipped</ext:activityState>
               <ext:projectionType>Casual</ext:projectionType>
            </bpel:empty>
         </bpel:sequence>
      <bpel:elseif>
         <ext:activityId>elseif 1</ext:activityId>
         <bpel:condition>
            bpel:getVariableProperty('availabilityInfo',
            'product:inProduction') > 0
         </bpel:condition>
         <bpel:sequence>
            <ext:activityId>sequence 4</ext:activityId>
            <bpel:invoke name="checkReorder"
              inputVariable="product" outputVariable="productOffers"
              operation="getProductOffers">
               <ext:activityId>invoke 2</ext:activityId>
               <ext:activityState>Completed</ext:activityState>
               <ext:projectionType>Casual</ext:projectionType>
            </bpel:invoke>
            <bpel:assign validate="no" name="saveReorderOffers">
               <ext:activityId>assign 1</ext:activityId>
               <ext:activityState>Completed</ext:activityState>
               <bpel:copy>
                  <ext:activityId>copy 1</ext:activityId>
                  <ext:activityState>Completed</ext:activityState>
                  <bpel:from>$productOffers</bpel:from>
                  <bpel:to>$product.reorderOffers</bpel:to>
               </bpel:copy>
            </bpel:assign>
            <bpel:forEach parallel="no" counterName="Counter"
              name="ForEach">
               <bpel:startCounterValue>1</bpel:startCounterValue>
               <bpel:finalCounterValue>
```

```
                $productOffers.count
             </bpel:finalCounterValue>
             <ext:activityId>forEach 1</ext:activityId>
             <ext:activityState>Executing</ext:activityState>
             <ext:projectionType>Loop</ext:projectionType>
             <bpel:scope>
                <ext:activityId>scope 1</ext:activityId>
                <bpel:sequence>
                <ext:activityId>sequence 5</ext:activityId>
                <bpel:flow name="Flow">
                   <ext:activityId>flow 1</ext:activityId>
                   <ext:activityState>Executing</ext:activityState>
                   <ext:projectionType>Casual</ext:projectionType>
                   <bpel:receive name="receiveOffer"
                     variable="productOffer" operation="offerRequest">
                      <ext:activityId>receive 2</ext:activityId>
                      <ext:activityState>Completed</ext:activityState>
                      <ext:projectionType>Casual</ext:projectionType>
                      <bpel:sources>
                         <bpel:source linkName="link1"></bpel:source>
                         <bpel:source linkName="link2"></bpel:source>
                      </bpel:sources>
                   </bpel:receive>
                   <bpel:invoke name="checkDeliveryDate"
                     inputVariable="product"
                     outputVariable="productDeliveryDate"
                     operation="getDeliveryDate">
                      <ext:activityId>invoke 3</ext:activityId>
                      <ext:activityState>Executing</ext:activityState>
                      <ext:projectionType>Casual</ext:projectionType>
                      <bpel:targets>
                         <bpel:target linkName="link1"></bpel:target>
                      </bpel:targets>
                      <bpel:sources>
                         <bpel:source linkName="link3"></bpel:source>
                      </bpel:sources>
                   </bpel:invoke>
                   <bpel:assign validate="no" name="assignOffer">
                      <ext:activityId>assign 2</ext:activityId>
                      <ext:activityState>Inactive</ext:activityState>
                      <ext:projectionType>Casual</ext:projectionType>
                      <bpel:copy>
                         <ext:activityId>copy 2</ext:activityId>
                         <bpel:from>$productDeliveryDate</bpel:from>
                         <bpel:to>$product.deliveryDate</bpel:to>
                      </bpel:copy>
                      <bpel:copy>
                         <ext:activityId>copy 3</ext:activityId>
                         <bpel:from>$productPrice</bpel:from>
                         <bpel:to>$product.price</bpel:to>
                      </bpel:copy>
                      <bpel:targets>
                         <bpel:target linkName="link3"></bpel:target>
                         <bpel:target linkName="link4"></bpel:target>
                      </bpel:targets>
                   </bpel:assign>
                   <bpel:invoke name="checkPrice" inputVariable="product"
                     outputVariable="productPrice" operation="getPrice">
                      <ext:activityId>invoke 4</ext:activityId>
                      <ext:activityState>Ready</ext:activityState>
                      <ext:projectionType>Casual</ext:projectionType>
                      <bpel:targets>
```

```
                        <bpel:target linkName="link2"></bpel:target>
                     </bpel:targets>
                     <bpel:sources>
                        <bpel:source linkName="link4"></bpel:source>
                     </bpel:sources>
                  </bpel:invoke>
                  <bpel:links>
                     <bpel:link name="link1"></bpel:link>
                     <bpel:link name="link2"></bpel:link>
                     <bpel:link name="link3"></bpel:link>
                     <bpel:link name="link4"></bpel:link>
                  </bpel:links>
               </bpel:flow>
               </bpel:sequence>
            </bpel:scope>
         </bpel:forEach>
         <bpel:invoke name="evaluateResults"
           inputVariable="productOffers"
           outputVariable="evaluatedProductOffers"
           operation="evaluateResults">
            <ext:activityId>invoke 5</ext:activityId>
            <ext:activityState>Inactive</ext:activityState>
            <ext:projectionType>Casual</ext:projectionType>
         </bpel:invoke>
      </bpel:sequence>
   </bpel:elseif>
   <bpel:else>
      <ext:activityId>else 1</ext:activityId>
      <bpel:sequence>
         <ext:activityId>sequence 6</ext:activityId>
         <bpel:while name="While">
            <ext:activityId>while 1</ext:activityId>
            <ext:activityState>Skipped</ext:activityState>
            <ext:projectionType>Loop</ext:projectionType>
            <ext:iterationCount>0</ext:iterationCount>
            <bpel:condition>
               $similarProductRequest &ne; nil
            </bpel:condition>
            <bpel:invoke name="findSimilarProduct"
              inputVariable="product"
              outputVariable="similarProductRequest"
              operation="findSimilarProduct">
               <ext:activityId>invoke 6</ext:activityId>
               <ext:activityState>Skipped</ext:activityState>
               <ext:projectionType>Casual</ext:projectionType>
               <bpel:compensationHandler>
                  <ext:activityId>compensationHandler 1</ext:activityId>
                  <ext:activityState>Skipped</ext:activityState>
                  <ext:projectionType>Casual</ext:projectionType>
                  <bpel:invoke name="cancelSearch"
                    operation="cancelProductSearch">
                  </bpel:invoke>
               </bpel:compensationHandler>
            </bpel:invoke>
         </bpel:while>
         <bpel:wait name="Wait">
            <ext:activityId>wait 1</ext:activityId>
            <ext:activityState>Skipped</ext:activityState>
            <ext:projectionType>Casual</ext:projectionType>
            <bpel:for>PT10M</bpel:for>
         </bpel:wait>
         <bpel:invoke name="summarizeResults">
```

```
                <ext:activityId>invoke 7</ext:activityId>
                <ext:activityState>Skipped</ext:activityState>
                <ext:projectionType>Casual</ext:projectionType>
            </bpel:invoke>
         </bpel:sequence>
      </bpel:else>
   </bpel:if>
   <bpel:reply name="productOffering" operation="productRequest"
     variable="productOffers">
      <ext:activityId>reply 1</ext:activityId>
      <ext:activityState>Inactive</ext:activityState>
      <ext:projectionType>Casual</ext:projectionType>
   </bpel:reply>
   </bpel:sequence>
</bpel:process>
```

Listing 1: BPEL-Code für das Produktanfrage Beispiel

2.2 BPMN

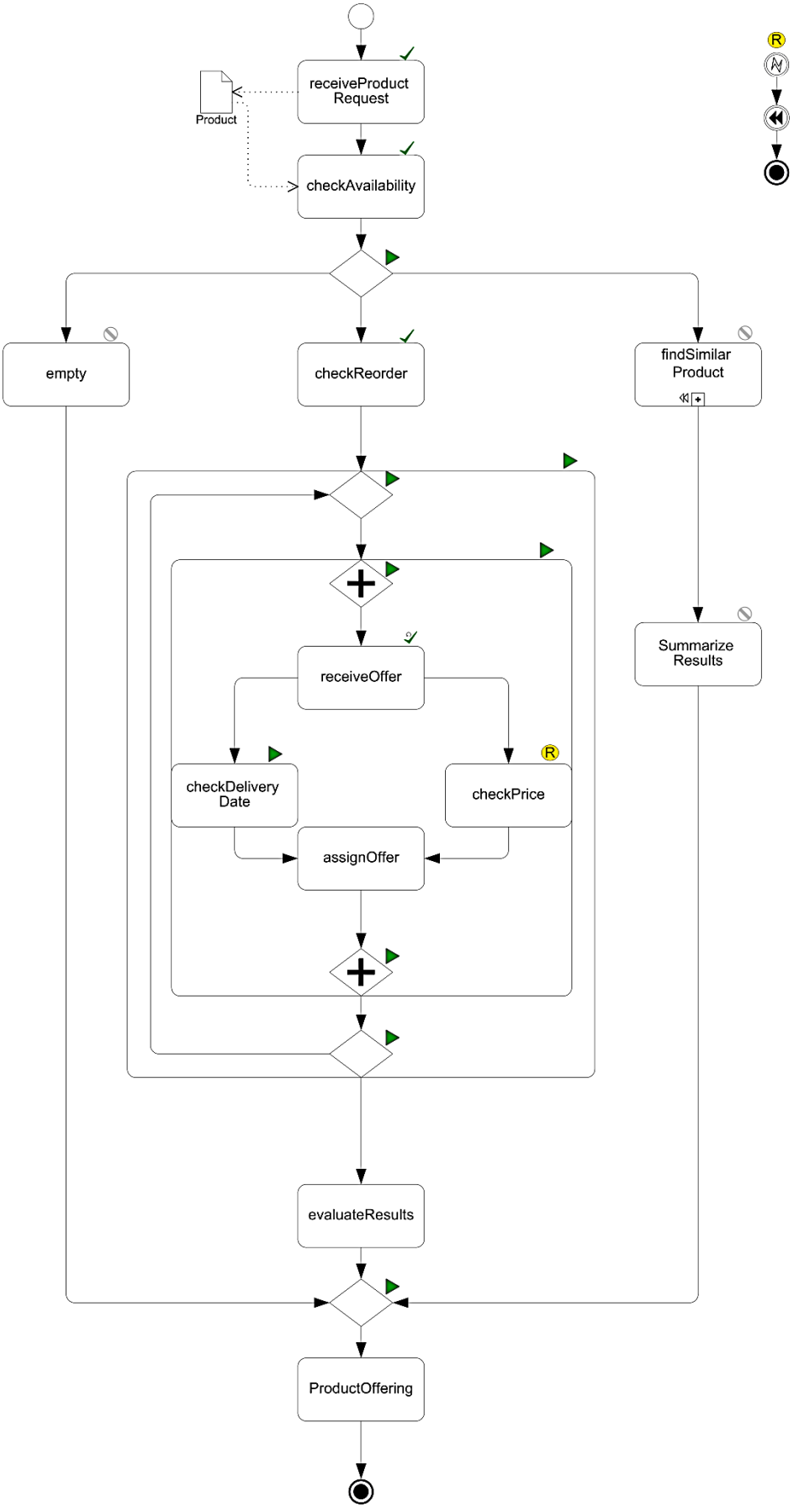
receiveProduct Request
Product
checkAvailability
empty
checkReorder
findSimilar Product
receiveOffer
checkDelivery Date
checkPrice
assignOffer
Summarize Results
evaluateResults
ProductOffering

Abbildung 2: Das Produktanfrage Beispiel in BPMN

2.3 Beispiel der Projektion

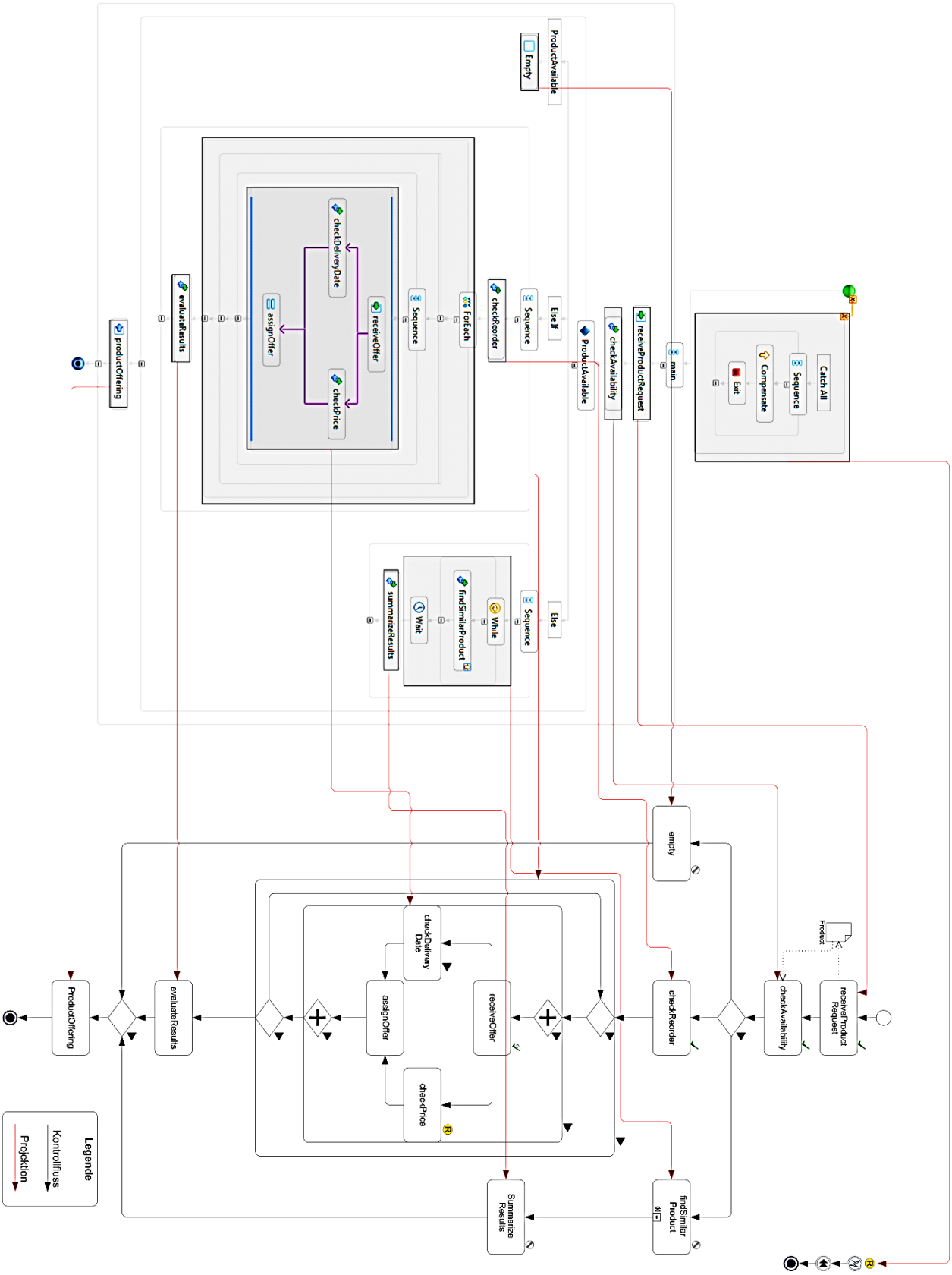

Abbildung 3: Die BPEL zu BPMN Zuordnungen für das Beispiel

3 Grundlagen

Die fachlichen Grundlagen, die für diese Arbeit relevant sind, werden im Folgenden näher beschrieben. Eine kurze Einführung in BPEL soll dem Leser einen Einblick in den Aufbau und die Verwendung der Sprache geben. Um eine Grundlage für die weitere Arbeit zu schaffen, werden die wichtigsten Sprachkonstrukte aus BPEL näher erläutert. Darauf folgt dann eine Einführung in BPMN, sowie eine Erläuterung der, für diese Arbeit relevanten, Sprachkonstrukte aus BPMN [4]. Die Erläuterungen der Sprachkonstrukte beider Sprachen dienen zur Veranschaulichung und zur Einführung in das Thema und stellen keinen Ersatz der Spezifikationen dar. Da die Spezifikationen nur auf Englisch verfügbar sind, werden zur Unterstützung des Verständnisses der Leser die Beschreibungen auf Deutsch formuliert. Dabei werden die Informationen aus den Spezifikationen eingegrenzt. Für weiterführende Informationen wird auf die Spezifikationen verwiesen.

3.1 Definitionen

3.1.1 Projektion

Eine Projektion ist die Übertragung des Zustandes einer Aktivität in BPEL auf ein Konstrukt in BPMN.

3.1.2 Soundness

Die Soundness einer Projektion wird durch folgende Regeln bestimmt:

1. die Projektion muss surjektiv sein. Es muss also gelten:

    X Elemente im BPEL Model

    Y Elemente im BPMN Model

    f() Zustandszusammenführungsfunktion

Für alle Elemente in Y muss es mindestens ein Element in X geben, für das gilt f(x) = y.

2. die Projektion muss gewisse Konsistenzregeln einhalten:

    (1) Eine Aktivität kann nur den Zustand *Executing* haben, wenn die vorherige Aktivität im Zustand *Completed* ist.

    (2) Ist eine Aktivität im Zustand *Compensated* oder *Faulted*, dann darf dieser Zustand nicht mehr geändert werden.

    (3) Gateways bleiben im Zustand *Executing* solange nicht alle Aktivitäten zwischen den beiden zusammengehörenden Gateways im Zustand *Completed* sind.

    (4) Alle Aktivitäten auf einem toten Pfad werden automatisch in den Zustand *Skipped* gesetzt.

    (5) Eine Aktivität mit mehreren Vorgängern kann erst ausgeführt werden, wenn alle Vorgänger im Zustand *Completed* sind.

    (6) Der Zustand von Schleifen wird aus den Zuständen aller enthaltenen Aktivitäten berechnet und wird erst auf *Completed* gesetzt, wenn alle enthaltenen Zustände *Completed* sind.

    (7) Für jede Aktivität muss es die Möglichkeit geben ausgeführt zu werden.

3.1.3 Vollständigkeit

Vollständigkeit wird erreicht, wenn es für jede Projektion eine Zustandsüberführungsregel gibt.

3.2 BPEL

BPEL ist eine XML-basierte Sprache zur Beschreibung von Abläufen aus technischer Sicht. Mit BPEL können Dienste, welche über Web Services angeboten werden, innerhalb eines Geschäftsprozesses orchestriert werden. Unter Orchestrierung versteht man die Anordnung von unterschiedlichen Web Services in einem ausführbaren Geschäftsprozess. Im Bereich der Orchestrierung von Web Services ist BPEL der de-facto Standard. Für die Modellierung von automatisierten Geschäftsprozessen stehen in BPEL standardisierte Sprachkonstrukte zur Verfügung. Diese Sprachkonstrukte sind über ein XML-Schema definiert. Die aktuelle Version ist BPEL 2.0 [1]. Es gibt zudem Erweiterungen von BPEL, wie zum Beispiel BPEL4People [6], das die Integration von Personen im Geschäftsprozess erlaubt.

Die Tatsache, dass es keine standardisierte, visuelle Notation gibt, ist ein viel diskutiertes Problem von BPEL. Im Moment stellt jeder Hersteller eines BPEL Frameworks eine eigene Notation zur Verfügung. [7] zeigt, dass in der Praxis zur visuellen Modellierung häufig BPMN [4] verwendet und das BPMN-Model dann durch eine Transformation in ein BPEL-Model umgewandelt wird. Auf BPMN wird in Kapitel 3.4 näher eingegangen.

Da es bereits verschiedene Quellen zu der Entstehungsgeschichte von BPEL oder zur Zielsetzung der Sprache gibt, wird in dieser Arbeit nicht näher auf diese Themen eingegangen. Für weiterführende Informationen zur Entstehungsgeschichte wird an dieser Stelle auf [8] und [9] und zur Zielsetzung auf [10] verwiesen.

Ein Beispiel für den allgemeinen Aufbau eines BPEL-Modells wurde in Kapitel 2 gegeben und dieses Beispiel wird im nächsten Unterkapitel weiter verwendet um die einzelnen Konstrukte näher zu beschreiben.

3.3 Konstrukte in BPEL

In diesem Unterkapitel werden die in der BPEL-Spezifikation beschriebenen Konstrukte erläutert und durch Beispiele veranschaulicht. Die Arbeit orientiert sich bei der Unterteilung der Konstrukte an der BPEL2.0-Spezifikation. Basis Aktivitäten, Strukturierte Aktivitäten, Scopes, Variablen und andere Konstrukte werden im Folgenden beschrieben und erläutert. Zu jedem Konstrukt werden eine kurze Beschreibung und soweit möglich ein Teilbeispiel aus dem großen Beispiel in Kapitel 2 angegeben.

3.3.1 Basis Aktivitäten

3.3.1.1 <assign>

Die <assign> Aktivität wird zum einen verwendet um Variablen Werte zuzuweisen und zum anderen um neue Daten via Ausdrücken zu konstruieren und einzufügen. Ein <assign> Konstrukt kann zwischen einer und beliebig vielen Zuweisungen enthalten. Jede Zuweisung wird dabei als einzelnes <copy> Element beschrieben. Während der Ausführung werden entweder alle oder keine der Zuweisungen ausgeführt.

In Listing 2 ist ein <assign> Beispiel mit einem einzelnen <copy> Element aus dem Beispiel aus Kapitel 2 zu sehen.

```
<bpel:assign validate="no" name="saveReorderOffers">
   <bpel:copy>
      <bpel:from>$productOffers</bpel:from>
      <bpel:to>$product.reorderOffers</bpel:to>
   </bpel:copy>
</bpel:assign>
```

Listing 2: <assign>

3.3.1.2 <empty>

Die <empty> Aktivität ist eine leere Aktivität, die keine auszuführende Funktion hat. Ein Anwendungsbeispiel ist die Fehlerbehandlung bei der ein Fehler abgefangen und unterdrückt werden muss. Des Weiteren kann die <empty> Aktivität einen Synchronisationspunkt in einem <flow> darstellen.

Ein Beispiel für eine <empty> Aktivität aus dem Beispiel in Kapitel 2 ist in Listing 3 zu sehen.

```
<bpel:empty name="Empty">
</bpel:empty>
```

Listing 3: <empty>

3.3.1.3 <exit>

Zum sofortigen Beenden einer BPEL-Prozessinstanz wird die <exit> Aktivität verwendet. Alle laufenden Aktivitäten müssen sofort beendet werden, ohne <terminationHandler>, <faultHandler> oder <compensationHandler> zu beachten.

Listing 4 zeigt ein Beispiel für die <exit> Aktivität aus dem großen Beispiel in Kapitel 2.

```
<bpel:exit name="Exit">
</bpel:exit>
```

Listing 4: <exit>

3.3.1.4 <invoke>

Zum Aufrufen von Web Services, die von Service Providern angeboten werden, wird die <invoke> Aktivität verwendet. Die Aktivität kann synchron, das bedeutet es wird auf die Antwort gewartet bevor die Ausführung weitergeht, oder asynchron, es wird ohne auf die Antwort zu warten weiter ausgeführt, aufgerufen werden. Innerhalb einer <invoke> Aktivität können Elemente zur Fehlerbehandlung oder zur Kompensation definiert werden, ohne dass ein <scope> verwendet werden muss.

In Listing 5 und 6 folgen zwei Beispiele für die <invoke> Aktivität. Das erste Beispiel zeigt einen kurzen Aufruf per <invoke>, im zweiten Beispiel wird zusätzlich ein Element zur Kompensation, ein <compensationHandler>, verwendet.

```
<bpel:invoke name="checkAvailability" inputVariable="product"
  outputVariable="availabilityInfo"
  operation="requestAvailability">
</bpel:invoke>
```

Listing 5: <invoke> ohne <compensationHandler>

```
<bpel:invoke name="findSimilarProduct"
  inputVariable="product"
  outputVariable="similarProductRequest"
  operation="findSimilarProduct">
   <bpel:compensationHandler>
      <bpel:invoke name="cancelSearch"
        operation="cancelProductSearch">
      </bpel:invoke>
   </bpel:compensationHandler>
</bpel:invoke>
```

Listing 6: <invoke> mit <compensationHandler>

3.3.1.5 <receive>

Mit der <receive> Aktivität wird auf eine bestimmte Nachricht gewartet. Ist das Attribut createInstance auf den Wert yes gesetzt, dann wird mit dieser <receive> Aktivität ein BPEL-Prozess gestartet.

In Listing 7 ist ein Beispiel für die <receive> Aktivität zu sehen.

```
<bpel:receive name="receiveProductRequest" variable="product"
  createInstance="yes" operation="sendProductRequest">
</bpel:receive>
```

Listing 7: <receive>

3.3.1.6 <reply>

Um in einer Request-Response Interaktion auf eine Anfrage durch <receive>, <pick> oder <extensionActivity> zu antworten wird die <reply> Aktivität verwendet.

In Listing 8 ist eine <reply> Aktivität aus dem Beispiel in Kapitel 2 zu sehen.

```
<bpel:reply name="productOffering" operation="productRequest"
  variable="productOffers">
</bpel:reply>
```

Listing 8: <reply>

3.3.1.7 <rethrow>

Mit der <rethrow> Aktivität kann ein aktiver <faultHandler> einen Fehler an den übergeordneten <faultHandler> weiterreichen.

Im Beispiel aus der BPEL-Spezifikation [1, S.132] leitet die <rethrow> Aktivität alle Fehler an den übergeordneten <faultHandler> weiter.

```
<catchAll>
   <sequence>
      <compensate />
      <rethrow />
   </sequence>
</catchAll>
```

Listing 9: <rethrow>

3.3.1.8 <throw>

Wenn ein Geschäftsprozess explizit einen internen Fehler signalisieren muss, wird die <throw> Aktivität verwendet. Jeder Fehler muss mit einem QName identifizierbar sein. Optional können weitere Informationen über den Fehler an die <faultHandlers> weitergegeben werden, um Fehlermeldungen an andere Services zu erstellen.

In Listing 10 folgt ein einfaches Beispiel für eine <throw> Aktivität, die keine Fehlerdaten weitergibt [1, S. 95].

```
<throw xmlns:FLT="http://example.com/faults"
   faultName="FLT:OutOfStock" />
```

Listing 10: <throw>

3.3.1.9 <wait>

Mit der <wait> Aktivität kann entweder für eine bestimmte Zeitdauer, mit <for>, oder auf einen bestimmten Zeitpunkt, mit <until>, gewartet werden.

Listing 11 zeigt eine <wait> Aktivität aus dem Beispiel in Kapitel 2 bei dem 10 Minuten gewartet wird.

```
<bpel:wait name="Wait">
   <bpel:for>PT10M</bpel:for>
</bpel:wait>
```

Listing 11: <wait>

3.3.2 Strukturierte Aktivitäten

3.3.2.1 <flow>

Mit Hilfe der <flow> Aktivität können parallele Abläufe modelliert werden. Die <flow> Aktivität ist erst beendet, wenn alle Aktivitäten im <flow> beendet sind. Durch <link>, <transitionCondition>, <joinCondition> und Dead-Path-Eliminierung kann das Verhalten innerhalb der <flow> Aktivität beeinflusst werden.

In Listing 12 ist eine <flow> Aktivität mit vier internen Aktivitäten aus dem Beispiel in Kapitel 2 zu sehen.

```
<bpel:flow name="Flow">
   <bpel:receive name="receiveOffer"
     variable="productOffer" operation="offerRequest">
      <bpel:sources>
         <bpel:source linkName="link1"></bpel:source>
         <bpel:source linkName="link2"></bpel:source>
      </bpel:sources>
   </bpel:receive>
   <bpel:invoke name="checkDeliveryDate"
     inputVariable="product"
     outputVariable="productDeliveryDate"
     operation="getDeliveryDate">
      <bpel:targets>
         <bpel:target linkName="link1"></bpel:target>
      </bpel:targets>
      <bpel:sources>
         <bpel:source linkName="link3"></bpel:source>
      </bpel:sources>
```

```
   </bpel:invoke>
   <bpel:assign validate="no" name="assignOffer">
      <bpel:copy>
         <bpel:from>$productDeliveryDate</bpel:from>
         <bpel:to>$product.deliveryDate</bpel:to>
      </bpel:copy>
      <bpel:copy>
         <bpel:from>$productPrice</bpel:from>
         <bpel:to>$product.price</bpel:to>
      </bpel:copy>
      <bpel:targets>
         <bpel:target linkName="link3"></bpel:target>
         <bpel:target linkName="link4"></bpel:target>
      </bpel:targets>
   </bpel:assign>
   <bpel:invoke name="checkPrice" inputVariable="product"
     outputVariable="productPrice" operation="getPrice">
      <bpel:targets>
         <bpel:target linkName="link2"></bpel:target>
      </bpel:targets>
      <bpel:sources>
         <bpel:source linkName="link4"></bpel:source>
      </bpel:sources>
   </bpel:invoke>
   <bpel:links>
      <bpel:link name="link1"></bpel:link>
      <bpel:link name="link2"></bpel:link>
      <bpel:link name="link3"></bpel:link>
      <bpel:link name="link4"></bpel:link>
   </bpel:links>
</bpel:flow>
```

Listing 12: <flow>

3.3.2.1.1 <link>

Mit <link>s wird innerhalb der <flow> Aktivität der Kontrollfluss festgelegt. Ein <link> hat dabei einen Ursprung und ein Ziel. In einer Aktivität wird der Ursprung durch ein Unterelement im <sources> Element dargestellt und ein Ziel durch ein Unterelement im <targets> Element. Die BPEL2.0 Spezifikation beschreibt einige Einschränkungen, welche für das <link> Element gelten. Zum einen wird die Erzeugung von Zyklen ausgeschlossen. Eine Aktivität mit einem logischen Vorgänger zu verlinken und damit Zyklen zu erzeugen ist untersagt. Des Weiteren dürfen <link>s das <flow> Konstrukt, in welchem sie definiert wurden, nicht verlassen. Um <link>s innerhalb von <while>, <repeatUntil>, <forEach>, <eventHandlers> oder <compensationHandler> verwendet zu können, ist es erforderlich, dass die <link>s in einem <flow> Element innerhalb der Konstrukte eingebettet werden, damit sie die umgebenden Konstrukte nicht verlassen können. In den Konstrukten <catch> und <catchAll>, sowie dem <terminationHandler> sind eingehende <link>s untersagt.

In Listings 12 sind Beispiele für <link>s zu sehen.

3.3.2.1.2 <transitionCondition>

Um die Übergangsbedingung für den Ursprung eines <link>s festlegen zu können, muss die betreffende Aktivität im <source> Unterelement eine <transitionCondition> beinhalten. Nach Beendigung der Aktivität wird diese Bedingung ausgewertet und dabei entweder auf true oder false gesetzt. Wird die <transitionCondition> nicht angegeben wird immer auf true ge-

setzt. Der Zustand eines <link>s wird nicht nur durch die <transitionCondition> festgelegt, sondern ist auch noch abhängig von Fehlern und der Dead-Path-Elimination.

Ein einfaches Beispiel aus der BPEL-Spezifikation [1, S.182].

```
<source linkName="assess-to-setMessage">
      <transitionCondition>
            $risk.level='low'
      </transitionCondition>
</source>
```

Listing 13: <transitionCondition>

3.3.2.1.3 <joinCondition>

Das Gegenstück zur <transitionCondition> im Ziel eines <link>s ist die <joinCondition>. Hat ein <link> in seinem <targets> Unterelement eine <joinCondition>, dann wird in diesem Element festgelegt, unter welchen Bedingungen die Aktivität ausgeführt wird. Unter Auswertung der Zustände der eingehenden <link>s, wird ein bool'scher Ausdruck auf true oder false gesetzt, das bedeutet zur Auswertung müssen die Zustände der <link>s bereits evaluiert sein. Ist die <joinCondition> true, dann wird die Aktivität ausgeführt. Ist die <joinCondition> nicht explizit angegeben, dann wird ein logisches ODER verwendet, um zu entscheiden, ob die Aktivität ausgeführt wird. In diesem Fall muss also nur ein eingehender <link> den Zustand true haben. Wird die <joinCondition> mit false ausgewertet, dann wird die Dead-Path-Elimination angewendet.

Das Beispiel aus [1, S.110] zeigt eine <joinCondition>, die eine AND-Verknüpfung für „buyToSettle" und „sellToSettle" festlegt.

```
<targets>
      <joinCondition>$buyToSettle and $sellToSettle</joinCondition>
      <target linkName="buyToSettle" />
      <target linkName="sellToSettle" />
</targets>
```

Listing 14: <joinCondition>

3.3.2.1.4 Dead-Path-Elimination

Die BPEL-Engine muss Fälle, in welchen bestimmte Aktivitäten nicht mehr erreichbar sind, von selbst auflösen. Das ist die sogenannte Dead-Path-Elimination. Wie im vorherigen Kapitel beschrieben, kann eine <joinCondition> zu false ausgewertet werden, in diesem Fall muss die BPEL-Engine die Dead-Path-Elimination durchführen. Durch diesen Vorgang werden alle Zustände von <link>s, die die nun nicht auszuführende Aktivität als Ursprung haben, auf false gesetzt. Dies geschieht unabhängig von etwaigen <transitionCondition>s und kann dazu führen, dass ganze Pfade von der Ausführung ausgenommen werden. Diese Pfade nennt man Dead Paths. Damit die Dead-Path-Elimination durchgeführt wird, muss im betreffenden <flow> das Attribut suppressJoinFailure auf yes gesetzt sein, ansonsten wird, im Falle einer zu false ausgewerteten <joinCondition>, von der BPEL-Engine ein Fehler erzeugt.

3.3.2.2 <forEach>

Um eine Aktivität mehrfach ausführen zu können, verwendet man die <forEach> Aktivität. Innerhalb einem eigenen <scope> werden alle auszuführenden Aktivitäten der <forEach>-Schleife modelliert. Des Weiteren können die Ausführungen sequenziell oder parallel ausge-

führt werden und bei Bedarf kann eine <completionCondition> angegeben werden, mit welcher die Ausführung auch frühzeitig beendet werden kann.

Listing 15 zeigt eine <forEach> Schleife aus dem Beispiel in Kapitel 2. Die enthaltene <flow> Aktivität ist in Listing 12 zu sehen.

```
<bpel:forEach parallel="no" counterName="Counter"
  name="ForEach">
   <bpel:startCounterValue>1</bpel:startCounterValue>
   <bpel:finalCounterValue>
      $productOffers.count
   </bpel:finalCounterValue>
   <bpel:scope>
      <bpel:sequence>
         <bpel:flow name="Flow">
         ...
         </bpel:flow>
      </bpel:sequence>
   </bpel:scope>
</bpel:forEach>
```

Listing 15: <forEach>

3.3.2.3 <if>

Für bedingte Aktivitäten stellt die BPEL-Engine die <if> Aktivität zur Verfügung. Wie von höheren Programmiersprachen bekannt, können <else> und <elseif> Konstrukte verwendet werden, um eine komplexere bedingte Aktivität zu modellieren. Die Bedingungen werden mit dem <condition> Element angegeben.

In Listing 16 ist eine <if> Aktivität aus dem Beispiel in Kapitel 2 zu sehen. Die zur Übersichtlichkeit weggelassenen Teile können Listing 1 entnommen werden.

```
<bpel:if name="ProductAvailable">
   <bpel:condition>
      bpel:getVariableProperty('availabilityInfo',
      'inventory:level') > 0
   </bpel:condition>
   <bpel:sequence>
   ...
   </bpel:sequence>
   <bpel:elseif>
      <bpel:condition>
         bpel:getVariableProperty('availabilityInfo',
         'product:inProduction') > 0
      </bpel:condition>
      <bpel:sequence>
      ...
      </bpel:sequence>
   </bpel:elseif>
   <bpel:else>
      <bpel:sequence>
      ...
      </bpel:sequence>
   </bpel:else>
</bpel:if>
```

Listing 16: <if>

3.3.2.4 <pick>

Wenn aus mehreren alternativen Nachrichten eine ausgewählt und verarbeitet werden soll, wird die <pick> Aktivität verwendet. Im Unterelement <onMessage> wird dann die, für die spezifische Nachricht angegebene, Aktivität ausgeführt. Mit dem <onAlarm> Element lässt sich ein Timeout für die <pick> Aktivität einstellen, an dem diese dann auch ohne Eingang einer Nachricht beendet wird. Wie bei der <receive> Aktivität kann das Attribut createInstance auf yes gesetzt werden um einen BPEL-Prozess zu starten.

Ein Beispiel aus der BPEL-Spezifikation [1, S.101 f.] ist in Listing 17 zu sehen.

```
<pick>
   <onMessage partnerLink="buyer"
      portType="orderEntry"
      operation="inputLineItem"
      variable="lineItem">
      <!-- activity to add line item to order -->
   </onMessage>
   <onMessage partnerLink="buyer"
      portType="orderEntry"
      operation="orderComplete"
      variable="completionDetail">
      <!-- activity to perform order completion -->
   </onMessage>
   <!-- set an alarm to go off
        3 days and 10 hours after the last order line -->
   <onAlarm>
      <for>'P3DT10H'</for>
      <!-- handle timeout for order completion -->
   </onAlarm>
</pick>
```

Listing 17: <pick>

3.3.2.5 <repeatUntil>

Die <repeatUntil> Aktivität führt enthaltene Aktivitäten solange aus, bis die durch <condition> angegebene Bedingung erfüllt ist. Die <repeatUntil>-Schleife wird dabei mindestens einmal ausgeführt.

Listing 18 zeigt ein Beispiel für eine <repeatUntil> Schleife.

```
<bpel:repeatUntil name="checkStock">
   <ext:iterationCount>10</ext:iterationCount>
   <bpel:invoke name="checkStock">
      ...
   </bpel:invoke>
   <bpel:condition>
      bpel:getVariableProperty(`availabilityInfo`,
      `product:inStock`) < 15
   </bpel:condition>
</bpel:repeatUntil>
```

Listing 18: <repeatUntil>

3.3.2.6 <sequence>

Innerhalb der <sequence> Aktivität werden mehrere Aktivitäten sequentiell ausgeführt. Wenn alle enthaltenen Aktivitäten ausgeführt wurden, wird die <sequence> Aktivität beendet.

In Listing 19 ist eine <sequence> mit einer <while> Schleife, einer <wait> Aktivität und einer <invoke> Aktivität aus dem Beispiel in Kapitel 2 zu sehen.

```
<bpel:sequence>
   <bpel:while name="While">
      <bpel:condition>
         $similarProductRequest &ne; nil
      </bpel:condition>
      <bpel:invoke name="findSimilarProduct"
        inputVariable="product"
        outputVariable="similarProductRequest"
        operation="findSimilarProduct">
         <bpel:compensationHandler>
            <bpel:invoke name="cancelSearch"
              operation="cancelProductSearch">
            </bpel:invoke>
         </bpel:compensationHandler>
      </bpel:invoke>
   </bpel:while>
   <bpel:wait name="Wait">
      <bpel:for>PT10M</bpel:for>
   </bpel:wait>
   <bpel:invoke name="summarizeResults">
   </bpel:invoke>
</bpel:sequence>
```

Listing 19: <sequence>

3.3.2.7 <while>

Genau wie in der <repeatUntil>-Schleife kann hier, unter Berücksichtigung der Bedingung im <condition> Element, eine Aktivität mehrfach ausgeführt werden. Der Unterschied besteht darin, dass die Bedingung zu Beginn des Durchlaufs geprüft wird und es somit auch möglich ist, dass die <while>-Schleife nicht einmal durchlaufen wird.

Eine <while> Schleife aus dem Beispiel in Kapitel 2 ist in Listing 20 zu sehen.

```
<bpel:while name="While">
   <bpel:condition>
      $similarProductRequest &ne; nil
   </bpel:condition>
   <bpel:invoke name="findSimilarProduct"
     inputVariable="product"
     outputVariable="similarProductRequest"
     operation="findSimilarProduct">
      <bpel:compensationHandler>
         <bpel:invoke name="cancelSearch"
           operation="cancelProductSearch">
         </bpel:invoke>
      </bpel:compensationHandler>
   </bpel:invoke>
</bpel:while>
```

Listing 20: <while>

3.3.3 Scopes

3.3.3.1 <compensate>

Die <compensate> Aktivität darf nur innerhalb eines <compensationHandler>s, eines <terminationHandler>s oder eines <faultHandlers>, also innerhalb eines <catch> oder <catchAll>, aufgerufen werden. Mit dieser Aktivität wird in allen abgeschlossenen <scope>s die Kompensation während eines Fehlerfalles gestartet. Die betreffenden, abgeschlossenen <scope>s müssen dazu, nimmt man die XML Darstellung als Ausgangspunkt, auf derselben oder auf einer tieferen Ebene im Zweig wie der Handler liegen.

Ein Beispiel für <compensate> innerhalb eines <faultHandlers> ist in Listing zu sehen.

3.3.3.2 <compensateScope>

Für die <compensateScope> Aktivität gelten dieselben Regeln wie für die <compensate> Aktivität, bis auf die Besonderheit, dass man einen bestimmten zu kompensierenden <scope> angeben kann.

Listing 21 zeigt ein kurzes Beispiel aus [1, S.139]:

```
<compensationHandler>
   <sequence>
      <compensateScope target="S2" />
   </sequence>
</compensationHandler>
```

Listing 21: <compensateScope>

3.3.3.3 <scope>

<scope>s sind Bereiche in welchen, für die enthaltenen Aktivitäten, bestimmte gültige Einstellungen und Angaben gemacht werden. Zu diesen Einstellungen zählen: <faultHandler>, <eventHandler>, <compensationHandler>, <terminationHandler>, <correlationSets>, <partnerLinks>, <messageExchanges> und <variables>.

In Listing 22 ist eine <scope> Aktivität aus dem Beispiel in Kapitel 2 zu sehen. Die inneren Aktivitäten der <flow> Aktivität sind in Listing 12 dargestellt.

```
<bpel:scope>
   <bpel:sequence>
      <bpel:flow name="Flow">
      ...
      </bpel:flow>
   </bpel:sequence>
</bpel:scope>
```

Listing 22: <scope>

3.3.4 Variablen

3.3.4.1 <variable>

Um in BPEL Variablen zu verwendet wird das <variable> Konstrukt verwendet. Es gibt in BPEL lokale und globale Variablen. Wird ein <variable> Konstrukt innerhalb eines <scope>s verwendet, dann ist die angegebene Variable nur innerhalb dieses <scope>s gültig und man

spricht von einer lokalen Variable. Wird die Variable auf der <process> Ebene deklariert, dann ist sie allgemein gültig und man spricht von einer globalen Variable. Die Variablentypen werden in WSDL- oder XML-Schema-Dateien definiert.

Listing 23 zeigt die <variables> aus dem Beispiel in Kapitel 2.

```
<bpel:variables>
   <bpel:variable name="product" messageType="tns:productType">
   </bpel:variable>
   <bpel:variable name="availabilityInfo"
     messageType="tns:availabilityType">
   </bpel:variable>
   <bpel:variable name="productOffers" messageType="tns:productType">
   </bpel:variable>
   <bpel:variable name="productOffer" messageType="tns:productType">
   </bpel:variable>
   <bpel:variable name="productDeliveryDate" messageType="tns:date">
   </bpel:variable>
   <bpel:variable name="productPrice" messageType="tns:float">
   </bpel:variable>
   <bpel:variable name="evaluatedProductOffers"
     messageType="tns:productArrayType">
   </bpel:variable>
   <bpel:variable name="similarProductRequest"
     messageType="tns:productType">
   </bpel:variable>
</bpel:variables>
```

Listing 23: <variable>

3.3.4.2 <validate>

Um Variablen gegen ihre Datendefinitionen aus der jeweilgen XML oder WSDL zu validieren, wird die <validate> Aktivität verwendet.

Listing 24 zeigt ein Beispiel zum Validieren der zuvor genannten Variablen:

```
<validate variables=" product availabilityInfo productOffers
  productOffer productDeliveryDate productPrice
  evaluatedProductOffers similarProductRequest "  />
```

Listing 24: <validate>

3.3.5 Andere Konstrukte

3.3.5.1 <catch>

Die <catch> Aktivität ermöglicht es, während der Fehlerbehandlung, auf bestimmte Fehler zu reagieren. Hierfür muss der Bezeichner und der Nachrichtentyp des Fehlers angegeben werden.

Das folgende Beispiel eines <faultHandlers> aus [1, S.19] in Listing 25 zeigt die Fehlerbehandlung auf <process> Ebene.

```
<faultHandlers>
   <catch faultName="lns:cannotCompleteOrder"
      faultVariable="POFault"
      faultMessageType="lns:orderFaultType">
      <reply partnerLink="purchasing"
         portType="lns:purchaseOrderPT"
```

```
         operation="sendPurchaseOrder" variable="POFault"
         faultName="cannotCompleteOrder" />
   </catch>
</faultHandlers>
```

Listing 25: <catch>

3.3.5.2 <catchAll>

Alle Fehler, die nicht mit der <catch> Aktivität abgefangen und behandelt werden, können mit der <catchAll> Aktivität behandelt werden.

Im Beispiel in Listing 26 wird bei allen auftretenden Fehlern abgebrochen [1, S.162].

```
<faultHandlers>
   <catchAll>
      <exit />
   </catchAll>
</faultHandlers>
```

Listing 26: <catchAll>

3.3.5.3 <compensationHandler>

Alle für die Kompensation auszuführenden Aktionen werden im <compensationHandler> zusammengefasst. Durch die Aktivitäten im <compensationHandler> soll ein vorheriger Zustand so gut es geht wieder hergestellt werden. Zusätzlich gibt es noch die Möglichkeit einen <compensationHandler> direkt an einer <invoke> Aktivität anzubringen.

In Listing 6 wurde ein Beispiel für einen <compensationHandler> innerhalb einer <invoke> Aktivität vorgestellt. Folgend noch ein Beispiel für einen <compensationHandler> in einem <scope> [1, S.145].

```
<scope name="Q" isolated="true">
   <compensationHandler>
      <sequence name="undoQ_Seq">...</sequence>
   </compensationHandler>
   <sequence name="doQ_Seq">...</sequence>
</scope>
```

Listing 27: <compensationHandler>

3.3.5.4 <extensions>

Mit Hilfe der <extensions> kann BPEL, um Attribute, neue Aktivitäten oder ein verändertes Laufzeitverhalten der BPEL-Engine, erweitert werden. Mit dem <extensions> Element werden in dieser Arbeit zusätzliche Elemente zur Speicherung von benötigten Informationen für die Zustandsübertragung erstellt.

Ein Beispiel aus der BPEL-Spezifikation, in dem der Prozess um das Attribut „uniqueUserFriendlyName“ erweitert wird [1, S.162].

```
<extensions>
   <extension
      namespace="http://example.com/bpel/some/extension"
      mustUnderstand="yes" />
</extensions>

<receive partnerLink="homeInfoVerifier"
   operation="##opaque" variable="##opaque"
```

```
   ext:uniqueUserFriendlyName="receive verification
   result" />
```

Listing 28: <extensions>

3.3.5.5 <faultHandlers>

Wie bereits beschrieben werden mit den Konstrukten <catch> und <catchAll> Fehler abgefangen und behandelt. Diese können nur innerhalb eines <faultHandlers> verwendet werden. Einzige Ausnahme dieser Regel stellt die <invoke> Aktivität dar, hier können die beiden Konstrukte direkt angebracht werden, ohne das <faultHandlers> Konstrukt zu verwenden.

In Listing 29 ist der <faultHandlers> auf Prozessebene aus dem Beispiel in Kapitel 2 dargestellt.

```
<bpel:faultHandlers>
   <bpel:catchAll>
      <bpel:sequence>
         <bpel:compensate>
         </bpel:compensate>
         <bpel:exit name="Exit">
         </bpel:exit>
      </bpel:sequence>
   </bpel:catchAll>
</bpel:faultHandlers>
```

Listing 29: <faultHandlers>

3.3.5.6 <import>

Mit Hilfe des <import> Konstrukts wird innerhalb von BPEL Prozessen eine Abhängigkeit von externen XML Schemata oder WSDL Definitionen angezeigt. Das <import> Konstrukt wird direkt unterhalb des <process> Elements eingefügt.

Listing 30 zeigt ein Beispiel aus [1, S.169]:

```
<import importType="http://schemas.xmlsoap.org/wsdl/"
      location="shippingProperties.wsdl"
      namespace="http://example.com/shipping/properties/" />
```

Listing 30: <import>

3.3.5.7 <partnerLinks>

Damit ein BPEL Prozess mit den beteiligten Web Services, beziehungsweise Business Partnern, kommunizieren kann, werden <partnerLinks>, <partnerLinkType>s, roles und portTypes verwendet. Diese Elemente werden in dieser Arbeit nicht näher erläutert, da sie für die Zustandsübertragung nicht von Bedeutung sind. Für weiterführende Informationen wird auf [1, S.36 ff.] verwiesen.

3.3.5.8 <process>

Der <process> ist das Hauptkonstrukt eines BPEL Prozesses. Innerhalb des <process> Konstrukts werden globale Einstellungen für den BPEL Prozess vorgenommen. Dazu zählen <extensions>, <import>s, <partnerLinks>, <variables>, <correlationSets>, <messageExchanges>, <eventHandlers> und <faultHandlers>. In der BPEL Spezifikation wird das als root-context bezeichnet. Im <process> Konstrukt darf nur eine Aktivität enthalten sein. In den

meisten Fällen ist das entweder eine <sequence>, <scope> oder <flow> Aktivität. Ein <process> Konstrukt kann entweder ein *executable process* sein oder ein *abstract process*. Ein *executable process* beschreibt das Verhalten eines BPEL-Prozesses und kann auf einer BPEL-Engine ausgeführt werden. Bei einem *abstract process* dagegen lassen sich Prozessdetails verbergen um, zum Beispiel, Geschäftspartnern die nötigen Schnittstellen offen zu legen, ohne Einsicht auf die inneren Abläufe zu geben. Zum Verbergen von Details wird die <opaqueActivity> verwendet.

In Kapitel 2 wurde ein Beispiel für einen kompletten <process> gezeigt. Weitere Beispiele sind in [1, S.19 ff.] zu finden.

3.3.5.9 <terminationHandler>

Wird der komplette BPEL Prozess oder nur ein <scope> durch einen Fehler beendet, bietet der <terminationHandler> die Möglichkeit, durch Ausführung der darin enthaltenen Aktivitäten, den durch den Fehler entstandenen Schaden zu begrenzen.

Das Beispiel in Listing 31 zeigt den Default-<terminationHandler> [1, S.132].

```
<terminationHandler>
   <compensate />
</terminationHandler>
```

Listing 31: <terminationHandler>

3.3.5.10 <eventHandlers>

Um auf Ereignisse während der Ausführung eines BPEL-Prozesses zu reagieren werden die <eventHandlers> verwendet. Es muss sich mindestens ein Element vom Typ <onEvent> oder vom Typ <onAlarm> darin befinden und die in jenem Element enthaltene Aktivität muss vom Typ <scope> sein. <eventHandlers> können entweder direkt in das <process> Konstrukt oder in ein <scope> Konstrukt integriert werden.

3.3.5.10.1 <onEvent>

Ein <onEvent> Element in einem <eventHandlers> Konstrukt reagiert auf das Eintreffen einer bestimmten Nachricht.

In der BPEL-Spezifikation ist ein Beispiel zur Veranschaulichung des <onEvent> Elements zu finden [1, S.141].

```
<process name="orderCar">
   ...
   <eventHandlers>
      <onEvent partnerLink="buyer"
         portType="ns:car"
         operation="haltOrder"
         messageType="ns:haltOrderMsgType"
         variable="haltDetails">
         <scope>
            <exit />
         </scope>
      </onEvent>
      ...
   </eventHandlers>
   ...
</process>
```

Listing 32: <eventHandlers> mit <onEvent>

3.3.5.10.2 <onAlarm>

Ein <onAlarm> Element wird für zeitgesteuerte Ereignisse eingesetzt. Die bereits bekannten Elemente <for> oder <until> können darin enthalten sein. Es gibt zwei Unterschiede bei der Verwendung dieser Elemente in einem <onAlarm> Element im Vergleich zur Verwendung in einer <pick> Aktivität. Der eine Unterschied ist, dass mit <repeatEvery> die Ausführung solange wiederholt ausgeführt werden kann, bis der <process> oder der <scope>, der diese <eventHandlers> beinhaltet, beendet ist. Und der zweite Unterschied ist, dass innerhalb von <onAlarm> Elementen nur ein <scope> Konstrukt verwendet werden darf.

Ein kurzes <eventHandlers> Beispiel mit dem <onAlarm> Element aus [1, S.142]:

```
<eventHandlers>
   <onAlarm>
      <for>$orderDetails.processDuration</for>
      ...
   </onAlarm>
   ...
</eventHandlers>
```

Listing 33: <eventHandlers> mit <onAlarm>

3.4 BPMN 2.0

BPMN [4] ist eine Sprache zur graphischen Modellierung von Geschäftsprozessen. Es existieren noch andere Sprachen zur visuellen Modellierung von Geschäftsprozessen, aber durch die große Anzahl namhafter Unternehmen [4, S.17 ff.], die an der BPMN-Spezifikation beteiligt sind, ist BPMN die am häufigsten verwendete Sprache zur visuellen Modellierung von Geschäftsprozessen, also der quasi Standard. Die OMG, Object Management Group [11], hat die Spezifikation für BPMN definiert und veröffentlicht, wobei die aktuelle Version BPMN2.0 ist. BPMN wurde entwickelt, um Geschäftsleuten die Erstellung von einfach verständlichen, grafischen Darstellungen ihrer Geschäftsprozesse zu ermöglichen. Des Weiteren soll BPMN eine Brücke über die Kluft zwischen dem Entwurf und der Implementierung von Geschäftsprozessen erstellen und die Kommunikation zwischen Geschäftsleuten und technisch orientierten Mitarbeitern, wie zum Beispiel Programmierern, vereinfachen.

In BPMN wird eine Tokensemantik ähnlich der in Petri-Netzen verwendet. Ein Token kann man sich dabei als ein Element vorstellen, das während der Ausführung des Prozesses die einzelnen Konstrukte durchläuft. Beim Starten des Prozesses wird ein Token bildlich auf das Start-Event gelegt und wird dann von Konstrukt zu Konstrukt geschoben. An Verzweigungen kann das Token verschiedene Wege einschlagen und es werden nur die Konstrukte ausgeführt, auf denen das Token landet. Ist eine parallele Ausführung, zum Beispiel durch ein Parallel Gateway, vorgesehen, so wird an jeden Pfad ein Token weitergegeben und diese durchlaufen den Prozess dann unabhängig voneinander. Alle Tokens müssen an einem End-Event konsumiert werden um den Prozess zu beenden. Weiterführende Informationen über die Tokensemantik in BPMN sind in der Spezifikation zu finden [4, S.427 ff.].

Auf die genaue Entstehungsgeschichte von BPMN wird in dieser Arbeit nicht näher eingegangen. Für explizitere Informationen über das Themengebiet wird auf die Homepage der Object Management Group [11] und auf die BPMN-Spezifikation [4] verwiesen.

3.5 Konstrukte in BPMN

Die in der BPMN-Spezifikation [4] beschriebenen Konstrukte werden in diesem Unterkapitel erläutert. Um den Umfang einzugrenzen werden die Konstrukte hier nur durch kurze Beschreibungen erläutert. Zur ausführlicheren Beschreibung wird auf die BPMN-Spezifikation verwiesen und hierfür werden zu allen Konstrukten die Seitenangaben in der Spezifikation angegeben. In diesem Unterkapitel findet keine nähere Beschreibung der einzelnen Attribute statt. Diese Informationen sind ebenfalls der BPMN-Spezifikation zu entnehmen. Für weiterführende Informationen wird das Buch „BPMN Method & Style“ von Bruce Silver [12] empfohlen.

3.5.1 Events

Wie in der BPMN-Spezifikation beschrieben (vgl. [4, S. 233 ff.]), ist ein Event ein Ereignis, das während der Ausführung eines Geschäftsprozesses eintritt. Diese Events beeinflussen den weiteren Ablauf des Geschäftsprozesses und haben normalerweise einen Auslöser, im Weiteren Trigger genannt, oder eine Auswirkung, im Weiteren Result genannt. Ein Event kann viele verschiedene Dinge darstellen, wie zum Beispiel den Start eines Tasks, das Ende eines Tasks, die Zustandsänderung eines Dokuments oder eine eingehende Nachricht.

Es gibt drei Haupttypen von Events: Das Start-Event, siehe Abbildung 4, das End-Event, siehe Abbildung 5, und das Intermediate-Event, siehe Abbildung 6. Ein Start-Event wird als

Kreis mit einfachem Rand dargestellt und gibt an wo ein Geschäftsprozess startet. Ein End-Event wird als Kreis mit fettem Rand dargestellt und gibt an wo ein Geschäftsprozess endet. Der dritte Haupttyp, das Intermediate-Event, wird als Kreis mit doppeltem Rand dargestellt und wird überall dort eingesetzt, wo während des Geschäftsprozesses ein Ereignis eintritt.

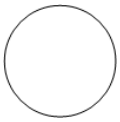

Abbildung 4: Start-Event

Abbildung 5: End-Event

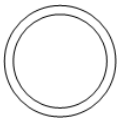

Abbildung 6: Intermediate-Event

Die drei Haupttypen werden in zwei Arten unterteilt. Events mit einem Trigger werden als catching Event, sie warten auf den Eingang eines Events, und Events mit einem Result werden als throwing Events, sie sind der Auslöser eines Events, bezeichnet. Während Start-Events immer catching Events und End-Events immer throwing Events sind, können Intermediate-Events von beiden Arten sein. Zur Veranschaulichung der beiden Arten in der Darstellung werden catching Events als normal gezeichnetes Symbol und throwing Events als ausgemaltes Symbol innerhalb des Kreises dargestellt.

In dieser Arbeit wird nur eine Teilmenge von BPMN, wie sie in [13] definiert wurde, beschrieben. Die folgende Abbildung zeigt eine Übersicht der verschiedenen Event-Typen. Diese werden anschließend näher erläutert.

	„Catching“	**„Throwing“**
Message		
Timer		
Error		
Compensation		
Terminate		

Tabelle 2: Übersicht über die verwendeten Events aus BPMN2.0

3.5.1.1 Message-Event

Ist der Message-Event ein catching Event, dann wird auf eine eingehende Nachricht gewartet und anschließend entsprechend dieser Nachricht eine Aktivität gestartet. Handelt es sich um einen throwing Event, dann wird eine Nachricht versendet. Ist der Message-Event am Rand einer Aktivität angebracht, dann handelt es sich um eine Ausnahmebehandlung.

Abbildung 7: Message Events

3.5.1.2 Timer-Event

Zum Starten von Prozessen zu einer bestimmten Zeit oder nach einem Zeitplan wird der Timer-Event verwendet. Ist es ein Intermediate-Event dann kann der Timer-Event auch zur Ablaufverzögerung eines Prozesses verwendet werden. Angebracht an den Rand einer Aktivität dient er zur Ausnahmebehandlung zu einem bestimmten Zeitpunkt oder nach Ablauf einer Zeitperiode.

Abbildung 8: Timer Events

3.5.1.3 Error-Event

Der Error-Event wird zur Fehlerbehandlung genommen. Es kann am Rand einer Aktivität als Intermediate-Event oder als End-Event auftreten. Ist der Event ein Intermediate-Event, dann wird auf ein Fehlerereignis gewartet. Als End-Event wird ein Fehler dem zugehörigen catching Event übermittelt.

Abbildung 9: Error Events

3.5.1.4 Compensation-Event

Wird der Compensation-Event als Catching-Event am Rand einer Aktivität verwendet, dann wird, sobald eine Kompensation notwendig ist, die angegebene Ausführung gestartet. Wird der Compensation-Event dagegen als Throwing-Event verwendet, dann signalisiert dieser, dass eine Kompensation notwendig ist.

Abbildung 10: Compensation Events

3.5.1.5 Terminate-Event

Mit dem Terminate-Event werden alle noch laufenden Aktivitäten im aktuellen Prozess sofort beendet. Beim Beenden durch den Terminate-Event werden keine weiteren Behandlungen, wie zum Beispiel die Kompensation, mehr gestartet.

Abbildung 11: Terminate Event

3.5.2 Aktivität

In BPMN ist eine Aktivität ([4, S.151 ff.]) ein Arbeitsschritt, der während eines Geschäftsprozesses ausgeführt wird. Dabei kann es sich um einen einzelnen, atomaren Arbeitsschritt handeln, ein Task, oder um mehrere zusammengesetzte Arbeitsschritte, dabei spricht man dann von einem Sub-Prozess. Dargestellt werden Aktivitäten durch ein Rechteck mit abgerundeten Ecken. Marker am unteren Rand des Rechtecks geben Aufschluss darüber, welches Verhalten die Aktivität hat. Mit einem Label auf der Rechtecksfläche kann eine Beschreibung der Aktivität angegeben werden.

Abbildung 12: Piktogramm einer Aktivität

3.5.3 Task

Ein einzelner Arbeitsschritt, ein Task ([4, S.156 ff.]), kann drei verschiedene Marker haben. Einen Loop-Marker, einen Multi-Instance-Marker oder einen Compensation-Marker. Der Loop-Marker hat eine wiederholende, sequenzielle Ausführungssemantik und der Multi-Instance-Marker eine parallele, mehrfach Ausführungssemantik. Somit können die beiden Marker nicht gemeinsam auftreten, da sich die Semantik überschneiden würde. Der Compensation-Marker gibt an, dass der Task eine Kompensationsfunktion hat. Der Compensation-Marker kann gemeinsam mit dem Loop-Marker oder dem Multi-Instance-Marker auftreten. Der Multi-Instance Marker wird in dieser Arbeit nicht weiter verwendet.

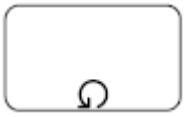

Abbildung 13: Loop-Marker

Abbildung 14: Compensation-Marker

3.5.3.1 Sub-Prozess

Ein Sub-Prozess ([4, S.173 ff.]) ist ein Teil-Prozess innerhalb des Geschäftsprozesses. Er kann wie der Prozess beliebig viele Aktivitäten, Gateways, Events und Sequence-Flows beinhalten. Ein Sub-Prozess kann in zweierlei Arten angezeigt werden. Zum einen die „collapsed“ Ansicht, bei der die Details des Sub-Prozesses versteckt sind. Die Darstellung des Sub-Prozesses beinhaltet in diesem Fall einen weiteren Marker am unteren Rand, ein Plus-Zeichen umgeben von einem kleinen Rechteck. Die zweite Ansicht ist die „expanded“ Ansicht. In dieser sieht man die Details des Sub-Prozesses umgeben von einem Rechteck mit abgerundeten Ecken.

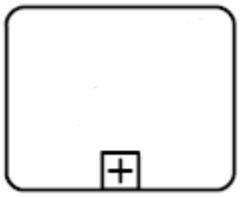

Abbildung 15: collapsed Sub-Prozess

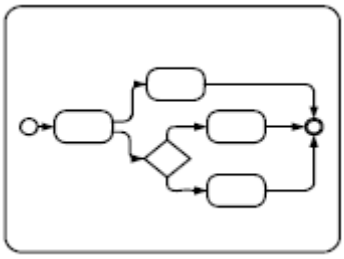

Abbildung 16: expanded Sub-Prozess

Zusätzlich zum „collapsed“-Marker werden vier weitere Marker definiert. Der Loop-Marker, der Compensation-Marker, der Error-Marker und der Termination-Marker. Wie beim Task hat auch hier der Loop-Marker eine wiederholende, sequenzielle Ausführungssemantik. Der Compensation-Marker gibt an, dass der Sub-Prozess als Kompensation von zuvor ausgeführten Aktivitäten genutzt wird. Der Error-Marker wird eingeführt um den <faultHandlers> aus BPEL darstellen zu können und der Termination-Marker wird für die Darstellung des <terminationHandler>s verwendet.

Abbildung 17: Loop

Abbildung 18: Compensation

Abbildung 19: Error

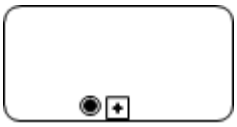

Abbildung 20: Termination

3.5.4 Gateways

Gateways ([4, S.287 ff.]) dienen zur Steuerung der Ablaufreihenfolge der Aktivitäten. Mit deren Hilfe können Verzweigungen und Entscheidungen, sowie Zusammenführungen dargestellt werden. Ein Gateway wird in BPMN 2.0 durch eine Raute abgebildet. Die jeweils spezifische Semantik wird durch einen Marker innerhalb der Raute angegeben. Wie in [4, S.90 ff.] können Gateways Null oder mehr eingehende Sequence Flows haben. Wenn das Gateway keinen eingehenden Sequence Flow hat und es keinen Start-Event für den Prozess gibt, dann soll das verzweigende Verhalten des Gateways beim Instanziieren des Prozesses durchgeführt werden. Genauso können Gateways Null oder mehr ausgehende Sequence Flows haben. Aber dabei ist zu beachten, dass ein Gateway entweder mehrere eingehende oder mehrere ausgehende Sequence Flows haben muss.

3.5.4.1 Exclusive Gateway

Das verzweigende Exclusive Gateway stellt eine Entscheidung im Prozessablauf dar. Es werden alternative Pfade erstellt, wobei in jeder Prozessinstanz nur einer der Pfade genommen werden kann. Eine Entscheidung kann man sich als Frage vorstellen, die zu einem bestimmten Zeitpunkt im Prozess gestellt wird. Diese Frage hat eine definierte Menge an alternativen Antworten. Jede Antwort hat eine zugehörige Bedingung, welche zu einem bestimmten ausgehenden Sequence Flow gehört. Exclusive Gateways können einen Marker haben, der wie ein großgeschriebenes X aussieht. In der BPMN 2.0 Spezifikation wird darauf hingewiesen, dass ein Diagramm konsistent in der Verwendung dieses internen Markers sein sollte. Es sollte also nicht vorkommen, dass in einem Diagramm Gateways mit und Gateways ohne Marker verwendet werden.

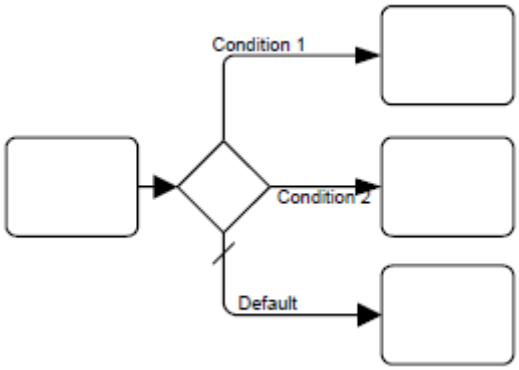

Abbildung 21: Exclusive Gateway ohne Marker

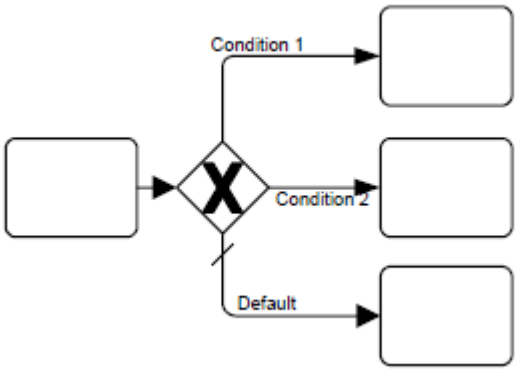

Abbildung 22: Exclusive Gateway mit Marker

Wie in den beiden Abbildungen zu sehen ist, kann ein Default-Pfad angegeben werden. Wurden alle Bedingungen auf den Wahrheitswert *false* evaluiert, wird dieser Default-Pfad verwendet. Wird kein Default-Pfad spezifiziert und alle Bedingungen sind auf *false* evaluiert, so wird ein Runtime Error erzeugt.

Ein zusammenführendes Exclusive Gateway wird zur Zusammenfassung von mehreren alternativen Pfaden verwendet. Jeder eingehende Sequence Flow wird zum ausgehenden Sequence Flow weitergeleitet.

3.5.4.1.1 Event-Based Gateway

Wie der Name schon sagt, sind Event-Based Gateways abhängig von eintretenden Ereignissen. Ein bestimmtes Ereignis, meistens eine empfangene Nachricht, bestimmt welcher Pfad weiter genommen wird. Wenn zum Beispiel eine Firma auf die Antwort eines Kunden wartet, dann wird es zwei unterschiedliche Mengen von Aktivitäten geben, je nachdem welche Antwort vom Kunden kommt. Die Kundenentscheidung bestimmt also den weiteren Pfad im Prozess. Das Empfangen einer Nachricht kann als Intermediate Event mit einem Nachrichten-Trigger modelliert werden. Neben Nachrichten können auch Timer als Trigger verwendet werden.

Das Piktogramm des Event-Based Gateways wird als Raute mit dem Symbol für Multiple Intermediate Events im Innern dargestellt.

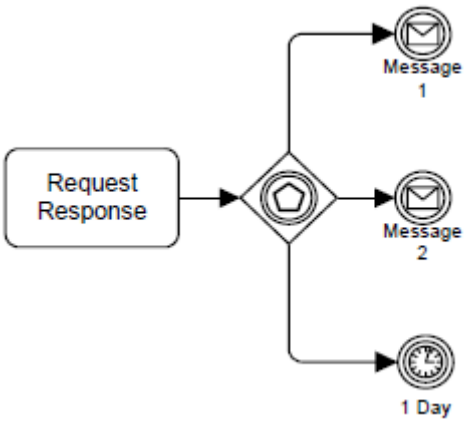

Abbildung 23: Event-Based Gateway

Event-Based Gateways können auch zum Starten von Prozessen verwendet werden. Beim Exclusive Event-Based Gateway wird nur der als erstes angestoßene Event gestartet. Alle anderen Pfade des Gateways sind dann nicht mehr gültig. Das Piktogramm hierfür ist eine Raute mit dem Multiple Start Event in der Mitte.

Abbildung 24: Exclusive Event-Based Gateway

3.5.4.2 Parallel Gateway

Das Parallel Gateway wird verwendet um parallele Flows zu synchronisieren oder zu erstellen. Das Parallel Gateway erstellt parallele Pfade ohne irgendwelche Bedingungen zu überprüfen. Jeder ausgehende Sequence Flow bekommt einen Token vom Gateway und bei synchronisierenden Gateways wird auf alle eingehenden Sequence Flows gewartet bevor der ausgehende Flow ausgelöst wird. Als Piktogramm wird für das Parallel Gateway ein Plus-Zeichen innerhalb der Raute verwendet.

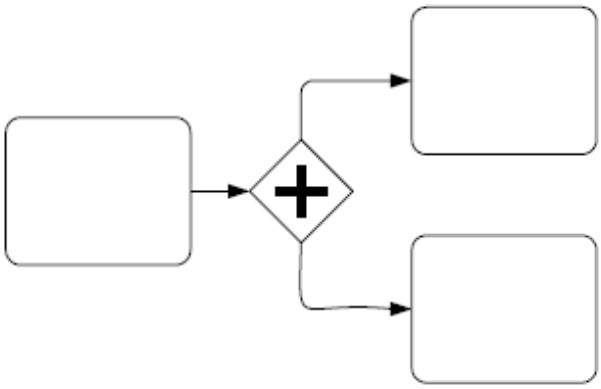

Abbildung 25: erstellendes Parallel Gateway

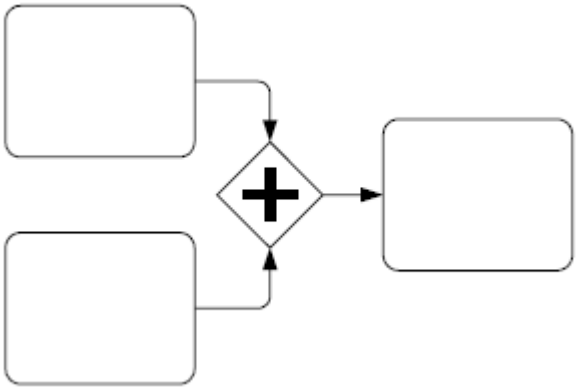

Abbildung 26: synchronisierendes Parallel Gateway

3.5.5 Message Flow

Um die Kommunikation zwischen zwei Teilnehmern in einem Prozess darzustellen verwendet man Message Flow Verbindungen ([4, S.120 ff.]) in BPMN. Die Verbindung muss zwischen zwei unterschiedlichen Pools sein, wobei nicht nur der Pool selbst, sondern auch Objekte innerhalb des Pools als Verbindungspunkt verwendet werden kann. Es ist aber nicht möglich zwei Objekte innerhalb desselben Pools zu verbinden.

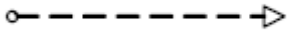

Abbildung 27: Message Flow Connection

3.5.6 Sequence Flow

Zur Festlegung der Reihenfolge, in welcher die Aktivitäten in einem Prozess ausgeführt werden, werden die Sequence Flow Verbindungen ([4, S.97 ff.]) verwendet. Jede Verbindung hat immer genau ein Ursprung und ein Ziel. Als Ursprung oder Ziel kommen Events, Aktivitäten oder Gateways in Frage, wobei es bestimmte Einschränkungen gibt. Start-Events dürfen keine eingehenden Verbindungen und End-Events keine ausgehenden Verbindungen haben. Es dürfen keine Verbindungen über die Grenzen eines Sub-Prozesses hinaus verwendet werden und Pools, Lanes, Data Objects und Annotations dürfen überhaupt nicht verbunden werden. Diese drei Konstrukte werden mit Message Flows verbunden.

Abbildung 28: Sequence Flow

Außer den normalen Verbindungen gibt es noch zwei weitere Arten. Zum einen die Conditonal Sequence Flows und zum anderen die Default Sequence Flows. Die Condtional Sequence Flows haben eine Übergangsbedingung, mit welcher entschieden wird, ob der Token weitergeschickt wird. Typischerweise werden Conditonal Sequence Flows nur an Gateways oder an Aktivitäten verwendet. Handelt es sich um eine ausgehende Verbindung einer Aktivität, dann muss eine kleine Raute zusätzlich zum Piktogramm der normalen Sequence Flow Verbindung vorne angehängt werden. Handelt es sich jedoch um einen Gateway, dann muss diese zusätzliche Raute nicht angehängt werden. Ist der Ursprung des Conditional Sequence Flows eine Aktivität, dann muss es mindestens eine weitere ausgehende Verbindung geben, ist es ein Gateway, dann darf dieses Gateway nicht vom Typ Parallel oder Event sein.

Abbildung 29: Conditional Sequence Flow

Ein Sequence Flow, der als Ursprung entweder ein Exclusive Gateway, ein Inclusive Gateway, ein Complex Gateway oder eine Aktivität hat, kann auch als Default Sequence Flow definiert werden. Dieser Sequence Flow wird einen Marker haben, der ihn als Standard

kennzeichnet. Der Default Sequence Flow bekommt ein Token, wenn alle anderen ausgehenden Verbindungen nicht mehr gültig sind, also deren Bedingungen nicht zutreffen.

Abbildung 30: Default Sequence Flow

3.5.7 Pools

In einem Prozess repräsentiert ein Pool ([4, S.112 ff.]) einen Teilnehmer des Prozesses. Ein Teilnehmer kann eine spezifische Rolle, wie zum Beispiel ein Unternehmen, oder eine allgemeinere Rolle, wie zum Beispiel einen Kunden, darstellen. Dabei kann der innere Prozess des Pools sichtbar oder verborgen sein. Pools werden zur Visualisierung des Nachrichtenaustausches zwischen mehreren Teilnehmern genutzt.

Abbildung 31: Pool

3.5.8 Data Association

Data Associations werden verwendet um den Datenfluss zwischen Data Objects, siehe Kapitel 3.5.9, und anderen Konstrukten zu visualisieren. Ein Beispiel wäre die Verwendung von Daten aus einem Data Object in einer Aktivität.

Abbildung 32: Data Association

3.5.9 Data Object

Mit Hilfe des Data Objects ([4, S.205ff.]) können die Diagramme der Prozesse mit zusätzlichen Informationen ausgestattet werden. Zur Visualisierung wird ein Aussehen eines Blatt Papiers genommen. Data Objects können nicht per Sequence Flow Connections, sondern nur per Data Associations verbunden werden.

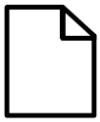

Abbildung 33: Data Object

4 Abbildungen von BPEL zu BPMN

Um einen Geschäftsprozess, der mit BPEL ausgeführt wird, mit BPMN überwachen zu können, muss das BPEL-Modell zuerst in ein BPMN-Modell transformiert werden. Ein Beispiel für ein BPEL-Model und dem zugehörigen BPMN-Model ist in Kaiptel 2.3 zu sehen. Eine Möglichkeit diese Transformation zu bewerkstelligen ist ein BPEL-Modell einem manuell erstellten BPMN-Modell gegenüber zu stellen und die Projektionen der einzelnen Aktivitäten manuell zu bestimmen. Dabei muss auf die Einhaltung von Soundness und Vollständigkeit geachtet werden. Die zweite Möglichkeit besteht darin das BPMN-Modell aus dem BPEL-Modell generieren zu lassen. Bei dieser Variante kann angenommen werden, dass das generierte Modell, auf Grund der wohldefinierten, verwendeten Regeln, vollständig und sound ist.

Die Motivation dieser Arbeit ist, dass ein Konzept erstellt werden soll, mit dem es ermöglicht wird das Tool BPI um die Überwachung eines BPEL-Prozesses durch BPMN zu erweitern. In Hinsicht auf den automatisierten Ablauf des Tools, wird die zweite Möglichkeit gewählt. Für die Umsetzung des Konzepts soll in einem ersten Schritt das Tool BPI um die Generierung eines BPMN-Modells aus einem BPEL-Modell erweitert werden.

In den folgenden Abschnitten dieses Kapitels werden bereits untersuchte und definierte Konzepte zur Transformation von BPEL zu BPMN erläutert und weiterverwendet.

4.1 Transformation zwischen den Sprachen

In [8] wurde eine umfassende und detaillierte Untersuchung von Ansätzen zur Transformation von BPEL-Modellen zu BPMN-Modellen durchgeführt. In diesem Abschnitt werden die wichtigsten Erkenntnisse und Ergebnisse zusammengefasst. Für tiefergehende Fragen wird auf [8, S.62 ff.] verwiesen.

Als Grundlage für die Transformation zwischen BPEL und BPMN kann das in der BPMN Spezifikation [4, S.445 ff.] vorgestellte Mapping von BPMN zu BPEL dienen. Zusätzlich werden in [13] Muster zur Visualisierung von BPEL Aktivitäten vorgestellt. Weiter wurden die drei Strategien Flattening, Hierarchy-Preservation und Hierarchy-Maximization aus [14] und [15] in Bezug auf die Transformation zwischen BPEL und BPMN untersucht. Bei der Flattening Strategie werden alle strukturierten Aktivitäten in BPEL vereinfacht dargestellt, indem die, in den strukturierten Aktivitäten enthaltenen, Aktivitäten von zwei Gateways ummantelt werden. Dabei werden für die <flow> Aktivität AND-Gateways und für die <if>, <pick> und <while> Aktivitäten XOR-Gateways verwendet. Probleme bei dieser Strategie sind hauptsächlich in der Umsetzung der Dead-Path-Eliminierung aus BPEL zu finden.

Die zweite Strategie, die Hierarchy-Preservation, transformiert alle strukturierten Aktivitäten aus BPEL zu Sub-Prozessen in BPMN. Da in BPMN keine Sequence Flow Connection als Ziel oder Quelle eine Aktivität aus einem Sub-Prozess haben darf, wird für diese Strategie die Teilmenge „Structured BPEL“ aus BPEL definiert. Darin sind alle BPEL Konstrukte enthalten, nur mit der Ausnahme, dass keine <link>s in den Aktivitäten enthalten sein dürfen.

Das Konzept der Hierarchy-Maximization kombiniert die beiden vorherigen Strategien. Strukturierte Aktivitäten werden mit der Hierarchy-Preservation als Sub-Prozess transformiert, solange die Aktivität keine <link>s beinhaltet. In diesem Fall wird die Flattening Strategie angewandt, um die komplexe Struktur aufzulösen.

Die Ergebnisse, wie die einzelnen Aktivitäten aus BPEL in BPMN dargestellt werden können, werden in den folgenden Unterkapiteln beschrieben und anschließend wird eine Tabelle als

Übersicht für alle Zuordnungen präsentiert. Bis auf die <forEach> Aktivität, die leicht verändert wurde, wurden die Transformationen aus [8] verwendet.

4.1.1 Basis Aktivitäten

4.1.1.1 <assign>

Die <assign> Aktivität kann in BPMN als Task oder als Sub-Prozess dargestellt werden. Wird die Darstellung als Sub-Prozess gewählt, dann werden alle enthaltenen <copy> Elemente als Task innerhalb des Sub-Prozesses dargestellt.

Genügt der überwachenden Person des Prozesses ein gröberer Detailgrad, so können mehrere Aktivitäten zu einem Task zusammengefasst werden. Beispiele für solche Zusammenfassungen wären eine <sequence> aus <assign>, <invoke> und <assign> oder eine <sequence> aus <receive> und <reply> als Task darzustellen.

4.1.1.2 <empty>

Die <empty> Aktivität wird in BPMN als Task dargestellt.

4.1.1.3 <exit>

Die <exit> Aktivität wird in BPMN als Throwing Intermediate Terminate Event dargestellt.

4.1.1.4 <invoke>

Die <invoke> Aktivität kann in BPMN als Task oder als Sub-Prozess dargestellt werden. Bei der Darstellung als Sub-Prozess können eingebettete <faultHandlers> und <compensationHandler> innerhalb des Sub-Prozesses separat dargestellt werden.

4.1.1.5 <receive>

Die <receive> Aktivität kann in BPMN als Task oder als Catching Intermediate Message Event dargestellt werden.

4.1.1.6 <reply>

Die <reply> Aktivität kann in BPMN als Task oder als Throwing Intermediate Message Event dargestellt werden.

4.1.1.7 <rethrow>

Die <rethrow> Aktivität wird in BPMN als Throwing Intermediate Error Event dargestellt.

4.1.1.8 <throw>

Die <throw> Aktivität wird in BPMN als Throwing Intermediate Error Event dargestellt.

4.1.1.9 <wait>

Die <wait> Aktivität wird in BPMN als Catching Intermediate Timer Event dargestellt.

4.1.2 Strukturierte Aktivitäten

4.1.2.1 <flow>

Die <flow> Aktivität wird in BPMN als Sub-Prozess mit jeweils einem Parallel-Gateway als Start- und Endpunkt dargestellt. Die im <flow> enthaltenen Aktivitäten werden zwischen den beiden Gateways separat abgebildet. Im <flow> enthaltene <link>s werden als Sequence Flow Connections abgebildet.

4.1.2.2 <forEach>

Die <forEach> Aktivität wird in BPMN als Sub-Prozess mit jeweils einem Exclusive-Gateway als Start- und Endpunkt dargestellt. Der im <forEach> enthaltene <scope> wird zwischen den beiden Gateways separat dargestellt.

4.1.2.3 <if>

Die im <if> Konstrukt enthaltenen Aktivitäten werden durch jeweils ein Exclusive-Gateway als Start- und Endpunkt umgeben. Für den if-Zweig und für jeden elseif-Zweig innerhalb des <if> Konstrukts werden einzelne Pfade mit einer Sequence Flow Connection erstellt. Für den else-Zweig wird eine Default Sequence Flow Connection erstellt. Jeder Pfad bildet dabei eine <sequence> Aktivität im <if> Konstrukt ab.

4.1.2.4 <pick>

Die im <pick> Konstrukt enthaltenen Aktivitäten werden durch ein Event-Based-Gateway als Startpunkt und ein Exclusive-Gateway als Endpunkt umgeben. Die Aktivitäten innerhalb des <pick> Konstrukts werden separat abgebildet.

4.1.2.5 <repeatUntil>

Die <repeatUntil> Aktivität wird in BPMN entweder als Sub-Prozess mit einem Loop-Marker dargestellt oder mit jeweils einem Exclusive-Gateway als Start- und Endpunkt, die die enthaltenen Aktivitäten umschließen. Zwischen dem Endpunkt und dem Startpunkt wird eine Default Sequence Flow Connection erstellt, um die RepeatUntil-Schleifen-Semantik herzustellen.

4.1.2.6 <sequence>

Die einzelnen Aktivitäten innerhalb der <sequence> Aktivität werden in BPMN separat dargestellt und mit Sequence Flow Connections verbunden.

4.1.2.7 <while>

Die <while> Aktivität wird in BPMN entweder als Sub-Prozess mit einem Loop-Marker dargestellt oder mit jeweils einem Exclusive-Gateway als Start- und Endpunkt, die die enthaltenen Aktivitäten umschließen. Zwischen dem Endpunkt und dem Startpunkt wird eine Sequence Flow Connection erstellt und als ausgehende Verbindung eine Default Sequence Flow Connection, um die While-Schleifen-Semantik herzustellen.

4.1.3 Scopes

4.1.3.1 <compensate>

Die <compensate> Aktivität wird in BPMN als Throwing Intermediate Compensation Event dargestellt.

4.1.3.2 <compensateScope>

Die <compensate> Aktivität wird in BPMN als Throwing Intermediate Compensation Event dargestellt.

4.1.3.3 <scope>

Die <scope> Aktivität wird in BPMN als Sub-Prozess dargestellt. <faultHandlers>, <compensationHandler> und <eventHandler> werden innerhalb des Sub-Prozesses ebenfalls dargestellt.

4.1.4 Variablen

4.1.4.1 <variable>

Die Informationen aus den <variable> Konstrukten werden in BPMN durch Data Objects abgebildet.

4.1.4.2 <validate>

Die <validate> Aktivität wird in BPMN als Task dargestellt.

4.1.5 Andere Konstrukte

4.1.5.1 <catch>

Die Abbildung des <catch> Elements wird in Kapitel 4.1.5.4 behandelt.

4.1.5.2 <catchAll>

Die Abbildung des <catchAll> Elements wird in Kapitel 4.1.5.4 behandelt.

4.1.5.3 <compensationHandler>

Die <compensationHandler> Aktivität wird in BPMN als Sub-Prozess mit einem Compensation-Marker dargestellt.

4.1.5.4 <faultHandlers>

Die Struktur eines <faultHandlers> Konstrukts in BPMN beginnt mit einem Error Start Event und die eigentliche Fehlerbehandlung wird wie bei der <if> Aktivität abgebildet. Für jedes <catch> Element und das <catchAll> Element wird ein eigener Pfad erstellt. Die <catch> Elemente werden durch normale Sequence Flow Connections und das <catchAll> Element durch eine Default Sequence Flow Connection verbunden.

4.1.5.5 <process>

Das <process> Konstrukt wird in BPMN als Pool mit allen enthaltenen Aktivitäten und Handlern dargestellt.

4.1.5.6 <terminationHandler>

Die <terminationHandler> Aktivität wird in BPMN als Sub-Prozess mit einem Termination-Marker dargestellt.

4.1.5.7 <onEvent>

Das <onEvent> Element einer <eventHandler> Aktivität wird in BPMN als Catching Intermediate Message Event dargestellt.

4.1.5.8 <onAlarm>

Das <onAlarm> Element einer <eventHandler> Aktivität wird in BPMN als Catching Intermediate Timer Event dargestellt.

In der folgenden Tabelle sind nochmal alle Zuordnungen zusammengefasst. Wenn es mehrere Möglichkeiten für eine Zuordnung gibt, wurde eine Default-Zuordnung angegeben. Die Default-Zuordnungen sind unterstrichen.

BPEL Aktivität	**BPMN Zuordnung**
Basis Aktivitäten	
<assign>	Task oder Sub-Prozess mit den <copy> Elementen als einzelne Tasks
<empty>	Task
<exit>	Throwing Intermediate Terminate Event
<invoke>	Task oder Sub-Prozess mit <faultHandlers> und <compensationHandler>
<receive>	Task oder Catching Intermediate Message Event
<reply>	Task oder Throwing Intermediate Message Event
<rethrow>	Throwing Intermediate Error Event
<throw>	Throwing Intermediate Error Event
<wait>	Catching Intermediate Timer Event
Strukturierte Aktivitäten	
<flow>	Sub-Prozess mit Parallel-Gateways als Start- und Endpunkt der inneren Aktivitäten, <link>s als Sequence Flow Connections
<forEach>	Sub-Prozess mit Loop-Marker oder Exclusive-Gateways als Start- und Endpunkt des

	inneren <scope>
<if>	Exclusive-Gateways als Start- und Endpunkt, Default Sequence Flow Connection beim else-Pfad
<pick>	Event-Based-Gateway als Startpunkt und Exclusive-Gateway als Endpunkt, separate Projektionen der internen Aktivitäten
<repeatUntil>	Sub-Prozess mit Loop-Marker oder Exclusive-Gateways als Start- und Endpunkt, Default Sequence Flow Connection zwischen End- und Startpunkt
<sequence>	Sequence Flow Connections verbinden die einzelnen Aktivitäten
<while>	Sub-Prozess mit Loop-Marker oder Exclusive-Gateways als Start- und Endpunkt, normale Sequence Flow Connection zwischen End- und Startpunkt, Default Sequence Flow Connection als ausgehende Verbindung
Scopes	
<compensate>	Throwing Intermediate Compensation Event
<compensateScope>	Throwing Intermediate Compensation Event
<scope>	Sub-Prozess mit allen Handlern
Variablen	
<variable>	Data Object
<validate>	Task
Andere Konstrukte	
<catch>	Siehe <faultHandlers>
<catchAll>	Siehe <faultHandlers>
<compensationHandler>	Sub-Prozess mit einem Compensation-Marker
<faultHandlers>	Error Start Event und Exclusive-Gateways als Start- und Endpunkt für alle <catch> und <catchAll>, jedes <catch> ein eigener Pfad, der <catchAll> Pfad hat eine Default Sequence Flow Connection
<process>	Pool mit allen Aktivitäten und Handlern
<terminationHandler>	Sub-Prozess mit einem Termination-Marker

<onEvent>	Catching Intermediate Message Event
<onAlarm>	Catching Intermediate Timer Event

Tabelle 3: Übersicht über alle Zuordnungen

4.2 Probleme nach der Transformation

Als größtes Problem nach der Transformation zwischen BPEL und BPMN hat sich die Dead-Path-Eliminierung aus BPEL herausgestellt. Die Dead-Path-Eliminierung stellt für das Tokenkonzept in BPMN eine große Herausforderung dar, da eine Unterscheidung gemacht werden muss, ob eine Aktivität ausgeführt oder übersprungen wird.

Die Dead-Path-Eliminierung widerspricht der Tokensemantik in BPMN, weil dabei Zustände von Konstrukten verändert werden, die nicht von einem Token abgelaufen werden. In [8, S.115 f.] wird eine Lösung durch die Einführung von unterschiedlichen Tokens erläutert. Dabei werden zusätzlich zu den normalen Tokens Anti-Tokens verwendet, um tote Pfade abzulaufen und die entsprechenden Aktivitäten in den Zustand *Skipped* zu setzen.

In dieser Arbeit werden die Zustände in BPMN, die von der Dead-Path-Eliminierung betroffen sind, intern berechnet und so direkt auf den Zustand *Skipped* gesetzt. Der Bruch mit der Tokensemantik in BPMN wird damit in Kauf genommen, da keine Nachteile bezüglich der Ausführung erkennbar sind. Um die interne Berechnung zu gewährleisten müssen bestimmte Regeln bezüglich der Existenz von Zuständen eingehalten werden.

1. Wenn eine Aktivität im Zustand *Inactive* ist, müssen alle folgenden Aktivitäten ebenfalls in diesem Zustand sein.
2. Wenn eine Aktivität im Zustand *Ready* ist, dann müssen alle folgenden Aktivitäten im Zustand *Inactive* und die vorhergehenden Aktivitäten im Zustand *Completed* sein.
3. Wenn eine Aktivität im Zustand *Executing* ist, dann müssen alle folgenden Aktivitäten im Zustand *Inactive* und die vorhergehenden Aktivitäten im Zustand *Completed* sein.
4. Wenn eine Aktivität im Zustand *Completed* ist, dann müssen alle vorhergehenden Aktivitäten ebenfalls im Zustand *Completed* sein.

5 Beschreibung einer Projektion

5.1 Zustände

Den Betrachter eines ausführenden Geschäftsprozesses können zwei unterschiedliche Arten von Zuständen interessieren. Zum einen der Zustand des gesamten Prozesses und zum anderen der Zustand einzelner Aktivitäten innerhalb des Prozesses. Basierend auf den Informationen aus [16] und [17] werden im Folgenden die möglichen Zustände und die Zustandsübergänge beschrieben. Die Abbildungen 34 und 35 zeigen die Zustandslebenszyklen basierend auf diesen Informationen.

In diesem Kapitel wird der Begriff Anwender verwendet. Ein Anwender ist die Person, die den Geschäftsprozess überwacht und bei Bedarf die Ausführung stoppen kann. Der Anwender muss im natürlichen Geschäftsleben keine einzelne Person sein, der Begriff dient hier nur zur Veranschaulichung.

Der komplette Prozess kann sich in einem der vier folgenden Zustände befinden: *Running*, *Faulted*, *Completed* und *Terminated*.

Wird die Ausführung eines Geschäftsprozesses gestartet, in BPEL geschieht dies durch den Empfang einer entsprechenden Nachricht, so wird der Zustand auf *Running* gesetzt. Solange keine Fehler auftreten oder die Ausführung vom Anwender beendet wird, bleibt der Prozess im Zustand *Running*. Tritt ein Fehler während der Ausführung auf, wird die Fehlerbehandlung gestartet und anschließend der Zustand des Prozesses von *Running* zu *Faulted* geändert.

Der Prozess selbst und jeder <scope> haben eine Fehlerbehandlung. Ist keine explizite Fehlerbehandlung angegeben, wird eine Default-Fehlerbehandlung ausgeführt. Bei der Default-Fehlerbehandlung werden alle Fehler innerhalb eines <scope>s an die übergeordnete Fehlerbehandlung weitergegeben. Wurde eine explizite Fehlerbehandlung angegeben, werden die beinhalteten Aktivitäten ausgeführt. Wird während der Fehlerbehandlung die Kompensationsmethode aufgerufen, werden alle Kompensationsbehandlungen gestartet. Durch diese sollen bereits ausgeführte Aktivitäten wieder, soweit möglich, rückgängig gemacht werden. Dadurch können Aktivitäten innerhalb des Prozesses, die bereits im Zustand *Completed* sind, in den Zustand *Compensating* gesetzt werden, während der Prozess selbst im Zustand *Running* verbleibt.

Nach Beendigung der Fehlerbehandlung ändert sich der Zustand des Prozesses zu *Faulted*. Wird der Prozess durch den Anwender beendet, wird dieser auf den Zustand *Terminated* gesetzt. Dabei wird vorher keine Fehlerbehandlung durchgeführt. Läuft die Ausführung dagegen bis zum Ende des Prozesses ohne das Auftreten eines Fehlers durch, wird der Zustand auf *Completed* gesetzt.

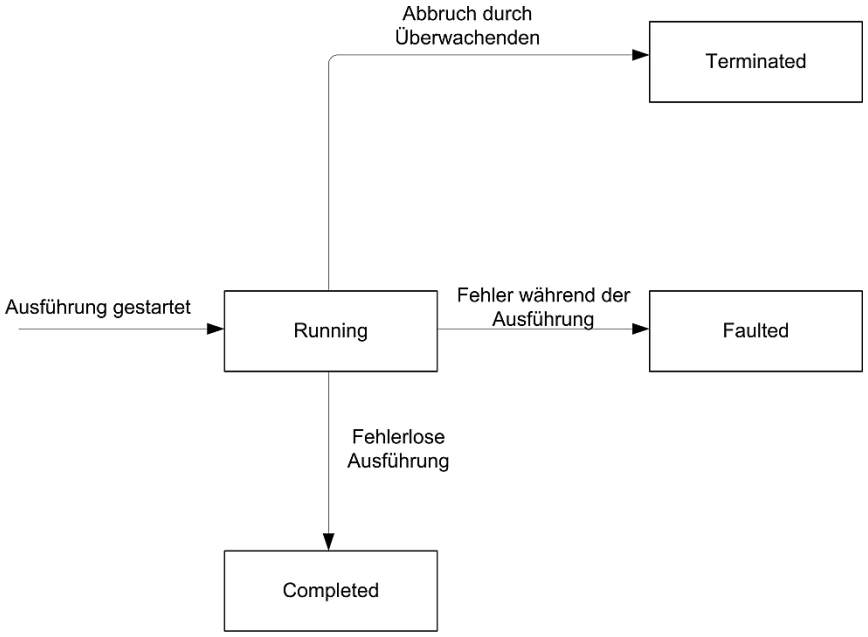

Abbildung 34: Zustandslebenszyklus für einen Geschäftsprozess

Die hier beschriebenen Zustände für den Prozess werden in dieser Arbeit durch Symbole veranschaulicht. In der folgenden Tabelle ist eine Übersicht mit allen Zuständen und den zugehörigen Symbolen zu sehen.

Name	**Symbol**	**Beschreibung**
Running		Der Prozess wird momentan ausgeführt.
Completed		Der Prozess wurde erfolgreich beendet.
Faulted		Der Prozess ist fehlgeschlagen.
Terminated		Der Prozess wurde abgebrochen.

Tabelle 4: Zustände des Prozesses

Für die Aktivitäten gibt es mehr Zustände als für den Prozess an sich. Hier wird unterschieden zwischen den Zuständen *Inactive*, *Ready*, *Skipped*, *Executing*, *Completed*, *Iteration Completed*, *Compensating*, *Compensated*, *Faulting*, *Faulted* und *Terminated*.

Nach der Initialisierung des Prozesses werden alle Aktivitäten in den Zustand *Inactive* gesetzt. In diesem Zustand kann eine Aktivität nicht ausgeführt werden. Der Zustand *Skipped* bedeutet, dass die Aktivität im weiteren Prozessverlauf nicht ausgeführt wird. Durch eine Dead-Path-Eliminierung kann eine Aktivität, die entweder im Zustand *Inactive* oder *Ready* ist, in den Zustand *Skipped* gesetzt werden. Innerhalb einer <if> Aktivität werden alle Aktivitäten, die nicht im ausgeführten Zweig liegen, ebenfalls in den Zustand *Skipped* gesetzt.

Ist eine Aktivität in einer <sequence> enthalten, so wird ihr Zustand auf *Ready* gesetzt, sobald die vorherige Aktivität in dieser <sequence> den Zustand *Completed* erreicht hat. Innerhalb eines <flow>s wird eine Aktivität in den Zustand *Ready* gesetzt, wenn sie entweder als erste Aktivität vorkommt oder, falls die Aktivität <link>s beinhaltet, wenn alle Aktivitäten, deren <link>s als Ziel diese Aktivität haben, den Zustand *Completed* erreicht haben.

Ist eine Aktivität im Zustand *Ready*, ergeben sich drei Möglichkeiten in der Fortführung des Prozesses. Entweder wird die Aktivität gestartet und somit in den Zustand *Executing* gesetzt oder ein alternativer Pfad in der Ausführung einer <if> Aktivität wird gewählt und die Aktivität wird in den Zustand *Skipped* gesetzt, wobei sie in diesem Fall nicht ausgeführt wird. Die dritte Möglichkeit besteht im Übergang in den Zustand *Terminated*, falls der Anwender den Prozess beendet.

Für eine Aktivität im Zustand *Executing* gibt es vier mögliche Zustandsübergänge im weiteren Ablauf des Prozesses. Die erste Möglichkeit besteht darin, dass ein Fehler während der Ausführung auftritt und somit der Zustand der Aktivität zu *Faulting* übergeht. Ist eine explizite Fehlerbehandlung angegeben, wird diese ausgeführt und der Zustand geht anschließend zu *Completed* über. In diesem Fall wird der Prozess weiter ausgeführt. Gibt es keine explizite Fehlerbehandlung, wird die Default-Fehlerbehandlung durchgeführt und der Fehler wird an die übergeordnete Fehlerbehandlung weitergegeben. Wenn es keine übergeordnete Fehlerbehandlung gibt, der Fehler befindet sich bereits auf der Prozessebene, werden der Zustand der Aktivität, sowie der Zustand des Prozesses, nach Beendigung der Fehlerbehandlung in den Zustand *Faulted* gesetzt und die Ausführung wird beendet.

Als zweites gibt es die Möglichkeit, dass der Prozess vom Anwender abgebrochen wird. In diesem Fall gehen alle Aktivitäten, die entweder im Zustand *Executing* oder im Zustand *Ready* sind, sofort in den Zustand *Terminated* über, ohne eine Fehler- oder eine Kompensationsbehandlung zu starten.

Weiter gibt es die Möglichkeit, dass die Ausführung der Aktivität normal verläuft und nach Beendigung der Aktivität der Zustand auf *Completed* gesetzt wird.

Als letztes gibt es den Anwendungsfall, dass eine Schleife durchlaufen wird. In diesem Fall wird die Aktivität in den Zustand *Iteration Completed* gesetzt und am Ende des Schleifendurchlaufs wird, durch eine angegebene Bedingung, entschieden, ob die Schleife nochmals zu durchlaufen ist. Im Falle, dass die Schleife nochmals durchlaufen wird, werden alle internen Aktivitäten, die im Zustand *Iteration Completed* sind, in den Zustand *Inactive* gesetzt und der Zustand der Schleife bleibt auf *Executing*. Wird die Schleife nicht nochmals durchlaufen, werden der Zustand der Schleife und die Zustände alle internen Aktivitäten auf *Completed* gesetzt.

Wird die Kompensationsbehandlung einer Aktivität gestartet, so geht der Zustand von *Completed* zu *Compensating* über. Nach Beendigung der Kompensation wird der Zustand auf *Compensated* gesetzt.

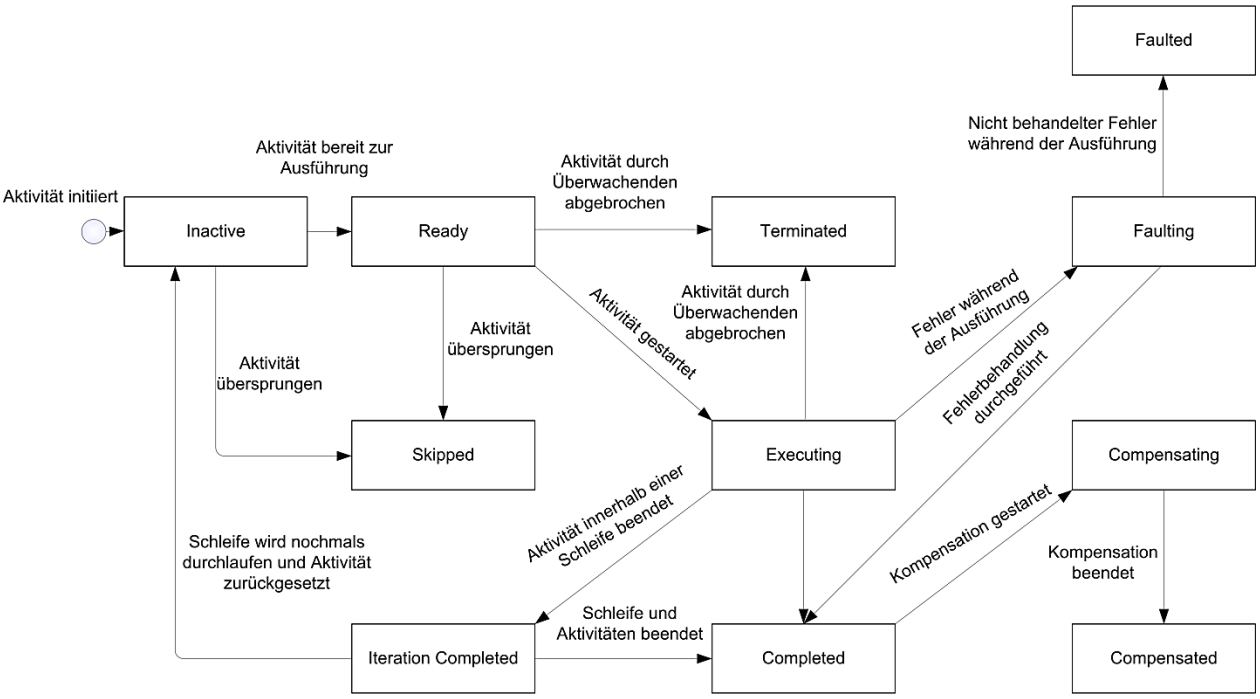

Abbildung 35: Zustandslebenszyklus von Aktivitäten

Die für die Aktivitäten beschriebenen Zustände werden in dieser Arbeit durch verschiedene Symbole veranschaulicht. In der folgenden Tabelle ist eine Übersicht mit allen Zuständen und den zugehörigen Symbolen zu sehen.

Name	Symbol	Beschreibung
Inactive		Die Aktivität wurde noch nicht ausgeführt und die Voraussetzungen zum Starten sind noch nicht erfüllt.
Ready		Die Aktivität wurde noch nicht ausgeführt, aber die Voraussetzungen zum Starten sind erfüllt.
Skipped		Die Aktivität wurde übersprungen.
Executing		Die Aktivität wird im Moment ausgeführt.
Completed		Die Aktivität wurde erfolgreich beendet.
Iteration Completed		Ein Durchlauf der Schleife wurde erfolgreich beendet und es wird geprüft ob ein weiterer durchgeführt wird.
Compensated		Die Aktivität wurde kompensiert.
Faulted		Die Aktivität ist fehlgeschlagen.
Terminated		Die Aktivität wurde abgebrochen.

Tabelle 5: Zustände der Aktivitäten

5.2 Muster zur Zustandsübertragung

In [18] haben die Autoren durch ihre Arbeit mehrere Muster zur Zustandsübertragung zwischen technischen Prozessen, zum Beispiel beschrieben durch BPEL, und abstrakten Prozessen, zum Beispiel beschrieben durch BPMN, identifiziert und erstellt. Um diese Muster abzuleiten, haben sie die Transformation von Konstrukten in abstrakten Prozessen zu Konstrukten in technischen Prozessen untersucht und anschließend haben sie eine Lösung spezifiziert, um den Zustand des abstrakten Prozesses aus dem Zustand des technischen Prozesses abzuleiten. Im Folgenden erläutere ich die von mir verwendeten Muster aus [18]. Für Informationen über die weiteren Muster wird auf [SLLMS11, S.7 ff.] verwiesen.

5.2.1 Direct State Propagation Pattern

Eine direkte Projizierung des Zustandes einer technischen Aktivität auf den Zustand einer abstrakten Aktivität ist die einfachste Möglichkeit zur Übertragung des Zustandes. Dieses Pattern kann immer angewendet werden, wenn eine einzelne technische Aktivität auf eine einzelne abstrakte Aktivität abgebildet werden kann. Im Beispiel aus Kapitel 2 wird die <invoke> Aktivität „checkAvailbility“ in BPEL als Task in BPMN dargestellt, was ein Beispiel für das Direct State Propagation Pattern ist. Dabei wird der Zustand der <invoke> Aktivität direkt auf die Task in BPMN übertragen.

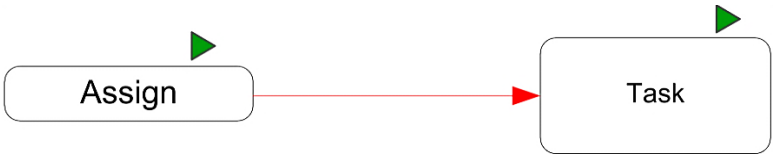

Abbildung 36: Direct State Propagation

5.2.2 State Combination Pattern

Es kann vorkommen, dass mehrere technische Aktivitäten eine einzige abstrakte Aktivität implementieren. Der Zustand der abstrakten Aktivität wird aus einer Funktion über die Zustände von mehreren technischen Aktivitäten abgeleitet, welche nicht unbedingt verbunden sein müssen. Es können beliebige Funktionen zur Zustandszusammenführung definiert werden. Eine <assign>-Aktivität, eine <invoke>-Aktivität und eine zweite <assign>-Aktivität im BPEL-Model können als eine BPMN-Aktivität dargestellt werden. Die Zustandszusammenführungsfunktion könnte nun lauten: Wenn eine der BPEL-Aktivitäten im Zustand *Executing* ist, dann ist die BPMN-Aktivität ebenfalls im Zustand *Executing*. Wenn alle drei BPEL-Aktivitäten im Zustand *Completed* sind, dann ist auch die BPMN-Aktivität im Zustand *Completed*.

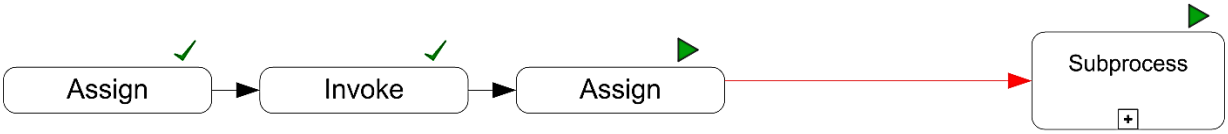

Abbildung 37: State Combination

5.2.3 Complex State Distribution Pattern

Das Complex State Distibution Pattern basiert auf dem Direct State Propagation Pattern. Wenn eine einzelne Aktivität im technischen Model mehrere Aktivitäten im abstrakten Model implementiert, dann wird der Zustand der technischen Aktivität auf die zugehörigen abstrakten Aktivitäten verteilt.

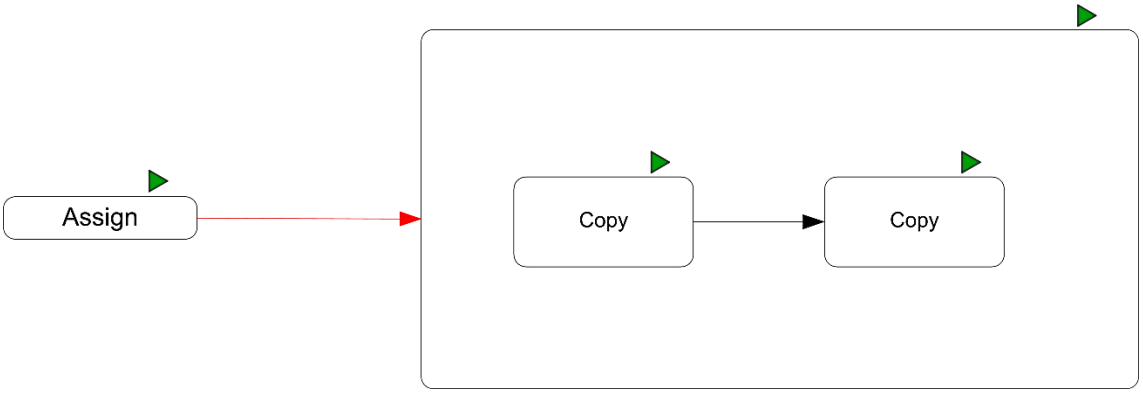

Abbildung 38: Complex State Distribution

5.3 BPEL und BPMN Erweiterungen

Um die Projektion von Zuständen durchführen zu können, müssen neue XML-Tags als Erweiterungen eingeführt werden. Im weiteren Verlauf werden die neuen XML-Tags vorgestellt und beschrieben. In BPEL werden die Erweiterungen mit dem <extension> Element beschrieben und eingeführt. BPMN stellt ebenfalls Elemente zu Erweiterung zur Verfügung. Das Element in BPMN wird wie in BPEL „extension“ genannt und Unterstützt die Erweiterbarkeit, indem neue Attribute oder Elemente für BPMN durch XML-Schemas definiert werden können. Die neuen Attribute oder Elemente müssen anschließend mit dem „extension“ Element importiert werden. Für weitergehende Informationen zur Erweiterbarkeit von BPMN wird auf [4, S.57 ff.] verwiesen.

5.3.1 activityId

Jede Aktivität in BPEL erhält eine eindeutige activityId. Diese wird aus dem Namen der Aktivität, sowie einer Zahl, getrennt durch ein Leerzeichen, zusammengesetzt. Das erste Vorkommen einer <assign> Aktivität hätte die activityId „assign 1“, die Zweite „assign 2“ und so weiter. Die activityId wird für die Generierung der Mappings-Datei, zum Auslesen der Zustände und zur Anwendung der Projektionen benötigt.

```
<bpel:invoke name="checkAvailability"
  inputVariable="product"
  outputVariable="availabilityInfo"
  operation="requestAvailability">
   <ext:activityId>invoke 1</ext:activityId>
   <ext:activityState>Completed</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
</bpel:invoke>
```

Listing 34: <invoke> Aktivität mit einer <ext:activityId>

5.3.2 projectionId

Jede Projektion erhält eine einzigartige projectionId. Diese wird aus dem Namen des BPMN-Konstrukts, sowie einer Zahl, getrennt durch ein Leerzeichen, zusammengesetzt. Das erste Vorkommen eines Tasks hätte die projectionId „task 1“, die Zweite „task 2“ und so weiter. Zusammen mit der activityId wird die projectionId zur Generierung der Mappings-Datei und zur Anwendung der Projektionen verwendet.

```
<projection>
   <MappingFrom>
      <activityId>assign 10</activityId>
      <activityId>invoke 8</activityId>
      <activityId>assign 11</activityId>
   </MappingFrom>
   <MappingTo>
      <projectionId>Task 3</projectionId>
   </MappingTo>
   <projectionRule>Combination</projectionRule>
</projection>
```

Listing 35: Beispiel einer Projektion aus der Mappings-Datei mit der <projectionId>

In der BPMN-Datei wird das Attribut „id“ zur Speicherung der projectionId genommen. Das Attribut „id“ ist ein Standard-Attribut in BPMN und stellt einen eindeutigen Bezeichner für jedes Konstrukt dar.

```
<subProcess id="subprocess 5" name="findSimilarProduct" />
```

Listing 36: Attribut "id" in einem BPMN-Konstrukt

5.3.3 activityState

Der XML-Tag activityState gibt den aktuellen Zustand der Aktivität an, der auf das Konstrukt in BPMN übertragen wird. Wie in Kapitel 5.1 beschrieben, gibt es für Aktivitäten die Zustände *Inactive*, *Ready*, *Skipped*, *Executing*, *Terminated*, *Iteration Completed*, *Completed*, *Faulting*, *Faulted*, *Compensating* und *Compensated*.

```
<bpel:invoke name="checkAvailability"
  inputVariable="product"
  outputVariable="availabilityInfo"
  operation="requestAvailability">
   <ext:activityId>invoke 1</ext:activityId>
   <ext:activityState>Completed</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
</bpel:invoke>
```

Listing 37: <invoke> Aktivität mit dem <ext:activityState> *Completed*

In BPMN wird der Zustand in einem Attribut gespeichert. Die BPMN-Datei wird zu diesem Zweck durch das Attribut „state“ erweitert.

```
 <task id="task 2" name="checkAvailability" state="Completed" />
```

Listing 38: Task mit dem state *Completed*

5.3.4 processState

Der XML-Tag processState ist das Pendant zum activityState für den kompletten Prozess. Für den Prozess kann nicht der activityState verwendet werden, da der Prozess einen ande-

ren Zustandsraum als die Aktivitäten hat. Der Zustandsraum für den Prozess umfasst, wie in Kapitel 5.1 beschrieben, die Zustände *Running*, *Completed*, *Faulted* und *Terminated*.

```
<bpel:process name="Beispiel"
  targetNamespace="http://sample.bpel.org/bpel/sample"
  suppressJoinFailure="yes"
  xmlns:tns="http://sample.bpel.org/bpel/sample"
  xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
  abstractProcessProfile=
  "http://docs.oasis-open.org/wsbpel/2.0/process/abstract
  /simple-template/2006/08">
   <ext:processState>Running</ext:processState>
   <bpel:import location="Beispiel.wsdl"
     namespace=http://sample.bpel.org/bpel/sample
     importType="http://schemas.xmlsoap.org/wsdl/" />
```

Listing 39: Der Beispiel-Prozess mit dem <ext:processState> *Running*

5.3.5 projectionType

Mit dem XML-Tag projectionType wird zwischen casual und loop unterschieden. Der Typ loop beinhaltet alle Schleifen, die es in BPEL gibt. Dazu zählen <while>, <repeatUntil> und <forEach>. Für diese Konstrukte wird ein weiteres Attribut, der iterationCount, zum korrekten Abbilden der Zustände benötigt. Alle anderen Konstrukte sind im Typ casual zusammengefasst. Der projectionType kann um weitere Typenarten erweitert werden.

```
<bpel:invoke name="checkAvailability"
  inputVariable="product"
  outputVariable="availabilityInfo"
  operation="requestAvailability">
   <ext:activityId>invoke 1</ext:activityId>
   <ext:activityState>Completed</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
</bpel:invoke>
```

Listing 40: <invoke> Aktivität mit einem <ext:projectionType>

5.3.6 iterationCount

Um in Schleifen die korrekte Darstellung des aktuellen Zustands abbilden zu können, wird der iterationCount eingeführt. Der iterationCount gibt an in welchem Durchlauf die Schleife sich befindet. Bei der <forEach> Schleife wird der iterationCount auch zur Überprüfung der Bedingung eingesetzt. Hat der iterationCount zu Beginn des Schleifendurchlaufs denselben Wert wie der finalCounterValue der <forEach> Schleife, so ist die Schleife vollständig durchgelaufen und wird beendet. Bei der <while> und <repeatUntil> Schleife gibt es keinen finalCounterValue, da die Schleife solange durchlaufen wird bis die angegebene Bedingung erfüllt ist. Das schließt nicht aus, dass in der Bedingung vom iterationCount gebraucht gemacht wird.

```
<bpel:forEach parallel="no" counterName="Counter" name="ForEach">
   <bpel:startCounterValue>1</bpel:startCounterValue>
   <bpel:finalCounterValue>$productOffers.count</bpel:finalCounterValue>
   <ext:activityId>forEach 1</ext:activityId>
   <ext:activityState>Executing</ext:activityState>
   <ext:projectionType>Loop</ext:projectionType>
   <ext:iterationCount>5</ext:iterationCount>
   <bpel:scope>
   ...
```

```
   </bpel:scope>
</bpel:forEach>
```

Listing 41: <forEach> Aktivität mit dem <ext:iterationCount> 5

5.4 Zustandsübertragungen von BPEL nach BPMN

5.4.1 <assign>

Die <assign> Aktivität kann in BPMN als Task oder als Sub-Prozess abgebildet werden. Wird die Aktivität als Task abgebildet, dann wird das Direct State Propagation Pattern verwendet. Der Zustand der <assign> Aktivität wird also direkt als Zustand auf den Task übertragen.

Wird die <assign> Aktivität nicht als Task, sondern als Sub-Prozess abgebildet, dann muss unterschieden werden, ob der Sub-Prozess expandiert dargestellt wird oder nicht. Ist der Sub-Prozess nicht expandiert dargestellt, dann wird, wie beim Task, der Zustand direkt übertragen. Bei einer expandierten Darstellung wird das State Distribution Pattern verwendet. Hierbei wird der Zustand der <assign> Aktivität auf alle Tasks im Sub-Prozess übertragen.

Die Abbildung zeigt ein Beispiel für den Zustand *Executing*.

```
<bpel:assign validate="no" name="assignOffer">
   <ext:activityId>assign 2</ext:activityId>
   <ext:activityState>Executing</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
   <bpel:copy>
      <ext:activityId>copy 2</ext:activityId>
      <bpel:from>$productDeliveryDate</bpel:from>
      <bpel:to>$product.deliveryDate</bpel:to>
   </bpel:copy>
   <bpel:copy>
      <ext:activityId>copy 3</ext:activityId>
      <bpel:from>$productPrice</bpel:from>
      <bpel:to>$product.price</bpel:to>
   </bpel:copy>
   <bpel:targets>
      <bpel:target linkName="link3"></bpel:target>
      <bpel:target linkName="link4"></bpel:target>
   </bpel:targets>
</bpel:assign>
```

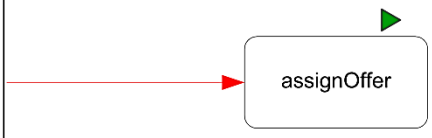

Abbildung 39: Projektion einer <assign> Aktivität

5.4.2 <empty>

Die <empty> Aktivität wird in BPMN als Task dargestellt und somit greift das Direct State Propagation Pattern.

```
<bpel:empty name="Empty">
   <ext:activityId>empty 1</ext:activityId>
   <ext:activityState>Skipped</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
</bpel:empty>
```

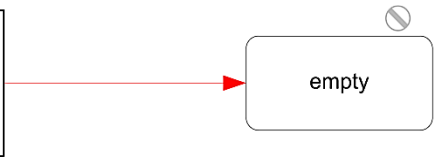

Abbildung 40: Projektion einer <empty> Aktivität

5.4.3 <exit>

Für die Abbildung der <exit> Aktivität kann das Terminate-End-Event in BPMN verwendet werden. Der Zustand betrifft hier nicht direkt das Event, sondern den kompletten Prozess.

Wird die <exit> Aktivität ausgeführt, so wird der <process> Zustand entsprechend verändert und alle laufenden Aktivitäten werden unverzüglich beendet.

In Abbildung 41 ist der Aspekt zu sehen, dass die <exit> Aktivität Einfluss auf den Zustand des Prozesses hat.

Abbildung 41: Einfluss der <exit> Aktivität auf den Prozess

5.4.4 <invoke>

Bei der <invoke> Aktivität kommen wieder mehrere Darstellungsansätze in Frage. Falls die <invoke> Aktivität keine <faultHandlers> und <compensationHandler> hat, dann wird sie als Task dargestellt und somit kommt das Direct State Propagation Pattern zum Einsatz und der Zustand wird direkt übertragen. Kommen jedoch innerhalb der <invoke> Aktivität eine <faultHandlers> Aktivität, eine <compensationHandler> Aktivität oder beide Aktivitäten zum Einsatz, dann wird diese Aktivität als Sub-Prozess dargestellt. Bei einer nicht expandierten Darstellung des Sub-Prozesses wird wiederum der Zustand direkt, mit Hilfe des Direct State Propagation Patterns, übertragen. Die Übertragung des Zustandes wird komplexer sobald eine expandierte Darstellung der <invoke> Aktivität verwendet wird. Hier müssen die unterschiedlichen Zustände betrachtet werden. Ist die <invoke> Aktivität im Zustand *Faulting*, dann wird der dargestellte Sub-Prozess des <faultHandlers> auf den Zustand *Executing* gesetzt und der Zustand des Sub-Prozesses für die <invoke> Aktivität bleibt ebenfalls im Zustand *Executing*, bis die Abarbeitung des <faultHandlers> beendet ist. Für den <compensationHandler> gelten dieselben Regeln wie für den <faultHandlers> Bereich.

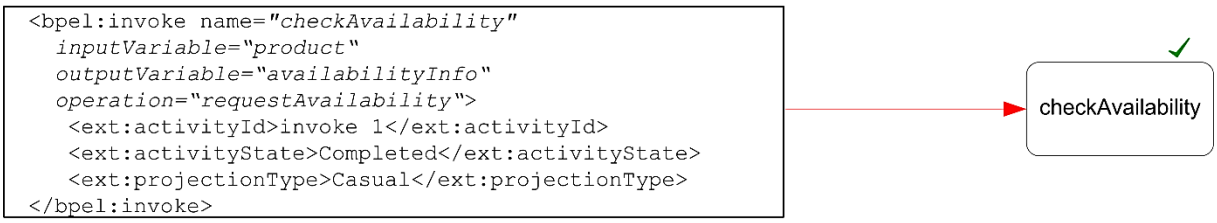

Abbildung 42: Projektion einer <invoke> Aktivität

5.4.5 <receive>

Die <receive> Aktivität kann in BPMN als Task dargestellt werden. Hierbei wird das Direct State Propagation Pattern zum direkten Übertragen des Zustandes verwendet.

```
<bpel:receive name="receiveProductRequest"
  variable="product" createInstance="yes"
  operation="sendProductRequest">
   <ext:activityId>receive 1</ext:activityId>
   <ext:activityState>Completed</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
</bpel:receive>
```

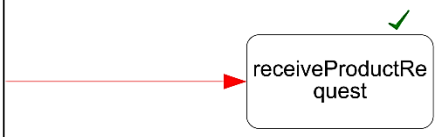

Abbildung 43: Projektion einer <receive> Aktivität

5.4.6 <reply>

Die <reply> Aktivität wird als Task abgeleitet. Der Zustand für die <reply> Aktivität wird ebenfalls direkt mit dem Direct State Propagation Pattern übertragen.

```
<bpel:reply name="productOffering"
  variable="productOffers"
  operation="productRequest">
   <ext:activityId>reply 1</ext:activityId>
   <ext:activityState>Inactive</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
</bpel:reply>
```

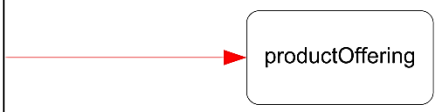

Abbildung 44: Projektion einer <reply> Aktivität

5.4.7 <rethrow>

Ein <rethrow> wird in BPMN als Throwing Intermediate Error Event dargestellt. Die <rethrow> Aktivität startet den übergeordneten <faultHandlers> und setzt dessen Zustand auf *Executing*, während der Zustand des aktuellen <faultHandlers> auf *Completed* gesetzt wird.

```
<bpel:rethrow name="Rethrow">
   <ext:activityId>rethrow 1</ext:activityId>
   <ext:activityState>Skipped</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
</bpel:rethrow>
```

Abbildung 45: Projektion einer <rethrow> Aktivität

5.4.8 <throw>

Mit <throw> kann ein benutzerdefinierter Fehler geworfen werden und durch Eintreten dieses Fehlers wird der zugehörige <faultHandlers> in den Zustand *Executing* gesetzt. Die <throw> Aktivität wird in BPMN, genau wie die <rethrow> Aktivität, als Throwing Intermediate Error Event dargestellt.

```
<bpel:throw xmlns:FLT="http://example.com/faults"
  faultName="FLT:OutOfStock">
   <ext:activityId>throw 1</ext:activityId>
   <ext:activityState>Executing</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
</bpel:throw>
```

Abbildung 46: Projektion einer <throw> Aktivität

5.4.9 <wait>

Die <wait> Aktivität wird durch einen Catching Intermediate Timer Event dargestellt. Der Zustand bleibt während der Wartezeit im Zustand *Executing* und wird auf *Completed* gesetzt, sobald die <wait> Aktivität beendet wurde.

In Abbildung 47 ist ein Beispiel für eine <wait> Aktivität mit einer Wartezeit von 10 Minuten zu sehen.

```
<bpel:wait name="Wait">
   <ext:activityId>wait 1</ext:activityId>
   <ext:activityState>Skipped</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
   <bpel:for>PT10M</bpel:for>
</bpel:wait>
```

Abbildung 47: Projektion einer <wait> Aktivität

5.4.10 <flow>

Als äußere Hülle eines <flow> Konstrukts dient ein Sub-Prozess. Im Innern des Sub-Prozesses kommt jeweils ein Parallel-Gateway an den Anfang und an das Ende des Konstrukts, um die parallele Ausführung innerhalb eines <flow>s zu ermöglichen. Die <link>s innerhalb des <flow> werden als Sequence Flow Connections und alle Aktivitäten werden separat je nach Aktivitätentyp in BPMN abgebildet.

Wird die <flow> Aktivität als nicht expandierter Sub-Prozess dargestellt, kommt das State Combination Pattern zum Einsatz. Ist der Sub-Prozess expandiert dargestellt, kommt ebenfalls das State Combination Pattern zum Einsatz, aber zusätzlich sind die beiden Parallel-Gateways des <flow>s im Zustand *Executing*. Solange mindestens eine der im <flow> enthaltenen Aktivitäten im Zustand *Executing* ist, bleiben die Parallel-Gateways in ihrem Zustand. Sind alle Aktivitäten im Zustand *Completed*, werden die Parallel-Gateways auch in den Zustand *Completed* gesetzt. Durch die Tokensemantik von BPMN würde es durch die Zustände der beiden Gateways zu Problemen kommen, da der durchlaufende Token dupliziert wird. Diese Problematik wird in Kapitel 5.5.4 behandelt.

```
<bpel:flow name="Flow">
   <ext:activityId>flow 1</ext:activityId>
   <ext:activityState>Executing</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
   <bpel:receive name="receiveOffer" …>
      <ext:activityId>receive 2</ext:activityId>
      <ext:activityState>Completed</ext:activityState>
      <ext:projectionType>Casual</ext:projectionType>
      ...
   </bpel:receive>
   <bpel:invoke name="checkDeliveryDate" ...>
   ...
   <bpel:links>
   ...
   </bpel:links>
</bpel:flow>
```

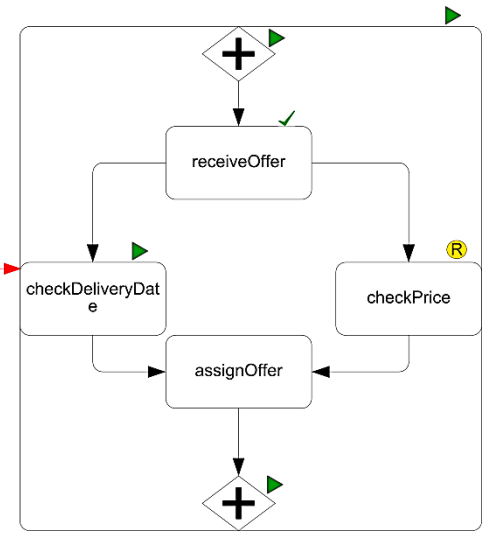

Abbildung 48: Projektion einer <flow> Aktivität

5.4.11 <forEach>

Diese Aktivität wird durch einen Sub-Prozess dargestellt. Ist der Sub-Prozess nicht expandiert dargestellt, kommt das State Combination Pattern zum Einsatz. In der expandierten Darstellung wird auch das State Combination Pattern zum Einsatz, wobei, wie bei der <flow> Aktivität, die Exclusive-Gatways ebenfalls einen Zustand erhalten. Sie sind während der Ausführung der <forEach> Aktivität im Zustand *Executing*. Nach Beendigung der Schleife, entweder durch einen kompletten Durchlauf oder durch eine <completionCondition>, wird der Zustand der <forEach> Aktivität, sowie die Zustände der Exclusive-Gateways, auf *Completed* gesetzt. Durch die Tokensemantik von BPMN würde es durch die Zustände der beiden Gateways zu Problemen kommen, da der durchlaufende Token dupliziert wird. Diese Problematik wird in Kapitel 5.5.4 behandelt.

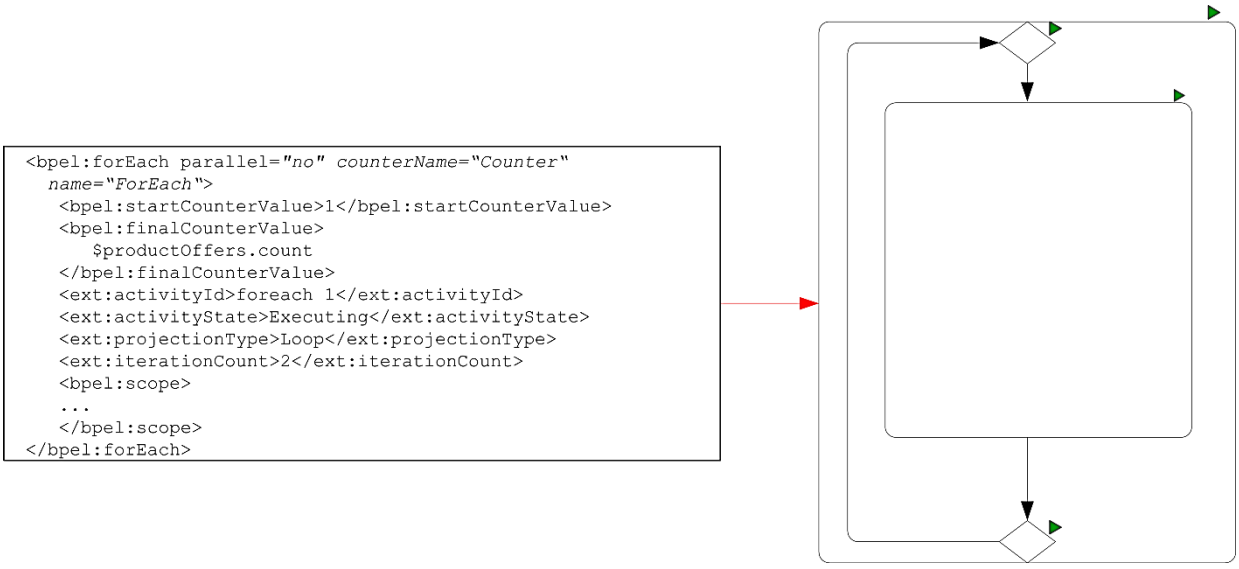

Abbildung 49: Projektion einer <forEach> Aktivität

5.4.12 <if>

Ein <if> Konstrukt wird als Sub-Prozess dargestellt. Bei einer nicht expandierten Darstellung kommt das State Combination Pattern zum Einsatz. In der expandierten Darstellung kommt ebenfalls das State Combination Pattern zum Einsatz. In dieser Darstellung hat der Sub-Prozess vor und hinter den beinhalteten Aktivitäten jeweils ein Exclusive-Gateway, die während der Ausführung der <if> Aktivität im Zustand *Executing* bleiben. Das Start-Gateway hat je einen ausgehenden Sequence Flow für den if- und den else-Pfad, sowie für jeden elseif-Pfad. Da nur einer der Pfade ausgeführt werden kann, werden die Zustände der Aktivitäten auf den anderen Pfaden intern berechnet und automatisch in den Zustand *Skipped* gesetzt. Der else-Pfad hat eine Default Sequence Flow Connection, wird also standardmäßig als auszuführender Pfad genommen. Auf dem gewählten Pfad werden die Aktivitäten der Reihe nach, wie in einer <sequence>, ausgeführt. Nach Beendigung der <if> Aktivität werden deren Zustand, sowie die Zustände der Exclusive-Gateways, auf *Completed* gesetzt. Durch die Tokensemantik von BPMN würde es durch die Zustände der beiden Gateways, sowie durch die Dead-Path-Eliminierung, zu Problemen kommen, da der durchlaufende Token dupliziert wird. Diese Problematik wird in Kapitel 5.5.4 behandelt.

```
<bpel:if name="ProductAvailable">
   <ext:activityId>if 1</ext:acitivityId>
   <ext:activityState>Executing</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
   <bpel:condition>
      bpel:getVariableProperty(`availabilityInfo`,
      `inventory:level`) > 0
   </bpel:condition>
   ...
<bpel:elseif>
   ...
</bpel:elseif>
<bpel:else>
   ...
</bpel:else>
</bpel:if>
```

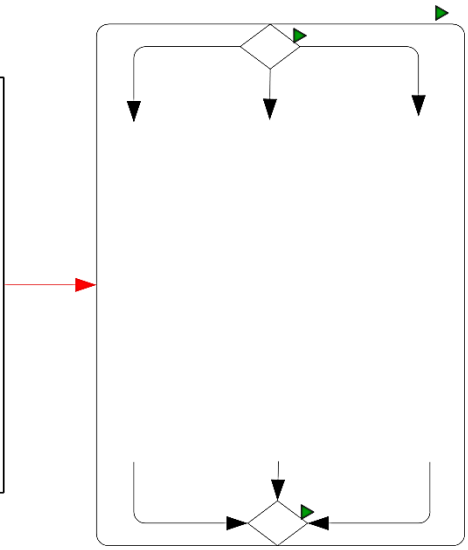

Abbildung 50: Projektion einer <if> Aktivität

5.4.13 <pick>

Das <pick> Konstrukt wird ebenfalls durch zwei umgebende Gateways abgebildet. Am Anfang wird ein Event-Based-Gateway und am Ende ein Exclusive-Gateway verwendet. Die Aktivitäten innerhalb des <pick> Konstrukts werden separat behandelt. Das Konstrukt selbst hat keinen eigenen Zustand, aber die beiden Gateways bekommen einen. Solange mindestens eine Aktivität innerhalb der <pick> Aktivität nicht im Zustand *Completed* ist, sind die beiden Gateways im Zustand *Executing*. Wurden alle internen Aktivitäten beendet, so werden die beiden Gateways in den Zustand *Completed* gesetzt. Durch die Tokensemantik von BPMN würde es durch die Zustände der beiden Gateways zu Problemen kommen, da der durchlaufende Token dupliziert wird. Diese Problematik wird in Kapitel 5.5.4 behandelt.

```
<bpel:pick name="productRequest">
   <bpel:onMessage operation="startProductSearch" variable="product">
      <ext:activityId>event 3</ext:activityId>
      <ext:activityState>Inactive</ext:activityState>
      <ext:projectionType>Casual</ext:projectionType>
   </bpel:onMessage>
   <bpel:onAlarm>
      <ext:activityId>event 4</ext:activityId>
      <ext:activityState>Inactive</ext:activityState>
      <ext:projectionType>Casual</ext:projectionType>
      <bpel:for>`PT12H`</bpel:for>
   </bpel:onAlarm>
</bpel:pick>
```

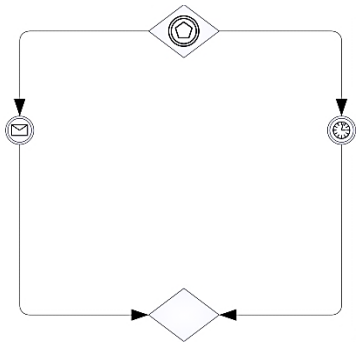

Abbildung 51: Projektion einer <pick> Aktivität

5.4.14 <repeatUntil>

Die <repeatUntil> Aktivität kann als nicht expandierter Sub-Prozess mit einem Loop-Marker oder als expandierter Sub-Prozess dargestellt werden. In beiden Fällen wird der Zustand per State Combination Patter übertragen. Bei der expandierten Darstellung wird am Anfang und am Ende des Sub-Prozesses jeweils ein Exclusive-Gateway gestellt. Man definiert dann für das hintere Gateway eine Default-Verbindung, die zurück auf das erste Gateway führt um die Schleife weiterlaufen zu lassen und definiert zudem eine Verbindung, die bei Erfüllung der angegebenen Bedingung gewählt wird. Die Aktivitäten innerhalb der <repeatUntil> Aktivität werden separat abgebildet und zwischen die beiden Exclusive-Gateways gesetzt. Die beiden

Gateways erhalten ebenfalls einen Zustand. Während der Ausführung der <repeatUntil>-Schleife wird der Zustand der beiden Gateways auf *Executing* gesetzt. Nach Beendigung der Schleife werden der Zustand der <repeatUntil> Aktivität, sowie die Zustände der Exclusive-Gateways, auf *Completed* gesetzt. Durch die Tokensemantik von BPMN würde es durch die Zustände der beiden Gateways zu Problemen kommen, da der durchlaufende Token dupliziert wird. Diese Problematik wird in Kapitel 5.5.4 behandelt.

Ein Beispiel für die <repeatUntil> Aktivität mit einem nicht expandierten Sub-Prozess.

```
<bpel:repeatUntil name="checkStock">
   <ext:activityId>repeatuntil 1</ext:activityId>
   <ext:activityState>Executing</ext:activityState>
   <ext:projectionType>Loop</ext:projectionType>
   <ext:iterationCount>10</ext:iterationCount>
   <bpel:invoke name="checkStock">
      ...
   </bpel:invoke>
   <bpel:condition>
      bpel:getVariableProperty(`availabilityInfo`,
      `product:inStock`) < 15
   </bpel:condition>
</bpel:repeatUntil>
```

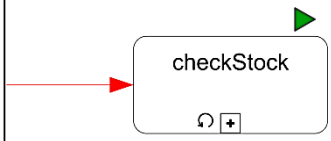

Abbildung 52: Projektion einer <repeatUntil> Aktivität

5.4.15 <sequence>

Eine aneinander Reihung mehrerer Aktivitäten, wie innerhalb der <sequence> Aktivität, kann in BPMN entweder wieder durch einen nicht expandierten Sub-Prozess oder durch die einzelnen Aktivitäten innerhalb der <sequence> dargestellt werden. Bei Verwendung eines nicht expandierten Sub-Prozesses wird das State Combination Pattern angewandt. Der Zustand des Sub-Prozesses ist also solange *Executing*, solange mindestens eine Aktivität innerhalb der <sequence> im Zustand *Executing* ist. Sind alle Aktivitäten innerhalb der <sequence> im Zustand *Completed*, dann wird auch der Zustand des Sub-Prozesses auf den Zustand *Completed* gesetzt. Werden die Aktivitäten aus der <sequence> allerdings einzeln in BPMN abgebildet, dann wird jede Aktivität einzeln betrachtet und die Regeln für jede dieser Aktivitäten müssen beachtet werden. Die <sequence> an sich hat in diesem Fall keinen gesonderten Zustand.

```
<bpel:sequence>
   <bpel:assign name="Assign1">
      <ext:activityId>assign 1</ext:activityId>
      <ext:activityState>Completed</ext:activityState>
      ...
   </bpel:assign>
   <bpel:invoke name="Invoke1">
      <ext:activityId>invoke 1</ext:activityId>
      <ext:activityState>Executing</ext:activityState>
      ...
   </bpel:invoke>
   <bpel:assign name="Assign2">
      <ext:activityId>assign 2</ext:activityId>
      <ext:activityState>Inactive</ext:activityState>
      ...
   </bpel:assign>
</bpel:sequence>
```

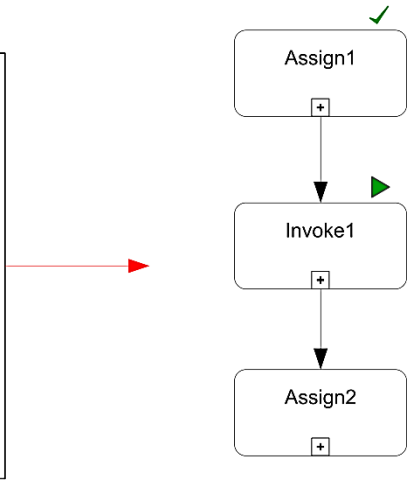

Abbildung 53: Projektion einer <sequence> Aktivität

5.4.16 <while>

Die <while> Aktivität wird genauso wie die <repeatUntil> Aktivität aufgebaut, nur dass hier die Default-Verbindung nicht zurück zum ersten Exclusive-Gateway führt, sondern die ausgehende Verbindung ist, mit der die Schleife beendet wird. Die Verbindung mit der <condition> führt, solange diese erfüllt wird, zurück zum ersten Exclusive-Gateway. Genau wie bei der <repeatUntil> Aktivität tritt auch hier das Problem mit der Tokensemantik von BPMN auf. Diese Problematik wird in Kapitel 5.5.4 behandelt.

Das Beispiel in Abbildung 54 zeigt die <while> Schleife aus dem Beispiel in Kapitel 2. Zur Unterstützung der Verständlichkeit wird das BPMN Konstrukt expandiert und mit Start- und End-Event dargestellt.

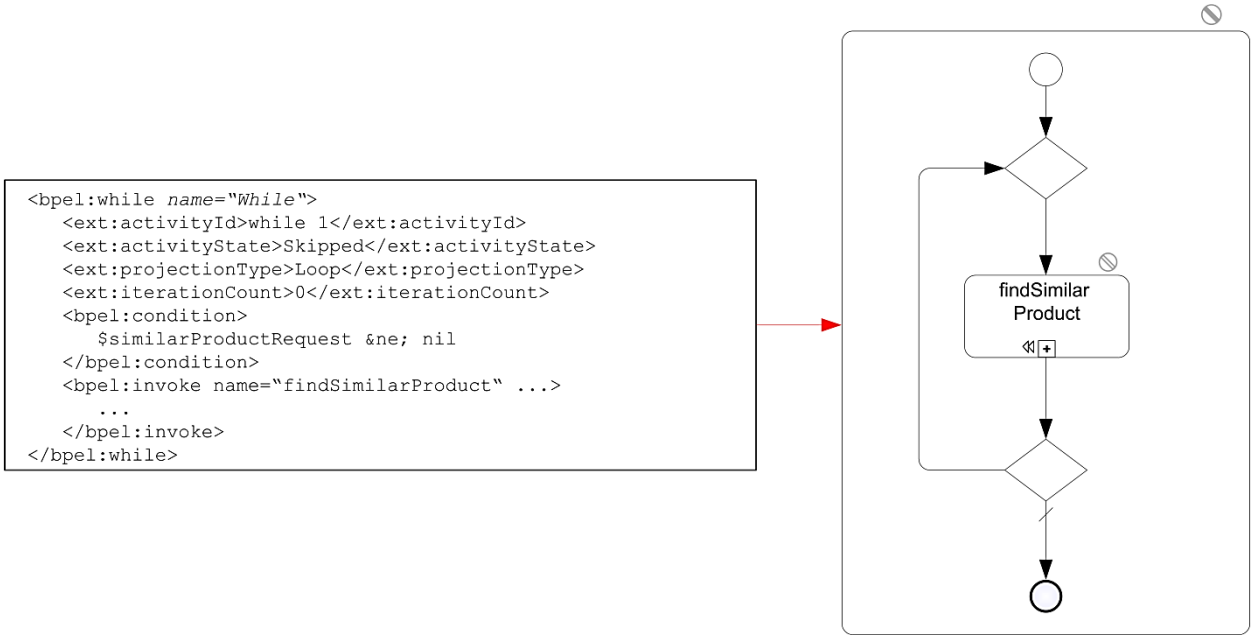

Abbildung 54: Projektion einer <while> Aktivität

5.4.17 <scope>

Alle Aktivitäten innerhalb der <scope> können einzeln dargestellt werden und somit kommt für jede Aktivität die jeweilige Zustandsübertragungsregel zum Einsatz. Wird die <scope> Aktivität jedoch als einzelner nicht expandierter Sub-Prozess dargestellt, dann wird das State Combination Pattern genutzt um die Zustände zusammenzufassen. Es wird eine Funktion über alle internen Aktivitäten erstellt, die eine Aussage über den Zustand der <scope>, als übergeordneten Sub-Prozess, macht.

5.4.18 <variable>

Variablen können in BPMN durch ein Data Object dargestellt werden. Der Zustand entspricht dabei dem aktuellen Wert der Variablen.

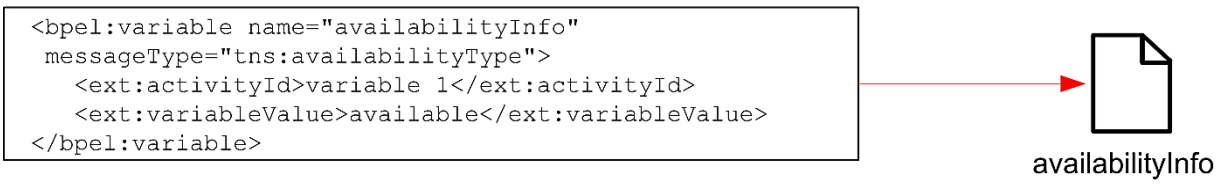

Abbildung 55: Projektion einer <variable> Aktivität

5.4.19 <validate>

Die <validate> Aktivität wird in BPMN durch einen Task dargestellt. Der Zustand wird mit dem Direct State Propagation Pattern übertragen.

```
<bpel:validate variables=" shipRequest shipNotice
itemsShipped " name="Validate">
   <ext:activityId>validate 1</ext:activityId>
   <ext:activityState>Executing</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
</bpel:validate>
```

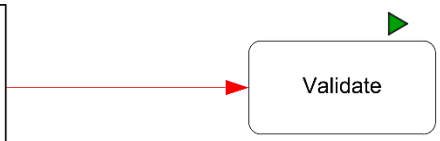

Abbildung 56: Projektion einer <validate> Aktivität

5.4.20 <compensationHandler>

Der <compensationHandler> wird als Sub-Prozess mit einem Compensation-Marker in BPMN dargestellt. Die Darstellung als Sub-Prozess hat den Vorteil, dass nicht zwischen einer Basic Activity und einer Structured Activity unterschieden werden muss. Bei der Darstellung als nicht expandierter Sub-Prozess wird der Zustand direkt übertragen, es kommt also das Direct State Propagation Pattern zum Einsatz. Handelt es sich jedoch um eine expandierte Darstellung des Sub-Prozesses, dann werden die Aktivitäten innerhalb des <compensationHandler>s einzeln behandelt und die entsprechenden Zustandsübertragungsregeln müssen verwendet werden.

```
<bpel:Invoke name="findSimilarProduct"
  inputVariable="product"
  outputVariable="similarProductRequest"
  operation="findSimilarProduct">
   <bpel:compensationHandler>
      <ext:activityId>invoke 6</ext:activityId>
      <ext:activityState>Skipped</ext:activityState>
      <ext:projectionType>Casual</ext:projectionType>
      <bpel:invoke name="cancelSearch"
        operation="cancelProductSearch0">
      </bpel:invoke>
   </bpel:compensationHandler>
</bpel:invoke>
```

Abbildung 57: Projektion einer <compensationHandler> Aktivität

5.4.21 <faultHandlers>

Die Darstellung des <faultHandlers> in BPMN wird durch einen Error Start Event, gefolgt von einem Exclusive-Gateway gelöst. Um die unterschiedlichen <catch> Elemente darstellen zu können, wird je ein Exclusive-Gateway vor und hinter die Fehlerbehandlung gesetzt. Die beiden Exclusive-Gateways haben ebenfalls einen Zustand. Wird der <faultHandlers> ausgeführt, sind die beiden Gateways im Zustand *Executing*. Nach Beendigung des <faultHandlers> werden die beiden Gateways in den Zustand *Completed* gesetzt. Für jedes <catch> Element gibt es eine eigene Verbindung, die vom ersten Exclusive Gateway ausgeht und beim zweiten Exclusive-Gateway gibt es synchron für jedes <catch> Element eine eingehende Verbindung. Die Aktivitäten innerhalb der <catch> Elemente werden einzeln behandelt und für jede wird die jeweilige Zustandsübertragungsregel verwendet. Gibt es nur ein <catchAll> Element und keine <catch> Elemente, dann können die Gateways wegfallen und

die Aktivitäten innerhalb des <catchAll> Elements werden direkt an das Error Start Event gehängt.

Durch die Tokensemantik von BPMN würde es durch die Zustände der beiden Gateways zu Problemen kommen, da der durchlaufende Token dupliziert wird. Diese Problematik wird in Kapitel 5.5.4 behandelt.

```
<bpel:faultHandlers>
   <ext:activityId>faulthandlers 1</ext:activityId>
   <ext:activityState>Ready</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
   <bpel:catchAll>
      <bpel:sequence>
         <ext:activityId>sequence 1</ext:activityId>
         <bpel:compensate>
            <ext:activityId>compensate 1</ext:activityId>
            <ext:activityState>Inactive</ext:activityState>
            <ext:projectionType>Casual</ext:projectionType>
         </bpel:compensate>
         <bpel:exit name="Exit">
            <ext:activityId>exit 1</ext:activityId>
            <ext:activityState>Inactive</ext:activityState>
            <ext:projectionType>Casual</ext:projectionType>
         </bpel:exit>
      </bpel:sequence>
   </bpel:catchAll>
</bpel:faultHandlers>
```

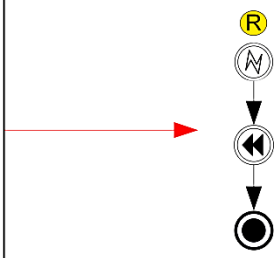

Abbildung 58: Projektion einer <faultHandlers> Aktivität

5.4.22 <process>

Das <process> Konstrukt enthält den kompletten BPEL-Prozess mit allen Aktivitäten und Elementen. Der Prozess wird in BPMN als Pool dargestellt und er ist solange im Zustand *Running*, solange eine Aktivität innerhalb des Prozesses läuft. Es wird also das State Combination Pattern verwendet. Gibt es auf Prozessebene ein <faultHandlers> Konstrukt, dann wird dieses wie im entsprechenden Kapitel beschrieben behandelt.

Ein Beispiel für den kompletten Prozess ist in Kapitel 2 zu sehen.

5.4.23 <terminationHandler>

Der <terminationHandler> wird in BPMN durch einen Sub-Prozess mit einem Termination-Marker dargestellt. Die enthaltenen Aktivitäten werden separat behandelt. Der Zustand wird durch das State Combination Pattern übertragen. Solange eine der internen Aktivitäten im Zustand *Executing* ist, ist auch der <terminationHandler> im Zustand *Executing*. Sobald alle internen Aktivitäten beendet sind, wird der Zustand auf *Completed* gesetzt.

```
<bpel:terminationHandler name="Terminate">
   <ext:activityId>terminationhandler 1</ext:activityId>
   <ext:activityState>Executing</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
   <bpel:assign name="Assign">
      <ext:activityId>assign 1</ext:activityId>
      <ext:activityState>Completed</ext:activityState>
      ...
   </bpel:assign>
   <bpel:invoke name="Invoke">
      <ext:activityId>invoke 1</ext:activityId>
      <ext:activityState>Executing</ext:activityState>
      ...
   </bpel:invoke>
</bpel:terminationHandler>
```

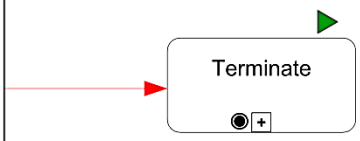

Abbildung 59: Projektion einer <terminationHandler> Aktivität

5.4.24 <eventHandlers>

Das <onEvent> Konstrukt kann in BPMN durch einen Catching Intermediate Message Event, gefolgt von einem Task beziehungsweise einem Sub-Prozess für die enthaltene Aktivität, dargestellt werden. Synchron dazu kann das <onAlarm> Konstrukt durch einen Catching Intermediate Timer Event, gefolgt von einem Task beziehungsweise einem Sub-Prozess für die enthaltene Aktivität, dargestellt werden. Solange eine Aktivität innerhalb des <scope> im Zustand *Executing* ist, ist auch der Event im Zustand *Executing*. Wenn der <scope> beendet ist, wird das Event in den Zustand *Completed* gesetzt.

```
<bpel:eventHandlers>
   <bpel:onEvent operation="querySearchStatus">
      <ext:activityId>event 1</ext:activityId>
      <ext:activityState>Executing</ext:activityState>
      <ext:projectionType>Casual</ext:projectionType>
      <bpel:scope>...</bpel:scope>
   </bpel:onEvent>
   <bpel:onEvent operation="cancelSearch">
      <ext:activityId>event 2</ext:activityId>
      <ext:activityState>Inactive</ext:activityState>
      <ext:projectionType>Casual</ext:projectionType>
      <bpel:scope>...</bpel:scope>
   </bpel:onEvent>
</bpel:eventHandlers>
```

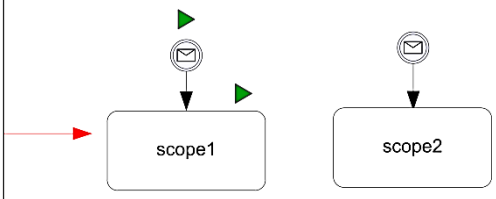

Abbildung 60: Projektion einer <eventHandlers> Aktivität

Eine Übersicht über alle Zustandsüberführungsregeln:

BPEL Aktivität	Zustandsüberführungsregel
Basis Aktivitäten	
<assign>	Direct State Propagation Pattern oder State Distribution Pattern
<empty>	Direct State Propagation Pattern
<exit>	Direct State Propagation Pattern
<invoke>	Direct State Propagation Pattern oder State Combination Pattern
<receive>	Direct State Propagation Pattern
<reply>	Direct State Propagation Pattern

<rethrow>	Direct State Propagation Pattern
<throw>	Direct State Propagation Pattern
<wait>	Direct State Propagation Pattern
Strukturierte Aktivitäten	
<flow>	State Combination Pattern
<forEach>	State Combination Pattern
<if>	State Combination Pattern
<pick>	State Combination Pattern
<repeatUntil>	State Combination Pattern
<sequence>	Jede Aktivität wird separat behandelt
<while>	State Combination Pattern
Scopes	
<compensate>	Direct State Propagation Pattern
<compensateScope>	Direct State Propagation Pattern
<scope>	Direct State Propagation Pattern oder State Combination Pattern
Variablen	
<variable>	Direct State Propagation Pattern
<validate>	Direct State Propagation Pattern
Andere Konstrukte	
<catch>	Direct State Propagation Pattern
<catchAll>	Direct State Propagation Pattern
<compensationHandler>	Direct State Propagation Pattern oder State Combination Pattern
<faultHandlers>	Direct State Propagation Pattern
<process>	State Combination Pattern

<terminationHandler>	Direct State Propagation Pattern oder State Combination Pattern
<onEvent>	Direct State Propagation Pattern
<onAlarm>	Direct State Propagation Pattern

Tabelle 6: Übersicht über die Zustandsüberführungsregeln

Die <flow>, <forEach>, <if>, <repeatUntil>, <pick> und <while> Aktivitäten haben das State Combination Pattern als Default, alle anderen Aktivitäten werden Default mäßig mit dem Direct State Propagation Pattern abgebildet.

5.5 Ablauf einer Projektion

Um einen Geschäftsprozess, der mit BPEL ausgeführt wird, mit BPMN überwachen zu können, muss zu allererst ein BPMN-Modell aus dem BPEL-Modell generiert werden. Zur Generierung des BPMN-Modells werden zwei Dinge benötigt. Zum einen das BPEL-Modell und zum anderen Informationen über die Zuordnung der Aktivitäten. Im ersten Schritt wird aus der BPEL-Datei die Mappings-Datei erstellt. In der Mappings-Datei werden die Informationen über die Zustandsüberführungen, wie die verwendete Zustandsüberführungsregel und die beteiligten activityIds und projectionIds, gespeichert. Für die Generierung werden Default-Zustandsüberführungsregeln bestimmt, mit denen eine erste Version der Mappings-Datei erstellt werden kann. Eine gewisse Individualisierbarkeit wird dadurch erreicht, dass die Mappings-Datei nach der Generierung manuell verändert werden können.

Die Generierung der BPMN-Datei ist der zweite Schritt im Ablauf. Mit Hilfe der Mappings-Datei wird aus der BPEL-Datei eine zustandslose BPMN-Datei generiert. Es wird über alle Projektionen aus der Mappings-Datei iteriert und für jede activityId wird der entsprechende Aktivitäten-Name und die Struktur aus der BPEL-Datei geholt und abgespeichert. Danach wird ein BPMN-Konstrukt mit Hilfe der gespeicherten Daten entsprechend der gespeicherten Zustandsüberführungsregel aus der Mappings-Datei erzeugt.

Wie bereits erwähnt kann anschließend die Mappings-Datei manuell bearbeitet werden, um die dargestellten Konstrukte nachträglich zu verändern. Wurde die Datei nach den individuellen Wünschen verändert, muss die BPMN-Datei erneut generiert werden, um die Änderungen darzustellen. Nach der erneuten Generierung kann die Zustandsüberführung gestartet werden. Mit den Informationen aus der BPEL-Engine wird aus der BPEL-Datei eine zustandsbehaftete BPEL-Datei. Jetzt können die aktuellen Zustandsinformationen mit Hilfe der Mappings-Datei auf die BPMN-Datei übertragen werden und generiert so eine zustandsbehaftete BPMN-Datei.

Alle Schritte des Ablaufs werden in diesem Kapitel detailliert beschrieben und durch Beispiele verdeutlicht.

Eine Übersicht über den Ablauf ist in Abbildung 61 zu sehen.

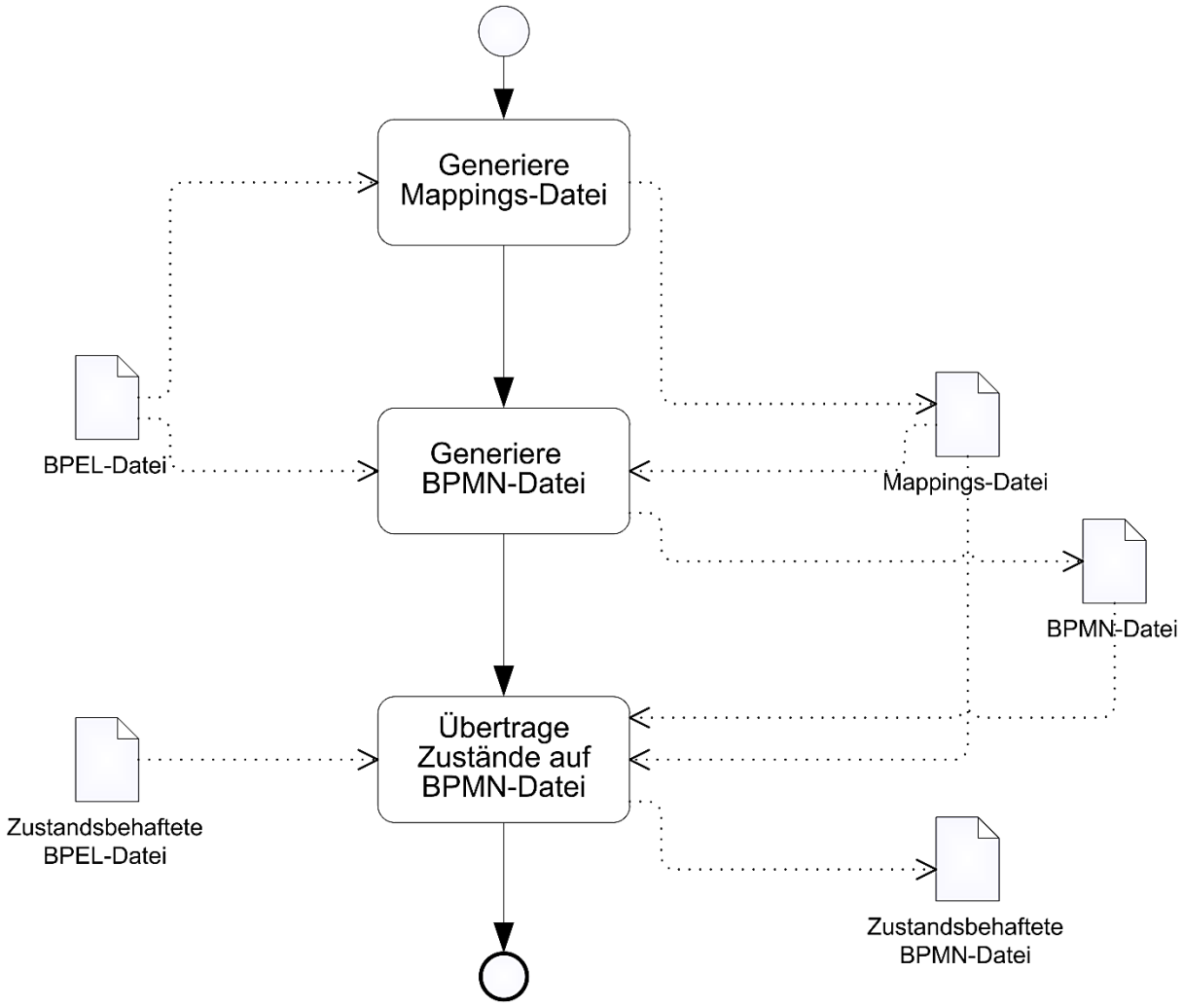

Abbildung 61: Übersicht über den Ablauf

Als erklärendes Beispiel wird der Ablauf an der <while> Schleife gefolgt von einer <wait> Aktivität aus dem Beispiel in Kapitel 2 gezeigt. In Listing 42 ist die normale BPEL-Datei für beide Aktivitäten zu sehen.

```
<bpel:while name="While">
   <ext:activityId>while 1</ext:activityId>
   <ext:activityState> </ext:activityState>
   <ext:projectionType>Loop</ext:projectionType>
   <ext:iterationCount>0</ext:iterationCount>
   <bpel:condition>
      $similarProductRequest != nil
   </bpel:condition>
   <bpel:invoke name="findSimilarProduct" inputVariable="product"
     outputVariable="similarProductRequest" operation="findSimilarProduct">
      <ext:activityId>invoke 6</ext:activityId>
      <ext:activityState> </ext:activityState>
      <ext:projectionType>Casual</ext:projectionType>
      <bpel:compensationHandler>
         <ext:activityId>compensationHandler 1</ext:activityId>
         <ext:activityState> </ext:activityState>
         <ext:projectionType>Casual</ext:projectionType>
         <bpel:invoke name="cancelSearch" operation="cancelProductSearch">
            <ext:activityId>invoke 7</ext:activityId>
            <ext:activityState> </ext:activityState>
```

```
            <ext:projectionType>Casual</ext:projectionType>
         </bpel:invoke>
      </bpel:compensationHandler>
   </bpel:invoke>
</bpel:while>
<bpel:wait name="Wait">
   <ext:activityId>wait 1</ext:activityId>
   <ext:activityState> </ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
   <bpel:for>PT10M</bpel:for>
</bpel:wait>
```

Listing 42: <while> Schleife gefolgt von einer <wait> Aktivität in BPEL

5.5.1 Mappings XML-Schema

Um festlegen zu können welche Aktivitäten auf welche Konstrukte in BPMN projiziert werden, wird eine Mappings XML-Datei erstellt. Darin sind alle Projektionen enthalten. Eine Projektion enthält dabei eine oder mehrere Aktivitäten aus BPEL und eine oder mehrere Konstrukte aus BPMN. Die Aktivitäten sind durch die activityId und die Konstrukte durch die projectionId einzigartig gekennzeichnet. Zudem wird durch projectionRule die Transformationsregel angegeben, mit der die Projektion durchgeführt wird. In Listing 43 wird das XML-Schema beschrieben und danach folgt ein Beispiel für die Mappings-Datei.

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema                                attributeFormDefault="unqualified"
elementFormDefault="qualified"                                    version="1.0"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="projections">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" name="projection">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="MappingFrom">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element maxOccurs="unbounded" name="activityId"
                     type="xsd:string" />
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
              <xsd:element name="MappingTo">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element maxOccurs="unbounded" name="projectionId"
                     type="xsd:string" />
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
              <xsd:element name="projectionRule" type="xsd:string" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 43: Mappings XML-Schema

In Listing 44 ist die Mappings-Datei für die <while> Schleife gefolgt von der <wait> Aktivität zu sehen.

```
<?xml version="1.0" encoding="UTF-8"?>
<projections>
   ...
   <projection>
      <MappingFrom>
         <activityId>while 1</activityId>
         <activityId>invoke 6</activityId>
         <activityId>compensationHandler 1</activityId>
         <activityId>invoke 7</activityId>
      </MappingFrom>
      <MappingTo>
         <projectionId>SubProcess 5</projectionId>
      </MappingTo>
      <projectionRule>Combination</projectionRule>
   </projection>
   <projection>
      <MappingFrom>
         <activityId>wait 1</activityId>
      </MappingFrom>
      <MappingTo>
         <projectionId>Task 10</projectionId>
      </MappingTo>
      <projectionRule>Direct</projectionRule>
   </projection>
   ...
</projections>
```

Listing 44: Mappings-Datei für das <while> und <wait> Beispiel

In dem Beispiel sind zwei Projektionen zu sehen. Bei der ersten wird eine <while> Schleife in BPMN als Sub-Prozess dargestellt. Als Zustandsüberführungsregel kommt das State Combination Pattern zum Einsatz. Bei der zweiten Projektion wird eine <wait> Aktivität in BPMN als ein einzelner Task dargestellt. Hierbei kommt als Zustandsüberführungsregel das Direct State Propagation Pattern zum Einsatz.

Zur Generierung der Mappings-Datei wird der Pseudo-Code in Listing 45 verwendet:

```
GenerateMappings(BPELProcess p){
   //schreibe den XML-Header für die Mappings-Datei
   writeHead;
   Array activityList = getAllActivities(p);
   FOR each activity in activityList
      activityId = getId(activity);
      type = getType(activity);
      mapping = getDefaultMapping(type);
      rule = getRule(mapping);
      projectionIds = generateNewProjectionId(mapping);
      //schreibe den XML-Code für die Projektion
      writeProjection(activityId, projectionIds, rule);
   NEXT;
   //schreibe das XML-Closing für die Mappings-Datei
   writeTail;
}
```

Listing 45: Generierung der Mappings-Datei

5.5.2 Zustandsüberführungsregeln

In Kapitel 5.2 wurden drei Muster zur Zustandsüberführung vorgestellt. Diese werden in diesem Kapitel in Zusammenhang mit dieser Arbeit definiert und in Pseudo-Code beschrieben. In Hinsicht auf die weitere Entwicklung können weitere Zustandsüberführungsregeln definiert und umgesetzt werden.

5.5.2.1 Direct State Propagation Pattern

Das Direct State Propagation Pattern überführt den Zustand einer Aktivität in BPEL in den Zustand eines Konstrukts in BPMN. Da der Zustand einfach übernommen wird, ist dies das einfachste der drei Muster. Die Methode für das Muster bekommt als Eingabe eine activityId und eine projectionId, liest dann den aktuellen Zustand der Aktivität aus und schreibt ihn zum passenden Konstrukt in BPMN.

```
Directstate (activityID string, projectionID string) {
   projectionState  = getState(activityID);
   write(projectionID, projectionState);
}
```

Listing 46: Pseudo-Code zum Direct State Propagation Pattern

5.5.2.2 State Combination Pattern

Für das State Combination Pattern gibt es zwei Methoden. Eine für normale Aktivitäten und eine für Schleifen. Die normalen Aktivitäten umfassen <invoke>, <flow>, <if>, <pick>, <scope>, <compensationHandler>, <faultHandlers>, <process> und <terminationHandler>. Als Schleifen werden <forEach>, <repeatUntil> und <while> bezeichnet. Beim State Combination Pattern werden mehrere Zustände aus BPEL auf einen Zustand in BPMN abgebildet. Um das zu erreichen müssen bestimmte Bedingungen erfüllt sein, die im Folgenden für beide Methoden beschrieben werden.

Die Methode für die normalen Aktivitäten bekommt als Eingabe mehrere activityIds und eine projectionId. Als erstes wird die Variable projectionState auf einen leeren String gesetzt und anschließend wird über alle activityIds iteriert. Solange der projectionState auf *Completed* steht wird die activityId abgehandelt, ist dies nicht der Fall werden die restlichen Aktivitäten nicht weiter beachtet, da eine weitere Änderung des Zustands eine Inkonsistenz in Bezug auf die in Kapitel 4.2 vorgestellten Regeln verursachen würde. Während der Abarbeitung der Aktivität, wird der activityState ausgelesen und anschließend je nach Zustand der projectionState verändert. Bei *Inactive* wird der projectionState ebenfalls auf *Inactive* gesetzt. Dasselbe gilt für die Zustände *Skipped*, *Executing*, *Compensated*, *Completed*, *Faulted* und *Terminated*. Beim Zustand *Ready* wird der bisherige projectionState abgefragt, ist dieser *Completed*, so wird der projectionState auf *Executing* gesetzt, ansonsten auf *Ready*. Abschließend wird der projectionState zum passenden Konstrukt in der BPMN-Datei geschrieben.

```
CombinationState (activityIDs array, projectionID string){
   projectionState = "";
   //ändere den projectionState abhängig von allen activityStates
   FOR each activityID in activityIds
      IF pprojectionState == Completed | Iteration Completed
      activityState = getState(activityID);
      CASE activityState:
         Inactive: projectionState = Inactive;
         Ready: IF projectionState == Completed
               then  projectionState = Executing
```

```
                ELSE projectionState = Ready;
          Skipped: projectionState = Skipped;
          Executing: projectionState = Executing;
          Compensated:  projectionState = Compensated;
          Completed: projectionState = Completed;
          Faulted: projectionState = Faulted;
          Terminated: projectionState = Terminated;
       ESAC;
       FI;
    NEXT;
    //schreibe den Zustand in die BPMN-Datei
    write(projectionID, projectionState);
}
```

Listing 47: Pseudo-Code für das normale State Combination Pattern

Für die Methode für Schleifen wird die obere Methode erweitert. Als weitere Eingaben kommen der iterationCount und der finalCounterValue hinzu. Wobei der finalCounterValue nur für die <forEach> Schleife von Bedeutung ist. Für die <repeatUntil> und <while> Schleifen wird eine Bedingung in der BPEL-Datei angegeben. Die Zustände werden durch den Zustand *Iteration Completed* ergänzt. Nach der Abarbeitung aller activityIds kommt eine Abfrage über den Zustand hinzu. Ist der projectionState nach der Abarbeitung der <forEach> Schleife im Zustand *Iteration Completed*, dann wird geprüft, ob der finalCounterValue erreicht wurde. Für <repeatUntil> und <while> wird geprüft, ob die angegebene Bedingung erfüllt wurde. Bei der <while> Schleife wird diese Abfrage vor der inneren Ausführung gemacht und nicht am Ende. Trifft der Fall zu, dass die jeweilige Bedingung erfüllt ist, wird der projectionState auf den Zustand *Executing* und alle activityIds im Zustand *Iteration Completed* werden in den Zustand *Inactive* gesetzt. Ansonsten werden der projectionState und alle activityStates auf *Completed* gesetzt.

```
CombinationState (activityIDs array, projectionID string, iterationCount
int, finalCounterValue int, condition){
   projectionState = "";
   FOR each activityID
      IF projectionstate == Completed | Iteration Completed
      activityState = getState(activityID);
      CASE activityState:
         Inactive: projectionState = Inactive;
         Ready: IF projectionState == Completed
                then  projectionState = Executing
                ELSE projectionState = Ready;
         Skipped: projectionState = Skipped;
         Executing: projectionState = Executing;
         Compensated:  projectionState = Compensated;
         Completed: projectionState = Completed;
         Faulted: projectionState = Faulted;
         Terminated: projectionState = Terminated;
         Iteration Completed: projectionState = Iteration Completed;
      ESAC;
      FI;
   NEXT;
   //Abschnitt für die <forEach> Schleife
   //Hier wird geprüft, ob der finalCounterValue erreicht wurde
   IF projectionState == Iteration Completed
      IF iterationCount < finalCounterValue
         projectionState = Executing;
         FOR each activityID
            activityState = getState(activityID);
            IF activityState == Iteration Completed
```

```
               activityState = Inactive;
            FI;
         NEXT;
       ELSE
         projectionState = Completed;
         FOR each activityID
            activityState = getState(activityID);
            IF activityState == Iteration Completed
               activityState = Completed;
            FI;
         NEXT:
      FI;
   FI;
   write(projectionID, projectionState);
}
```

Listing 48: Pseudo-Code für das State Combination Pattern für Schleifen

5.5.2.3 State Distribution Pattern

Das State Distribution Pattern überführt den Zustand einer Aktivität in BPEL in den Zustand mehrerer Konstrukte in BPMN. Der Zustand der Aktivität wird für alle Konstrukte übernommen. Die Methode für das Muster bekommt als Eingabe eine activityId und mehrere projectionId, liest dann den aktuellen Zustand der Aktivität aus und schreibt ihn zu allen passenden Konstrukten in BPMN.

```
DistributionState (activityID string, projectionIDs array){
   projectionState  = getState(activityID);
   FOR each projectionID
      write(projectionID, projectionState);
   NEXT;
}
```

Listing 49: Pseudo-Code für das State Distribution Pattern

5.5.3 Generierung der BPMN-Datei

5.5.3.1 XML-Elemente in BPMN

In diesem Unterkapitel werden die wichtigsten XML-Tags für die BPMN-Datei vorgestellt. Das <process>-Tag für den eigentlichen Prozess wird vom <definitions>-Tag umgeben. In diesem Tag werden Informationen wie zum Beispiel der Namespace angegeben. Alle anderen Tags haben einen Identifier, kurz eine id, und einen Namen. Zusätzlich gibt es noch die Attribute sourceRef, gibt die Quelle an, targetRef, gibt das Ziel an und gatewayDirection, gibt an ob das Gateway erstellend (diverging) oder synchronisierend (converging) ist. Für detaillierte Informationen über die verwendeten XML-Tags wird auf [19] verwiesen.

In Listing 50 ist die der Teil der BPMN-Datei für die <while> Schleife und die folgende <wait> Aktivität zu sehen.

```
<subProcess id="subprocess 5" name="findSimilarProduct" />
<sequenceFlow id="sequenceflow 25" sourceRef="subprocess 5"
          targetRef="task 10" />
<task id="task 10" name="Wait" />
```

Listing 50: BPMN-Datei für die <while> Schleife und die <wait> Aktivität

Das Listing 51 zeigt die BPMN-Datei für das Beispiel aus Kapitel 2.

```
<definitions id="customerProductRequest"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://schema.omg.org/spec/BPMN/2.0 BPMN20.xsd" >
  <process id="process 1" name="main_process" processType="None">
     <startEvent id="Start" />
     <intermediateCatchEvent id="event 1" name="processFaultHandler">
        <errorEventDefinition>
           <subProcess id="subprocess 1" name="faultHandlers">
              <exclusiveGateway id="exclusivegateway 1"
                gatewayDirection="diverging" />
              <sequenceFlow id="sequenceflow 1"
                sourceRef="exclusivegateway 1" targetRef="event 2" />
              <intermediateThrowEvent id="event 2" name="compensate">
                 <compensateEventDefintion />
              </intermediateThrowEvent>
              <sequenceFlow id="sequenceflow 2" sourceRef="event 2"
                targetRef="event 3" />
              <endEvent id="event 3" name="terminate">
                 <terminateEventDefintion />
              </endEvent>
              <sequenceFlow id="sequenceflow 3" sourceRef="event 3"
                targetRef="exclusivegateway 2" />
              <exclusiveGateway id="exclusivegateway 2"
                gatewayDirection="converging" />
           </subProcess>
        </errorEventDefinition>
     </intermediateCatchEvent>
     <sequenceFlow id="sequenceflow 4" sourceRef="Start"
       targetRef="task 1" />
     <task id="task 1" name="receiveProductRequest" />
     <sequenceFlow id="sequenceflow 5" sourceRef="task 1"
       targetRef="task 2" />
     <task id="task 2" name="checkAvailability" />
     <sequenceFlow id="sequenceflow 6" sourceRef="task 2"
       targetRef="subprocess 2" />
     <subProcess id="subprocess 2" name="productAvailable">
        <sequenceFlow id="sequenceflow 7" sourceRef="subprocess 2"
          targetRef="exclusivegateway 3" />
        <exclusiveGateway id="exclusivegateway 3"
          gatewayDirection="diverging" />
        <sequenceFlow id="sequenceflow 8" sourceRef="exclusivegateway 3"
          targetRef="task 3">
           <conditionExpression xsi:type="tFormalExpression">
           ${inventoryLevel > 0}
           </conditionExpression>
        </sequenceFlow>
        <task id="task 3" name="empty" />
        <sequenceFlow id="sequenceflow 9" sourceRef="task 3"
          targetRef="exclusivegateway 4" />
        <sequenceFlow id="sequenceflow 10" sourceRef="exclusivegateway 3"
          targetRef="task 4">
           <conditionExpression xsi:type="tFormalExpression">
           ${inventoryLevel = 0} && ${inProduction > 0}
           </conditionExpression>
        </sequenceFlow>
        <task id="task 4" name="checkReorder" />
        <sequenceFlow id="sequenceflow 11" sourceRef="task 4"
          targetRef="subprocess 3" />
        <subprocess id="subprocess 3" name="forEach">
           <exclusiveGateway id="exclusivegateway 5"
             gatewayDirection="diverging" />
           <sequenceFlow id="sequenceflow 12"
```

```
            sourceRef="exclusivegateway 5" targetRef="subprocess 4" />
          <subProcess id="subprocess 4" name="flow">
             <sequenceFlow id="sequenceflow 13" sourceRef="subprocess 4"
               targetRef="parallelgateway 1" />
             <parallelGateway id="parallelgateway 1"
               gatewayDirection="diverging" />
             <sequenceFlow id="sequenceflow 14"
               sourceRef="parallelgateway 1" targetRef="task 5" />
             <task id="task 5" name="receiveOffer" />
             <sequenceFlow id="sequenceflow 15" sourceRef="task 5"
               targetRef="task 6" />
             <sequenceFlow id="sequenceflow 16" sourceRef="task 5"
               targetRef="task 7" />
             <task id="task 6" name="checkDeliveryDate" />
             <task id="task 7" name="checkPrice" />
             <sequenceFlow id="sequenceflow 17" sourceRef="task 6"
               targetRef="task 8" />
             <sequenceFlow id="sequenceflow 18" sourceRef="task 7"
               targetRef="task 8" />
             <task id="task 8" name="assignOffer" />
             <sequenceFlow id="sequenceflow 19" sourceRef="task 8"
               targetRef="parallelgateway 2" />
             <parallelGateway id="parallelgateway 2"
               gatewayDirection="converging" />
          </subProcess>
          <sequenceFlow id="sequenceflow 20" sourceRef="parallelgateway 2"
            targetRef="exclusivegateway 6" />
          <exclusiveGateway id="exclusivegateway 6"
            gatewayDirection="converging" />
          <sequenceFlow id="sequenceflow 21"
            sourceRef="exclusivegateway 6"
            targetRef="exclusivegateway 5" />
       </subProcess>
       <sequenceFlow id="sequenceflow 22" sourceRef="exclusivegateway 6"
         targetRef="task 9" />
       <task id="task 9" name="evaluateResults" />
       <sequenceFlow id="sequenceflow 23" sourceRef="task 9"
         targetRef="exclusivegateway 4" />
       <sequenceFlow id="sequenceflow 24" sourceRef="exclusivegateway 3"
         targetRef="subprocess 5">
          <conditionExpression xsi:type="tFormalExpression">
          ${inventoryLevel = 0} && ${inProduction = 0}
          </conditionExpression>
       </sequenceFlow>
       <subProcess id="subprocess 5" name="findSimilarProduct" />
       <sequenceFlow id="sequenceflow 25" sourceRef="subprocess 5"
         targetRef="task 10" />
       <task id="task 10" name="summarizeResults" />
       <sequenceFlow id="sequenceflow 26" sourceRef="task 10"
         targetRef="exclusivegateway 4" />
       <exclusiveGateway id="exclusivegateway 4"
         gatewayDirection="converging" />
    </subProcess>
    <sequenceFlow id="sequenceflow 27" sourceRef="exclusivegateway 4"
      targetRef="task 11" />
    <task id="task 11" name="productOffering" />
    <sequenceFlow id="sequenceflow 28" sourceRef="task 11"
      targetRef="End" />
    <endEvent id="End" />
  </process>
</definitions>
```

Listing 51: BPMN-Datei

5.5.3.2 Generierung

Die BPEL-Datei und die Mappings-Datei dienen als Grundlage für die Generierung der BPMN-Datei. Es wird über alle Projektionen aus der Mappings-Datei iteriert und für jede activityId wird der entsprechende Aktivitäten-Name aus der BPEL-Datei geholt und abgespeichert. Danach wird ein BPMN-Konstrukt entsprechend der gespeicherten Zustandsüberführungsregel aus der Mappings-Datei erzeugt. Die XML-Tags für BPMN wurden dafür in Kapitel 5.5.3.1 eingeführt.

Der Algorithmus in Listing 52 zeigt die Generierung der BPMN-Datei in Pseudo-Code:

```
GenerateBPMN(BPELProcess p, MappingsDatei m){
//schreibe den XML-Header für die BPMN-Datei
writeHead;
Array projections = getProjections(m);
FOR each projection in projections
   activityId = getAcitivityId(projection);
   name = getName(activityId);
   projectionId = getProjectionId(projection);
   rule = getRule(projection);
   //Zur Einhaltung der Struktur wird der BPEL-Prozess bei der
   //Generierung als Parameter mitgegeben
   writeBPMNContruct(p, name, projectionId, rule);
NEXT;
//schreibe das XML-Closing für die BPMN-Datei
writeTail;
}
```

Listing 52: Generierung der BPMN-Datei

5.5.4 Probleme bei der Projektion

Bei allen Aktivitäten, die mit Hilfe von Gateways dargestellt werden, würde das Problem der duplizierten Tokens auftreten. Um dieses zu umgehen werden die Zustände der Gateways nicht übertragen, sondern intern aus den Zuständen aller, zwischen den beiden Gateways enthaltenen, Aktivitäten berechnet. Während mindestens eine dieser Aktivitäten im Zustand *Executing* ist, sind die beiden Gateways ebenfalls im Zustand *Executing*. Sobald alle enthaltenen Aktivitäten im Zustand *Completed* sind, werden auch die beiden Gateways in den Zustand *Completed* gesetzt.

5.6 Anpassungen an der Mappings-Datei

Nach der Generierung der Mappings-Datei ist es möglich, die enthaltenen Projektionen manuell zu verändern. Dabei müssen die activityIds korrekt weiterverwendet werden, da es sonst zu Problemen bei der Generierung der BPMN-Datei kommen kann. Wurden alle gewünschten Änderungen vollzogen, kann, mit Hilfe der geänderten Mappings-Datei, die BPMN-Datei erneut erzeugt werden. Dafür wird überprüft, ob die Mappings-Datei verändert wurde, und falls dies der Fall ist, wird die generateBPMN-Methode erneut aufgerufen.

```
checkMappingsFile(BPELProcess p, MappingsDatei m){
   BOOL fileStateChanged = checkForChanges(m);
   IF fileStateChanged == true
      generateBPMN(p, m);
   FI
```

```
}
```

Listing 53: checkMappingsFile als Pseudo-Code

5.7 Mapping der Zustände

Als letzter Schritt wird die Übertragung der Zustände gemacht. Die zustandsbehaftete BPEL-Datei, die Mappings-Datei und die BPMN-Datei bilden die Grundlage für die Generierung der zustandsbehafteten BPMN-Datei.

In Listing 54 ist die zustandsbehaftete BPEL-Datei für das Beispiel mit der <while> Schleife gefolgt von der <wait> Aktivität zu sehen.

```
<bpel:while name="While">
   <ext:activityId>while 1</ext:activityId>
   <ext:activityState> </ext:activityState>
   <ext:projectionType>Loop</ext:projectionType>
   <ext:iterationCount>0</ext:iterationCount>
   <bpel:condition>
      $similarProductRequest != nil
   </bpel:condition>
   <bpel:invoke name="findSimilarProduct" inputVariable="product"
     outputVariable="similarProductRequest" operation="findSimilarProduct">
      <ext:activityId>invoke 6</ext:activityId>
      <ext:activityState>Skipped</ext:activityState>
      <ext:projectionType>Casual</ext:projectionType>
      <bpel:compensationHandler>
         <ext:activityId>compensationHandler 1</ext:activityId>
         <ext:activityState>Skipped</ext:activityState>
         <ext:projectionType>Casual</ext:projectionType>
         <bpel:invoke name="cancelSearch" operation="cancelProductSearch">
            <ext:activityId>invoke 7</ext:activityId>
            <ext:activityState>Skipped</ext:activityState>
            <ext:projectionType>Casual</ext:projectionType>
         </bpel:invoke>
      </bpel:compensationHandler>
   </bpel:invoke>
</bpel:while>
<bpel:wait name="Wait">
   <ext:activityId>wait 1</ext:activityId>
   <ext:activityState>Skipped</ext:activityState>
   <ext:projectionType>Casual</ext:projectionType>
   <bpel:for>PT10M</bpel:for>
</bpel:wait>
```

Listing 54: zustandsbehaftete BPEL-Datei für die <while> Schleife und die <wait> Aktivität

In Listing 55 ist die zustandsbehaftete BPMN-Datei für das Beispiel mit der <while> Schleife gefolgt von der <wait> Aktivität zu sehen.

```
<subProcess id="subprocess 5" name="findSimilarProduct" state="Skipped" />
<sequenceFlow id="sequenceflow 25" sourceRef="subprocess 5"
          targetRef="task 10" />
<task id="task 10" name="Wait" state="Skipped" />
```

Listing 55: zustandsbehaftete BPMN-Datei für die <while> Schleife und die <wait> Aktivität

6 Zusammenfassung und Ausblick

Das Thema dieser Diplomarbeit war die sprachübergreifende Überwachung von Geschäftsprozessen. Da es sich um ein großes Themengebiet handelt, wurde das Thema für diese Diplomarbeit auf die beiden Sprachen BPEL und BPMN eingeschränkt. Nach dem Schaffen von Grundlagen durch die Beschreibung der wichtigsten BPEL und BPMN Konstrukte, sowie der Erläuterung an Beispielen, wurde eine Analyse der Transformation zwischen BPEL und BPMN gemacht. Die Darstellungen der jeweiligen BPEL Aktivitäten in BPMN wurden erläutert und in einer Tabelle zusammengefasst.

Im Hauptteil der Arbeit wurde ein Ansatz zur Zustandsübertragung beschrieben und drei Muster für die Zustandsübertragung vorgestellt. Als Grundlage der Zustandsübertragung wurden zwei Zustandsräume, für den Prozess und für die Aktivitäten, definiert. Diese gelten für BPEL und BPMN gleichermaßen. Zur Darstellung der Beispiele wurden Symbole für die verschiedenen Zustände eingeführt und in Tabellen zusammengefasst.

Danach folgten die notwendigen Änderungen an den Dateiformaten für die beiden Sprachen und deren Umsetzung. Darauf folgte die Beschreibung der Zustandsübertragung für die verwendeten BPEL Aktivitäten und deren Darstellung in BPMN. Nach Schaffung der Grundlagen wurde der Ablauf zur Generierung einer BPMN-Datei aus einer BPEL-Datei beschrieben. Als ersten Schritt wurde die zusätzliche Mappings-Datei eingeführt, in der die Zuordnungen der BPEL Aktivitäten zu den BPMN Konstrukten festgehalten werden. Mit Hilfe der Mappings-Datei wurde dann aus der BPEL-Datei die BPMN-Datei erzeugt. Zur Unterstützung des Verständnisses wurden die Schritte zusätzlich in Code-Form dargestellt.

Als letzten Schritt wurde die Individualisierbarkeit der BPMN-Datei erläutert. Die Möglichkeit die Mappings-Datei von Hand ändern zu können, bietet genau diese Individualisierbarkeit der BPMN-Datei. Wurden die in der Mappings-Datei enthaltenen Projektionen wunschgemäß verändert, kann die BPMN-Datei erneut aus der Mappings-Datei und der BPEL-Datei generiert werden.

Für die Zukunft bleibt abzuwarten, ob es eine standardisierte Darstellung von BPEL auf BPMN geben wird. Das hier vorgestellte Konzept beruht zwar auf den Standards von BPEL und BPMN, aber eine standardisierte Darstellung wurde bisher nicht von einer Organisation erstellt. Bisher ist auch noch keine Lösung in Sicht, da sich die OMG nur mit der Transformation von BPMN zu BPEL beschäftigt und OASIS hat in der BPEL-Spezifikation geschrieben, dass eine Transformation von BPEL zu BPMN „out of scope“ ist. Daher bleibt nur die Hoffnung, dass sich die Einstellungen dazu in Zukunft noch ändern.

Als Erweiterung zu dieser Arbeit könnten alle Zustandsübertragungsmuster aus [18] umgesetzt werden. Zudem wäre es ein interessantes Thema, die BPMN-Datei nicht generieren zu lassen, sondern eine selbst erstellte Datei zu verwenden. Dabei müssten aber Probleme mit der Soundness und der Vollständigkeit von Projektionen betrachtet und gelöst werden. Zudem wäre es natürlich denkbar Zustandsüberführungen zwischen anderen Sprachen, nicht nur BPEL und BPMN, zu betrachten und auszuarbeiten.

Literaturverzeichnis

[1] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guizar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. v. d. Rijn, P.Yendluri, A. Yiu:

OASIS Web Services Business Process Execution Language Version 2.0, BPEL 2.0, 2007

[2] G.Latuske:

Sichten auf Geschäftsprozesse als Werkzeug zur Darstellung laufender Prozessinstanzen

Universität Stuttgart, Diplomarbeit Nr. 3036, 2010

[3] Business Process Illustrator

http://sourceforge.net/projects/bpi/

[4] Object Management Group:

Business Process Model and Notation Version 2.0, BPMN 2.0, 2011

[5] BPEL Designer Project

http://www.eclipse.org/bpel/

[6] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. König, F. Leymann,
R. Müller, G. Pfau, K. Plösser, R. Rangaswamy, A. Rickayzen, M. Rowley, P. Schmidt,
I. Trickovic, A. Yiu, M. Zeller:

WS-BPEL Extension for People (BPEL4People), Version 1.0, 2007

[7] Business Process Management Software

http://bpmsoftware.wordpress.com/free-bpa-tools/

[8] D. Schumm:

Graphische Modellierung von BPEL Prozessen unter der Verwendung der BPMN Notation

Universität Stuttgart, Diplomarbeit Nr. 2720, 2008

[9] OASIS, Advanced open standards for the information society

http://www.oasis-open.org/

[10] F. Leymann, D. Roller, S. Thatte:

Goals oft he BPEL4WS Specification, 2003

[11] Object Management Group

http://www.omg.org/

[12] B. Silver:

BPMN Method and Style: A Levels-Based Methodology for BPM Process Modeling and Improvement Using BPMN 2.0, 2009

[13] C. Ouyang, M. Dumas, A. H. M. Hofstede, W. M. P. van der Aalst:

Pattern-based Translation of BPMN Process Models to BPEL Web Services

[14] J. Mendling, K. B. Lassen, U. Zdun:

Transformation Strategies between Block-Oriented and Graph-Oriented Process Modelling Languages

[15] J. Recker, J. Mendling:

On The Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages, 2006

[16] D. Karastoyanova, R. Khalaf, R. Schroth, M. Paluszek, F. Leymann:

BPEL Event Model, Universität Stuttgart, 2006

[17] T. Steinmetz:

Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE

Universität Stuttgart, Diplomarbeit Nr. 2729, 2008

[18] D. Schumm, G. Latuske, F. Leymann, R. Mietzner, T. Scheibler:

State Propagation For Business Process Monitoring On Different Levels Of Abstraction

Universität Stuttgart, 2011

[19] JBPMN, Darstellung von BPMN in XML-Notation:

http://docs.jboss.com/jbpm/v4/devguide/html_single/#basicConstructsEvents

Abbildungsverzeichnis

Listingsverzeichnis

Tabellenverzeichnis

Erklärung

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

Stuttgart, den 04.08.2011

_____

(Eike Klenk)