Institut für Architektur von Anwendungssystemen (IAAS)

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Diplomarbeit Nr. 2720

# Graphische Modellierung von BPEL Prozessen unter Verwendung der BPMN Notation

David Schumm

| | |
|---|---|
| Studiengang: | Informatik |
| Prüfer: | Prof. Dr. Frank Leymann |
| Betreuer: | Dr. Dimka Karastoyanova |
| | Dipl. Inf. Jörg Nitzsche |
| begonnen am: | 12.11.2007 |
| beendet am: | 15.04.2008 |
| CR-Klassifikation: | D.2.2, H.4.1, H.5.2 |

## Kurzfassung

Der nächste Evolutionsschritt in der Architektur von IT-Systemen gibt den Unternehmen die Kontrolle über ihre IT zurück. Service-Orientierte Architekturen (SOA) auf Basis von Web Service Technologie ermöglichen die Trennung der Businesslogik von der Anwendungslogik. Der Industriestandard zur Orchestrierung von Web Services (WS), die Business Process Execution Language (BPEL), ermöglicht dies auf einem zuvor nie dagewesenen Niveau. Allerdings ist BPEL eine XML-basierte Sprache. Als solche ist sie für Business Analysten und Manager, die eigentlichen Architekten der Geschäftsprozesse, schwierig in der Handhabung und dazu wenig intuitiv.

Für die Modellierung von Geschäftsprozessen war lange Zeit kein allgemein akzeptierter Standard in Sicht. Nun deutet alles darauf hin, dass mit der Business Process Modeling Notation (BPMN) ein solcher Standard gefunden ist. Diese Darstellungsform von Geschäftsprozessen kann die uneinheitlichen Flussdiagramme ablösen, die zurzeit für diesen Zweck verwendet werden. BPMN würde damit für eine hohe Präzision, ein klares Verständnis und eine sehr gute Portabilität und Interoperabilität in der Darstellung und Kommunikation von Geschäftsprozessen sorgen.

Die Verbindung dieser beiden Standards in einem Werkzeug zur graphischen Prozessmodellierung ist der Inhalt und zugleich das Ziel dieser Arbeit. Es wird versucht, die „Business-IT Gap“ - also die Kluft, die zwischen Business und IT besteht - ein Stück weit zu überwinden.

Die Schwierigkeiten und Probleme, die dabei auftreten, werden analysiert und diskutiert. Sowohl die fachlichen, als auch die technischen Hintergründe werden umfassend dargestellt. Der „State of the Art“ der Forschung und Entwicklung wird betrachtet, um daraus eine Verbindung von BPMN und BPEL auf wissenschaftlicher Basis ableiten zu können. Diese Verbindung wird mittels einer praktischen Umsetzung demonstriert und anschließend werden die gewonnenen Erkenntnisse zusammengefasst. Zum Abschluss werden Möglichkeiten und Grenzen der hier präsentierten Lösung aufgezeigt und es wird ein Blick in eine mögliche Zukunft gewagt.

# Inhaltsverzeichnis

## Abbildungsverzeichnis

## Listingverzeichnis

## Tabellenverzeichnis

# 1 Einleitung

## 1.1 Einführung

Als Einführung in das Themengebiet ist es hilfreich, zunächst einen kurzen Blick in die Vergangenheit werfen, um die aktuellen Entwicklungen in der Gegenwart besser zu erfassen (siehe [AaHe02, S. 25 f.]):

*1975-1985: database management - "take data management out of the applications"*

*1985-1995: user-interface management - "take the user interface out of the applications"*

*1995-2005: workflow management - "take the business processes out of the applications"*

Wir schreiben mittlerweile das Jahr 2008, *Web Services (WS)* sind die führende Technologie um das Schlüsselproblem zu lösen, dem sich die Unternehmen heute stellen müssen: die Anwendungsintegration (AI). Sowohl die unternehmensinterne Anwendungsintegration (Enterprise Application Integration, EAI), als auch die Integration mit Geschäftspartnern (Business Process Integration, BPI) kann durch eine lose Kopplung von Anwendungen unter der Verwendung von Web Service Schnittstellen erreicht werden.

Genau an dieser Stelle spielt die *Business Process Execution Language (BPEL)* eine wichtige Rolle: Mit dieser Sprache können einzelne Komponenten, die eine Web Service Schnittstelle anbieten, zu einem koordinierten Geschäftsprozess zusammengefasst werden. Dabei spricht man auch von *Web Service Orchestration* oder auch von *Workflows*[1]. Der so definierte Geschäftsprozess bietet nach außen hin wiederum eine Web Service Schnittstelle an und kann somit auch in weitere Prozesse integriert werden. Mit BPEL kann sowohl die interne Implementierung eines Prozesses definiert, als auch der Grad der Offenlegung in Business-To-Business (B2B) Interaktionen festgelegt werden (vgl. [ACKM04, S. 284]).

Die Trennung der Geschäftsprozesse von der darin verwendeten Anwendungslogik ermöglicht es einem Unternehmen, flexibler und schneller auf neue Anforderungen zu reagieren. In diesem Zusammenhang wird das *Business Process (Re-)Engineering (BPR)* von *Business Analysten* oder auch *Process Modelers* durchgeführt (vgl. [LeRo00, S. 62]). Das Business Engineering befasst sich mit der Analyse, Spezifikation und der Modellierung von Geschäftsprozessen und den damit in Zusammenhang stehenden Ressourcen und deren Organisation (vgl. [LeRo00, S. 30 ff.]). *Business Process Reengineering* steht für ein fundamentales Überdenken und eine radikale Neukonstruktion der Geschäftsprozesse mit dem Ziel, gravierende Verbesserungen in Bezug auf die Kosten, die Zeit, die Qualität, den Durchsatz und die angebotenen Dienstleistungen zu erreichen.

*Business Analysten* verwenden für die Modellierung der Geschäftsprozesse graphische Modellierungswerkzeuge; jedoch sind die Darstellungsweise und die Semantik der modellierten Prozesse dabei von Tool zu Tool mehr oder weniger unterschiedlich. Dies ist in vielerlei Hinsicht kontraproduktiv, beispielsweise für die Abstimmung mit der zuständigen IT, die diesen Prozess implementieren soll oder auch für die Kommunikation untereinander. Mit mittlerweile über 50 unterstützenden Werkzeugen (vgl. [BFV07]) zeichnet sich die *Business Process Modeling Notation (BPMN)* immer mehr als Standard für die Modellierung von Geschäftsprozessen ab und kann diesem Missstand entgegenwirken.

[1] Der Begriff Workflow wird in [WfMC08] definiert: „the automation of business procedures or 'workflows' during which documents, information or tasks are passed from one participant to another in a way that is governed by rules or procedures."

„BPMN creates a standardized bridge for the gap between the business process design and process implementation" (siehe [BPMN1.1, S.1]). Diese Aussage unterstreicht das eigentliche Ziel der BPMN, eine graphische Notation für die Beschreibung von Geschäftsprozessen zu liefern, die von allen Mitwirkenden gleichermaßen verwendet werden kann. Angefangen von den Business Analysten, die die Prozesse entwerfen, weiter zu den technischen Entwicklern, die die Prozesse implementieren, bis hin zu den Managern und Administratoren, die diese Prozesse überwachen und verwalten. Eine detailliertere Abgrenzung von Business Analysten und technischen Entwicklern im Zusammenhang mit dem Business Process Management (BPM) und der Business Process Execution (BPE) wird in [Dubr07] dargestellt.
Als ein weiteres, ebenso wichtiges Ziel wird angegeben, dass die BPMN zur Visualisierung von Sprachen für die Implementierung von Geschäftsprozessen, wie beispielsweise BPEL, eine Business-orientierte Notation zur Verfügung stellt (vgl. [BPMN1.1, S.1]).

## 1.2 Motivation

Es gibt gegenwärtig zahlreiche Werkzeuge, mit denen BPEL graphisch modelliert werden kann. In [BPEL2.0] werden die Sprache und deren Semantik spezifiziert, es ist aber keine entsprechende Visualisierung der Konstrukte[2] in dieser Sprache angegeben. Die graphische Darstellung hängt in diesen Werkzeugen daher voll und ganz von dem jeweiligen Hersteller ab. Ebenso gibt es zahlreiche Werkzeuge, die die graphische Modellierung von Geschäftsprozessen mit der BPMN-Notation ermöglichen. Die Werkzeuge, die beide Standards verbinden, sind mit erheblichen Problemen konfrontiert. Die Motivation für diese Arbeit liegt darin begründet, diese Probleme zu erfassen und zu überwinden.

Die Möglichkeiten, auf welche Art und in welchem Umfang BPMN als Visualisierung von BPEL verwendet werden kann sind somit der Gegenstand dieser Arbeit. Es gibt bereits mehrere Ansätze zu diesem Thema, häufig wird dabei von einem nahezu beliebigen BPMN-Diagramm ausgehend versucht, dieses in einen BPEL-Prozess zu übersetzen. Die dabei verwendeten Algorithmen sind zum Teil sehr weit entwickelt, doch es stellen sich in der praktischen Verwendung bestimmte Schwierigkeiten ein, z. B. bei dem Monitoring des generierten BPEL-Codes. In dieser Arbeit werden daher die bestehenden Ansätze zur Verbindung von BPMN und BPEL diskutiert und deren Defizite aufgezeigt. Das eigentliche Ziel ist die Erarbeitung und die Demonstration eines alternativen Ansatzes, der die dabei identifizierten Probleme nicht aufweist.

[2] Der Begriff „Konstrukt" (Construct) bezeichnet ein Element in einer Sprache. Es ist ein relativ abstrakter Begriff, beispielsweise wird der Aufruf eines Web Services in BPEL mit dem Konstrukt `<invoke>` beschrieben, wobei dieses Konstrukt möglicherweise weitere Konstrukte beinhaltet. Je nach dem Kontext, in dem es verwendet wird, kann ein Konstrukt eine unterschiedliche Semantik haben.

## 1.3 Aufgabenstellung

Für die Modellierung und Implementierung EDV-gestützter Geschäftsprozesse soll ein Werkzeug entwickelt werden, welches die Möglichkeit bietet, diese Prozesse graphisch abzubilden, sowie automatisch Code für deren Implementierung zu erzeugen.

Als Standard für die graphische Modellierung soll die *Business Process Modeling Notation (BPMN)* in der Version 1.1 verwendet werden. Der dabei erzeugte Code soll dem OASIS Standard zur Orchestrierung von Web Services entsprechen, der *Business Process Execution Language (BPEL)* in der Version 2.0. Als der Modellierung zu Grunde liegendes Metamodell soll ebenfalls BPEL eingesetzt werden.

Um auf Basis der Sprache BPEL mit der graphischen Notation BPMN die Modellierung von Geschäftsprozessen zu ermöglichen, müssen entsprechende Abbildungen, wie sich BPEL Konstrukte mittels BPMN Piktogrammen darstellen lassen, vorgestellt werden. Es soll zusätzlich gezeigt werden, wie sich alternative Abbildungen in dem Werkzeug umsetzen lassen.

Für die Realisierung des Werkzeugs sollen erweiterbare Anwendungen zur Modellierung von BPMN und BPEL sowie Frameworks betrachtet werden, die die Entwicklung graphischer Modellierungswerkzeuge unterstützen. In einer Gegenüberstellung soll ermittelt werden, womit die Realisierung am besten bewerkstelligt werden kann.

## 1.4 Aufbau der Arbeit

Die vorliegende Arbeit ist in sechs Kapitel unterteilt:

Kapitel 1 umfasst eine Einführung in das Themengebiet, die Aufgabenstellung sowie zusätzliche formale Informationen zu dieser Arbeit.
In Kapitel 2 werden die Hintergründe dieser Arbeit vorgestellt. Dazu wird eine Einführung in die Standards BPEL und BPMN gegeben. Diese Standards werden anschließend von einander abgegrenzt, bevor sie im Detail betrachtet werden. Danach werden bestehende Ansätze präsentiert, auf welche Weise diese Standards verbunden werden können und welche Schwierigkeiten sich dabei ergeben.
Anschließend werden in Kapitel 3 Frameworks und erweiterbare Anwendungen vorgestellt, mit denen die geforderte Verbindung von BPEL und BPMN realisierbar wäre. Eine Entscheidung zugunsten einer dieser Möglichkeiten wird dabei mit dem Hauptgesichtspunkt der Erweiterbarkeit getroffen und begründet. Die Frameworks Microsoft DSL Tools, die Eclipse Frameworks EMF, GEF, GMF und GEMS, das Web-basierte Framework Oryx sowie die Anwendungen SOA Tools Platform BPMN, Oryx BPEL4Chor, Eclipse BPEL Designer und ein im Rahmen der Diplomarbeit [Kapl06] erstellter und in der Studienarbeit [Schu07] erweiterter graphischer BPEL Editor werden betrachtet.
Nachfolgend wird in Kapitel 4 auf die Eigenheiten, die Vorteile sowie auf die Nachteile der bestehenden Verfahren zur Verbindung von BPMN und BPEL eingegangen. Aus dieser Diskussionsgrundlage heraus werden anschließend theoretische Überlegungen angestellt, wie sich einfache und komplexe BPEL Konstrukte mit einfachen und komplexen BPMN Strukturen darstellen lassen. Ebenso wird erörtert, welche verschiedenen Möglichkeiten und Varianten sich dabei anbieten.
Im darauf folgenden Kapitel 5 wird die Realisierung und Implementierung dieser theoretischen Überlegungen beschrieben und vorgestellt.
Abschließend werden in Kapitel 6 die gewonnenen Erkenntnisse zusammengefasst, Möglichkeiten und Grenzen der realisierten Lösung angesprochen und es wird ein Ausblick auf zukünftige Arbeiten gegeben.

## 1.5 Zusätzliche Informationen

Die dieser Arbeit zu Grunde liegenden Spezifikationen, sowie der Großteil der verwendeten Literatur, ebenso wie die verwendeten Softwaredokumentationen, sind in Englisch verfasst und haben keine offizielle deutsche Übersetzung. Die in dieser Arbeit dargestellten Übersetzungen wurden durch den Autor in Eigenarbeit erstellt. Sie wurden nicht offiziell autorisiert und geben daher die Intentionen der Verfasser möglicherweise nicht exakt wieder. Im Zweifelsfall sollte der entsprechende Originaltext zu Rate gezogen werden.

Die Fachbegriffe aus diesen Quellen werden in der Regel im englischen Original verwendet. Bei der Einführung neuer, nicht-gängiger Begriffe wird eine deutsche Übersetzung unter Zuhilfenahme von [Pons06] vorgenommen.

Um die Elemente von BPEL und BPMN leichter unterscheiden zu können, werden verschiedene Schriftarten verwendet. Zum Beispiel wird das BPEL-Element zur Dokumentation `<documentation>` geschrieben, ein vergleichbares BPMN-Element ist *Annotation* und wird kursiv geschrieben.

Diese Arbeit umfasst außer den gängigen Referenzen auf Literatur auch Referenzen auf zum Teil nur online verfügbare Publikationen von Unternehmen, Spezifikationen sowie Software und deren Dokumentationen. Um die verwendete Literatur, Publikationen und Spezifikationen von der verwendeten Software und deren Dokumentation im Text besser unterscheiden zu können, wird in dieser Arbeit eine leicht unterschiedliche Notation für deren Referenzierung verwendet. Für eine derartige Notation ist bislang kein Standard verfügbar, daher muss an dieser Stelle eine eigene Form festgelegt werden: Referenzen auf Literatur, Spezifikationen und andere Publikationen werden mit eckigen Klammern dargestellt, wie beispielsweise [ACKM04], [eCla06] und [BPMN1.1]. Ein vollständiger Nachweis dieser Quellen ist im Literaturverzeichnis verfügbar. Referenzen auf Software und deren zugehörige Dokumentation werden ebenfalls mit eckigen Klammern dargestellt, dazu wird ein „S:“ vorangestellt, wie beispielsweise [S:EclipseBPEL], [S:JDK5] und [S:MSDSL]. Ein vollständiger Nachweis, der die Homepage des Herstellers, die Versionsnummer sowie Links zu Dokumentationen umfasst, ist im Softwareverzeichnis verfügbar.

Im Anhang befindet sich zusätzlich ein Glossar, welches einem Abkürzungsverzeichnis entspricht, das alle Abkürzungen enthält, die in dieser Arbeit vorkommen. Für eine einfachere Navigation in dieser Arbeit in Printform ist darüber hinaus ein vollständiges Inhaltsverzeichnis im Anhang aufgeführt. In elektronischer Form im PDF-Format sind zu diesem Zweck auch entsprechende Lesezeichen und verlinkte Querverweise vorhanden.

# 2 Hintergrund

In diesem Kapitel werden die fachlichen Grundlagen dieser Arbeit vorgestellt und diskutiert. Anfangs wird ein kurzer Überblick über die Sprache BPEL und die Notation BPMN gegeben. Die Beziehung, die zwischen diesen Standards besteht, wird anschließend in einer Gegenüberstellung analysiert. Um eine graphische Darstellung von BPEL-Prozessen mit der BPMN-Notation hinreichend präzise formulieren zu können, werden die Konstrukte, die in [BPEL2.0] und [BPMN1.1] spezifiziert sind, nachfolgend in deutscher Sprache vorgestellt. Dies liefert keinen Ersatz für die eigentlichen Spezifikationen, es erfüllt jedoch den Zweck, das in dieser Arbeit zu Grunde liegende Verständnis dieser Konstrukte aufzuzeigen und auf diese Beschreibung in der später vorgestellten Verbindung direkt verweisen zu können. Darüber hinaus ist keine profunde Übersetzung bekannt, die stattdessen referenziert werden könnte. Zum Abschluss dieses Kapitels werden bestehende Ansätze zur Verbindung von BPMN und BPEL beschrieben.

## 2.1 Einführung in BPEL

Es gibt mehrere Möglichkeiten, wie eine Einführung in BPEL gegeben werden kann. Beispielsweise könnte man mit dem Technologie-Stack für *Service-Orientierte Architekturen (SOA)* auf Basis von *Web Service (WS)* Technologie beginnen (siehe dazu [inno07]) und beschreiben, welchen Platz BPEL hierbei einnimmt (siehe hierfür [WCLS06] und [ACKM04]). Ebenso könnte man mit einer Darstellung der Konzepte und Techniken von Workflow Management Systemen (WfMS) beginnen (vgl. dazu [LeRo00], [AaHe02], [WfMC08]) und analysieren, welche dieser Konzepte mit BPEL umgesetzt werden, beziehungsweise auf welche Art und Weise.

### 2.1.1 Ziele der Architektur

In dieser Arbeit findet eine weitere Möglichkeit Verwendung. BPEL wird anhand der Ziele vorgestellt, die die Architekten dieser Sprache bei dem Entwurf verfolgt haben. In [LRT03] werden zehn wesentliche Ziele beschrieben, die die Essenz der Überlegungen und Prinzipien widerspiegeln, welche die Spezifikation von BPEL entscheidend bestimmt haben:

1. BPEL4WS (Anm.: so lautet der ursprüngliche Name, BPEL for Web Services) soll Geschäftsprozesse definieren, die mithilfe von Web Service Schnittstellen mit anderen Komponenten, sprich Web Services, kommunizieren. Die von diesen Komponenten zur Verfügung gestellten Operationen sollen mit der Web Service Description Language (WSDL, siehe [W3C01]) beschrieben werden. Die Geschäftsprozesse stellen dabei wiederum selbst einen Web Service dar, der mit WSDL beschrieben wird. Die laufende Entwicklung von WSDL muss daher in zukünftigen Versionen von BPEL4WS berücksichtigt werden (vgl. [LRT03, S. 1]).
2. Die Definition der Geschäftsprozesse mit BPEL4WS findet unter Verwendung einer XML-basierten Sprache statt. BPEL4WS beschäftigt sich weder mit der graphischen Darstellung von Prozessen, noch definiert es eine bestimmte Designmethodik für Prozesse (vgl. [LRT03, S. 2]).
3. BPEL4WS soll Konzepte zur Orchestrierung von Web Services definieren, die sowohl für die externe Sicht auf den Geschäftsprozess (`Abstract Process`) als auch für die interne Sicht (`Executable Process`) verwendet werden können. Es ist abzusehen, dass jedes dieser Verwendungsmuster spezifische Erweiterungen erfordert (vgl. [LRT03, S. 3]).
4. BPEL4WS soll hierarchische und auch Graph-artige Kontrollstrukturen bieten. Deren kombinierte Verwendung soll dabei so nahtlos wie möglich sein, *„allow their usage to be blended as seamlessly as possible“* (vgl. [LRT03, S. 4]).

5. Für die Bearbeitung von Daten werden von BPEL4WS eingeschränkte Funktionen zur Verfügung gestellt. Daten, die für den Prozess oder den Kontrollfluss relevant sind, sollen damit in einfacher Weise bearbeitet werden können. Die tatsächlichen Funktionen zur Datenverarbeitung werden durch die Web Services bereitgestellt, die in dem Prozess aufgerufen werden (vgl. [LRT03, S. 4]).
6. BPEL4WS soll einen Mechanismus zur Identifikation von Prozessinstanzen unterstützen. Dieser Mechanismus soll die Definition des Identifikationsmerkmals auf der Ebene des Nachrichtenaustausches zwischen den Anwendungen erlauben. Das möglicherweise aus mehreren Teilen zusammengesetzte Identifikationsmerkmal ist unabhängig von dem zugrunde liegenden Kommunikationsprotokoll und kann sich im Laufe der Zeit (Anm.: des Prozessverlaufs) ändern, es ist sozusagen kontextsensitiv (vgl. [LRT03, S. 5]).
7. Als Hauptmechanismus für den Lebenszyklus von Prozessen soll BPEL4WS die implizite Erstellung und Terminierung von Prozessinstanzen unterstützen. Erweiterte Operationen für den Lebenszyklus von Prozessen wie Suspend/ Resume (Unterbrechen/Fortfahren) können in zukünftigen Versionen der Spezifikation berücksichtigt werden (vgl. [LRT03, S. 5]).
8. BPEL4WS soll ein Transaktionsmodell für lange andauernde Prozesse definieren (long-running business processes). Dieses soll auf erprobten Techniken wie *Compensation Actions* und *Scoping* basieren, dadurch soll im Fehlerfall *Recovery* für bestimmte Teile eines Geschäftsprozesses unterstützt werden (vgl. [LRT03, S. 6]).
9. BPEL4WS soll für die Komposition von einzelnen Prozessen zu größeren, zusammengesetzten Prozessen das Web Service Modell verwenden. Dies ist möglich, indem ein Prozess selbst wiederum als Web Service zur Verfügung gestellt wird (siehe Ziel 1). Eine Verbindung mit WS-Policy kann dieses Modell darüber hinaus noch leistungsfähiger machen (vgl. [LRT03, S. 6]).
10. BPEL4WS soll soweit wie möglich auf passende, bestehende Web Service Spezifikationen aufbauen, sowohl auf Standards als auch auf Vorschläge für Standards (Standard Proposals). Diese sollen dabei „in a composable and modular manner“ eingesetzt werden (Anm.: Also besteht BPEL4WS aus spezifischen Anforderungen und bezieht weitere Spezifikationen an passenden Stellen modular mit ein). Die Spezifikation von bestimmten Anforderungen soll nur dann erfolgen, wenn dies nicht bereits in einer anderen Web Service Spezifikation geleistet wurde (vgl. [LRT03, S. 6]).

Aus Platzgründen kann in dieser Arbeit nicht ausführlich auf die eigentlichen Überlegungen, Prinzipien und Hintergründe eingegangen werden, die zu diesen Zielen geführt haben. Selbst in [LRT03] kann nur ein knapper Überblick über die komplexen Sachverhalte vermittelt werden. Als vertiefende Literatur eignen sich [WCLS06], [ACKM04], [BeNe97], [LeRo00], die Web Service Spezifikationen [BPEL1.1] beziehungsweise [BPEL2.0], WSDL, WS-Addressing, WS-Policy und SOAP sowie [RFC2119] zur Klärung der in den Spezifikationen verwendeten Begriffe zur Anforderungsbeschreibung.

### 2.1.2 Möglichkeiten zur Prozessbeschreibung

Die vollständige Beschreibung eines BPEL Prozesses erstreckt sich selbst bei überschaubaren Prozessen über mehrere Bildschirmseiten. Um dennoch einen Einblick in die Struktur eines BPEL Prozesses und in die XML-Syntax zu geben, wird nun ein Ausschnitt aus diesem Prozess gezeigt. In dem ersten Listing wird dieser Ausschnitt in Pseudo-Code beschrieben, das zweite Listing stellt den eigentlichen BPEL-Code dar.

```
shipOrder := receive();
if (shipComplete) then
   shipNotice := shipRequest;
   send(shipNotice);
else
   while (itemsShiped < itemsTotal) do  ...
```

**Listing 1: Ausschnitt aus einem BPEL Prozess in Pseudo-Code**

```
<sequence>
   <receive partnerLink=”customer” operation=”shippingRequest”
            variable=”shipRequest”>
      <correlations>
         <correlation set=”shipOrder” initiate=”yes” />
      </correlations>
   </receive>
   <if>
   <condition>
      bpel:getVariableProperty(‘shipRequest’,’props:shipComplete’)
   </condition>
   <sequence>
      <assign>
         <copy>
            <from variable=”shipRequest” property=”props:shipOrderID” />
            <to variable=”shipNotice” property=”props:shipOrderID” />
         </copy>
         <copy>
            <from variable=”shipRequest” property=”props:itemsCount” />
            <to variable=”shipNotice” property=”props:itemsCount” />
         </copy>
      </assign>
      <invoke partnerLink=”customer” operation=”shippingNotice”
              inputVariable=”shipNotice”>
         <correlations>
            <correlation set=”shipOrder” pattern=”request” />
         </correlations>
      </invoke>
   </sequence>
   <else>
      <sequence>
         <assign>
            <copy>
               <from>0</from>
               <to>$itemsShipped</to>
            </copy>
         </assign>
         <while>
            <condition>
               $itemsShipped &lt;
                  bpel:getVariableProperty(‘shipRequest’,’props:itemsTotal’)
            </condition>
            <sequence> ...
```

**Listing 2: Ausschnitt aus einem BPEL-Prozess**

Dieser Ausschnitt zeigt zudem, wie gesprächig (verbose) die Darstellung in XML ist, also wie viel Raum sie einnimmt. In der Regel werden BPEL Prozesse mithilfe eines graphischen Modellierungswerkzeuges erstellt. Diese bieten eine graphische Repräsentation von BPEL Prozessen sowie Funktionalität zur Validierung und Codegenerierung. Die Anwendungen, die in 3.2.3 und 3.2.4 vorgestellt werden, sind derartige graphische BPEL Modellierungswerkzeuge.

### 2.1.3 Entstehungsgeschichte und Weiterentwicklung

Die Spezifikation wurde erstmals unter dem Namen *BPEL4WS* (BPEL for Web Services) (vgl. [BPEL1.0]) im Juli 2002 veröffentlicht. Es ist auf die unterschiedlichen Anforderungen der beteiligten Firmen zurückzuführen, dass BPEL eine sowohl hierarchische als auch graphbasierte Sprache ist. So stellt BPEL4WS eine Konvergenz der Sprache *XLANG*[3] (Web Services for Business Process Design) der Firma Microsoft und der Sprache *WSFL*[4] (Web Service Flow Language 1.0) der Firma IBM dar (vgl. [BPEL1.0, S. 2]). Ebenso war *BEA Systems* an der Spezifikation beteiligt.
Mit der im Mai 2003 veröffentlichten Version [BPEL1.1] kamen weitere Autoren der Firmen *SAP AG* und *Siebel Systems* hinzu. In dieser Version wurden Fehler korrigiert, ungenaue Beschreibungen geklärt und auch auf neue Entwicklungen im Bereich der WS Technologie reagiert, wie beispielsweise auf *WS-Addressing* (vgl. [BPEL1.1, S. 12 f.]). Diese Spezifikation hat sich gegen konkurrierende Spezifikationen wie die *Business Process Modeling Language (BPML)* durchgesetzt und ist durch die breite Unterstützung in der Industrie zu einem de facto Standard in diesem Bereich avanciert. Sie wurde im Jahre 2003 an die *Organization for the Advancement of Structured Information Standards (OASIS)* zur Standardisierung übergeben.
Die Standardisierung durch OASIS wurde im April 2007 vorerst abgeschlossen und die Spezifikation unter dem Namen *WS-BPEL* veröffentlicht (vgl. [BPEL2.0]). Die Änderungen im Vergleich zu *BPEL4WS* betreffen hauptsächlich das Naming und die Syntax von Konstrukten. Es wurden zudem weitere Aktivitäten eingeführt (zum Beispiel 2.4.1.3, 2.4.2.2) und unklare Sachverhalte geklärt. Eine Auflistung aller vorgenommenen Änderungen ist in [Schu07, S. 2 ff.] aufgeführt.

[3] siehe [BPEL1.0, S. 70].
[4] siehe ebenso [BPEL1.0, S. 70].

## 2.2 Einführung in BPMN

### 2.2.1 Ziele der Notation

In der Einleitung dieser Arbeit wurde ein Zitat vorangestellt, das die *Business Process Modeling Notation (BPMN)* sinngemäß als standardisierte Überbrückung der Kluft zwischen dem Design von Geschäftsprozessen und deren Implementierung beschreibt (vgl. [BPMN1.1, S.1]). Auf diese Kluft wird in [BPMN1.1, S.11] durch einen direkten Vergleich mit BPEL4WS näher eingegangen. Eine Wertung durch den Autor dieser Arbeit wird an dieser Stelle noch nicht vorgenommen, diese findet erst in der Analyse der Verbindung von BPMN und BPEL in Abschnitt 4.1 statt. Der Vergleich von BPMN und BPEL wurde so direkt wie möglich übersetzt:

Durch BPEL4WS könnte ein komplexer Geschäftsprozess in einem komplexen, zusammenhanglosen und unintuitiven Format organisiert werden, mit dem ein Software System oder ein Programmierer sehr gut umgehen kann. Dieses Format wäre allerdings für die Business Analysten und Manager schwer verständlich, die mit den Aufgaben betraut sind, die Prozesse zu entwerfen, zu steuern und zu überwachen und dabei an Flussdiagramme gewöhnt sind. Es gibt also eine Ebene der Interoperabilität oder auch der Portabilität, die von den XML- und Web Service-basierten Sprachen wie BPEL4WS nicht adressiert wird, nämlich die *menschliche* Ebene. Um diese Kluft zu überwinden bietet BPMN die graphische Notation, die auf ausführende Sprachen wie BPEL4WS abgebildet werden kann (vgl. [BPMN1.1, S.11]).

Mit BPMN wird ebenso die zwischenmenschliche Interoperabilität möglich: Standardisierte Ablaufdiagramme, so genannte *Business Process Diagrams (BPD)*, beschreiben Geschäftsprozesse auf eine einheitliche, eindeutige und für alle Beteiligten verständliche Art und Weise (vgl. [Whit05b, S. 4 f.]).

### 2.2.2 Möglichkeiten zur Modellierung

Für die Beschreibung von Geschäftsprozessen stehen in einem *BPD* zahlreiche Möglichkeiten zur Verfügung. Sie erinnern an UML Aktivitätsdiagramme (vgl. [Oest06, S. 302 ff.]), sind jedoch weit mehr auf die Bedürfnisse von Unternehmen ausgerichtet. Die Spezifikation definiert Konstrukte für die Darstellung mehrerer Geschäftspartner (*Pools*) und dem damit verbundenen Informationsaustausch (*Message Flow Connections)*. Für die zu erledigenden Aufgaben (*Activities*) lassen sich Zuständigkeitsbereiche (*Lanes*) darstellen. Deren Abfolge wird durch Entscheidungen (*Gateways*) und Verbindungen (*Sequence Flow Connections*) festgelegt. Dabei können auch parallele Abläufe erzeugt und wieder synchronisiert werden. Zudem kann auf besondere Ereignisse (*Events*) reagiert werden, die innerhalb und außerhalb des Prozesses auftreten. Außerdem können zusätzliche Daten (*Annotation, Data Object*), beispielsweise zu Dokumentationszwecken, in das Diagramm integriert werden (*Association*). Da Geschäftsprozesse häufig Transaktionen darstellen, werden zusätzliche Möglichkeiten angeboten, dies aussagekräftig abzubilden (*Compensation, Cancel, Timer*).

Mit diesen Konstrukten ist die Beschreibung komplexer Geschäftsprozesse möglich, wie ein Beispiel aus [Whit05b, S. 25] illustriert:

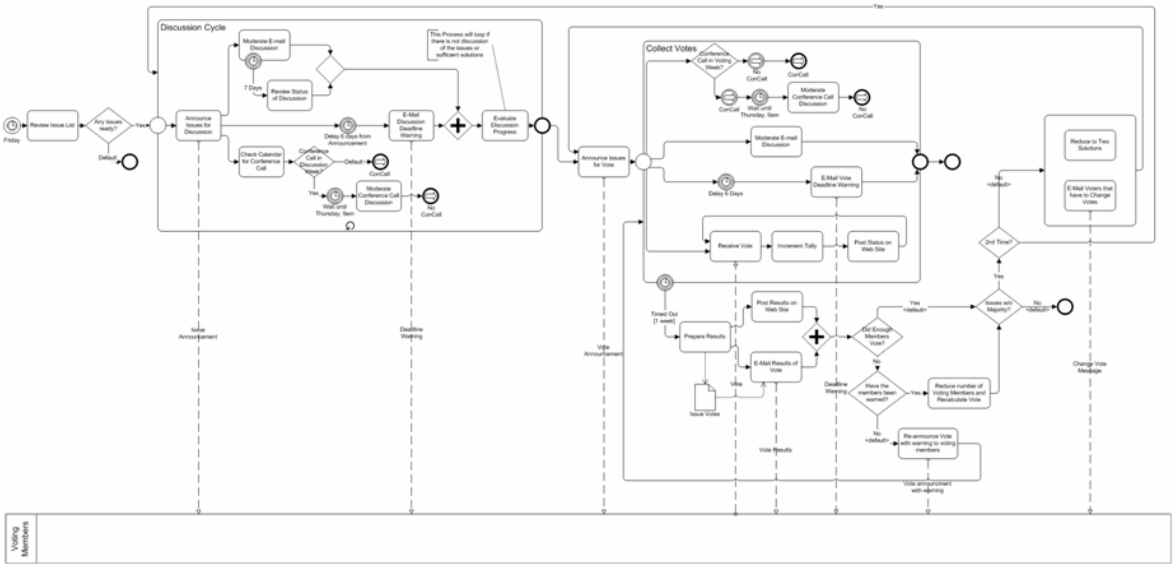


**Abbildung 1: Geschäftsprozess mit BPMN**

### 2.2.3 Metamodell und Kontrollfluss

In der Spezifikation [BPMN1.0] (ebenso in [BPMN1.1]) wurde die Semantik eines BPD nicht formal definiert, es ist aber „not just a notation“ (siehe [Whit05b, S. 9]), denn die Semantik wird im Beschreibungstext der graphischen Elemente festgelegt. Aus diesem Text konnte beispielsweise in [WSPE07] bereits ein inoffizielles UML Klassendiagramm (vgl. [Oest06, S. 241 ff.]) erstellt werden, welches die Datenstruktur von BPMN 1.0 anschaulich beschreibt.

Für die Beschreibung des Kontrollflusses (dieser Ausdruck wird in BPMN ausdrücklich nicht verwendet, siehe dazu [BPMN1.1, S. 102]) in den Diagrammen wird in [BPMN1.1] ein Konzept eingesetzt, das mit „Tokens“ arbeitet: Ein Token ist ein Beschreibungskonstrukt, das dazu verwendet wird, den Ablauf in einem Prozess zur Laufzeit zu beschreiben. Durch die Nachverfolgung, wie das Token die Aktivitäten in einem Prozess durchläuft, wie es durch alternative Pfade gelenkt wird, in parallele Pfade aufgeteilt und wieder zusammen geführt wird, ist der normale Ablauf definiert. Ein Token besitzt eine Menge von Merkmalen, die es eindeutig identifiziert, das *TokenId Set* (vgl. [BPMN1.1, S. 37, 297]).

Für die Verarbeitung der Tokens sind mehrere grundsätzliche Regeln definiert. Prozesse werden durch bestimmte Ereignisse, wie zum Beispiel eingehende Nachrichten gestartet. An einem *Start-Event*, das für den Start und somit für die Instanziierung des Prozesses verantwortlich ist, beginnt der Lebenszyklus der Tokens. Für jede Verbindung, die von diesem Event ausgeht, wird ein Token erzeugt. Diese Tokens beginnen dann, durch den Prozess zu fließen, man spricht hierbei auch von *Sequence Flow* (vgl. [BPMN1.1, S. 38]). Bei parallelen Abläufen in dem Prozess werden einzelne Tokens zudem in mehrere Untertokens aufgeteilt. Daher ist die *TokenId* als Menge von Identifikationsmerkmalen definiert. Aufgeteilte Tokens werden im weiteren Verlauf des Prozesses wieder zusammengeführt, dabei dürfen allerdings keine Untertokens „verloren gehen“ (vgl. [BPMN1.1, S. 87]). Alle Tokens, die an dem *Start Event* erzeugt werden, müssen letztendlich an einem *End-Event* oder an einem *Task* ohne ausgehende *Sequence Flow* Verbindung ankommen (vgl. [BPMN1.1, S. 37, 41]). Der Prozess läuft also so lange, bis alle Tokens auf diese Weise konsumiert wurden (vgl. [BPMN1.1, S. 41, 43]).

Die folgende Abbildung aus [Asti07, S. 40] visualisiert das Konzept der Tokens zur Beschreibung des Kontrollflusses anschaulich. Die Tokens werden durch rote Punkte dargestellt, die von *Activities* konsumiert und nach deren Abschluss weitergegeben werden. Der erste Übergang (hier: *Parallel Gateway*) hat die Aufteilung des Start-Tokens in zwei Untertokens zur Folge (vgl. dazu auch [BPMN1.1, S. 115, 199 ff.]):

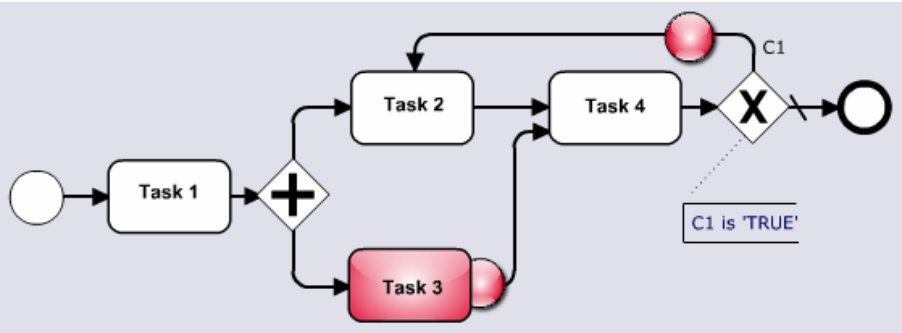


**Abbildung 2: Token zur Beschreibung des Kontrollflusses**

Aus diesem Beispiel geht auch hervor, dass für eine Aufteilung, beziehungsweise Zusammenführung von parallelen oder alternativen Pfaden nicht zwingend ein *Gateway* verwendet werden muss, allerdings hat dies eine unterschiedliche Semantik zur Folge. In [BPMN1.1] wird die Zusammenführung ohne *Gateway* als „*Uncontrolled Flow*“ bezeichnet. Bei parallelen Abläufen findet dabei keine Synchronisierung statt, lediglich die Pfade werden zusammengeführt.

*„In Figure 10.30 (Anm.: die folgende Abbildung), the upstream behavior is parallel. Thus, there will be two Tokens arriving (at different times) at activity “D.” Since the flow into activity “D” in uncontrolled, each Token arriving at activity “D” will cause a new instance of that activity. This is an important concept for modelers of BPMN should understand. In addition, this type of merge is the Workflow Pattern Multiple Merge.”* [BPMN1.1, S. 119]

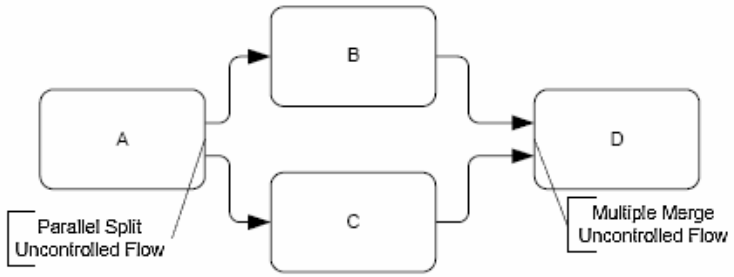


**Abbildung 3: Uncontrolled Flow**

Es ist durch diese Beschreibung noch nicht eindeutig klar, ob diese Tokens dann einzeln durch den weiteren Prozess fließen, oder ob die Aktivität „D“ auf alle „möglichen“ Tokens wartet. Die Beschreibung des Patterns „Multiple Merge“ in [AHKB01, S. 13] klärt dies auf. Die Pfade werden zusammengelegt, jedoch werden die einzelnen Threads nicht zu einem vereinigt, dadurch laufen die parallelen Abläufe getrennt nebeneinander her, aber auf ein und demselben Pfad. Dieses Verständnis besteht auch in [BPMN1.1, S. 257].

*“uncontrolled flow (no Gateway) […] means that all activity instances SHALL generate a token that will continue when that instance is completed.”*

### 2.2.4 Best Practice

Es sind im Allgemeinen mehrere Definitionen von *Best Practice* gängig, häufig wird es als Erfolgsmethode, bewährte Praxis, Standardverfahren oder Erfolgsrezept umschrieben. In BPMN gibt es für zahlreiche Anforderungen mehrere Möglichkeiten, diese abzubilden. Beispielsweise lässt sich Parallelität auf mehrere Arten beschreiben: Man kann sie mithilfe von *Gateways* definieren, alternativ können mehrere *Sequence Flow* Verbindungen von einer *Activity* ausgehen oder die parallelen Aktivitäten befinden sich in einem *Sub-Process* ohne eine eingehende *Sequence Flow* Verbindung (vgl. [BPMN1.1, S. 86 f.]).

Die Veröffentlichungen in diesem Bereich haben noch keine ausreichende Fundierung über ein *Best Practice* in der Modellierung mit BPMN erbringen können. Ein wichtiger Schritt dafür ist eine Analyse, welche von den in BPMN verfügbaren Konstrukten von den Business Analysten und Managern am häufigsten eingesetzt werden. In [Mueh07, S. 24] wird in diesem Zusammenhang beispielsweise festgestellt, dass der *Conditional Flow* ohne eine *Gateway* (siehe 2.5.4.1) nahezu keine Beachtung findet. Das lässt die Schlussfolgerung zu, dass Verzweigungen in der Regel mit *Gateways* modelliert werden und sich dieses Vorgehen wohl in der Praxis bewährt hat, also für diese Anforderung ein *Best Practice* darstellt.

### 2.2.5 Entstehungsgeschichte und Weiterentwicklung

BPMN wurde im Jahre 2004 in der Version 1.0 veröffentlicht. Die Spezifikation (zu der Zeit noch kein Standard der OMG) basiert auf der Arbeit des Herausgebers und Hauptautors Stephen A. White und den über 50 Mitgliedern der für Notation zuständigen Arbeitsgruppe der *Business Process Management Initiative (BPMI)*, die bereits im Jahre 2001 gegründet wurde. Nach dem Fusion der BPMI mit der *Object Management Group (OMG)* wurde [BPMN1.0] im Jahre 2006 als offizieller OMG Standard veröffentlicht (vgl. [Whit05b, S. 3], [OMG08]).

Die Unterstützung in der Industrie hat seitdem rasant an Fahrt gewonnen, bereits an dem Entwurf für BPMN 1.1 (siehe [BPMN1.1]) sind nahezu 50 Unternehmen beteiligt. So sind zu den Firmen, die von Anfang an beteiligt waren, wie z. B. IBM, Sybase, Lombardi und Intalio, weitere namhafte Firmen hinzugekommen. Dazu gehören unter anderem Adobe, BEA, Borland, Hewlett-Packard, Software AG, SAP, Tibco und Boeing. Eine vollständige Auflistung der beteiligten Firmen ist den Spezifikationen [BPMN1.0, S. 6 f.] und [BPMN1.1, S.8 ff.]) zu entnehmen. Es wurde bereits in der Einleitung dieser Arbeit beschrieben, dass mittlerweile über 50 Werkzeuge (vgl. [BFV07]) die Modellierung mit BPMN unterstützen. Auch diese Entwicklung trägt dazu bei, dass sich BPMN zunehmend als de facto[5] Standard für die Modellierung von Geschäftsprozessen durchsetzt. Diese Aussage stellt lediglich die Meinung des Autors dar und damit soll keinesfalls der Wert oder die Bedeutung anderer Notationen für die Prozessmodellierung, wie beispielsweise Ereignisgesteuerte Prozessketten (EPK), gering geschätzt werden.

[5] Brent Miller beschreibt in http://www.ibm.com/developerworks/autonomic/library/ac-edge2/, dass ein Standard nur von ganz bestimmten Organisationen zu einem „de jure“ Standard erhoben werden kann. Nach Miller gehören dazu beispielsweise ISO, IEEE, W3C und IETF. Wesentlich ist nach Miller, dass es sich bei der Organisation um eine „recognized authority“ auf deren Tätigkeitsgebiet handelt. In diesem Sinne nennt er auch die DMTF und OASIS. Nach dieser Argumentation kann BPEL as „de jure“ Standard bezeichnet werden, ebenso könnte auf diese Weise für die OMG und BPMN argumentiert werden. Die Abkürzungen der angesprochenen Organisationen sind im Glossar enthalten.

Im Juni 2007 hat die OMG ein *Request for Proposals (RFP)* veröffentlicht. Darin wird um Vorschläge und Anregungen gebeten, wie die Spezifikation BPMN und die Spezifikation *Business Process Definition Metamodel (BPDM)* harmonisiert und vereint werden können. Die BPDM wird ebenso innerhalb der OMG entwickelt, von der *Business Modeling & Integration (BMI) Domain Task Force (DTF)*. BPDM könnte es ermöglichen, Geschäftsprozesse zwischen verschiedenen Modellierungswerkzeugen unter der Verwendung von *XML Metadata Interchange (XMI)* auszutauschen (vgl. [BPDM S. 2], [Whit05b, S. 10]). Die resultierende Sprache soll sowohl eine graphische Notation, ein Metamodell und auch ein Austauschformat dafür definieren. Unter dem neuen Namen *Business Process Model and Notation (BPMN 2.0)* wird das Akronym BPMN beibehalten, die resultierende Spezifikation geht jedoch weit über das BPMN hinaus, das in dieser Arbeit vorgestellt wird (vgl. [BPMN2.0, S. 1 ff.].

Eine bildliche Darstellung der Entstehungsgeschichte von BPMN und BPEL und den dabei beteiligten Organisationen wird in [Bart05] gezeigt:

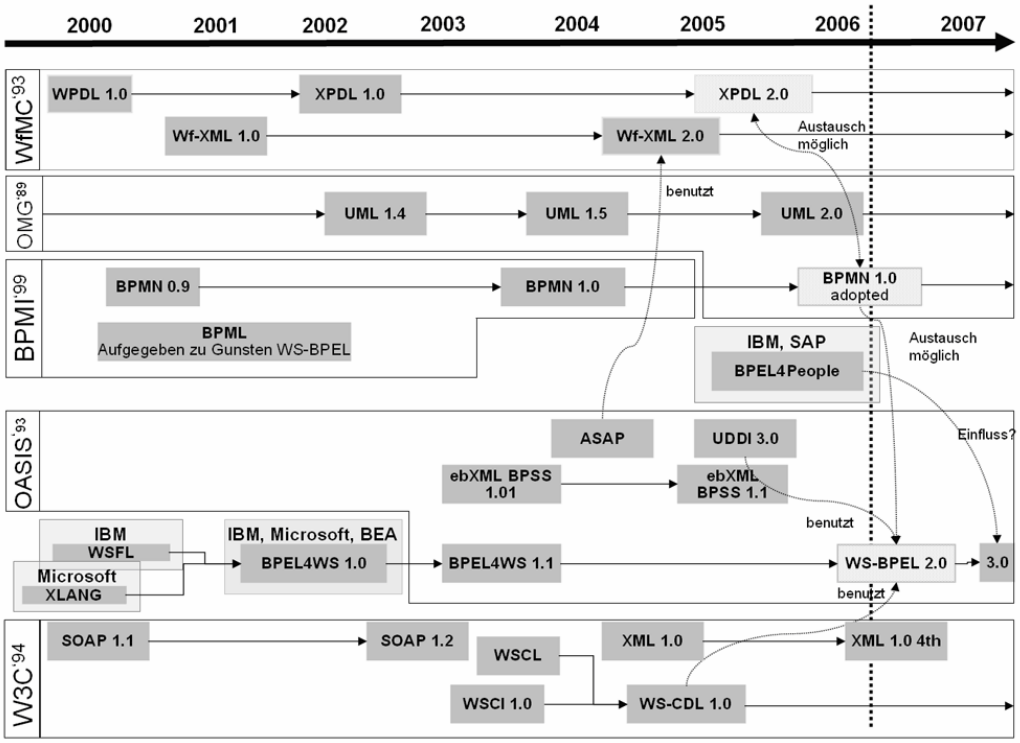


**Abbildung 4: Entstehungsgeschichte von BPMN und BPEL**

## 2.3 Gegenüberstellung

In dem folgenden Abschnitt werden die Sprachen BPMN und BPEL von einander abgegrenzt. Die Gegenüberstellung wird am Anfang auf einem eher hohen Level vollzogen (siehe 2.3.1), erst danach werden die Sprachkonzepte etwas detaillierter gegenübergestellt (siehe 2.3.2). Zum Schluss wird ein Vergleich auf der Ebene der Ausdruckskraft der Sprachen vollzogen (siehe 2.3.3).

### 2.3.1 Zuordnung zu Anwendungsbereichen

Die Abbildung aus [Whit05a, S. 8] zeigt, dass BPMN mit strategischen Beratern für Geschäftsprozesse im Grunde eine andere Zielgruppe hat als BPEL, das eher technische Experten bedient. Gerade zwischen diesen beiden Zielgruppen kann BPMN jedoch auch ein Bindeglied darstellen, in dem es eine standardisierte Darstellungsform der Prozesse, und somit eine bessere Kommunikation zwischen den Zielgruppen und deren Experten untereinander ermöglicht:

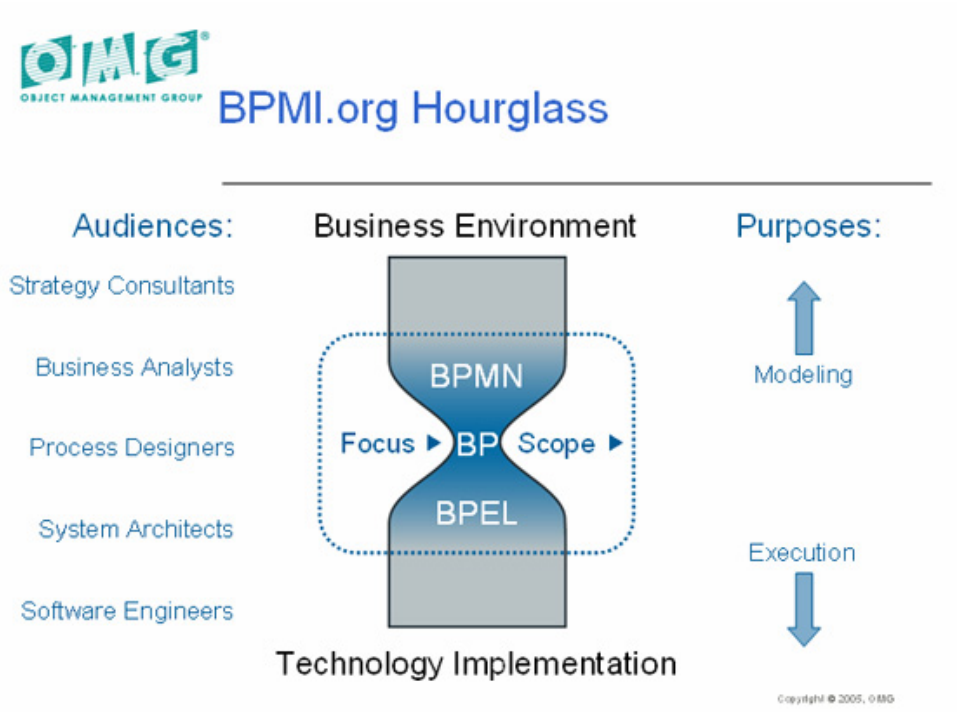


**Abbildung 5: BPMI.org Hourglass**

Die Workflow Management Coalition (WfMC) (siehe [WfMC08]) hat ein Referenzmodell entworfen, das die Interoperabilität von Workflow-Systemen durch die Standardisierung von Schnittstellen verbessert. Die nachfolgende Abbildung aus [Mueh07a, S. 20] zeigt eine Einordnung von BPMN und BPEL in dieses Referenzmodell. Hierbei ist BPMN die modellierende Sprache und BPEL die ausführende Sprache:

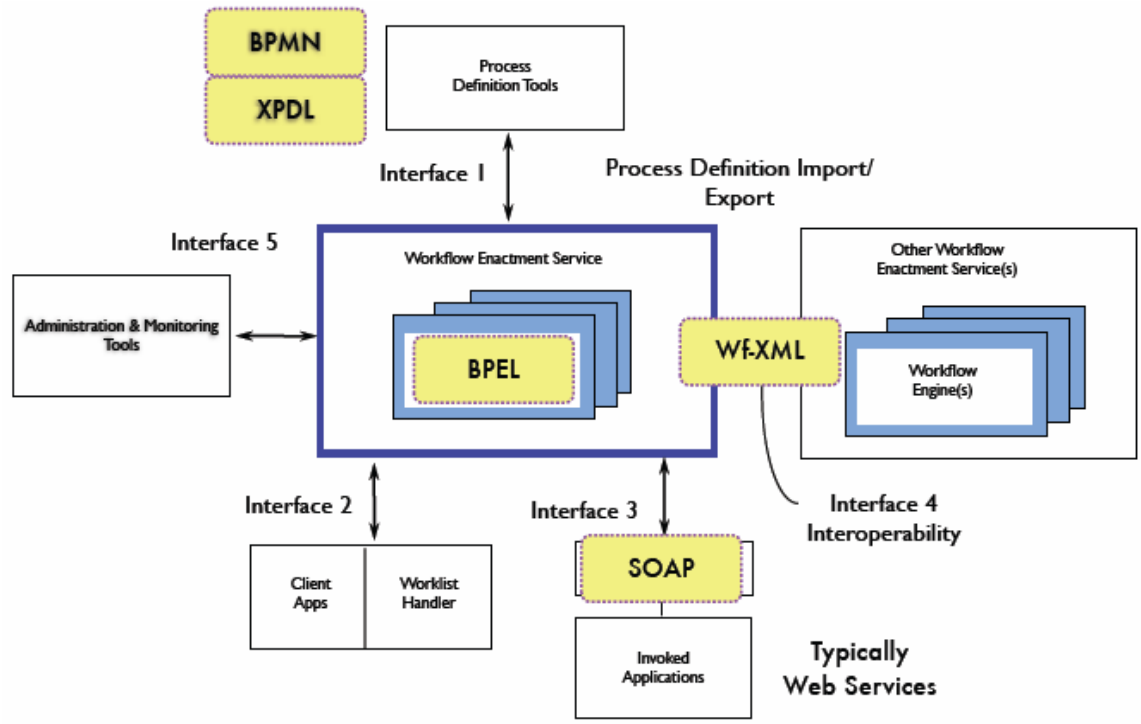


**Abbildung 6: BPMN und BPEL im Referenzmodel von WfMC**

Das Interface I (die Schnittstelle dazwischen) standardisiert, wie unterschiedliche Workflow-Systeme die Prozessmodelle sowie organisatorische Informationen austauschen können (vgl. [LeRo00, S. 118]). Die Workflow Process Definition Language (WPDL) wurde zu diesem Zweck im Jahr 1999 vorgeschlagen. Diese Sprache wurde allerdings von den Herstellern nur eingeschränkt unterstützt und war nicht XML-basiert. Sie diente jedoch als Ausgangspunkt für den im Jahre 2002 veröffentlichen Standard XML Process Definition Language (XPDL), in der die in WPDL enthaltenen Konzepte erweitert und geändert wurden. XPDL ist also ein Format zum Austausch und der Serialisierung von Prozessmodellen und kann daher zum Speichern von BPMN-Diagrammen eingesetzt werden (vgl. [Aals03, S. 1 f.], [WfMC08]).

### 2.3.2 Sprachkonzeptionen

BPMN wurde als graphische Notation zur Darstellung von Geschäftsprozessen entworfen. Während der Entwurfszeit dieser Notation hat sich BPEL als wichtigster Standard für die Orchestrierung von Web Services hervorgetan, dies wurde in der Spezifikation [BPMN1.0] berücksichtigt. BPEL wurde dabei nicht als „führendes System" angesehen, welches es zu visualisieren gelte, sondern gewisse Erweiterungen für BPMN implementierende Modellierungswerkzeuge wurden spezifiziert. In [BPMN1.1] sind für die graphischen Elemente bestimmte Attribute vorgesehen, die eine Abbildung auf einen BPEL-Prozess ermöglichen sollen, wie zum Beispiel das *QueryLanguage* Attribut an einem Business Process Diagram (BPD) (siehe [BPMN1.1, S. 31, 145]). Eine derartige Abbildung meint im Sinne von [Whit05a] die Übersetzung eines Diagramms in BPEL-Code. Die Sprache BPEL hingegen war als reine Ausführungssprache auf XML-Basis konzipiert, ohne eine Visualisierung der Konstrukte und Prozesse vorzugeben. Auf den ersten Blick scheinen sich die beiden Konzeptionen sehr gut zu ergänzen.
Die nachfolgende Workflow Pattern Analyse (siehe 2.3.3) wird allerdings offenbaren, dass es doch gewisse Unterschiede in der Ausdruckskraft der beiden Sprachen gibt. Besonders der zehnte Vergleichspunkt, das Pattern „Arbitrary Cycle" (beliebiger Ablauf), deutet auf eine ungleiche Konzeption hin. Dadurch treten in der Verbindung von BPMN zu einer ausführenden Sprache wie BPEL gewisse Schwierigkeiten auf, die in der vertiefenden Analyse in 2.6 sowie in 4.1 eingehend diskutiert werden. In [BPMN1.1, S. 1, 11] wird deutlich klargestellt, dass BPMN eine graphische Notation ist; für die Ausführung eines in BPMN modellierten Prozesses ist ein Mapping zu einer ausführenden Sprache notwendig.

*„This gap needs to be bridged with a formal mechanism that maps the appropriate visualization of the business processes (a notation) to the appropriate execution format (a BPM execution language)."* [BPMN1.1, S. 11]

In [MLZ06, S. 1 f.] wird diese Ungleichheit als *Graph-orientiertes* (bei BPMN) und *Block-orientiertes* (bei BPEL) „Kontrollfluss-Paradigma" in Worte gefasst:
Nach Mendling et al. definieren Graph-orientierte Sprachen zur Modellierung von Geschäftsprozessen den Kontrollfluss mit Hilfe von Kanten (arcs), die die zeitliche und logische Abhängigkeit zwischen Knoten darstellen. Die Arten von Knoten können von Sprache zu Sprache unterschiedlich sein. Block-orientierte Sprachen definieren den Kontrollfluss hingegen durch das Verschachteln (Nesten) der Sprachelemente und stellen dadurch Parallelität, Alternativen und Schleifen dar.
Während XLANG (siehe 2.1.3) eine rein Block-orientierte Sprache ist, sind in BPEL zusätzlich zu den Block-orientierten Konstrukten die Graph-orientierten Konzepte aus WSFL (siehe 2.4.2.1) vorhanden. Die Einschränkung, dass ein Graph (also in diesem Fall ein BPEL-`<flow>` mit `<link>`s, siehe dazu 2.4.2.1) azyklisch sein muss, rechtfertigt die Bezeichnung von BPEL als Block-orientierte Sprache nicht. In [MLZ06, S. 2] wird BPEL allerdings als Block-orientierte Sprache eingestuft, was den Sachverhalt unangemessen darstellt.

Die nachfolgende Abbildung aus [MLZ06, S. 3] zeigt die Gegenüberstellung dieser Paradigmen. Links auf dem Bild ist eine Graph-orientierte Notation dargestellt (in diesem Fall allerdings nicht BPMN). Rechts ist eine Block-orientierte Darstellung zu sehen, dabei handelt es sich um eine Visualisierung von BPEL-Code mit der Andeutung der Graph-orientierten Konzepte und der Blockstruktur:

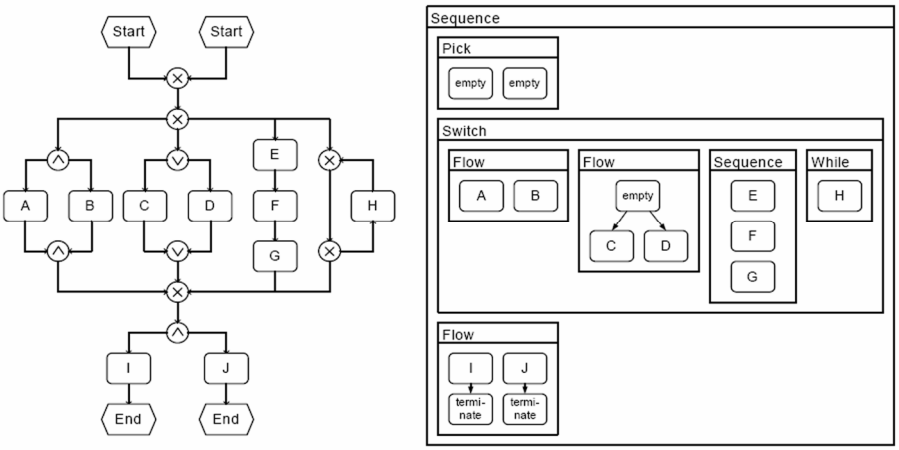


**Abbildung 7: Graph- und Block-orientierte Kontrollfluss-Paradigmen**

### 2.3.3 Workflow Pattern Analyse

Workflow Patterns stellen eine wissenschaftliche Möglichkeit dar, Sprachen zur Beschreibung von Geschäftsprozessen (technisch: Workflows) auf ihre Ausdrucksmächtigkeit hin zu überprüfen und zu vergleichen. In [AHKB03, S.1] werden diese Patterns definiert:

*"Differences in features supported by the various contemporary commercial workflow management systems point to different insights of suitability and different levels of expressive power. The challenge [...] is to systematically address workflow requirements, from basic to complex."*

Workflow Patterns stellen den „State of the Art" der wissenschaftlichen Werkzeuge für die Analyse von Sprachen zur Beschreibung von Geschäftsprozessen dar. Damit wurden unter anderem bereits UML Aktivitätsdiagramme, Ereignisgesteuerte Prozessketten (EPK), XML Process Definition Language (XPDL) und, für diese Arbeit entscheidend, BPMN und BPEL analysiert.
In [AHKB03] werden 20 Patterns definiert, die die Grundlage für die Evaluierung einer Sprache oder eines Produktes zu Beschreibung, beziehungsweise zur Modellierung von Geschäftsprozessen schaffen. In [RHAM06] wurde die Erforschung, beziehungsweise die Definition dieser Patterns fortgeführt. Die bereits identifizierten Patterns wurden überarbeitet und unter Zuhilfenahme von gefärbten Petri-Netzen (Coloured Petri-Net, CPN) präzisiert, zudem wurden 23 weitere Patterns vorgestellt. Diese Patterns beschreiben die Ausdruckskraft (Expressivität) einer Sprache in Bezug auf den Kontrollfluss. Es wurden darüber hinaus noch weitere Patterns zur Analyse vorgeschlagen, diese behandeln die *Data Perspective* und die *Resource Perspective* (vgl. [RHAM06, S. 3 f.]). Auf diese Patterns wird hier nicht eingegangen, es werden lediglich die Control-Flow Patterns betrachtet.

Es wurden bereits mehrere Analysen von BPMN und BPEL auf Basis der Workflow Patterns durchgeführt, diese unterscheiden sich dabei in der Version der betrachteten Spezifikation und in der Anzahl der analysierten Patterns. In [RHAM06] werden alle 43 Patterns auf die Spezifikation [BPEL1.1] und [BPMN1.0] angewendet. Diese Analyse basiert auf den Arbeiten [WADH03] für BPEL4WS und [WADH05] für BPMN. Die Analyse von Anwendungen zur Modellierung von BPEL-Prozessen liefern zum Beispiel [Muly05] und [Kram06]. Allerdings konnten darin die Änderungen und Neuerungen von [BPEL2.0] nicht berücksichtigt werden.

Eine umfassende Workflow Pattern Analyse für die aktuellen Versionen der Spezifikationen, [BPEL2.0] und [BPMN1.1], kann im Rahmen dieser Arbeit nicht erbracht werden. Eine zusammenfassende Gegenüberstellung der Resultate, die in den bestehenden Analysen erbracht wurden, ist hingegen gut möglich. Um weiterführende Betrachtungen einzelner Patterns und deren Unterstützung in BPEL oder BPMN zu ermöglichen, sind in der folgenden Tabelle zusätzlich relevante Seitenangaben der verwendeten Analysen aufgeführt.

Die nachfolgende Tabelle zeigt die Unterstützung der Control-Flow Patterns aus [RHAM06] in [BPMN1.0] und [BPEL1.1] unter Verwendung der Analysen in [WADH03], [WADH05] und [RHAM06]. Dabei kennzeichnet ein „+“ die direkte Unterstützung eines Patterns, „+ / -“ deutet an, dass das Pattern eingeschränkt unterstützt wird und „-“ zeigt an, dass das Pattern nicht direkt unterstützt wird. In manchen Fällen können nicht unterstützte Patterns durch Workarounds (Behelfslösungen) dennoch umgesetzt werden. Verschiedene Analysen kommen daher auf leicht unterschiedliche Ergebnisse (vgl. [Muly05], [Kram06]). Die ersten 20 Patterns sind die in [AHKB03] vorgestellten, die weiteren 23 Patterns sind die in [RHAM06] vorgestellten Patterns.

| Nr. | Pattern | BPEL 1.1 | BPMN 1.0 | Literatur |
|---|---|---|---|---|
| | | | | |
| *Basic Control-Flow* | | | | |
| 01 | Sequence | + | + | [WADH03, S. 4], [WADH05, S. 4 f.] |
| 02 | Parallel Split | + | + | [WADH03, S. 4 ff.], [WADH05, S. 4 f.] |
| 03 | Synchronization | + | + | [WADH03, S. 4 ff.], [WADH05, S. 4 f.] |
| 04 | Exclusive Choice | + | + | [WADH03, S. 6 f.], [WADH05, S. 4 f.] |
| 05 | Simple Merge | + | + | [WADH03, S. 6 f.], [WADH05, S. 4 f.] |
| | | | | |
| *Advanced Synchronization* | | | | |
| 06 | Multi-Choice | + | + | [WADH03, S. 7 f.], [WADH05, S. 6 f.] |
| 07 | Structured Synchronizing Merge | + | + | [WADH03, S. 8], [WADH05, S. 6 f.] |
| 08 | Multi-Merge | - | + | [WADH03, S. 8 f.], [WADH05, S. 7] |
| 09 | Structured Discriminator | - | +/- | [WADH03, S. 9], [WADH05, S. 7 f.] |
| | | | | |
| *Structural Patterns* | | | | |
| 10 | Arbitrary Cycles | - | + | [WADH03, S. 9], [WADH05, S. 8 f.] |
| 11 | Implicit Termination | + | + | [WADH03, S. 9], [WADH05, S. 8 f.] |
| | | | | |
| *Multiple Instance Patterns* | | | | |
| 12 | Multiple Instances without Sychronization | + | + | [WADH03, S. 9 f.], [WADH05, S. 9 f.] |
| 13 | Multiple Instances with a Priori Design-Time Knowledge | + | + | [WADH03, S. 10], [WADH05, S. 9 f.] |
| 14 | Multiple Instances with a Priori Run-Time Knowledge | - | + | [WADH03, S. 10], [WADH05, S. 9 f.] |
| 15 | Multiple Instance without a Priori Run-Time Knowledge | - | - | [WADH03, S. 10], [WADH05, S. 9 f.] |
| | | | | |
| *State-Based Patterns* | | | | |
| 16 | Deferred Choice | + | + | [WADH03, S. 10 f.], [WADH05, S. 10 ff.] |
| 17 | Interleaved Parallel Routing | +/- | - | [WADH03, S. 11 f.], [WADH05, S. 10 ff.] |
| 18 | Milestone | - | - | [WADH03, S. 13], [WADH05, S. 11 f.] |
| | | | | |
| *Cancellation Patterns* | | | | |
| 19 | Cancel Activity | + | + | [WADH03, S. 13], [WADH05, S. 12 f.] |
| 20 | Cancel Case | + | + | [WADH03, S. 13], [WADH05, S. 12 f.] |

**Tabelle 1: Workflow Pattern-Analyse von BPEL 1.1 und BPMN 1.0**

| Nr. | Pattern | BPEL 1.1 | BPMN 1.0 | Literatur |
|---|---|---|---|---|
| | | | | |
| 21 | Structured Loop | + | + | [RHAM06, S. 115], [RHAM06, S. 124] |
| 22 | Recursion | - | - | [RHAM06, S. 115], [RHAM06, S. 124] |
| 23 | Transient Trigger | - | - | [RHAM06, S. 116], [RHAM06, S. 124] |
| 24 | Persistent Trigger | + | + | [RHAM06, S. 116], [RHAM06, S. 124] |
| 25 | Cancel Region | +/- | +/- | [RHAM06, S. 116], [RHAM06, S. 124] |
| 26 | Cancel Multiple Instance Activity | - | + | [RHAM06, S. 116], [RHAM06, S. 124] |
| 27 | Complete Multiple Instance Activity | - | - | [RHAM06, S. 116], [RHAM06, S. 124] |
| 28 | Blocking Discriminator | - | +/- | [RHAM06, S. 116], [RHAM06, S. 124] |
| 29 | Cancelling Discriminator | - | + | [RHAM06, S. 116], [RHAM06, S. 124] |
| 30 | Structured N-out-of-M Join | - | +/- | [RHAM06, S. 116], [RHAM06, S. 124] |
| 31 | Blocking N-out-of-M Join | - | +/- | [RHAM06, S. 116], [RHAM06, S. 124] |
| 32 | Cancelling N-out-of-M Join | - | +/- | [RHAM06, S. 116], [RHAM06, S. 124] |
| 33 | Generalized AND-Join | - | + | [RHAM06, S. 116], [RHAM06, S. 124] |
| 34 | Static Partial Join for Multiple Instances | - | +/- | [RHAM06, S. 116], [RHAM06, S. 125] |
| 35 | Cancelling Partial Join for Multiple Instances | - | +/- | [RHAM06, S. 116], [RHAM06, S. 125] |
| 36 | Dynamic Partial Join for Multiple Instances | - | - | [RHAM06, S. 116], [RHAM06, S. 125] |
| 37 | Acyclic Synchronizing Merge | + | - | [RHAM06, S. 116], [RHAM06, S. 125] |
| 38 | General Synchronizing Merge | - | - | [RHAM06, S. 116], [RHAM06, S. 125] |
| 39 | Critical Section | + | - | [RHAM06, S. 116], [RHAM06, S. 125] |
| 40 | Interleaved Routing | + | +/- | [RHAM06, S. 116], [RHAM06, S. 125] |
| 41 | Thread Merge | +/- | + | [RHAM06, S. 116], [RHAM06, S. 125] |
| 42 | Thread Split | +/- | + | [RHAM06, S. 116], [RHAM06, S. 125] |
| 43 | Explicit Termination | - | + | [RHAM06, S. 116], [RHAM06, S. 125] |

**Tabelle 2: Analyse weiterer Patterns**

## 2.4 Konstrukte in BPEL 2.0

Die nachfolgenden Abschnitte stellen die in [BPEL2.0] zur Verfügung stehenden Konstrukte und deren Elemente dar. Zu jedem Konstrukt sind eine Kurzbeschreibung, relevante Seitenangaben in der Spezifikation sowie ein Code-Listing als Beispiel angegeben. Die Kurzbeschreibung umfasst weder eine vollständige Auflistung aller Attribute noch die zugehörigen XML Schema Definitionen (XSD); hierfür ist die Spezifikation [BPEL2.0] mit den angegebenen Seitenzahlen heranzuziehen. Die Gliederung ist an die in der Spezifikation verwendete Gliederung in einfache Aktivitäten (`Basic Activities` siehe 2.4.1), komplexe Aktivitäten (`Structured Activities`, siehe 2.4.2), Bereiche (`Scopes`, siehe 2.4.3) und `Variables` (siehe 2.4.4) angelehnt. Andere, dieser Gliederung nicht zuordenbare Konstrukte, sind in dem Abschnitt „Weitere Konstrukte“ (siehe 2.4.5) beschrieben.

### 2.4.1 Basic Activities

#### 2.4.1.1 <assign>

Die `<assign>` Aktivität[6] wird dazu verwendet, Variablen neue Werte zuzuweisen. Dabei kann ein `<assign>` Konstrukt eine beliebige Anzahl von Zuweisungen mittels mehrerer `<copy>` Elemente enthalten. Diese Aktivität ist überaus vielseitig einsetzbar und erweiterbar. So können neben *XPath* (XML Path Language) auch weitere `Expression Languages` für die Beschreibung der Zuweisungen verwendet werden. Ebenso kann die Aktivität zur dynamischen Bestimmung von Service Partnern eingesetzt werden (vgl. [BPEL2.0, S. 26, 59 ff.]). In [BPEL2.0, S. 63] wird beschrieben, dass diese Aktivität sowohl *Atomicity* als auch *Isolation* Sematik (im Sinne von ACID) hat, dass also entweder alle `<copy>`-Statements ausgeführt werden oder gar keines und dass die Ausführung unabhängig von anderen Aktivitäten ist. Hier ein Beispiel mit einer Zuweisung aus [BPEL2.0, S. 20]:

```
<assign>
   <copy>
      <from>$PO.customerInfo</from>
      <to>$shippingRequest.customerInfo</to>
   </copy>
</assign>
```

**Listing 3: <assign>**

#### 2.4.1.2 <empty>

Die `<empty>` Aktivität ist eine leere Anweisung (NOP, no operation) und ist beispielsweise bei der Synchronisation (bedingt) paralleler Aktivitäten nützlich (vgl. [BPEL2.0, S. 27, 95]). In [BPEL2.0, S. 130] wird eine Verwendung in der Fehlerbehandlung gezeigt:

```
<faultHandlers>
   <catchAll>
      <empty />
   </catchAll>
</faultHandlers>
```

**Listing 4: <empty>**

[6] In der Spezifikation [BPEL2.0] werden Anweisungen als „Aktivitäten“ bezeichnet. Der Oberbegriff der Sprachelemente in BPEL ist „Konstrukt“, wobei jede Aktivität ein Konstrukt ist, jedoch nicht jedes Konstrukt eine Aktivität.

#### 2.4.1.3 <extensionActivity>

Eine <extensionActivity> Aktivität wird dazu verwendet, BPEL durch neue, nicht im Standard enthaltene Aktivitäten auf eine dennoch standardisierte Art und Weise zu erweitern. Dabei kann eine <extensionActivity> auch zum Starten eines BPEL-Prozesses verwendet werden (vgl. [BPEL2.0, S. 30, 90, 95 f.]). Diese Aktivität stellt eine Neuerung im Vergleich zur Vorversion [BPEL1.1] dar.

```
<extensionActivity>
   <ext:suspend/>
</extensionActivity>
```

**Listing 5: <extensionActivity>**

#### 2.4.1.4 <exit>

Mit der <exit> Aktivität wird eine BPEL-Prozessinstanz unmittelbar beendet, eine Ausführung des <terminationHandler> findet also nicht statt (vgl. [BPEL2.0, S. 30, 96]). In der Vorversion [BPEL1.1] wurde diese Aktivität als <terminate> bezeichnet. Das folgende Beispiel aus [BPEL2.0, S. 162] zeigt eine Verwendung in der Fehlerbehandlung:

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

**Listing 6: <exit>**

#### 2.4.1.5 <invoke>

Mit der <invoke> Aktivität können Web Services, die von (Prozess-) Partnern angeboten werden, aufgerufen werden. Der Aufruf kann dabei synchron (request-response) oder asynchron (one-way) erfolgen. Die <invoke> Aktivität hat zudem die Möglichkeit, ohne die Verwendung eines <scope>s (in-line, siehe 2.4.3.3) Elemente zur Fehlerbehandlung und Kompensation (siehe 2.4.5.7, 2.4.5.3) zu definieren (vgl. [BPEL2.0, S. 25, 84 ff., 227 f.]). In [BPEL2.0, S. 86 f.] wird eine asynchrone Verwendung mit zusätzlicher Fehlerbehandlung (<catch>, allerdings ohne umgebenden <faultHandlers>) und einem <compensationHandler> gezeigt:

```
<invoke name="purchase" suppressJoinFailure="yes" partnerLink="Seller"
        portType="SP:Purchasing" operation="Purchase"
        inputVariable="sendPO" outputVariable="getResponse">
   <catch faultName="SP:rejectPO">...</catch>
   <compensationHandler>
      <invoke partnerLink="Seller" portType="SP:Purchasing"
              operation="CancelPurchase" inputVariable="getResponse"
              outputVariable="getConfirmation" />
   </compensationHandler>
</invoke>
```

**Listing 7: <invoke>**

#### 2.4.1.6 <receive>

Bei der <receive> Aktivität wird auf eine bestimmte Nachricht gewartet. Dabei kann mit einer <receive> Aktivität ein BPEL-Prozess gestartet werden. Dazu muss das Attribut createInstance auf den Wert yes gesetzt sein. Außer der <receive> Aktivität können auch <pick> (siehe 2.4.2.4) oder <extensionActivity> (siehe 2.4.1.3) zum Starten eines BPEL-

Prozesses eingesetzt werden (vgl. [BPEL2.0, S. 24, 89 ff.]). Den Start eines Prozesses mit <receive> stellt das Beispiel aus [BPEL2.0, S. 142] vor:

```
<receive name="getOrder"
   partnerLink="buyer"
   operation="order"
   variable="orderDetails"
   createInstance="yes" />
```

**Listing 8: <receive>**

#### 2.4.1.7 <reply>

Mit der <reply> Aktivität werden Nachrichten beantwortet (Response), die zuvor bei einer <receive> (siehe 2.4.1.6), <pick> (siehe 2.4.2.4) oder möglicherweise einer <extensionActivity> (siehe 2.4.1.3) Aktivität eingegangen sind (Request). Die empfangende Aktivität muss zu diesem Zweck in einer Request-Response Interaktion eingesetzt werden (vgl. [BPEL2.0, S. 25, 92 ff.]). In [BPEL2.0, S. 181] ist ein Beispiel angeführt, in dem <reply> bei einem Verarbeitungsfehler eingesetzt wird:

```
<catchAll>
   <reply partnerLink="customer"
      portType="lns:loanServicePT"
      operation="request" variable="error"
      faultName="unableToHandleRequest" />
</catchAll>
```

**Listing 9: <reply>**

#### 2.4.1.8 <rethrow>

Mit der <rethrow> Aktivität kann ein Fehler von einen Fault Handler, der den Fehler bereits behandelt, an den nächsten übergeordneten Fault Handler übergeben werden. (vgl. [BPEL2.0, S. 30, 96 f.]). Diese Aktivität stellt eine Neuerung im Vergleich zur Vorversion [BPEL1.1] dar. Das nachfolgende Beispiel aus [BPEL2.0, S. 132] zeigt, wie mit <rethrow> ein unbekannter Fehler (siehe <catchAll>, 2.4.5.7) an den übergeordneten Fault Handler übergeben wird:

```
<catchAll>
   <rethrow />
</catchAll>
```

**Listing 10: <rethrow>**

#### 2.4.1.9 <throw>

Die <throw> Aktivität wird normalerweise dazu verwendet, der BPEL-Engine einen Laufzeitfehler zu melden und eine entsprechende Fehlerbehandlung zu bewirken (vgl. [BPEL2.0, S. 26, 94 f.]). Dies wird in [BPEL2.0, S. 99] in einem Beispiel dargestellt, in dem durch <throw> die weitere Verarbeitung an den zugeordneten Fault Handler übergeben wird. Daran wird zudem deutlich, dass die <throw> Aktivität ebenso wie die Fault Handler nicht nur für Programmfehler verwendet werden dürfen:

```
<throw faultName="FLT:OutOfStock" variable="RestockEstimate" />
```

**Listing 11: <throw>**

#### 2.4.1.10 <wait>

Bei der <wait> Aktivität wird festlegt, dass entweder bis zu einem bestimmten Zeitpunkt (mit <until>) oder für eine gewisse Zeitdauer (mit <for>) gewartet werden soll (vgl. [BPEL2.0, S. 27, 95]). Dazu ein Beispiel, bei dem mittels <wait> auf einen bestimmten Zeitpunkt gewartet werden soll:

```
<wait>
   <until>'2008-04-30T18:00+01:00'</until>
</wait>
```

**Listing 12: <wait>**

### 2.4.2 Structured Activities

#### 2.4.2.1 <flow>

Eine <flow> Aktivität wird im Allgemeinen für die Beschreibung paralleler Abläufe verwendet. Dabei können <link>s (siehe 2.4.2.1.1) dazu verwendet werden, Kontrollabhängigkeiten zwischen den darin enthaltenen Aktivitäten festzulegen. Alle Aktivitäten in einem <flow>, die keinen festgelegten Vorgänger haben, können parallel verarbeitet werden (vgl. [BPEL2.0, S. 29, 102 ff.]). Die Regeln für die Übergänge, die mit <link>s festgelegt werden, sind nachfolgend behandelt. Hier ein Beispiel, bei dem die Aktivität A und B parallel ausgeführt werden. Aktivität C wird erst dann ausgeführt, wenn Aktivität A und B abgeschlossen sind:

```
<flow>
   <links>
      <link name="A-to-C" />
      <link name="B-to-C" />
   </links>
   <invoke name="A" >
      <sources>
         <source linkName="A-to-C" />
      </sources>
   </invoke>
   <invoke name="B" >
      <sources>
         <source linkName="B-to-C" />
      </sources>
   </invoke>
   <invoke name="C" >
      <targets>
         <target linkName="A-to-C" />
         <target linkName="B-to-C" />
      </targets>
   </invoke>
</flow>
```

**Listing 13: <flow>**

Die nachfolgende Abbildung dient dem Verständnis des <flow> Konstrukts und dessen Anwendungsmöglichkeiten. Zur einfacheren Lesbarkeit wird weder Anspruch auf syntaktische Korrektheit noch auf Vollständigkeit erhoben:

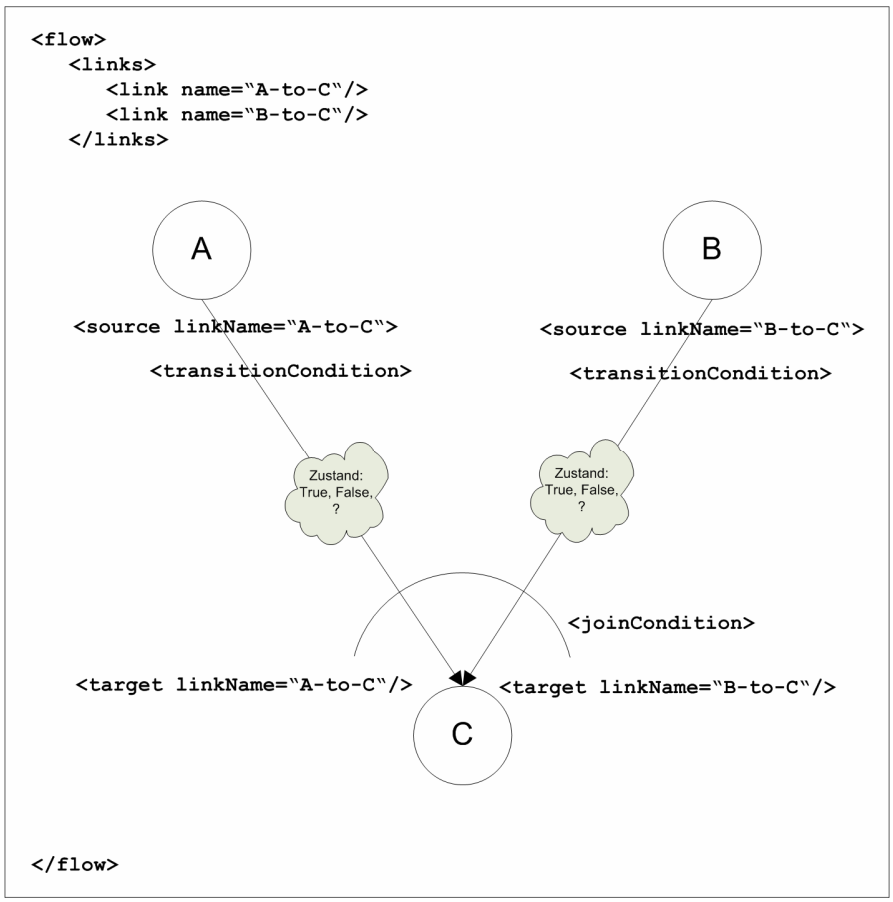


**Abbildung 8: Informelle Darstellung von <flow>**

#### 2.4.2.1.1 <link>

Die `<link>` Elemente werden innerhalb einer `<flow>` Aktivität definiert, sie legen in nicht notwendigerweise parallelen Abläufen den Kontrollfluss fest. Um festzulegen, wo ein `<link>` seinen Ursprung und sein Ziel hat, wird er an die entsprechende Aktivität gebunden. Eine Aktivität, die Ursprung eines `<link>` ist, hat hierfür ein `<sources>` Unterelement. Darin werden die von der Aktivität ausgehenden `<link>`-Elemente aufgeführt, unter dem Elementnamen `<source linkName="…">`. Analog hat eine Aktivität, die Ziel eines `<link>`s ist, ein `<targets>` Unterelement. In diesen Unterelementen sind die ausgehenden, beziehungsweise eingehenden `<link>` Elemente aufgeführt (vgl. [BPEL2.0, S. 102 ff.]).

Es gibt bestimmte Einschränkungen, wie die `<link>`-Elemente verwendet werden dürfen: Generell wird die Erzeugung von Zyklen ausgeschlossen, es ist also nicht erlaubt, eine Aktivität mit einem logischen Vorgänger zu verlinken und dadurch Zyklen zu erzeugen (vgl. [BPEL2.0, S. 105]). Auch dürfen sie bestimmte Grenzen (Boundaries) nicht überschreiten, so dürfen `<link>`s die wiederholenden Konstrukte `<while>` (siehe 2.4.2.7), `<repeatUntil>` (siehe 2.4.2.5), `<forEach>` (siehe 2.4.2.2), `<eventHandlers>` (siehe 2.4.5.12) sowie `<compensationHandler>` (siehe 2.4.5.3) nicht verlassen. Um `<link>`s in diesen Konstrukten verwenden zu dürfen, müssen sie daher in ein `<flow>` Element innerhalb der Konstrukte eingebettet sein (vgl. [BPEL2.0, S. 104 f.]). Des Weiteren dürfen sie das eigentliche `<flow>` Konstrukt, in dem sie definiert sind, nicht verlassen. In den Fehlerbehandlungskonstrukten `<catch>` und `<catchAll>`, sowie dem `<TerminationHandler>` sind eingehende `<link>`s untersagt.

#### 2.4.2.1.2 <transitionCondition>

Ist eine Aktivität der Ursprung eines <link>s, dann hat das <source> Unterelement eine <transitionCondition>, mit der die Bedingung zum Übergang festgelegt werden kann. Diese Bedingung wird nach Abschluss der Aktivität ausgewertet. Ist keine <transitionCondition>, angegeben, so wird der <link>-Zustand immer zu *true* ausgewertet, und zwar sobald die Aktivität abgeschlossen ist. Die <transitionCondition> ist nicht die einzige Bedingung, die den Zustand eines <link>s festlegt. Dieser kann ebenso durch faults und Dead-Path-Elimination (siehe 2.4.2.1.4) bestimmt werden (vgl. [BPEL2.0, S. 106 ff.]). Zur Verdeutlichung folgt ein Beispiel aus [BPEL2.0, S. 182] für eine <transitionCondition>:

```
<source linkName="receive-to-approval">
   <transitionCondition>
      $request.amount >= 10000
   </transitionCondition>
</source>
```

**Listing 14: <transitionCondition>**

#### 2.4.2.1.3 <joinCondition>

Wenn eine Aktivität das Ziel eines <link> ist, dann verfügt sein <targets> Unterelement über eine <joinCondition> (Bedingung zur Zusammenführung) mit der festgelegt wird, unter welchen Umständen die Aktivität ausgeführt werden darf. Eine solche <joinCondition> ist ein boolscher Ausdruck über die Zustände der eingehenden <link>-Elemente. Für die Auswertung der <joinCondition> müssen die Zustände der eingehenden <link>s also feststehen (vgl. [BPEL2.0, S. 106]).
Erst wenn dieser Ausdruck zum Wahrheitswert *true* ausgewertet ist, wird die Aktivität ausgeführt. Wenn diese <joinCondition> nicht explizit angegeben ist, dann wird die Aktivität ausgeführt, sobald alle Zustände der eingehenden <link>s feststehen und einer der <link>s den Zustand *true* hat (logisches „oder“). Wenn die <joinCondition> zum Wahrheitswert *false* ausgewertet wird, erfolgt die so genannte *Dead-Path-Elimination (DPE)* durch die *BPEL-Engine*, die den Prozess ausführt. Oder es findet bei einer bestimmten Einstellung eine Fehlerbehandlung durch einen Fault Handler statt. (vgl. [BPEL2.0, S. 105 f.]). Dazu ein Beispiel aus [BPEL2.0, S. 55], wie mit einer <joinCondition> eine AND-Verknüpfung der eingehenden <link>s festgelegt wird:

```
<targets>
   <target linkName="link1" />
   <target linkName="link2" />
   <joinCondition>$link1 and $link2</joinCondition>
</targets>
```

**Listing 15: <joinCondition>**

#### 2.4.2.1.4 Dead-Path-Elimination

Ein Workflow Management System (wie auch eine BPEL-Engine) muss Situationen, in denen bestimmte Aktivitäten im Ablaufgraphen nicht mehr erreichbar sind, automatisch auflösen (vgl. [LeRo00, S.146 ff.]). Bei dem <flow>-Konstrukt können sich folgende Situationen ergeben:
Die Auswertung des <targets>-Unterelements <joinCondition>, und somit auch die Auswertung der <transitionCondition>s der eingehenden <link>-Elemente, entscheiden darüber, ob eine Aktivität ausgeführt wird. Wird die <joinCondition> zu *false* ausgewertet, wird die Aktivität nicht ausgeführt und die ausführende BPEL-Engine führt die Dead-Path-Elimination (DPE) durch. Dies ist ebenso der Fall, wenn die <joinCondition> nicht explizit angegeben ist und die <transitionCondition>s aller eingehenden <link>s zu *false* ausgewertet werden.

Hierbei wird der Zustand von allen <link>s, die von der nicht auszuführenden Aktivität ausgehen, auf *false* gesetzt, unabhängig von ihrer <transitionCondition>. Dadurch können eventuell weitere <joinCondition>s (rekursiv) zu *false* ausgewertet werden. Die zugehörigen Aktivitäten werden übersprungen und auf diese Weise ganze Pfade von der Ausführung ausgenommen (dead paths). DPE wird nur durchgeführt, wenn das für den betreffenden <flow> gültige Attribut (vgl. [BPEL2.0, S. 23]) suppressJoinFailure auf yes gesetzt ist, andernfalls wird ein Fehler generiert (vgl. [BPEL2.0, S. 108 ff.]).

#### 2.4.2.2 <forEach>

Mit der <forEach> Aktivität können mehrfache Ausführungen von Aktivitäten realisiert werden. Diese Ausführungen können sequentiell oder parallel stattfinden und unter Umständen vorzeitig beendet werden. Die in diesem Konstrukt enthaltene (genestete) Aktivität muss dabei vom Typ <scope> (siehe 2.4.3.3) sein (vgl. [BPEL2.0, S. 28, 112 ff.]). Diese Aktivität stellt eine Neuerung im Vergleich zur Vorversion [BPEL1.1] dar. Nachfolgend ein Beispiel aus [Schu07, S. 19], bei dem die in der <scope>-Aktivität enthaltenen Aktivitäten sequentiell ausgeführt werden. Sobald 10 von maximal 50 Ausführungen erfolgreich sind, wird die Schleife beendet:

```
<forEach counterName="counter" parallel="no">
   <startCounterValue>1</startCounterValue>
   <finalCounterValue>50</finalCounterValue>
   <completionCondition>
      <branches successfulBranchesOnly="yes">10</branches>
   </completionCondition>
   <scope>...</scope>
</forEach>
```

**Listing 16: <forEach>**

#### 2.4.2.3 <if>

Die <if> Aktivität dient der bedingten Ausführung von Aktivitäten (vgl. [BPEL2.0, S. 27, 99]). In der Vorversion [BPEL1.1] wurde diese Aktivität als <switch> mit der dafür üblichen Syntax verwendet. Das folgende Beispiel, angelehnt an [BPEL2.0, S. 99] zeigt eine <if> Aktivität mit dem optionalen <elseif> und <else>. Wenn die <condition> erfüllt ist, wird die nachfolgende Aktivität ausgeführt und die <if>-Aktivität anschließend beendet, es werden also keine weiteren Zweige betrachtet:

```
<if>
   <condition>
      $stockResult > 100
   </condition>
   <flow> . . . </flow>
   <elseif>
      <condition>
         $stockResult >= 0
      </condition>
      <throw faultName="FLT:OutOfStock" />
   </elseif>
   <else>
      <throw faultName="FLT:ItemDiscontinued" />
   </else>
</if>
```

**Listing 17: <if>**

#### 2.4.2.4 <pick>

Die <pick> Aktivität wird eingesetzt, wenn mehrere verschiedene, alternative Nachrichten eintreffen können. Diese werden dann von der entsprechenden Unteraktivität in dem betreffenden <onMessage>-Element verarbeitet. Bei <pick> lässt sich mittels <onAlarm> ein Timeout (Zeitpunkt oder Zeitspanne) einstellen: Wird dieser erreicht, dann wird die <pick>-Aktivität auch ohne den Eingang einer Nachricht beendet.
Die <pick> Aktivität kann dazu eingesetzt werden, einen BPEL-Prozess zu starten. Dazu muss das createInstance Attribute auf yes gesetzt sein (vgl. [BPEL2.0, S. 28 f., 100 ff.]). Es folgt ein Beispielcode zu der Aktivität <pick>, als nicht-startende Aktivität (default) mit zwei möglichen Nachrichten und einem Timeout, angelehnt an [BPEL2.0, S. 101]:

```
<pick>
   <onMessage partnerLink="buyer" portType="orderEntry"
      operation="inputLineItem" variable="lineItem">
      <invoke name="processOrder" ... />
   </onMessage>
   <onMessage partnerLink="buyer" portType="orderEntry"
      operation="orderComplete" variable="completionDetail">
      <invoke name="completeOrder" ... />
   </onMessage>
   <onAlarm>
      <for>'P3DT10H'</for>
      <invoke name="timeout" ... />
   </onAlarm>
</pick>
```

**Listing 18: <pick>**

#### 2.4.2.5 <repeatUntil>

Die <repeatUntil> Aktivität wird dazu verwendet, die enthaltene Unteraktivität solange wiederholt auszuführen, bis die angegebene Bedingung erfüllt ist. Die in <repeatUntil> enthaltene Unteraktivität (in diesem Beispiel: <scope>) wird also mindestens einmal ausgeführt (vgl. [BPEL2.0, S. 28, 100]). Im Vergleich zu der Vorversion [BPEL1.1] stellt diese Aktivität eine Neuerung dar. Das folgende Beispiel aus [Schu07, S. 18] zeigt die <repeatUntil> Aktivität:

```
<repeatUntil>
   <scope>...</scope>
   <condition>$orderDetails > 100</condition>
</repeatUntil>
```

**Listing 19: <repeatUntil>**

#### 2.4.2.6 <sequence>

Bei einer <sequence> Aktivität werden alle enthaltenen (genesteten) Aktivitäten sequentiell ausgeführt und zwar in der Reihenfolge, in der sie aufgelistet sind. Diese Aktivität gilt dann als beendet, wenn die letzte der enthaltenen Aktivitäten abgeschlossen wurde. Mit dem <flow>-Konstrukt und entsprechend definierten <link>s (siehe 2.4.2.1) lässt sich diese Semantik ebenso modellieren (vgl. [BPEL2.0, S. 27, 98]). In [BPEL2.0, S. 165] ist ein Beispiel für eine derartige sequentielle Ausführung angegeben:

```
<sequence>
   <invoke operation="operation1" ... />
   <invoke operation="operation2" ... />
</sequence>
```

**Listing 20: <sequence>**

#### 2.4.2.7 <while>

Mit der <while> Aktivität kann angegeben werden, dass die darin enthaltene Unteraktivität wiederholt ausgeführt werden soll solange die angegebene <condition> (Bedingung) erfüllt ist (vgl. [BPEL2.0, S. 27, 99 f.]). Folgendes Beispiel zu der <while> Aktivität aus [BPEL2.0, S. 100] verdeutlicht dies:

```
<while>
   <condition>$orderDetails > 100</condition>
   <scope>...</scope>
</while>
```

**Listing 21: <while>**

### 2.4.3 Scopes

#### 2.4.3.1 <compensate>

Die <compensate> Aktivität startet in allen inneren, erfolgreich abgeschlossenen <scope>s die Kompensation bei einem Fehlerfall. Ein innerer <scope> ist dabei ein <scope>, der im selben Zweig (in XML Darstellung) wie der Handler und auf gleicher oder größerer Tiefe liegt. Dazu werden die entsprechenden <compensationHandler>, die an diese <scope>s angeheftet sind, von der ausführenden BPEL-Engine aufgerufen. Der Aufruf der <compensationHandler> (siehe 2.4.5.3) in den <scope>s erfolgt dabei in umgekehrter Reihenfolge zu ihrer Kontrollabhängigkeit (vgl. [BPEL2.0, S. 132 ff.]).
Diese Aktivität darf nur innerhalb eines <faultHandlers> (siehe 2.4.5.7), <compensationHandler> (siehe 2.4.5.3) oder <terminationHandler> (siehe 2.4.5.11) aufgerufen werden (vgl. [BPEL2.0, S. 30, 122 ff.]). Einen Aufruf der Kompensation im Fehlerfall zeigt [BPEL2.0, S. 121]:

```
<faultHandlers>
   <catch faultName="prefix:someFault">
      <compensate />
   </catch>
</faultHandlers>
```

**Listing 22: <compensate>**

#### 2.4.3.2 <compensateScope>

Die <compensateScope> Aktivität wird dazu verwendet, die Kompensation auf dem darin angegebenen inneren <scope> zu starten. Im Gegensatz zu <compensate> lässt sich dadurch ein spezifischer zu kompensierender <scope> angeben. Es gelten dieselben Einschränkungen zur Verwendung wie bei der <compensate> Aktivität (vgl. [BPEL2.0, S. 30, 122 ff.]). Das folgende Beispiel zeigt eine Verwendung von <compensateScope> aus [BPEL2.0, S. 123]:

```
<compensateScope target="RecordPayment"/>
```

**Listing 23: <compensateScope>**

#### 2.4.3.3 <scope>

Die <scope> Aktivität wird benötigt, um für die darin enthaltene Aktivität bestimmte, nur in diesem Bereich gültige Einstellungen und Angaben zu machen. Dazu gehören <faultHandlers>, <eventHandlers>, <compensationHandler>, <terminationHandler>, <correlationSets>, <partnerLinks>, <messageExchanges> und <variables> (vgl. [BPEL2.0, S. 29 f., 115 ff.]). Dieses Beispiel, entnommen aus [BPEL2.0, S. 146], zeigt die Definition eines <compensationHandler> für eine <sequence>-Aktivität:

```
<scope name="Y">
   <compensationHandler>
      <sequence name="undoY Seq">...</sequence>
   </compensationHandler>
   <sequence name="doY Seq">...</sequence>
</scope>
```

**Listing 24: <scope>**

### 2.4.4 Variables

#### 2.4.4.1 <variable>

Variablen in BPEL werden mit dem `<variable>` Konstrukt realisiert. Ihr Gültigkeitsbereich ist auf den `<scope>` begrenzt in dem sie deklariert ist. Für die Variablendeklaration können WSDL Nachrichtentypen oder XML Schema verwendet werden. Wird eine Variable auf der `<process>`-Ebene definiert, so spricht man auch von einer globalen Variable (vgl. [BPEL2.0, S. 45 ff.]). In dem Beispiel aus [BPEL2.0, S. 72] werden zwei Variablen definiert, die erste aus einem WSDL Nachrichtentyp, die zweite aus einem XML Schema:

```
<variable name="c2" messageType="x:person" />
<variable name="c3" element="x:address" />
```

**Listing 25: <variable>**

#### 2.4.4.2 <validate>

Die `<validate>` Aktivität dient dazu, Variablen gegen ihre zugehörige XML und WSDL Datendefinition zu validieren (vgl. [BPEL2.0, S. 26, 48 f.]):

```
<validate variables="orderDetails" />
```

**Listing 26: <validate>**

### 2.4.5 Weitere Konstrukte

#### 2.4.5.1 <catch>

Die Fehlerbehandlungsaktivität `<catch>` ermöglicht es, einen bestimmten Fehler zu behandeln, der durch einen Bezeichner und den Fehlernachrichtentyp angegeben wird (vgl. [BPEL2.0, S. 127 ff.]). In [BPEL2.0, S. 181] wird beispielhaft dargestellt, wie ein Fehler auf `<process>` Ebene durch Versenden einer Fehlernachricht an den Aufrufer des Prozesses behandelt werden könnte:

```
<faultHandlers>
   <catch faultName="lns:loanProcessFault" faultVariable="error"
          faultMessageType="lns:errorMessage">
      <reply partnerLink="customer" portType="lns:loanServicePT"
             operation="request" variable="error"
             faultName="unableToHandleRequest" />
   </catch>
</faultHandlers>
```

**Listing 27: <catch>**

#### 2.4.5.2 <catchAll>

Mit dem `<catchAll>` Konstrukt können alle übrigen, nicht mittels einem spezifischen `<catch>`-Konstrukts aufgefangenen Fehler behandelt werden (vgl. [BPEL2.0, S. 127 ff.]).

Hier ein Beispiel, in dem alle nicht gesondert beschriebenen Fehler an die übergeordneten <faultHandlers> (siehe 2.4.5.7) mittels <rethrow> (siehe 2.4.1.8) übergeben werden:

```
<faultHandlers>
   ...
   <catchAll>
      <rethrow />
   </catchAll>
</faultHandlers>
```

**Listing 28: <catchAll>**

### 2.4.5.3 <compensationHandler>

Ein <compensationHandler> fasst die zur Kompensation auszuführenden Aktionen zusammen. Er beinhaltet also die Aktivitäten, die einen vorigen Zustand so gut wie möglich wiederherstellen sollen (vgl. [BPEL2.0, S. 118 ff.]). Ein Beispiel für einen <compensationHandler> ist in 2.4.3.3 aufgelistet. Für die <invoke> Aktivität ist zusätzlich die Möglichkeit vorgesehen, einen <compensationHandler> direkt an der Aktivität (in-line) anzubringen, wie auch in [BPEL2.0, S. 87] dargestellt:

```
<invoke name="purchase" partnerLink="Seller" portType="SP:Purchasing"
        operation="Purchase" inputVariable="sendPO" outputVariable="getResponse">
   <compensationHandler>
      <invoke partnerLink="Seller" portType="SP:Purchasing"
              operation="CancelPurchase" inputVariable="getResponse"
              outputVariable="getConfirmation" />
   </compensationHandler>
</invoke>
```

**Listing 29: <compensationHandler> in <invoke>**

### 2.4.5.4 <correlationSets>

<correlationSets> werden benötigt um Nachrichten, die zwischen den am Ablauf beteiligten Diensten ausgetauscht werden, bestimmten Prozessinstanzen und darin enthaltenen Aktivitäten gezielt zuordnen zu können. Sie werden an einem <process> oder <scope> definiert und können mit den <correlations> Elementen an Aktivitäten verwendet werden, die Nachrichten versenden oder empfangen (vgl. [BPEL2.0, S. 75 ff.]). In [BPEL2.0, S. 81 f.] wird ein Beispiel mit einer <receive> Aktivität (siehe 2.4.1.6) aufgelistet:

```
<scope>
   <correlationSets xmlns:cor="http://example.com/supplyCorrelation">
      <correlationSet name="PurchaseOrder" properties="cor:customerID
                      cor:orderNumber" />
   </correlationSets>
   <receive partnerLink="Buyer" portType="SP:PurchasingPT"
            operation="PurchaseRequest" variable="PO">
      <correlations>
         <correlation set="PurchaseOrder" initiate="yes" />
      </correlations>
   </receive>
</scope>
```

**Listing 30: <correlationSets>**

#### 2.4.5.5 <documentation>

Das <documentation> Element stellt eine optionale Erweiterung der BPEL Konstrukte dar. Es dient als Platzhalter für die Dokumentation und kann beispielsweise reinen Text oder auch HTML enthalten (vgl. [BPEL2.0, S. 32]). Das folgende Listing aus [BPEL2.0, S. 20] zeigt ein <documentation> Element unterhalb von einer <invoke> Aktivität:

```
<invoke partnerLink="invoicing" portType="lns:computePricePT"
        operation="initiatePriceCalculation" inputVariable="PO">
   <documentation>
      Initial Price Calculation
   </documentation>
</invoke>
```

**Listing 31: <documentation>**

#### 2.4.5.6 <extensions>

Die <extensions> erlauben die Erweiterung von BPEL, beispielsweise können neue Attribute an Aktivitäten angefügt, völlig neue Aktivitäten integriert oder das Laufzeitverhalten der BPEL-Engine angepasst werden (vgl. [BPEL2.0, S. 164 f.]). In [BPEL2.0, S.175 f.] wird das an einem Attribut (uniqueUserFriendlyName) demonstriert:

```
<extensions>
   <extension namespace="http://example.com/bpel/some/extension"
              mustUnderstand="no" />
</extensions>

<invoke partnerLink="shipper" operation="shippingRequest"
        inputVariable="shippingRequestMsg"
        ext:uniqueUserFriendlyName="send shipping request to shipper"/>
```

**Listing 32: <extensions>**

#### 2.4.5.7 <faultHandlers>

Die Konstrukte <catch> und <catchAll> können nur innerhalb von einem übergeordneten <faultHandlers> verwendet werden, dieser dient mit anderen Worten als Container. Die Aktivität <invoke> stellt dabei eine Ausnahme dar: für eine einfachere Darstellung können die Konstrukte <catch> und <catchAll> direkt unterhalb dieser Aktivität angebracht werden. Ein <faultHandlers>-Konstrukt wird direkt im <process>-Element oder in den <scope>s definiert (vgl. [BPEL2.0, S. 127 ff.]). Das nachfolgende <faultHandlers>-Konstrukt beendet bei einem unbekannten Fehler den Prozess:

```
<faultHandlers>
   <catch faultName="noProblem">
      <empty/>
   </catch>
   <catchAll>
      <exit />
   </catchAll>
</faultHandlers>
```

**Listing 33: <faultHandlers>**

#### 2.4.5.8 <import>

Das `<import>` Konstrukt wird in BPEL Prozessen dazu verwendet, Abhängigkeiten zu externen XML Schemata oder WSDL Definitionen anzuzeigen (vgl. [BPEL2.0, S. 32 f.]). Dabei wird das <import> Konstrukt direkt unterhalb des `<process>` Elements eingefügt, wie beispielsweise in [BPEL2.0, S. 169]:

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

**Listing 34: <import>**

#### 2.4.5.9 <partnerLinks>

Die `<partnerLinks>`, `<partnerLinkType>`s, `roles` und `portTypes` werden zur Kommunikation des BPEL Prozesses mit den beteiligten Web Services, beziehungsweise Business Partnern benötigt (vgl. [BPEL2.0, S. 15 ff., 36 ff.]). Die Konzepte der Kommunikation von Web Services sind überaus komplex, der interessierte Leser ist an dieser Stelle auf weiterführende Literatur, [WCLS06], verwiesen. In [BPEL2.0, S. 169] wird ein `<partnerLink>` dargestellt:

```
<partnerLinks>
   <partnerLink name="customer" partnerLinkType="plt:shippingLT"
                partnerRole="shippingServiceCustomer" myRole="shippingService" />
</partnerLinks>
```

**Listing 35: <partnerLink>**

#### 2.4.5.10 <process>

Das `<process>` Konstrukt ist die äußere Klammer um einen BPEL Prozess. In diesem Konstrukt werden für den gesamten Prozess gültige Einstellungen vorgenommen. Dazu gehören `<extensions>`, `<import>`s, `<partnerLinks>`, `<variables>`, `<correlationSets>`, `<messageExchanges>`, `<eventHandlers>` und `<faultHandlers>`. Dies wird zusammenfassend auch als `root-context` bezeichnet. Das `<process>`-Konstrukt darf nur genau eine Aktivität enthalten, daher in der Regel eine `<sequence>`, ein `<scope>` oder eine `<flow>` Aktivität. Das Attribut für den Standard-XML-Namensraum (`xmlns`) gibt dabei an, ob es sich um einen `executable process` oder um einen `abstract process` handelt (vgl. [BPEL2.0, S. 21 ff.]).
Ein `executable process` beschreibt das tatsächliche Verhalten des BPEL-Prozesses und ist zur Ausführung auf einer BPEL-Engine bestimmt. Bei einem `abstract process` hingegen lassen sich Details von dem Prozess verbergen, beispielsweise um die Schnittstellen für Geschäftspartner offen zu legen, ohne die inneren Abläufe Preis zu geben. Dazu werden Erweiterungen spezifiziert, wie zum Beispiel die `<opaqueActivity>` (vgl. [BPEL2.0, S. 147 ff.]). In [BPEL2.0, S. 19] wird ein Prozess beispielhaft beschrieben, hier ein Ausschnitt:

```
<process name="purchaseOrderProcess"
         targetNamespace="http://example.com/ws-bp/purchase"
         xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
         xmlns:lns="http://manufacturing.org/wsdl/purchase">

   <extensions>...</extensions>
   <import ... />
   <partnerLinks>...</partnerLinks>
   <messageExchanges>...</messageExchanges>
   <variables>...</variables>
   <correlationSets>...</correlationSets>
   <faultHandlers>...</faultHandlers>
   <eventHandlers>...</eventHandlers>
```

```
   <sequence>...</sequence> <!-Die eigentliche Aktivität -->

</process>
```

**Listing 36: <process>**

### 2.4.5.11 <terminationHandler>

Wenn ein gesamter Prozess oder ein `<scope>` beendet werden muss, da ein Fehler nicht behandelt werden kann, hat man mit dem `<terminationHandler>`-Konstrukt die Möglichkeit, den Schaden zu begrenzen. Nach der Terminierung laufender Aktivitäten werden diejenigen Aktivitäten ausgeführt, die in dem `<terminationHandler>` enthalten sind (vgl. [BPEL2.0, S. 132, 135 f.]). In der Spezifikation ist hierfür mit dem Default-`<terminationHandler>` (vgl. [BPEL2.0, S. 132]) nur ein einziges Beispiel aufgeführt:

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

**Listing 37: <terminationHandler>**

### 2.4.5.12 <eventHandlers>

Die `<eventHandlers>` können eine beliebige Anzahl von `<onEvent>` oder `<onAlarm>` Elemente enthalten (mindestens jedoch eines). Mit diesen Elementen kann auf bestimmte Ereignisse reagiert werden. Ein `<eventHandlers>`-Konstrukt kann direkt in einem `<process>`-Element oder einem `<scope>` genestet werden und ist gültig, solange der `<process>`, beziehungsweise der `<scope>` aktiv ist. Aktiv bedeutet, dass er gestartet, aber noch nicht beendet wurde. Die in dem eigentlichen Handler enthaltene Aktivität muss vom Typ `<scope>` (siehe 2.4.3.3) sein (vgl. [BPEL2.0, S. 137 ff.]).

#### 2.4.5.12.1 <onEvent>

Jedes `<onEvent>`-Element in einem `<eventHandlers>`-Konstrukt steht für eine Reaktion auf das Eintreffen einer bestimmte Nachricht, „inbound messages that correspond to a WSDL operation“ (vgl. [BPEL2.0, S. 137 ff., 143 f.]). In [BPEL2.0, S. 141] wurde beispielsweise ein `<onEvent>` zum Abbruch einer Bestellung eingesetzt:

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

**Listing 38: <onEvent>**

#### 2.4.5.12.2 <onAlarm>

Das `<onAlarm>` Element wird für zeitgesteuerte Ereignisse eingesetzt. Dabei kann entweder mit dem Unterelement `<for>` eine Zeitspanne oder dem Unterelement `<until>` ein Zeitpunkt angegeben werden, an dem das Ereignis automatisch ausgelöst werden soll. Bei Verwendung in einem `<eventHandlers>`-Konstrukt unterscheidet es sich von dem `<onAlarm>` Element der `<pick>` Aktivität (siehe 2.4.2.4) um zwei wichtige Details: Bei `<eventHandlers>` kann zusätzlich das Element `<repeatEvery>` verwendet werden. Dadurch wird die Ausführung nach der darin angegebenen Zeitspanne erneut wiederholt. Diese Wiederholung wird solange durchgeführt, bis der `<scope>`, der diese `<eventHandlers>` beinhaltet, beendet ist (vgl. [BPEL2.0, S. 22, 141 f.]). Der zweite Unterschied ist, dass als auszuführende Aktivität nur ein `<scope>` (siehe 2.4.3.3) genestet werden darf (vgl. [BPEL2.0, S. 137]). Ein Anwendungsbeispiel aus [BPEL2.0, S. 142] zeigt das Element in Verwendung:

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

**Listing 39: <onAlarm>**

Die Beschreibung der Konstrukte aus [BPEL2.0] ist somit, bis auf die sprachlichen Unterschiede zu `Abstract Processes` (siehe 2.4.5.10), vollständig. Die XML Schema Definitionen zu den Konstrukten sind in [BPEL2.0, S. 216 ff.] aufgeführt.

## 2.5 Konstrukte in BPMN 1.1

In diesem Unterkapitel werden die in [BPMN1.1] zur Verfügung stehenden Piktogramme vorgestellt. Zusätzlich zu deren Namen sind relevante Seitenangaben in der Spezifikation sowie eine Kurzbeschreibung der Semantik angegeben. Auf die Attribute der einzelnen Elemente (vgl. [BPMN1.1, S. 245 ff.]) wird in diesem Abschnitt nicht näher eingegangen.

### 2.5.1 Events

Ein *Event* steht für ein Ereignis, das im Verlauf eines Geschäftsprozesses passiert, es beeinflusst den weiteren Ablauf des Prozesses. *Events* haben normalerweise einen *Trigger* (Auslöser, dann spricht man von einem *Catching Event*) oder ein *Result* (Auswirkung, dann spricht man von einem *Throwing Event*).
*Events* werden als Kreise dargestellt, der freie Platz in der Mitte dient für die Darstellung des *Triggers* (mit normal gezeichnetem Symbol) oder des *Results* (mit ausgemaltem Symbol). Die graphische Unterscheidung zwischen *Catching Events* und *Throwing Events* stellt eine Neuerung in BPMN 1.1 im Vergleich zur Vorgängerversion 1.0 dar (vgl. [BPMN1.0, S. 19], [BPMN1.1, S. 21]).

**Abbildung 9: Catching Event**

**Abbildung 10: Throwing Event**

Es wird zusätzlich unterschieden, wann ein Event sich auf den Prozessverlauf auswirkt: Ist der Kreis mit einem einfachen Rand gezeichnet, ist es ein *Start-Event*, mit doppeltem Rand ein *Intermediate-Event* und mit fettem Rand ein *End-Event*. Dabei sind *Start-Events* immer *Catching* und *End-Events* immer *Throwing* (vgl. [BPMN1.1, S. 18, 20 f., 36 ff.]).
Es ist festgelegt, dass ein *Start-Event* nur ausgehende *Sequence Flow* Verbindungen (siehe 2.5.4) haben darf. Das Event ist optional, wenn in einem Diagramm kein *Start-Event* enthalten ist, so werden alle Aktivitäten ohne eingehende *Sequence Flow* Verbindung gleichzeitig gestartet (vgl. [BPMN1.1, S. 61 ff.]). Ein *End-Event* darf nur eingehende *Sequence Flow* Verbindungen haben. Es ist ebenfalls optional, wenn es nicht verwendet wird, so agiert jede Aktivität ohne ausgehende *Sequence Flow* Verbindung als *End-Event*. Wenn allerdings ein *Start-Event* explizit verwendet wird, so ist für dieses ein explizites *End-Event* erforderlich (vgl. [BPMN1.1, S. 65 f.]). *Intermediate Events* können an dem Rand einer *Activity* (siehe 2.5.2) angebracht werden. Sie dienen dann der Behandlung von Ereignissen, Ausnahmen oder Fehlern, welche die *Activity* betreffen. (vgl. [BPMN1.1, S. 45 ff.]).

**Abbildung 11: Start-Event**

**Abbildung 12: Intermediate-Event**

**Abbildung 13: End-Event**

Die folgende Abbildung aus [BPMN1.1, S. 49] stellt eine Übersicht über die verschiedenen *Event*-Typen dar, die in BPMN spezifiziert sind. Diese werden anschließend erläutert.

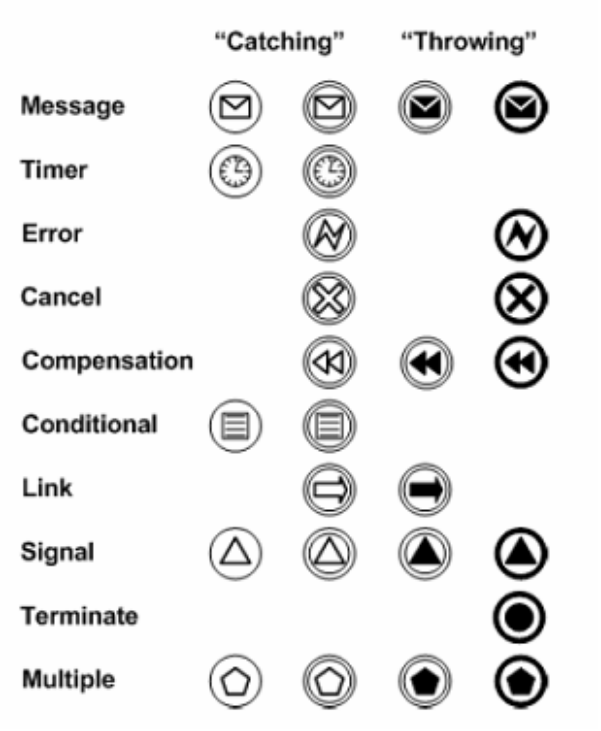


**Abbildung 14: Übersicht der Events in BPMN 1.1**

### 2.5.1.1 Message-Event

Das *Message-Event* wird entweder dazu verwendet, im Prozess auf eine eingehende Nachricht zu warten (*Catching*) und nach deren Eintreffen entsprechend zu reagieren – also *Activities* zu starten – oder um eine Nachricht zu versenden (*Throwing*). Wenn es an den Rand einer *Activity* angebracht wird, dient es der Ausnahmebehandlung (vgl. [BPMN1.1, S. 39, 42, 45]).

**Abbildung 15: Message-Event**

### 2.5.1.2 Timer-Event

Mit dem *Timer-Event* können Prozesse zu einer bestimmten Zeit oder nach einem Ablaufplan gestartet werden, beispielsweise montags um 9.00 Uhr. Ebenso kann es innerhalb von Prozessdiagrammen als Verzögerungsmechanismus verwendet werden. Angeheftet an den Rand einer *Activity* bewirkt es eine Ausnahmebehandlung, sobald ein Zeitlimit oder Zeitpunkt erreicht ist (vgl. [BPMN1.1, S. 39, 44, 45]).

**Abbildung 16: Timer-Event**

#### 2.5.1.3 Error-Event

Das *Error-Event* dient zur Fehlerbehandlung: Als *Intermediate-Event* wird auf ein Fehlerereignis gewartet. Es kann in dieser Form nur an dem Rand einer *Activity* angebracht sein, eingehende *Sequence Flow* Verbindungen sind also unzulässig. In Verbindung mit dem (*Throwing*) *End-Event* wird ein Fehler dem zuständigen (*Catching*) *Intermediate-Event* übermittelt (vgl. [BPMN1.1, S. 42, 45]). Das *Throwing Intermediate Error-Event* (siehe [BPMN1.0, S. 45]) wurde in der Version 1.1 aus der Spezifikation entfernt.

**Abbildung 17: Error-Event**

#### 2.5.1.4 Cancel-Event

Im Zusammenhang mit transaktionalen Subprozessen kommt das *Cancel-Event* zum Einsatz. Als *Intermediate-Event* wird ein *Cancel* registriert und eine Transaktion abgebrochen, es kann allerdings nur an den Rand eines *Sub-Process* geheftet werden. Als *End-Event* wird das Ereignis an die *Intermediate-Events* signalisiert (vgl. [BPMN1.1, S. 42, 45]).

**Abbildung 18: Cancel-Event**

#### 2.5.1.5 Compensation-Event

Ein *Compensation-Event* kann auf zwei Arten verwendet werden: Zum einen kann es als *Catching-Event* (einfach gezeichnetes Symbol) an eine *Activity* angeheftet werden und falls Kompensation notwendig ist, diese Behandlung entsprechend starten. Zum anderen wird von einem *Throwing-Event* signalisiert (mit ausgemalten Symbolen), dass eine Kompensation erforderlich ist (vgl. [BPMN1.1, S. 42, 45]).

**Abbildung 19: Compensation-Event**

#### 2.5.1.6 Conditional-Event

Mit dem *Conditional-Event* können nahezu beliebige Ereignisse behandelt werden. Dafür wird eine Bedingung (*Condition Expression*) in einer nicht näher spezifizierten Sprache angegeben. Das *Event* wird ausgelöst, wenn der Ausdruck vom Wahrheitswert *false* zum Wahrheitswert *true* wechselt. Es kann sowohl zum Start von Prozessen, zur Ereignisbehandlung als auch im normalen Prozessablauf eingesetzt werden (vgl. [BPMN1.1, S. 39, 46]). In [BPMN1.0] wurde die Bezeichnung „*Rule-Event*“ verwendet.

**Abbildung 20: Conditional-Event**

### 2.5.1.7 Link-Event

*Link-Events* können innerhalb einer Prozessebene für Verbindungen verwendet werden, so kann man beispielsweise ein Prozessdiagramm über mehrere Bildschirmseiten verteilen und mit *Link-Events* verbinden, statt unüberschaubar lange *Sequence Flow* Verbindungen darzustellen. Die Verwendung von *Link-Events* über die Grenzen von *Sub-Processes* hinaus ist nicht erlaubt (vgl. [BPMN1.1, S. 46]).

**Abbildung 21: Link-Event**

### 2.5.1.8 Signal-Event

Im Gegensatz zu einer Nachricht ist ein Signal nicht an einen bestimmten Empfänger gerichtet. Durch ein Signal können die *Signal-Events* von mehreren Prozessen angesprochen werden. Das Senden als auch das Empfangen des Signals ist weder auf eine bestimmte Prozessinstanz, noch auf ein bestimmtes Prozessdiagramm beschränkt. Es kann sowohl im normalen Ablauf verwendet werden, als auch an den Rand einer *Activity* angebracht sein (vgl. [BPMN1.1, S. 39, 42, 46, 129]). Das *Signal-Event* wurde erst in BPMN 1.1 eingeführt und stellt im Vergleich zur Vorgängerversion BPMN 1.0 eine Neuerung dar.

**Abbildung 22: Signal-Event**

### 2.5.1.9 Terminate-Event

Ein *Terminate-Event* zeigt in einem laufenden Prozess an, dass alle noch laufenden Aktivitäten umgehend beendet werden sollen. Die Prozessinstanz wird ohne weitere Behandlung von *Events* (zum Beispiel *Compensation*) beendet (vgl. [BPMN1.1, S. 42]).

**Abbildung 23: Terminate-Event**

### 2.5.1.10 Multiple-Event

Um Ereignisse zusammenzufassen wird das *Multiple-Event* eingesetzt: Wenn beispielsweise viele verschiedene Nachrichten oder Signale einen Prozess starten können, lassen sich diese übersichtlich zu einem *Multiple-Event* zusammenfassen. Ebenso lässt sich dadurch das Verschicken von mehreren Nachrichten beim Prozessende zusammenfassen (vgl. [BPMN1.1, S. 39, 42, 47]).

**Abbildung 24: Multiple-Event**

### 2.5.2 Activities

Unter einer *Activity* (Aktivität) wird in BPMN ein Arbeitsschritt verstanden, der in einem Geschäftsprozess durchgeführt wird. Eine *Activity* kann dabei entweder ein einziger, atomarer Arbeitsschritt sein, dann ist von einem *Task*, also von einer einzelnen Aufgabe, die Rede. Wenn die *Activity* aus mehreren Arbeitsschritten zusammengesetzt ist, wird sie als *Sub-Process* bezeichnet.
*Activities* werden durch eine rechteckige Form mit abgerundeten Ecken dargestellt. Symbole am unteren Rand (so genannte *Marker*) auf der Innenseite des Rechtecks machen weitere Angaben über die *Activity*. Zum Beispiel ob sie mehrfach ausgeführt wird oder Kompensationszwecke erfüllt (vgl. [BPMN1.1, S. 53 ff.]). Die Beschreibung der *Activity* kann mittels eines *Labels* (Beschriftung) auf der Rechtecksfläche erfolgen (vgl. [BPMN1.1, S. 29]).

**Abbildung 25: Piktogramm für BPMN-Activity**

#### 2.5.2.1 Task

Ein *Task* (vgl. [BPMN1.1, S. 65 f.]) kann nach dem BPMN Standard drei verschiedene *Marker* haben, die Aufschluss über sein Verhalten geben: Einen *Loop Marker*, einen *Multi-Instance Marker* und einen *Compensation Marker*. Der *Loop-Marker* und der *Multi-Instance Marker* können nicht gemeinsam auftreten. Die Semantik des *Loop-Markers* ist die wiederholende, sequentielle Ausführung des *Tasks* und die Semantik des *Multi-Instance Markers* ist die parallele, mehrfache Ausführung des *Tasks*. Der *Compensation-Marker* zeigt an, dass der *Task* eine Kompensationsfunktion hat.
In [BPMN1.1, S. 66] wird angemerkt, dass Modellierungstools zusätzlich eigene *Marker* einführen können, die den *Task* genauer visuell beschreiben. Diese benutzerdefinierten *Marker* müssen allerdings zusammen mit den anderen *Markern* am unteren inneren Rand des *Task*-Rechtecks gruppiert sein. *Tasks* können durch das Attribut *TaskType* genauer beschrieben werden. Dafür ist jedoch keine Visualisierung spezifiziert, die Darstellung mit kleinen Piktogrammen ist daher herstellerabhängig. Dieses Attribut dient hauptsächlich zur Unterstützung von Übersetzungen in ausführbare Sprachen wie BPEL.
An dieser Stelle ein Hinweis, dass auf BPMN-Attribute in dieser Beschreibung nur eingeschränkt eingegangen wird, da die Visualisierung im Vordergrund dieser Arbeit steht.

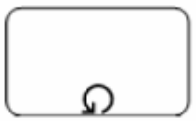

**Abbildung 26: Loop**

**Abbildung 27: Multi-Instance**

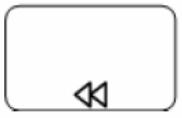

**Abbildung 28: Compensation**

### 2.5.2.2 Sub-Process

Ein *Sub-Process* (vgl. [BPMN1.1, S. 56 ff.]) stellt eine Aktivität höherer Ordnung dar. Sie kann aus beliebig vielen Aktivitäten (auch anderen *Sub-Processes*) zusammengesetzt sein. Um Prozesse auf einem niedrigeren Detailgrad betrachten zu können, lässt sich der Inhalt von einem *Sub-Process* verbergen. In diesem Fall ist ein umrandetes „Plus"-Symbol am unteren Rand des *Sub-Process* zu sehen, man spricht auch von der „collapsed"-Darstellung. Im anderen Fall, der „expanded"-Darstellung, werden die enthaltenen Elemente angezeigt und der *Sub-Process* als rechteckiger Rahmen mit abgerundeten Ecken dargestellt (vgl. [BPMN1.1, S. 21 f.]. Das explizite Anzeigen von einem *Start-Event* und einem *End-Event* ist in der *expanded* Darstellung nicht zwingend erforderlich, also optional.
*Sub-Processes* können vielseitig verwendet werden: Es ist möglich, sie als übergeordneten Kontext für die Ausnahmebehandlung der genesteten *Activities* einzusetzen, indem ein *Catching Error-Event* oder ein *Catching Cancel-Event* an den *Sub-Process* angeheftet wird, der Handler wird mit diesem *Event* verbunden (vgl. [BPMN1.1, S. 57]). Außerdem kann man damit Parallelität erzeugen. Alle Aktivitäten, die nicht explizit mit einem *Start-Event* oder einem *Sequence Flow Connection-Object* verbunden sind, werden parallel ausgeführt (vgl. [BPMN1.1, S. 58]).

Ähnlich wie bei einem *Task* können bei einem *Sub-Process* zusätzliche *Marker* angebracht werden. Diese werden neben dem „Plus"-Symbol anzeigt. Bei einem *Sub-Process* stehen vier solcher *Marker* zu Verfügung. Ein *Loop-Marker*, der die wiederholte Nacheinanderausführung des *Sub-Process* anzeigt. Man darf, wie bei einem *Task*, den *Loop-Marker* allerdings nicht zusammen mit dem *Multi-Instance Marker* verwenden. Dieser steht für die mehrfache, parallele Ausführung des *Sub-Process*. Der *Ad-Hoc Marker* visualisiert, dass die Reihenfolge, in der die Aktivitäten in dem *Sub-Process* ausgeführt werden, dynamisch zur Laufzeit bestimmt wird. Dieser *Marker* wird durch ein „Tilde"-Symbol dargestellt (vgl. [BPMN1.1, S. 58, 132 ff.]). Der *Compensation-Marker* zeigt an, dass der Zweck des *Sub-Process* die Kompensation von zuvor ausgeführten Aktivitäten ist. Er wird durch das Symbol für das Zurückspulen (Rewind, bekannt von Videorekordern) dargestellt.

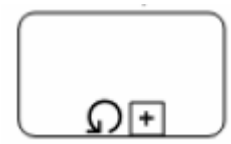

**Abbildung 29: Loop**

**Abbildung 30: Multi-Instance**

**Abbildung 31: Ad-Hoc**

**Abbildung 32: Compensation**

Zusätzlich zu den *Markern* kann ein *Sub-Process* als *Transaction* ausgezeichnet sein. Dazu wird der Rand mit doppelten Linien gezeichnet und ein *Cancel-Event* angeheftet (vgl. [BPMN1.1, S. 30, 63 ff.]).

**Abbildung 33: Transactional Sub-Process**

### 2.5.3 Gateways

*Gateways* sind Modellierungselemente, um die Abfolge der Aktivitäten zu steuern, Verzweigungen sowie Entscheidungen darzustellen und diese wieder zusammenzuführen. Die genaue Semantik eines *Gateways* wird durch einen *Marker* angezeigt der in die innere Fläche der Raute gezeichnet wird. In der Spezifikation wird angemerkt, dass Verbindungen an diese Form nicht auf die Ecken beschränkt sind (vgl. [BPMN1.1, S. 71]). In der Beschreibung der Gateways in [BPMN1.1] werden bei Verzweigungen und Übergängen auch die Begriffe *Split, Join, Fork* und *Merge* verwendet. *Split* und *Join* stehen dabei für Situationen, bei denen Parallelität auftritt und synchronisiert wird. Die Begriffe *Fork* und *Merge* werden bei der Beschreibung von Alternativen und deren Zusammenführung verwendet (vgl. [BPMN1.1, S. 287, 292, 294]).

**Abbildung 34: Piktogramm für BPMN-Gateway**

Die folgende Abbildung aus [BPMN1.1, S. 72] stellt eine Übersicht über die verschiedenen *Gateway*-Typen dar, die in BPMN spezifiziert sind. Deren Semantik wird anschließend näher erläutert.

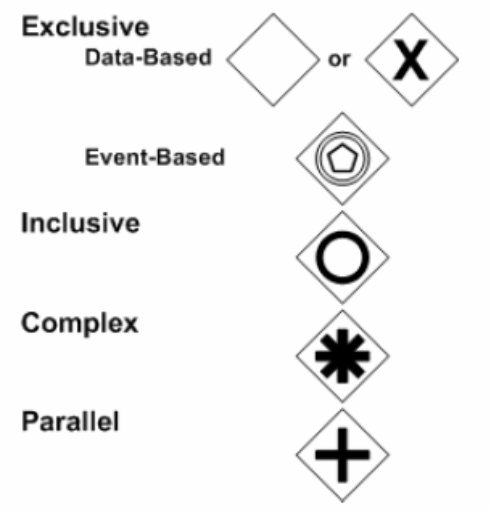


**Abbildung 35: Übersicht der Gateways in BPMN 1.1**

#### 2.5.3.1 Exclusive Gateway (XOR)

Für die Darstellung von Entscheidungen im Ablauf werden *Exclusive Gateways* verwendet. Dabei können mittels einer Bedingung (*Condition-Expression*) zwei oder mehr alternative Pfade angegeben werden (auch als *OR-Split* bezeichnet). Diese Pfade können mit diesem Gateway im weiteren Prozessverlauf wieder zusammengelegt werden (auch als *OR-Join* bezeichnet). Für die ausgehenden Pfade kann man sich die Bedingung als Frage vorstellen und jeder der Pfade stellt eine mögliche Antwort auf die Frage dar wie ein Switch/Case (vgl. [BPMN1.1, S. 74]). Es wird zwischen datenbasierten (*Data-Based*) und ereignisbasierten (*Event-Based*) *Exclusive Gateways* unterschieden. In [BPMN1.0] wurde dieses *Gateway* auch als *XOR* bezeichnet.

#### 2.5.3.1.1 Exclusive Data-Based Gateway

Die *Data-Based Gateways* sind der Spezifikation zufolge die am häufigsten verwendeten *Gateways*. Bei diesen *Gateways* werden boolsche Ausdrücke aufgestellt, die Daten aus dem Prozess einbeziehen. Die Auswertung eines solchen Ausdrucks bestimmt, welcher der ausgehenden Zweige verfolgt werden soll. Diese *Gateways* können auch auf andere Weise benutzt werden, nämlich um alternative Zweige wieder zusammenzuführen. Derartige *Gateways* sind allerdings nicht als Synchronisierungspunkt paralleler Zweige zu verstehen.
Die Beschreibung dieser *Gateways* ist wie die gesamte Spezifikation sehr allgemein gehalten, da gezielt Freiräume für verschiedene Anwendungsmöglichkeiten von BPMN geschaffen werden sollten. So wurde auch darauf verzichtet, eine Sprache, beziehungsweise ein Format für die *Condition-Expressions* festzulegen. Das *Gateway* kann mit einem „X“ als Marker versehen werden, damit es besser von anderen *Gateways* unterscheidbar ist. Dies ist jedoch für einen leichteren Umstieg von einem UML Aktivitätsdiagramm nicht unbedingt erforderlich (vgl. [BPMN1.1, S. 74 ff.], [Oest06 S. 302 ff.]). Alternativ können Verzweigungen auch mit einer *Conditional Sequence Flow Connection* ohne *Gateway* dargestellt werden (siehe 2.5.4.1).

**Abbildung 36: Exclusive Data-Based Gateway (XOR)**

#### 2.5.3.1.2 Exclusive Event-Based Gateway

Bei den *Event-Based Gateways* wird der weitere Verlauf dadurch bestimmt, welches von den *Events*, mit denen das *Gateway* verbunden ist, zuerst eintritt. Beispielsweise können auf diese Weise alternative Verzweigungen dargestellt werden, je nach dem, welche Art von Nachricht als erstes eintrifft. Als *Marker* wird dabei das *Catching Multi-Intermediate Event* eingesetzt (vgl. [BPMN1.1, S. 78 ff.]).

**Abbildung 37: Exclusive Event-Based Gateways**

### 2.5.3.2 Inclusive Gateway (OR)

*Inclusive Gateways* erlauben im Gegensatz zu den *Data-Based Gateways* die Auswertung, beziehungsweise die Weiterverfolgung mehrerer Pfade anstatt nur von Einem. Wird die *Condition-Expression* von einem der ausgehenden Pfade zum Wahrheitswert *true* evaluiert, werden die weiteren Alternativen ebenfalls betrachtet. Dadurch wird jeder Pfad, dessen Bedingung erfüllt ist, durchlaufen. Um diese Pfade wieder zusammenzuführen (auch als *Merge* bezeichnet), dient das Inclusive Gateway auch als Synchronisierungspunkt. Dabei wird auf ein *Token* (siehe 2.2.3) von jedem der genommenen Pfade gewartet. Als *Marker* wird ein Kreis eingesetzt (vgl. [BPMN1.1, S. 81 ff.]). In [BPMN1.0] wurde dieses *Gateway* auch als *OR* bezeichnet.

**Abbildung 38: Inclusive Gateway (OR)**

#### 2.5.3.3 Complex Gateway

*Complex Gateways* können dazu dienen, viele verschiedene *Gateways* einer komplexen Situation zusammenzufassen. Mit diesen *Gateways* lässt sich prinzipiell jedes andere *Gateway* oder eine Kombination daraus darstellen. Für das Eingangs- und Ausgangsverhalten des *Complex Gateways* werden dazu bestimmte Ausdrücke, so genannte *Complex-Expressions* angegeben. Dabei bleibt es den Modellierungstools überlassen, wie diese *Complex-Expressions* aussehen könnten, um komplexes Zusammenführungs- und Verzweigungsverhalten zu realisieren. Als *Marker* wird hierzu ein imperiales Stern-Symbol verwendet (vgl. [BPMN1.1, S. 84 ff.]).

**Abbildung 39: Complex Gateway**

#### 2.5.3.4 Parallel Gateway (AND)

*Parallel Gateways* erlauben es, parallele Abläufe zu erzeugen (mit mehreren ausgehenden Verbindungen, auch als *AND-Split* oder *Parallel-Split* bezeichnet) und ebenso wieder zu synchronisieren (mit mehreren eingehenden Verbindungen, auch als *AND-Join* bezeichnet). Sie stellen zwar nicht die einzige Möglichkeit dar, Parallelität zu erzeugen (siehe 2.5.4), aber sie machen diese am besten deutlich. Wird ein *Parallel Gateway* erreicht, so werden alle damit verbundenen Ausgangspfade verfolgt. Wird es zur Synchronisation verwendet, so wird auf alle verbundenen Eingänge gewartet. Als *Marker* wird ein „Plus“-Symbol verwendet (vgl. [BPMN1.1, S. 86 f.]). In [BPMN1.0] wurde diese Gateway auch als *AND* bezeichnet.

**Abbildung 40: Parallel Gateway (AND)**

### 2.5.4 Sequence Flow Connection

*Sequence Flow* Verbindungen werden in BPMN dazu verwendet die Reihenfolge festzulegen, in der die Aktivitäten in einem Prozess ausgeführt werden sollen. Eine solche Verbindung hat immer einen Ursprung und genau ein Ziel. Verzweigungen müssen daher mit mehreren *Sequence Flow* Verbindungen dargestellt werden (vgl. [BPMN1.1, S. 101]). Die Bedeutung von mehreren, aus einer *Activity* ausgehenden *Sequence Flow* Verbindungen ohne ein *Gateway* (siehe 2.5.3) kann entweder Parallelität oder Verzweigung sein (vgl. [BPMN1.1, S. 112, 115, 117, 119, 121, 122, 124]). Die Semantik hängt davon ab, ob die Verbindungen eine Bedingung haben (siehe 2.5.4.1) oder nicht, und ob eine Synchronisation paralleler Zweige stattfindet (vgl. 2.2.3, Tokenkonzept).

Verbindungen mit einer *Sequence Flow Connection* sind zwischen allen *Activities* (*Task* und *Sub-Process*), *Events* (*Start*, *Intermediate*, *End*) und *Gateways* erlaubt – mit den folgenden Einschränkungen:
Ein *Start-Event* darf keine eingehende Verbindung haben, ein *End-Event* darf keine ausgehende Verbindung haben und Elemente innerhalb eines *Sub-Process* dürfen nicht mit Elementen außerhalb des *Sub-Process* verbunden werden. *Pools*, *Lanes*, *Data Objects* und *Annotations* dürfen mit einer *Sequence Flow Connection* überhaupt nicht verbunden werden,

weder eingehend noch ausgehend (vgl. [BPMN1.1, S. 30]). Die Erzeugung von Zyklen ist somit erlaubt, jedoch nicht über die Grenzen (*Boundaries*) von einem *Sub-Process* oder *Pool* hinaus.

Die folgenden Abbildungen stellen die verschiedenen Ausprägungen der *Sequence Flow Connection* dar, die im Folgenden näher beschrieben werden:

**Abbildung 41: Sequence Flow**

**Abbildung 42: Conditional Sequence Flow**

**Abbildung 43: Default Sequence Flow**

#### 2.5.4.1 Conditional Sequence Flow

Eine *Sequence Flow* Verbindung kann ein *Condition Expression* Attribut (Übergangsbedingung) haben. Ist dieses Attribut gesetzt, wird die Verbindung als *Conditional Sequence Flow* bezeichnet. Wenn diese Verbindung ihren Ursprung an einem *Gateway* hat, ist sie allerdings optisch nicht von einer normalen *Sequence Flow* Verbindung zu unterscheiden. Geht sie jedoch von einer *Activity* aus, wird am Anfang eine Raute (*Diamond*) dargestellt (vgl. [BPMN1.1, S. 102]).
Es gibt aber mehrere Einschränkungen, wann ein *Conditional Sequence Flow* verwendet werden darf. Es ist nicht erlaubt, einen *Conditional Sequence Flow* mit dem Ursprung in einem *Exclusive Event Based Gateway*, einem *Complex Gateway*, einem *Parallel Gateway* oder einem *Event* zu verwenden (vgl. [BPMN1.1, S. 103]).

#### 2.5.4.2 Default Sequence Flow

Im Zusammenhang mit dem *Exclusive Data-Based Gateway*, bei dem aus mehreren Alternativen nur eine weiter verfolgt wird, kann eine *Sequence Flow* Verbindung als *Default* gekennzeichnet sein. Der damit verbundene Pfad wird dann verfolgt, wenn alle anderen Alternativen zum Wahrheitswert *false* evaluiert werden. Dasselbe gilt, wenn *Conditional Sequence Flow* Verbindungen direkt an einer *Activity* (*Task* oder *Sub-Process*) eingesetzt werden. Der Verbindungspfeil wird dann mit einem schrägen Strich am Ursprung dargestellt (vgl. [BPMN1.1, S. 102 f.]).

### 2.5.5 Weitere Elemente

#### 2.5.5.1 Message Flow Connection

Um den Austausch von Nachrichten zwischen verschiedenen *Pools* darzustellen wird eine *Message Flow* Verbindung benutzt. Als Verbindungspunkte können dabei entweder *Tasks*, *Sub-Processes*, *Catching / Throwing Message Events* oder die *Pools* selbst dienen. Selbst diejenigen *Tasks*, die sich in einem *Sub-Process* befinden, können als Ursprung oder als Ziel verwendet werden. Ein einziger *Task* kann dabei sowohl ausgehende als auch eingehende *Message Flow* Verbindungen haben (vgl. [BPMN1.1, S. 30 f., 103 ff.])

**Abbildung 44: Message Flow**

### 2.5.5.2 Association Connection

Eine *Association* wird benötigt, um zusätzliche Informationen mit BPMN-Elementen in Verbindung zu setzen. So können beispielsweise Texte oder Graphiken zu Dokumentationszwecken mit bestimmten *Tasks* verbunden werden. Normalerweise hat eine *Association* keine direkte Auswirkung auf den Prozess. Es gibt jedoch eine Ausnahme: *Activities*, die der Kompensation dienen, werden ebenfalls mit einer *Association* an die zu kompensierende Aktivität gebunden. Die Verbindungslinie einer *Association* wird gestrichelt dargestellt, optional mit einer Pfeilspitze (vgl. [BPMN1.1, S. 28, 105 ff.]).

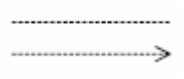

**Abbildung 45: Association**

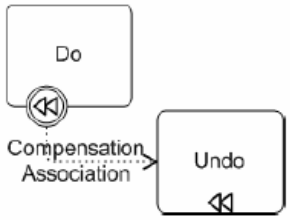


**Abbildung 46: Compensation Association**

### 2.5.5.3 Pools

In der BPMN Spezifikation wird ein *Pool* (Schwimmbecken) als Teilnehmer an einem Prozess beschrieben, im Sinne der Teilnehmer an einer B2B-Interaktion. Der Inhalt eines *Pools*, der private Prozess, kann sichtbar oder verborgen (*opaque*) sein. Dadurch lässt sich die Zusammenarbeit mit mehreren Partnern in Bezug auf den Nachrichtenaustausch visualisieren. Es ist in der Spezifikation nicht festgelegt, ob ein *Pool* vertikal oder horizontal ausgerichtet sein soll. Da allerdings die Mehrzahl der Beispiele in der Spezifikation über eine horizontale Ausrichtung verfügen, kann dies als Vorbild angenommen werden (vgl. [BPMN1.1, S. 29, 87 ff.]).

**Abbildung 47: Pool**

### 2.5.5.4 Lanes

Eine *Lane* (Bahn im Schwimmbecken) ist eine Unterteilung in einem *Pool*, die sich über dessen gesamte Länge erstreckt. *Lanes* dienen dazu, die verschiedenen Aktivitäten in einem Prozess zu organisieren und zu kategorisieren: Beispielsweise können *Lanes* die unterschiedlichen Rollen, Abteilungen, Führungsebenen oder Anwendungssysteme abbilden, die an einem Prozess beteiligt sind (vgl. [BPMN1.1, S. 19, 92 ff.]).

**Abbildung 48: Lanes**

### 2.5.5.5 Data Object

Um das Prozessdiagramm mit zusätzlichen Informationen anzureichern, bietet die BPMN-Spezifikation drei Artefakte: das *Data Object*, eine *Group* und eine *Annotation*.
Ein *Data Object*, dargestellt durch ein Blatt Papier, wird mittels einer oder mehrerer *Associations* mit Elementen des Prozessdiagramms verbunden. Damit können Input- und Outputdaten von Aktivitäten oder gar der Datenfluss modelliert werden, aber nur auf nicht-standardisierte Art und Weise. Eine Verbindung von *Data Objects* mittels *Sequence Flow* Verbindungen oder *Message Flow* Verbindungen wird ausgeschlossen (vgl. [BPMN1.1, S. 95 f.]).


**Abbildung 49: Data Object**

### 2.5.5.6 Group

Mit dem Artefakt *Group* können Elemente graphisch gruppiert werden, jedoch ohne Auswirkung auf die Semantik des Prozesses. Es ist als rein graphische Unterstützung zum besseren Verständnis gedacht und kann daher weder mit Prozesselementen verbunden werden, noch ist es an *Sub-Process Boundaries*, *Lanes* oder gar *Pools* gebunden. Das Artefakt wird mit einer gestrichelten Linie mit Punkten zwischen den Strichen gezeichnet (vgl. [BPMN1.1, S. 97 f.]).

**Abbildung 50: Group**

### 2.5.5.7 Annotation

Jedes Element des Prozessdiagramms kann mit dem Artefakt *Annotation* mit einem zusätzlichen Beschreibungstext versehen werden. Es dient wie das *Data Object* und die *Group* zu Dokumentationszwecken und wird mit einer durchgezogenen Linie und einem angedeuteten Rechteck dargestellt (vgl. [BPMN1.1, S. 97.]).

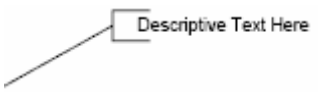


**Abbildung 51: Annotation**

## 2.6 Ansätze zur Verbindungen von BPMN und BPEL

Die Verbindung von BPMN und BPEL hat in beiden möglichen Richtungen einen bedeutsamen Anwendungsfall. Von BPMN zu BPEL kann eine Verbindung die direkte Ausführbarkeit von Geschäftsprozessen herstellen oder erzeugt zumindest ein Grundgerüst (Skeleton) dafür. Von BPEL zu BPMN stellt eine Verbindung eine auf einem Standard basierende Repräsentation eines technischen Workflows dar, der somit möglicherweise besser kommuniziert werden kann. Es ist im Moment noch ausdrücklich von *Verbindungen* die Rede, diese können nämlich auf ganz unterschiedliche Art und Weise ausgestaltet sein. Eine Verbindung kann mehrere Transformationsschritte umfassen, dabei mehr oder weniger verlustfrei sein, unterschiedliche Vorgaben machen und Einschränkungen an die zu Grunde liegenden Prozesse definieren. Wie bereits in 2.3.3 festgestellt, haben die Sprachen BPMN und BPEL eine unterschiedliche Ausdruckskraft, ebenso haben sie verschiedene Sprach-konzeptionen (siehe dazu 2.3.2). Beides wird in den verschiedenen Verbindungen unterschiedlich eingesetzt und berücksichtigt.

In diesem Abschnitt werden die Verbindungen nach der Richtung unterschieden, in der sie BPMN und BPEL verbinden. Zuerst werden die Ansätze zu Verfahren vorgestellt, die ein *Business Process Diagram (BPD)* in einen BPEL-`<process>` transformieren, danach wird die entgegengesetzte Richtung betrachtet. Anschließend wird eine bidirektionale Verbindung gezeigt, mit der Prozesse sowohl in BPMN als auch in BPEL bearbeitet werden können. Eine kritische Analyse und Diskussion der hier vorgestellten Ansätze wird erst in 4.1 vorgenommen, nachdem die für eine Realisierung zur Verfügung stehende Technik (siehe Kapitel 3) evaluiert wurde.

### 2.6.1 Von BPMN zu BPEL

Das folgende Zitat soll die Schwierigkeiten verdeutlichen, mit denen Business Analysten (und Transformationsalgorithmen) konfrontiert sind, wenn sie den Weg von BPMN zu BPEL beschreiten:

*“Not every flow you can draw in BPMN can be exported to BPEL – at least unless you draw it the ‘right way.’ I found that out the hard way after transcribing a client’s hand-drawn flowchart into a BPMN tool. Everything was great until I switched on BPEL validation, when whole blocks of process turned red with BPEL error messages. BPEL doesn’t like ‘interleaved’ process segments, looping back to previous steps, and other common features of user-drawn flowcharts.”* [Silv06]

#### 2.6.1.1 Ansatz von White, BPMI, OMG

##### 2.6.1.1.1 Beispielhafte Transformation von White

In der Arbeit von Stephen A. White (vgl. [Whit05a]) wird an einem Beispiel dargestellt, wie aus einem BPD ein BPEL-Prozess generiert werden könnte. Auch in der Spezifikation wird ausführlich dargestellt (vgl. [BPMN1.0, S.137 – 204], [BPMN1.1, S. 145 – 244]), wie eine Analyse eines BPD und dessen Übersetzung in BPEL im Detail aussehen könnte. Diese Verbindung (auch als Mapping bezeichnet) hatte als Zielsprache [BPEL1.1]. Die Standardisierung durch OASIS konnte daher noch nicht berücksichtigt werden[7], an dieser Stelle spielt dieser Umstand jedoch nur eine untergeordnete Rolle, zudem ist dies in den anderen Ansätzen ebenso der Fall.

[7] Manche Neuerungen in [BPEL2.0] wurden mit einem Workaround bereits schon vor dessen Veröffentlichung umgesetzt, beispielsweise zu der in BPEL4WS noch nicht vorhandenen For-Schleife (siehe [BPMN1.0, S. 148 ff.]). Für die zukünftige Version von BPMN ist eine erweiterte Unterstützung für ein Mapping zu [BPEL2.0] im Gespräch (vgl. [BPMN2.0, S.24]).

In [Whit05a] wird ein Mapping beschrieben, das vorwiegend BPEL-Graphstrukturen, also <flow> (siehe 2.4.2.1) einsetzt. Die Generierung von BPEL-Code wird darin an einem völlig unproblematischen Beispiel beschrieben, das die Ausdruckskraft von BPMN (siehe 2.3.3) nicht in ausreichendem Maße berücksichtigt. Die wesentlichen Schwierigkeiten einer Transformation werden in dieser eher normativen Beschreibung also außer Acht gelassen.

#### 2.6.1.1.2 Transformation in der Spezifikation von BPMN

Das in [BPMN1.1] dargestellte Mapping spricht die Schwierigkeit der überlappenden Schleifen und Sprünge (*interleaved segments*) an. Es wird dort festgestellt, dass eine einfache <while> Konstruktion das Problem nicht löst und auch <flow> nicht geeignet ist, da es keine Zyklen erlaubt. Als Lösungsansatz wird vorgeschlagen, mehrere (Teil-)Prozesse zu erzeugen (*derived and spawned*), die sich gegenseitig aufrufen. Dies wird an einem überschaubaren Beispiel mit fünf *Activities* und zwei *Gateways* ohne Parallelität auf nicht-formale Art dargestellt. Ein Algorithmus für eine automatische Transformation ist nicht angegeben, ebenso vermisst man an dieser Stelle eine Berücksichtigung von Gültigkeitsbereichen von Variablen und von Parallelität (vgl. [BPMN1.1, S. 203 ff., 212]).
Auch die Beschreibung zu der Übersetzung von *Gateways* bezieht die spätere Synchronisierung, Parallelität und das BPMN zugrunde liegende Tokenkonzept (siehe 2.2.3) nicht mit ein (vgl. zum Beispiel das *Inclusive Gateway (OR)*, [BPMN1.1, S. 181ff]).
An der Stelle, an der die Transformation der Sequence Flow Verbindungen (siehe 2.5.4) beschrieben ist, wird letztendlich klar, dass dieses Mapping noch nicht ausreichend spezifiziert ist:

*„A Sequence Flow may not have a specific mapping to a BPEL4WS in most situations. However, when there is a section of the Diagram that contains parallel activities, then Sequence Flow may map to the link element. Details of this mapping are TBD (Anm.: To Be Defined). In general, the set of Sequence Flow within a Pool will determine how BPEL4WS elements are derived and the boundaries of those elements."* [BPMN, S. 185]

### 2.6.1.2 Ansatz von Mendling, Recker, Lassen, Zdun

In [MLZ06] und [ReMe06a] werden zwei grundsätzliche Strategien vorgestellt, wie ein BPD in einen BPEL Prozess übersetzt werden kann. Sie setzen bestimmte, je nach Strategie unterschiedliche Restriktionen an ein BPMN-Diagramm (BPD) voraus.

#### 2.6.1.2.1 Element-Preservation

Die erste Strategie wird als *Element-Preservation* bezeichnet. Unter der Voraussetzung eines azyklischen BPD und der Beschränkung auf eine kleine Teilmenge von BPMN Elementen (vgl. [MLZ05, S.6]) wird eine 1:1 Beziehung hergestellt, die den Ablauf im Prozessdiagramm mit <flow> und <link>s (siehe 2.4.2.1 ff.) abbildet. Das Verfahren arbeitet zweistufig und ersetzt im zweiten Schritt, der *Element-Minimization*, Hilfskonstrukte zur Synchronisation wie <empty> (siehe 2.4.1.2) durch <joinCondition> (siehe 2.4.2.1.3) und <transitionCondition>s (vgl. [MLZ05, S. 11 ff.]). Dieses Vorgehen stellt in gewisser Weise eine formale Beschreibung des in 2.6.1.1.1 beispielhaft dargestellten Verfahrens dar, allerdings ohne die Berücksichtigung von beliebigen Abläufen (Arbitrary Cycles) im Diagramm.

#### 2.6.1.2.2 Structure-Identification

Die zweite Strategie wird als *Structure-Identification* bezeichnet. Dabei wird vorausgesetzt, dass Zyklen im BPD ausschließlich mit dem *XOR-Gateway* (siehe 2.5.3.1.1) erzeugt werden und sich nicht überlappen (vgl. [ReMe06a, S.10], [MLZ05, S.6]). Einfache Sequenzen werden in <sequence> überführt, *AND*-Blöcke (ohne Sprünge), ebenso wie *OR*-Blöcke werden in einen <flow> übersetzt und *XOR*-Blöcke werden zu einem <switch> abgebildet. Schleifen werden in <while> übersetzt, die Abbruchbedingung ist dabei mithilfe von einem <assign> innerhalb der Schleife realisiert. Ein *Start-Event* wird in ein <pick>, und ein *End-Event* in eine <terminate> Aktivität (sic!) abgebildet (vgl. [MLZ05, S.14 f.], [MLZ06, S.11 ff.]).

Unter *Structure-Maximization* wird in diesem Zusammenhang die mehrfache Anwendung der Identifikation von Strukturen verstanden, bis diese nicht weiter auf komplexe Konstrukte abbildbar sind. In einem vollständigen Ablauf einer Transformation werden die resultierenden komplexen Strukturen anschließend mit der *Element-Preservation* (siehe 2.6.1.2.1) in einen BPEL `<flow>` übersetzt (vgl. [MLZ05, S. 15]).

#### 2.6.1.2.3 Beispiel einer Transformation

In der nachfolgenden Abbildung aus [ReMe06b, S.23] ist eine Transformation nach den beschriebenen Verfahren beispielhaft dargestellt. In der erzeugten BPEL Struktur ist allerdings nicht berücksichtigt, dass das *AND Gateway* (siehe 2.5.3.4) synchronisierend auf alle Eingänge wartet. Die `<assign>` Aktivität links unten im Bild darf nicht vor der `<assign>` Aktivität rechts unten im Bild ausgeführt werden, sonst könnte eine *Race Condition* auftreten (vgl. [Tane01, S. 100 ff.]). Möglicherweise ist die Darstellung der BPEL Struktur lediglich unvollständig, ein `<link>` kann zwischen den Aktivitäten in den `<sequence>`-Blöcken zur Synchronisierung der parallelen Pfade eingesetzt werden. In diesem überschaubaren Beispiel ist ein solcher Fehler relativ leicht zu erkennen, jedoch ist „genaues Hinsehen“ bei komplexen Prozessen nicht ausreichend. Dies zeigt einen erheblichen Missstand auf. Die Korrektheit und die semantische Äquivalenz einer Transformation können nicht ohne weiteres bewiesen werden.

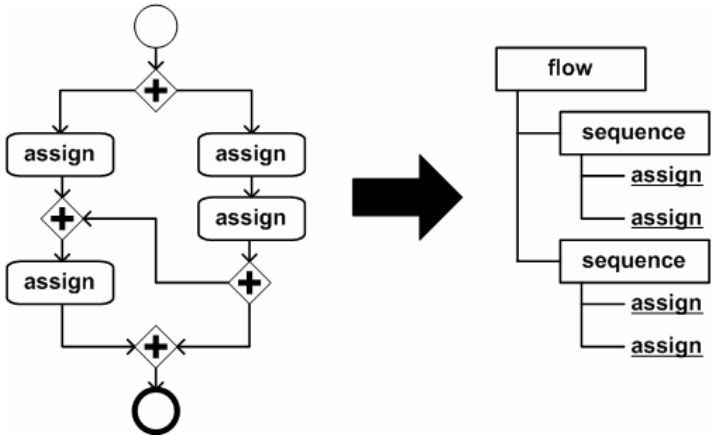


**Abbildung 52: Transformation nach Mendling et al.**

#### 2.6.1.2.4 Einschränkungen von BPMN

In [ReMe06b, S. 26] wird angesprochen, dass die Transformation nicht verlustfrei ist. Das betrifft hauptsächlich die Überlegungen zum Design eines Prozesses, die in BPEL-Code nicht mehr klar erkennbar sind. Für die Realisierung ihres Ansatzes schlagen sie eine Einschränkung der Sprachmächtigkeit von BPMN vor. Sie definieren dazu die Teilmengen *Structured BPMN* (siehe [MLZ05, S.6], [ReMe06a, S. 10]) und *Acyclic BPMN* (siehe [MLZ05, S.7], [ReMe06a, S. 10]), mit denen die Transformation zu BPEL zu bewerkstelligen ist.
Alternativ wird in [ReMe06b, S.26] eine Erweiterung von BPEL als Option in Betracht gezogen, was jedoch nicht im Detail vorgestellt wird. Vertiefende Literatur zu diesem Ansatz ist [ReMe06a], [ReMe06b], [MLZ05] und [MLZ06]. Die nachfolgende Abbildung aus [ReMe06b, S.24] zeigt tabellarisch die Anforderungen an ein BPD, um die beschriebenen Transformationsstrategien anwenden zu können:

| Transformation Strategies from BPMN to BPEL | Structured BPMN | Acyclic BPMN | All BPMN |
|---|---|---|---|
| Element-Preservation | - | + | - |
| Element-Minimization | - | + | - |
| Structure-Identification | + | - | - |
| Structure-Maximization | + | + | - |

**Abbildung 53: Einschränkungen der Transformationsstrategien**

#### 2.6.1.3 Ansatz von Ouyang, van der Aalst, Dumas, ter Hofstede, Lassen

Die bisher gezeigten Ansätze basieren auf der Identifikation von Mustern in einem BPD, die sich auf entsprechende Block-Strukturen in BPEL abbilden lassen. Die *Element-Preservation* Strategie aus 2.6.1.2 hat gezeigt, wie sich Strukturen, die nicht direkt durch ein BPEL Konstrukt darstellbar sind, mit einem BPEL-`<flow>` realisiert werden können. Allerdings wurde dafür die Sprachmächtigkeit von BPMN erheblich eingeschränkt. So wurde die Modellierung von beliebigen Abläufen (Arbitrary Cycles) schlicht ausgeschlossen. Auch in dem nun vorgestellten Ansatz werden Einschränkungen an ein BPD gemacht. Diese lassen jedoch erheblich mehr Spielraum als jeder andere Ansatz und bieten dadurch die zurzeit umfangreichste und vollständigste Möglichkeit einer Transformation von BPMN in BPEL.

##### 2.6.1.3.1 Anforderungen an die Transformation

In [ODHA07, S. 1 f.] werden drei Anforderungen formuliert, denen ein Verfahren zur Verbindung von BPMN und BPEL gerecht werden soll:

1. *Completeness*: Anwendbarkeit der Transformation auf ein BPMN-Modell mit beliebiger Topologie.
2. *Automation*: Fähigkeit des Transformationsverfahrens, Zielcode zu generieren, ohne menschliches Eingreifen für die Identifikation von Mustern (Patterns) zu erfordern.
3. *Readability*: Fähigkeit des Transformationsverfahrens, für Menschen verständlichen Zielcode zu generieren, um Anpassung, Testing und Debugging von diesem Code zu ermöglichen.

Bereits in [ODBH05] wurde ein Verfahren vorgestellt, das die ersten beiden Anforderungen für eine Teilmenge von BPMN erfüllen konnte. In diesem Verfahren (beschrieben in 2.6.1.3.6) wurde überaus viel Gebrauch von BPEL-`<eventHandlers>` (siehe 2.4.5.12) gemacht, um beliebige Topologien ohne menschliches Eingreifen übersetzen zu können. Allerdings war der generierte Code schlecht verständlich und der Kontrollfluss kaum noch nachvollziehbar (vgl. [ODHA06a, S.5]). Die Weiterentwicklung dieses Verfahrens (siehe [OADH06]) sollte eine bessere Lesbarkeit des generierten Codes erreichen und beinhaltete eine Identifikation von Mustern in einem BPD, die in entsprechende Block-Strukturen in BPEL übersetzt werden. Dabei wurde jedoch noch kein Gebrauch von `<link>`s in einem BPEL-`<flow>` gemacht:

*“However, since control links enable dead path elimination, such an extension, if not performed carefully, may hide errors such as deadlocks and livelocks in the source model during the translation, thus requiring verification technology for detection.”* [OADH06, S. 19]

Auf das in diesem Zitat angesprochene Problem wird in der Analyse der vorgestellten Ansätze in 4.1 eingegangen. Das Verfahren wurde mittlerweile dahingehend erweitert, dass auch von den graphartigen Strukturen in BPEL Gebrauch gemacht wird, allerdings nur wenn es keine andere Möglichkeit gibt (vgl. [ODHA07, S. 11 ff.]). Ebenso wurde die Identifikation von Mustern erweitert. Um der Anforderung der Vollständigkeit gerecht werden zu können, mussten bestimmte Einschränkungen an BPMN vorgenommen werden.

#### 2.6.1.3.2 Einschränkungen von BPMN

Die erste Einschränkung der Ausdruckskraft von BPMN ist durch die Definition des *Core BPD* gegeben. Dies ist eine echte Teilmenge der *Events*, *Activities*, *Gateways* und *Sequence Flow* Verbindungen, wie auch aus der folgenden Abbildung in [ODHA07, S.4] hervorgeht:

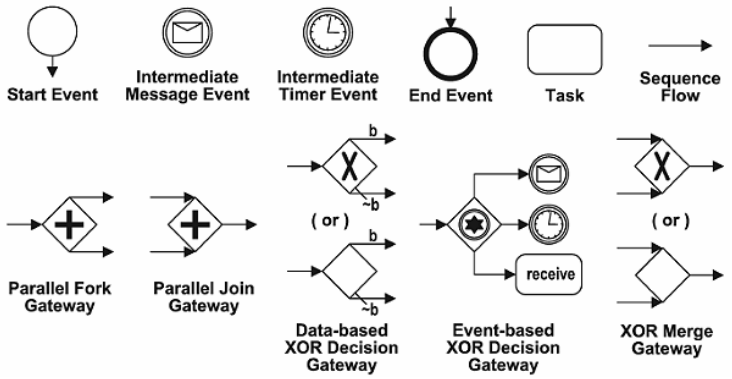


**Abbildung 54: Elemente in Core BPD**

Ausgehend von dieser Teilmenge werden durch die Definition des *Well-Formed Core BPD* weitere Einschränkungen vorgenommen. *Start-Events* dürfen nur eine ausgehende, *End-Events* nur eine eingehende *Sequence Flow* Verbindung haben. *Tasks* und *Intermediate Events* dürfen sowohl eine eingehende als auch eine ausgehende Sequence Flow Verbindung haben, jedoch nur genau eine. Nur *Gateways* dürfen mehr als eine eingehende, beziehungsweise ausgehende *Sequence Flow* Verbindung haben. Das in 2.2.3 dargestellte Konzept des *„Uncontrolled Flow“* wird damit vollständig von der Verwendung ausgeschlossen. Weitere Vorgaben sind die zwingende Verwendung eines *Default Sequence Flow* an einem *Exclusive Data-Based Gateway* und der Einsatz von *Intermediate Events* oder *Receiving Tasks* nach einem *Event-Based XOR Decision Gateway*. Darüber hinaus wird implizite Instanziierung und Terminierung von Prozessen (siehe [BPMN1.1, S. 37, 41]) in der Definition des *Well-Formed Core BPD* ausgeschlossen (vgl. [ODHA07, S. 5 f.]).
In [ODHA06b] und [ODHA07] werden die Ansätze und Entwicklungen von dieser Gruppe als kombiniertes Verfahren von vier einzelnen Schritten vorgestellt, die nun nach der Reihenfolge ihres Aufrufs im gesamten Algorithmus (siehe dazu [ODHA06b, S. 24]) vorgestellt werden.

#### 2.6.1.3.3 Well-Structured Pattern-Based Translation

Die *Well-Structured Pattern-Based Translation* identifiziert Muster der BPEL Konstrukte `<sequence>`, `<flow>` (im Sinne paralleler Aktivitäten ohne Synchronisation), `<switch>`, `<pick>` und `<while>` in einem BPD. Basis für die Identifikation ist eine formale Beschreibung von BPMN-Strukturen, die *Well-Defined Structured Patterns* (vgl. [ODHA07, S. 7 ff.]). Die nachfolgende Abbildung aus [ODHA07, S. 9] zeigt die Identifikation eines `<switch>` Musters, die Reduktion zu einer komplexen Komponente sowie dessen Abbildung in BPEL-Code:

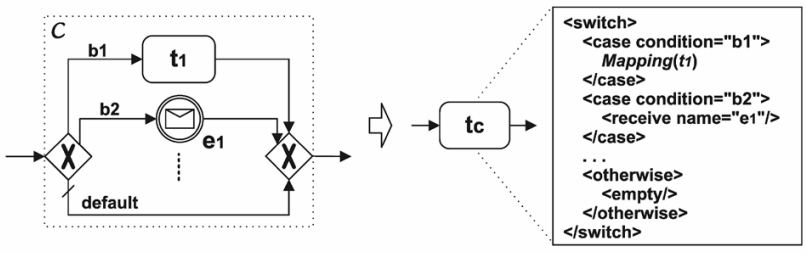


**Abbildung 55: Transformation in der Well-Structured Pattern-based Translation**

#### 2.6.1.3.4 Quasi-Structured Pattern-Based Translation

Die *Quasi-Structured Pattern-Based Translation* wird dazu eingesetzt, Muster zu erkennen, die beinahe (quasi) *Well-Structured* sind (siehe 2.6.1.3.3). Dies kommt erst dann zum Einsatz, wenn keine *Well-Structured* Muster mehr gefunden werden. Dabei werden in einer semantischen Äquivalenzumformung Teile des BPD umgeschrieben. Aus den (formal definierten) *Quasi-Structured Patterns* werden dadurch wiederum *Well-Structured Patterns* erzeugt (vgl. [ODHA07, S. 10 ff.]). Das auf diese Weise entstehende BPD wird mit dem vorigen Algorithmus (siehe 2.6.1.3.3) erneut überprüft und entsprechend auf komplexe Komponenten reduziert. Das folgende Bild aus [ODHA07, S. 11] zeigt die Transformation eines *Quasi* `<switch>` in einen *Well-Structured* `<switch>`:

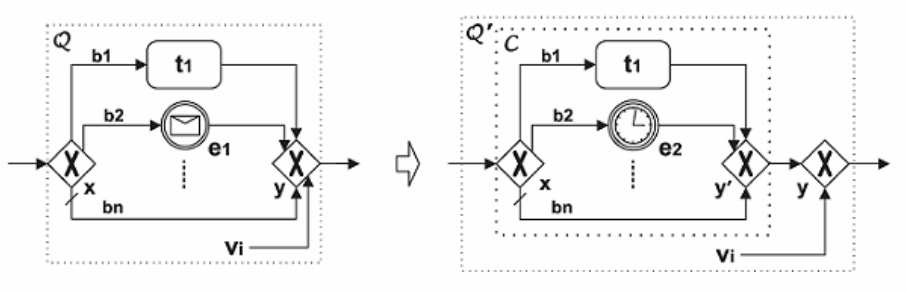


**Abbildung 56: Umformung eines Quasi-Structured Patterns**

#### 2.6.1.3.5 Generalised Flow-Pattern-Based Translation

Nachdem auch mit diesem Verfahren keine Muster mehr erkannt werden, wird die Generalised *Flow-Pattern-Based Translation* durchgeführt (vgl. [OADH06b, S. 11 ff.]). Dabei werden (Teil-) Strukturen in dem BPD in einen BPEL-`<flow>` und `<link>`s übersetzt. Diese Strukturen dürfen allerdings nur aus bereits identifizierten Mustern, einfachen *Tasks* und *Parallel Gateways* bestehen, außerdem darf eine solche Struktur keine Zyklen enthalten. Diese Einschränkungen sind notwendig, damit die Semantik der BPMN-Teilstruktur mit der Semantik des BPEL-Codes in Verbindung mit der Dead-Path-Elimination (siehe 2.4.2.1.4) übereinstimmt. In [ODHA07, S. 11] wird zudem erklärt, dass `<link>`s innerhalb der `<flow>` Aktivität nur dann verwendet werden, wenn es keine Möglichkeit zur Darstellung mit einer gegebenenfalls verschachtelten `<sequence>` Aktivität gibt. Dieses Vorgehen wird mit einer angeblich besseren Lesbarkeit des generierten BPEL-Codes begründet (vgl. [ODHA07, S. 12]).

#### 2.6.1.3.6 Event-Action Rule-Based Translation

Zuletzt sind von dem ursprünglichen BPD nur noch (reduzierte) Komponenten und beliebige Abläufe übrig, deren Bestandteile nicht mithilfe eines Musters erkannt werden können. Um dennoch eine vollständige Übersetzung in BPEL zu ermöglichen, wird die *Event-Action Rule-Based Translation* angewendet (vgl. [ODHA07, S. 23]). Diese Vollständigkeit wird allerdings teuer erkauft:

*"Using the event-action rule-based translation only as a last resort, reflects the desire to produce readable BPEL code."* [OADH06b, S. 25, hervorgehoben]

Eine vollständige Beschreibung dieses Ansatzes und seiner Entwicklung ist [ODBH05], [OADH06], [ODHA06a], [ODHA06b], [ODHA07] und [AaLa07] zu entnehmen. Darin sind auch Beispiele für die einzelnen Verfahren und ein Verweis auf die quelloffene Implementierung enthalten.

### 2.6.2 Von BPEL zu BPMN

In dieser Richtung der Verbindung beider Sprachen sind erheblich weniger Ergebnisse, Veröffentlichungen und Erkenntnisse als in der Gegenrichtung bekannt. Dies dürfte damit zusammenhängen, dass diese Verbindung als ein erheblich einfacheres Problem betrachtet wird (vgl. [Gao06, S. 1]).

#### 2.6.2.1 Ansatz von BPEL4WS, OASIS

Die Spezifikationen zu BPEL4WS liefern keine graphische Darstellung von Prozessen und keine Verbindung zu BPMN. Bereits in den Zielen der Spezifikation wird dies begründet: *"BPEL4WS is not concerned with the graphical representation of processes nor defines any particular design methodology for processes"* (siehe [LRT03, S. 2], 2.1.1). So ist es auch kaum verwunderlich, dass OASIS in der Standardisierung beschlossen hat, dass eine Verbindung zu BPMN *„out of scope"* des Standards [BPEL2.0] liegt (vgl. [OASI08]). Es bleibt abzuwarten, ob sich dies in folgenden Versionen ändert.

#### 2.6.2.2 Ansatz von Mendling, Recker, Lassen, Zdun

In [MLZ05] und in [ReMe06a] werden drei grundlegende Strategien beschrieben, um einen BPEL-Prozess in ein BPD zu überführen.

##### 2.6.2.2.1 Flattening

Zuerst wird ein Verfahren namens *Flattening* dargestellt. Dabei werden komplexe BPEL-Aktivitäten (Structured Activities) „eingeebnet". Eine solche BPEL-Aktivität wird in BPMN durch zwei *Gateways* repräsentiert, die die ursprünglich genesteten Aktivitäten umgeben. Ein `<flow>` wird durch *AND*, ein `<switch>` durch *XOR*, ein `<pick>` durch *XOR* und auch `<while>` wird durch *XOR Gateways* abgebildet (vgl. [MLZ05, S. 8 ff.]).
Der in [MLZ05, S. 8] angegebene *Flattening* Algorithmus in Pseudocode berücksichtigt allerdings nicht, dass ein BPEL-Prozess mehrere Prozess-startende Aktivitäten haben kann (vgl. [BPEL2.0, S. 34]). Weitaus bedeutsamer ist jedoch die unterschiedliche Semantik, die ein BPEL `<flow>` und ein BPMN *AND-Split / Join Gateway* haben: In BPEL wird die Semantik durch `<transitionConditions>` und `<joinConditions>` beschrieben, dazu kommt aber noch die Dead-Path-Elimination (siehe 2.4.2.1.4). Genau hier ist der Kern des Problems: Wenn ein Pfad in einem `<flow>` nicht weiter verfolgt werden kann, werden unerreichbare Aktivitäten übersprungen. Das Tokenkonzept (siehe 2.2.3) in BPMN funktioniert jedoch anders. Alle Untertokens, die an einem *AND-Split* entstehen, müssen an einem *AND-Join* wieder synchronisiert werden oder zumindest ein *End-Event* erreichen. Auf diese Problematik wird in 4.1 bei der kritischen Analyse der hier vorgestellten Verfahren näher eingegangen.

##### 2.6.2.2.2 Hierarchy-Preservation

Als zweites Verfahren wird *Hierarchy-Preservation* angeführt (vgl. [MLZ05, S. 9 ff.]). Dabei werden BPEL Structured Activities in BPMN *Sub-Processes* transformiert. Ein *Sub-Process* in BPMN (siehe 2.5.2.2) darf keine *Sequence Flow* Verbindungen haben, die als Quelle oder als Ziel eine innere Aktivität haben. Aus diesem Grund wird in [MLZ05, S. 7] die Teilmenge „Structured BPEL" definiert, das alle BPEL-Konstrukte (auch `<flow>`) einschließt, die lediglich keine `<link>`s enthalten dürfen. Damit bleibt der Block-Charakter der Aktivitäten in BPMN sichtbar. Zu dieser Strategie ist kein Pseudocode angegeben. Aber schon aus der verbalen Beschreibung geht hervor, dass die Semantik von BPEL Elementen nicht wie in der Spezifikation (damals: [BPEL1.1]) beschrieben in die Sematik von BPMN (siehe 2.2.3) umgesetzt wird, beziehungsweise dabei nicht ausreichend präzise ist. Ein Beispiel dazu ist `<terminate>`, das direkt in ein *End-Event* übersetzt wird. Bei Parallelität ist dies nicht korrekt, das passende Konstrukt wäre ein *Terminate-Event* (vgl. dazu [MLZ05, S. 9], 2.2.3 und 2.4.1.4).

#### 2.6.2.2.3 Hierarchy-Maximization

Unter der *Hierarchy-Maximization* Strategie wird in [MLZ05, S. 11] eine Kombination beider vorigen Strategien verstanden. Komplexe BPEL-Strukturen werden mit der *Hierarchy-Preservation* so weit wie möglich als entsprechende *Sub-Processes* in ihrer Form bewahrt. Hat eine solche Struktur `<link>`s, die die Grenzen der Struktur verlassen (zum Beispiel bei einer `<sequence>` innerhalb eines `<flow>`), so wird dies mit der *Flattening* Strategie aufgelöst. Die nachfolgende Abbildung aus [ReMe06b, S. 24] zeigt die Einschränkungen der Transformationsstrategien:

| Transformation Strategies from BPEL to BPMN | Structured BPEL | All BPEL |
|---|---|---|
| Flattening | + | + |
| Hierarchy-Preservation | + | - |
| Hierarchy-Maximization | + | + |

**Abbildung 57: Einschränkung der Transformationsstrategien**

### 2.6.3 Round Trip

Ein hybrides Modellierungswerkzeug, welches sowohl BPMN als auch BPEL unterstützt, wäre eine enorme Verbesserung für die Zusammenarbeit von Business Analysten und technischen Entwicklern. Ein solches Werkzeug muss in irgendeiner Form eine bidirektionale Transformation zwischen den beiden Sprachen ermöglichen. Unter der Berücksichtigung bestimmter Vorgaben und Beschränkungen müsste ein Werkzeug beide Metamodelle implementieren, also sowohl die entsprechenden Datenstrukturen vorsehen, als auch die Semantik, die Regeln und die Einschränkungen beider Sprachen mit einbeziehen. Dazu muss möglicherweise die Mächtigkeit von BPMN eingeschränkt werden, um nur solche BPMN-Prozesse zu erlauben, die in BPEL transformiert werden können. Eine notwendige Voraussetzung ist außerdem die exakte Definition der Transformation von BPMN zu BPEL (siehe 2.6.1) und von BPEL zu BPMN (siehe 2.6.2). In diesem Abschnitt wird lediglich die technische Problematik eines BPMN-BPEL Round Trips besprochen, eine ganzheitliche Perspektive dieses Round Trip Problems und der angrenzenden Bereiche wird in [Dubr07] dargestellt.

#### 2.6.3.1 Round Trip durch Modelltransformation

In [Gao06] wird ein Round Trip Verfahren beschrieben, das in einem proprietären Werkzeug zum Einsatz kommt. Die Transformation von BPMN zu BPEL findet mit mehrfachem „Process Rewrite“ statt, dabei werden BPMN-Strukturen umgeschrieben („Rewrite“), die nicht direkt in BPEL-Strukturen übersetzt werden können. Graphische Beispiele für die Transformation von BPEL in BPMN werden in [eCla06] zu den Prozessen „Purchase Order“ (siehe [BPEL1.1, S. 14 ff.]), „Auction“ (siehe [BPEL1.1, S. 101 ff.]) und „Loan Approval“ (siehe [BPEL1.1, S. 95 ff.]) vorgestellt. Dieser Ansatz wird hier nicht weiter behandelt, da das implementierte Verfahren nicht offen gelegt wurde und in dieser Arbeit daher nicht eingesetzt werden kann. In [ODHA07, S. 19] wird beschrieben, dass der Ansatz wohl auf dem in 2.6.1.1.2 dargestellten Verfahren basiert.

Die folgenden Abbildungen aus [Gao06, S. 1, 2] mit der nachfolgenden Interpretation eignen sich dennoch, um eine Kernproblematik des Round Trips aufzuzeigen:

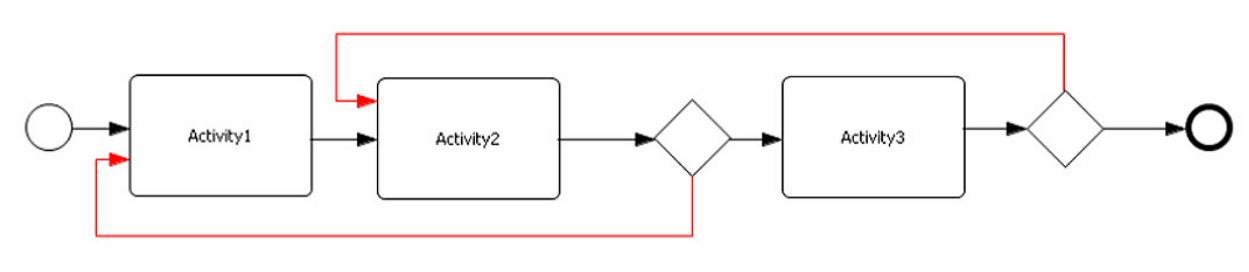


**Abbildung 58: BPD in BPMN mit Arbitrary Cycles**

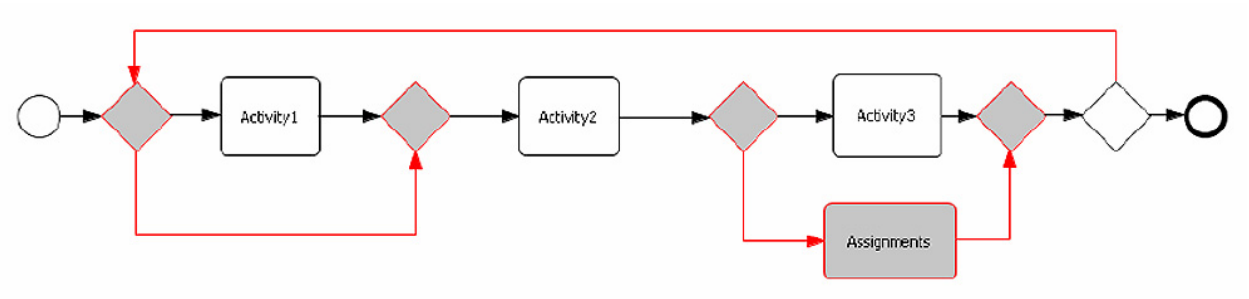


**Abbildung 59: BPD ohne Arbitrary Cycles nach Process Rewrite**

Um eine Transformation von BPMN nach BPEL zu ermöglichen, wird das BPD umgeschrieben und dabei werden die beliebigen Abläufe (Arbitrary Cycles) in BPMN durch wohlgeformte Strukturen in BPEL ersetzt. Dies wird in anderer Form auch in 2.6.1.3.4 durchgeführt. Doch wie ist dann der weitere Lebenszyklus? Wahrscheinlich werden von den technischen Entwicklern mehrere Tests durchgeführt, ein Debugging findet statt, technisch notwendige Anpassungen werden durchgeführt. Dies kann ein sowohl zeitaufwendiger als auch Kosten verursachender Ablauf sein. Wenn nun der Prozess in seiner Darstellung in BPMN geändert werden soll, was passiert dann? Wird der bereits bestehende BPEL-Prozess manuell angepasst und beide Modelle existieren nebeneinander? Müssen die BPEL-Artefakte neu generiert werden und alle (teuren) Anpassungen müssen von neuem durchgeführt werden? Oder kann das BPMN-Diagramm zuvor, durch eine Transformation von BPEL in BPMN, ein Update erfahren?

Eine Transformation von BPEL nach BPMN mag möglicherweise (dies wurde noch nicht bewiesen) keine fundamentalen Schwierigkeiten bereiten, wenn damit lediglich eine Reflexion des BPEL-Codes in der BPMN-Notation verstanden wird, wie zum Beispiel in 2.6.2.2 ansatzweise gezeigt. Das Ergebnis ist jedoch höchstwahrscheinlich nicht das ursprüngliche BPD, mit dem der Round Trip begonnen hat. Um ein „echtes“ Round Trip zu ermöglichen, müsste die Transformation von BPMN in BPEL invertierbar sein und bestenfalls dazu in der Lage sein, auch Änderungen „sinngemäß“ zu berücksichtigen. Dies wurde bisher von keinem der hier gezeigten Ansätze geleistet.

### 2.6.3.2 Round Trip durch gemeinsames Metamodell

In [DeSh06a] wird andeutungsweise beschrieben, dass eine zusätzliche Schicht zwischen BPMN und BPEL das Problem auseinander laufender Modelle beheben könnte. Die vielfältigen Schwierigkeiten, ein gemeinsames Metamodell zu definieren, werden dabei allerdings nicht einmal ansatzweise angedeutet. Das in 2.2.5 beschriebene BPDM könnte möglicherweise als gemeinsames Metamodell mit XMI als Austauschformat verwendet werden und dadurch eine solche Schicht hervorbringen. Es bleibt jedoch abzuwarten, ob und in welche Richtung die Entwicklung in diesem Bereich vorangetrieben wird. Bislang steht eine solche Schicht nicht zur Verfügung. Daher sind Modelltransformationen der gängige Weg, BPMN und BPEL zu verbinden.

# 3 Technische Grundlagen

In diesem Kapitel werden die technischen Möglichkeiten für die Realisierung des graphischen Editors diskutiert. An dieser Stelle ist anzumerken, dass für den graphischen Editor keine Zielplattform festgelegt ist. In diesem Sinne werden zuerst mehrere Frameworks zum Bau graphischer Editoren betrachtet (siehe 3.1). Diese Frameworks werden unter dem Gesichtspunkt analysiert, inwiefern sie den Bau einer Anwendung zur graphischen Modellierung von *Domain-Specific Languages* (Domänenspezifische Sprachen, DSL) unterstützen und ermöglichen. Anschließend (siehe 3.2) werden mehrere Open-Source Anwendungen vorgestellt, mit denen sich der geforderte graphische Editor durch Anpassung und Erweiterung realisieren lässt. Die gezeigten Realisierungsmöglichkeiten werden danach gegenübergestellt, wobei die zukünftige Erweiterbarkeit spielt eine wesentliche Rolle spielt (siehe 3.3). Anhand dieser Gegenüberstellung wird eines der vorgestellten Produkte für die Realisierung ausgewählt und diese Entscheidung sachlich und fachlich begründet.

## 3.1 Frameworks für graphische Editoren

Die in diesem Abschnitt vorgestellten Frameworks wurden bereits einer Vorauswahl unterzogen: Es wurden aus den auf dem Markt erhältlichen Frameworks diejenigen ausgewählt, die entweder breite Unterstützung in der Industrie, durch den Hersteller oder durch die Community erfahren. Darüber hinaus sollten sie für Forschungszwecke frei erhältlich sein.
Aus diesem Grund ist beispielsweise die Verwendung des kommerziellen Produkts *MetaEdit+* (siehe [S:MetaEdit+]) der Firma METACase zur Realisierung des graphischen Editors aus Kostengründen nicht möglich und wird daher hier nicht betrachtet.
Das ebenfalls kommerzielle Produkt *DSL Tools* (siehe [S:MSDSL]) der Firma Microsoft ist für Informatikstudenten der Universität Stuttgart für nicht-kommerzielle Anwendungen kostenfrei erhältlich. Dies war bisher allerdings nur durch das Abonnement des „Microsoft Developer Network – Academic Alliance“ möglich, welches für die Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart besteht. Seit Februar 2008 ist diese Plattform nun auch für Studenten von Universitäten ohne dieses Abonnement frei zugänglich (siehe [S:DreamSpark]). Die besonderen Umstände lassen die Verwendung im Rahmen der vorliegenden Arbeit daher zu.
Für die Eclipse Development Platform [S:Eclipse] existieren mehrere Open-Source Projekte, die den Bau graphischer Editoren unterstützen und ermöglichen, die wichtigsten stellen die Frameworks [S:EclipseEMF] und [S:EclipseGEF] dar, die in diesem Abschnitt vorgestellt werden. Darüber hinaus gibt es Bestrebungen, diese beiden Frameworks zu verbinden. Das Projekt [S:EclipseGMF] spielt an dieser Stelle eine wichtige Rolle, ein weiteres Projekt in diesem Bereich ist [S:GEMS]. Diese Eclipse-Frameworks kommen für die Realisierung des graphischen Editors ebenso wie [S:MSDSL] in Frage.
Während sich die [S:Eclipse]-basierten Frameworks und [S:MSDSL] in ihrer Architektur und der Vorgehensweise bei der Entwicklung einer Anwendung in gewisser Weise ähneln (siehe 3.3), wird in [S:Oryx] ein grundlegend anderer Ansatz verfolgt. Dieses Framework bietet die Möglichkeit, Web-basierte Modellierungswerkzeuge zu erstellen. Dabei werden Technologien wie SVG (Scalable Vector Graphics), RDF (Resource Description Framework) und JSON (JavaScript Object Notation), sowie die asynchrone Verwendung von JavaScript und XML (AJAX) eingesetzt. Ein weiterer wesentlicher Unterschied ist die Realisierung als generisches Framework, welches in der Erstellungsphase des Editors ohne Modelltransformationen arbeitet.

Bevor nun das Framework [S:MSDSL], die zusammenhängenden [S:Eclipse]-Frameworks und das Web-basierte Framework [S:Oryx] vorgestellt werden, ist es angebracht, mehrere Begriffe zu definieren.

Der erste Schritt bei der Erstellung eines graphischen Modellierungswerkzeugs für eine *Domain-Specific Language* (DSL) ist die Festlegung, beziehungsweise die Implementierung des zugrunde liegenden *Metamodells*.
Ein *Metamodell* definiert die in der DSL zur Verfügung stehenden Konstrukte und deren Relationen untereinander. Es umfasst auch eine präzise Beschreibung der Eigenschaften und des Verhaltens von Instanzen dieses *Metamodells*; mit anderen Worten von den *Modellen* einer DSL (vgl. [LeRo00, S. 62, 121]). Das *Metamodell* kann auf verschiedene Art und Weise beschrieben werden. Im Fall von [BPEL2.0] liegt das *Metamodell* in Form einer Spezifikation in Prosatext mit Codebeispielen und einer Beschreibung des *Datenmodells* in Form von *XML Schema Definitions* (XSD) vor. Der Text oder das Datenmodell für sich allein genommen würde das *Metamodell* nicht ausreichend präzise und nur unvollständig beschreiben. Beispielsweise lässt sich in einem XML Schema nur die Struktur eines `<flow>` Konstrukts (siehe 2.4.2.1) beschreiben, nicht jedoch die Semantik der `<link>`s (siehe 2.4.2.1.1) für gerichtete, azyklische Graphen.

### 3.1.1 Microsoft DSL Tools

*DSL Tools* von Microsoft ist ein im Jahre 2006 in der Version 1.0 veröffentlichtes Framework zum Bau von graphischen Editoren für *Domain Specific Languages (DSL)* und seitdem integrierter Bestandteil der Entwicklungsumgebung *Microsoft Visual Studio* (vgl. [S:MSVStudio], [S:DreamSpark]). Mit diesem Framework wird eine Strategie verfolgt, die von Microsoft als *Software Factories* bezeichnet wird. Diese *Factories* sollen im Sinne von *Model Driven Development (MDD)* eine kostengünstige, schnelle und zuverlässige Art der Softwareentwicklung ermöglichen, bei der das der Anwendung zugrunde liegende Modell im Vordergrund steht. Auf Basis von XML sowie proprietären Formaten werden Modelltransformationen, Codegenerierung und andere Formen der Automatisierung ermöglicht. Eine *Software Factory* wird als Entwicklungsumgebung verstanden, mit der eine bestimmte Art von Software schnell und effizient erstellbar ist (vgl. [S:MSDSL]).
In [Kent08] wird gezeigt, wie man mit [S:MSDSL] graphische Modellierungswerkzeuge für DSLs erstellen kann. Dabei werden zwei sehr unterschiedliche Beispiele gezeigt: Die erste DSL ist eine Sprache zur Modellierung und (Code-)Generierung von Wizards[8]. Das zweite Beispiel ist ein Modellierungswerkzeug für Aktivitätsdiagramme ohne weitere Codegenerierung. In [CJK07] werden weitere Beispiele dargestellt, dieses Buch stellt zudem das erste (und derzeit einzige) Handbuch zur Verwendung der *DSL Tools* dar.

[8] Wizards sind Hilfsprogramme zum Abfragen von Informationen und Ansteuern von Anwendungen oder Werkzeugen mit diesen Informationen.

### 3.1.1.1 Ablauf der Entwicklung mit DSL Tools

Der Entwicklungsprozess ist in mehrere Teile gegliedert bei denen verschiedene Tools, beziehungsweise Codegeneratoren dieses Frameworks zum Einsatz kommen. In [S:MSDSL] wird der Ablauf mit der folgenden Abbildung dargestellt:

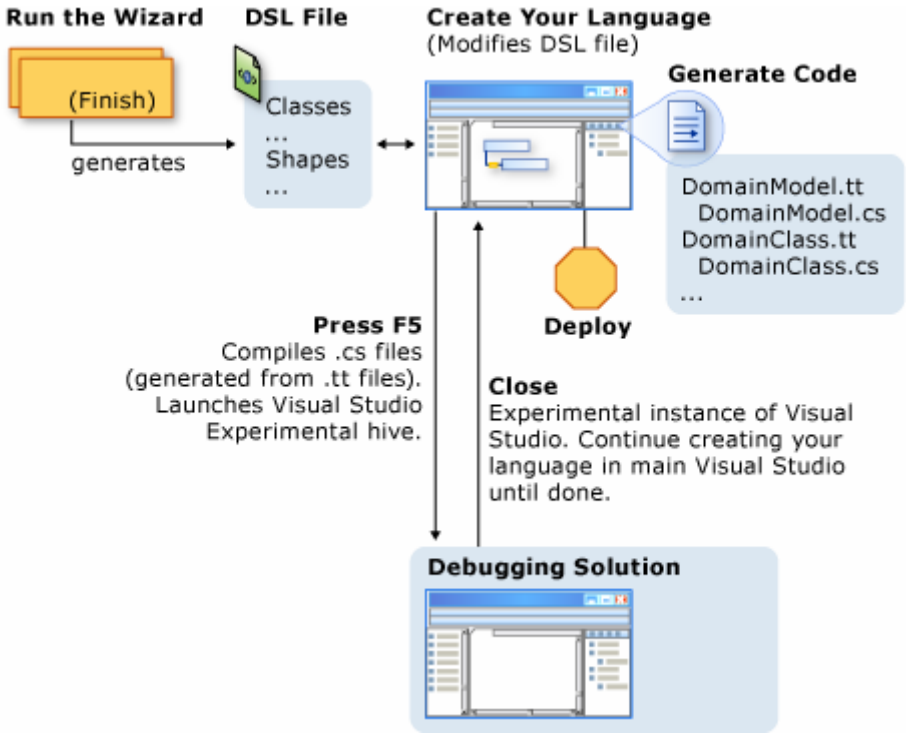


**Abbildung 60: Entwicklung mit DSL Tools**

Diese Abbildung vermittelt bereits eine ungefähre Vorstellung des Ablaufes, dennoch fehlt ihr eine präzise Beschreibung. Die in dieser Arbeit vorgestellte Notation *BPMN* (siehe 2.5) kann an dieser Stelle dazu eingesetzt werden, die Abläufe bei der Entwicklung mit [S:MSDSL] und die dabei erzeugten Artefakte genauer zu beschreiben. Die folgende Abbildung wurde mit dem auf [S:EclipseBPMN] (siehe 3.2.1) basierenden Werkzeug *Intalio BPMN Designer* erstellt (siehe [S:Intalio]) und zeigt den Ablauf der Entwicklung mit *DSL Tools* in BPMN-Notation, der anschließend erläutert wird:

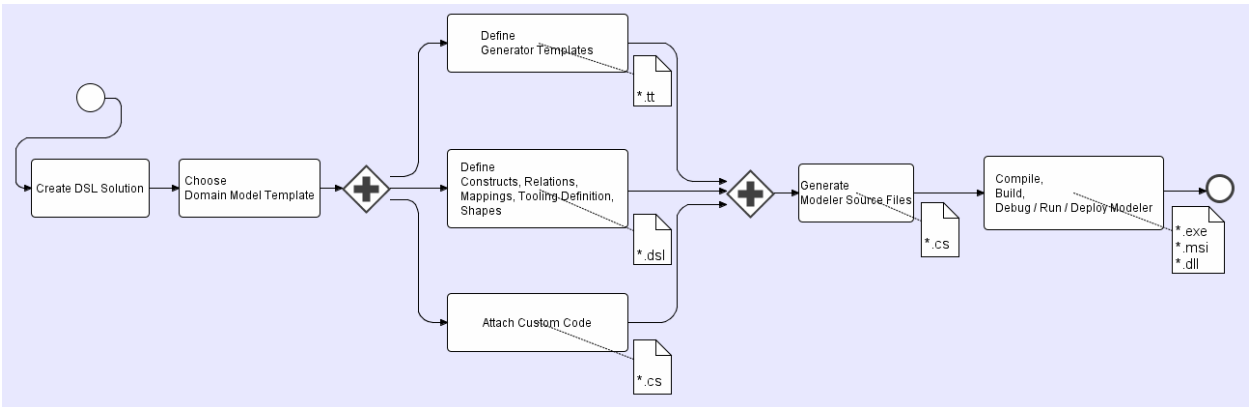


**Abbildung 61: Abläufe und Modelle in der Entwicklung mit DSL Tools**

Zu Beginn der Entwicklung wird mit *Visual Studio* ein neues *DSL Tools* Projekt (hier: DSL Solution) erstellt. Dabei kommt ein Wizard zum Einsatz, der verschiedene Vorlagen (*Domain Model Templates*, siehe unten) für das abzubildende Datenmodell bereithält (siehe 3.1.1.2). Nachdem ein solches Template ausgewählt wurde, findet der Großteil der Entwicklungsarbeit, wie die Definition der Konstrukte, Relationen, Mappings, Toolpallettendefiniton, Shapes etc. mit dem *Domain Model Designer* statt (siehe 3.1.1.3). Das so erstellte *Domain Model* wird dann mit *Text Templates* (siehe 3.1.1.4), die den Codegenerator steuern, zum Source Code des eigentlichen graphischen Modellierungswerkzeugs transformiert (siehe 3.1.1.5). Dieser Source Code wird durch eigenen Code erweitert und angepasst (siehe 3.1.1.6). Das entstandene graphische Modellierungswerkzeug kann dann gebaut, ausgeführt und deployed werden (siehe 3.1.1.7).

#### 3.1.1.2 Domain Model Templates

Zuerst wird das zugrunde liegende Datenmodell definiert, welches in den *DSL Tools* als *Domain Model* bezeichnet wird. Dazu wird ein graphisches Modellierungstool verwendet, der *Domain Model Designer*. Um bei diesem *Designer* nicht von Null anfangen zu müssen, wird beim Erstellen eines neuen *DSL Tools* Projekts ein Wizard gestartet. Dieser Wizard bietet mehrere *Designer Templates* an; diese Vorlagen sind für einen schnelleren Einstieg in das Design der eigenen DSL gedacht und beinhalten vordefinierte Konstrukte und deren graphische Darstellung für bestimmte Anwendungsbereiche sowie vordefinierte Codegenerierungs-templates. Es gibt folgende *Designer Templates*:

*Task Flow Diagrams:* Dieses Template bietet vordefinierte Konstrukte für die Modellierung von Geschäftsprozessen und Zustandsdiagrammen. Die wesentlichen Elemente sind Aktivitäten und Verbindungen zwischen diesen Elementen. Für die graphische Darstellung ist außer den Standardformen, wie abgerundete Rechtecke, Ovale und Kreise auch die Möglichkeit gegeben, direkt Grafikdateien wie JPEG, GIF oder BMP zu verwenden (vgl. [Kent08]). Die Darstellung von *Swimlanes* (siehe 2.5.5.4) ist in diesem Template ebenfalls bereits integriert. In [S:MSDSL] wird beschrieben, dass dieses Template UML-Aktivitätsdiagrammen sehr ähnelt, diese Spezifikation jedoch nicht vollständig umsetzt und auch eigene Erweiterungen eingebracht hat. In [Kent08] wird darauf hingewiesen, dass das Verschachteln (Nesten) von Aktivitäten allerdings schwierig umzusetzen ist.

*Class Diagrams:* Dieses Template bietet die Grundfunktionalität für die Modellierung von klassendiagrammähnlichen Strukturen. Dabei sind Klassen, Interfaces, Assoziationen und Relationen wie Generalisierung und Implementierung sowie Vererbung im Vorfeld definiert. In [S:MSDSL] wird beschrieben, dass dieses Template UML Klassendiagrammen sehr ähnelt, jedoch auch diese Spezifikation nicht vollständig beschreibt.

*Component Diagrams:* Dieses Template wird für die Modellierung von Software-Komponenten in verteilten Systemen verwendet. Dazu werden *Ports* zur Verfügung gestellt, die an Komponenten angebracht und untereinander verbunden werden können.

In *Minimal Language* sind lediglich eine Klasse und eine Relation definiert. Sie wird eingesetzt, um eine DSL von Grund auf zu modellieren.

#### 3.1.1.3 Domain Model Designer

Nachdem ein Template ausgewählt und ein *Domain Model* (.dsl) generiert wurde, beginnt die Modellierung des Datenmodells mit dem *Domain Model Designer.* In Ausnahmefällen kann auch die erzeugte XML-Modellbeschreibung direkt mit einem Editor bearbeitet werden, dies erfordert aber ein profundes Verständnis des *Domain Model* Formats. Die in [S:MSDSL] bereitgestellte Dokumentation kann dies zum Zeitpunkt der Erstellung der Arbeit jedoch nicht in dem nötigen Umfang leisten. An dieser Stelle ist die Anmerkung wichtig, dass sich die Online-Dokumentation während der Erstellung dieser Arbeit gerade im Aufbau befunden hat.

Eine Unterstützung des OMG-Standards *Meta-Object Facility (MOF)* oder *XML Metadata Interchange (XMI)* ist in dem *Domain Model Designer* nicht gegeben (vgl. [S:MSDSL]). In [Ozgu07, S. 28 ff.] wird ein Ansatz vorgestellt, dieses Format in ein EMF Ecore Modell (siehe 3.1.2.1) zu übersetzen, sowie einen Import von einem Ecore Modell zu ermöglichen. Eine praktische Umsetzung, die den Grad des Informationsverlusts zeigen würde, wurde allerdings nicht demonstriert. Eine derartige Kompatibilität wird daher bezweifelt.
Der *Domain Model Designer* wird unter anderem für das Mapping verwendet, also um graphische Elemente für die spätere Darstellung an die Konstrukte der DSL zu binden. Diese können auch direkt in dem *Domain Model Designer* erstellt werden. Ebenso werden in diesem Designer die Toolpallette, Shapes und Menüs definiert und an die zu modellierenden Konstrukte angebunden. Des Weiteren können zu den Konstrukten so genannte *Decorators* definiert werden. Das sind kleine Piktogramme, die in Abhängigkeit der Attributswerte an bestimmten Stellen innerhalb der Shape angezeigt werden. All diese Funktionen sind eigentlich einzelne Tools, daher der Name *DSL Tools*. Dadurch, dass sie in den *Domain Model Designer* integriert sind und somit unter einer einheitlichen Oberfläche angeboten werden, ist dies nicht mehr offensichtlich.
Die folgende Abbildung aus [Ozgu07, S. 36] zeigt die Modellierung mit dem *Domain Model Designer* unter Verwendung des *Class Diagrams* Template (links im Bild) und graphischen Mappings (rechts im Bild). In dieser Abbildung nicht zu sehen ist die Integration in die *Visual Studio* Entwicklungsumgebung.

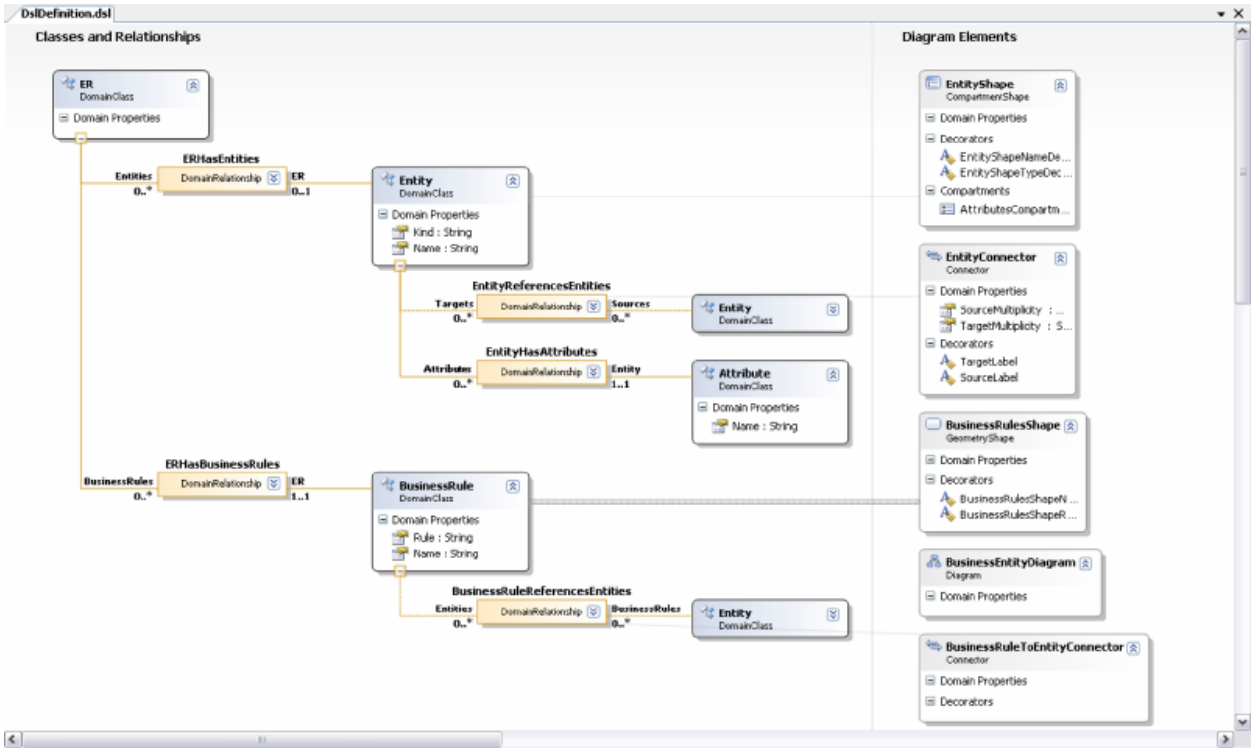


**Abbildung 62: Graphische Modellierung mit dem Domain Model Designer**

### 3.1.1.4 Text Templates

Die vorhergehende Auswahl des *Domain Model Templates* in 3.1.1.2 bewirkt außer der Generierung der Datenmodelldatei auch die Generierung vordefinierter Templates, die wiederum die Codegenerierung aus dem *Domain Model* steuern. Sie werden in [S:MSDSL] als *Text Templates* (.tt) bezeichnet. In diesen Templates wird sowohl für das ganze Modell als auch für die einzelnen Konstrukte in dem Modell festgelegt, wie diese Elemente in den Code für den Editor übersetzt werden sollen. Wenn die Möglichkeiten des gewählten *Domain Model*

*Templates* nicht ausreichen, so müssen diese *Text Templates* angepasst oder neu erstellt werden. Dazu kann entweder die Programmiersprache *Visual Basic* oder *Visual C#* verwendet werden. Die Zielsprache in den vordefinierten *Text Templates* ist *Visual C#*, die Verwendung einer anderen Zielsprache für den Bau einer *Software Factory* ist zwar möglich, wird aber wegen mangelnder Unterstützung beim Debugging nicht empfohlen (vgl. [S:MSDSL]). Eine Referenz für die Erstellung, die Funktionsweise und die Einsatzmöglichkeiten der *Text Templates* wird in [S:MSDSL] und in [CJK07] geliefert.

#### 3.1.1.5 Generierung der Modeler Source Files

Das in [S:MSDSL] unter der Funktion *Transform All Templates* zugänglich gemachte Tool führt die Generierung der Source Dateien des graphischen Modellierungswerkzeugs durch. Dabei wird das *Domain Model* mithilfe der *Text Templates* transformiert. In diesem Schritt wird auch Code für die Persistenzunterstützung des Modells und für die Integration in [S:MSVStudio] eingefügt. In [S:MSDSL] wird empfohlen, diese Transformation nach jeder Änderung des *Domain Model* durchzuführen, um eventuell auftretende Probleme in einem frühen Stadium zu erkennen und diese somit isoliert behandeln zu können.

#### 3.1.1.6 Custom Code

In [S:MSDSL] und auch in [Kent08] wird darauf hingewiesen, dass man mit der Oberfläche des *Domain Model Designer* nicht jedes Szenario abbilden kann, es ist jedoch ohne weiteres möglich, das *Domain Model* sowie den daraus generierten Code mit eigenem Code (Custom Code) zu erweitern. In [S:MSDSL] sind die gängigen Vorgehensweisen dazu beschrieben.
Die wichtigste Möglichkeit ist die Anpassung der *Modeler Source Files*, die durch die *Text Templates* generiert werden (siehe 3.1.1.5). Diese Codeanpassungen können auch die Verwendung von Interfaces und von Vererbung beinhalten. Um bei einer erneuten Generierung die eigenen Anpassungen und Erweiterungen vor der Überschreibung zu schützen, werden sie analog zu EMF (siehe 3.1.2.1) mit einem entsprechenden *Preservation Tag* versehen. (vgl. [S:MSDSL]).
Weiterhin kann an das ganze *Domain Model* oder an einzelne Konstrukte daraus Code angebracht werden, beispielsweise um Funktionen zur Validierung eines Modells zu integrieren. Diese Erweiterungen werden, durch die Entwicklungsumgebung *Visual Studio,* getrennt von dem Modell und den implementierenden Klassen, in so genannten *Partial Classes* verwaltet. Als Programmiersprache kommt *Visual C#* zum Einsatz, daher die Dateiendung *.cs* für C Sharp. Diese Codeerweiterungen werden von dem eigentlichen graphischen Modellierungswerkzeug beim Auftreten von bestimmten Ereignissen (Events) ausgeführt. Dazu können vordefinierte Events wie das Laden und Speichern eines Modells oder benutzerdefinierte Events wie der Rechtsklick auf ein bestimmtes Element oder einen *Decorator* verwendet werden. Der generierte Code und der eigene Code werden zur Compile-Zeit des eigentlichen Modellierungswerkzeugs gemerged (vgl. [S:MSDSL]).
Um in dem graphischen Modellierungswerkzeug das (graphisch) erstellte Modell in Code zu transformieren, werden mehrere Möglichkeiten angeboten. Das Modell kann entweder mithilfe von *Text Templates* (siehe 3.1.1.4) transformiert werden, mit einem DOM Parser wie [S:CodeDom] oder [S:CodeSmith] durchlaufen werden oder eine eigene Implementierung führt die Codegenerierung aus dem Modell durch. Die Verwendung von *Text Templates* bietet den enormen Vorteil, dass diese auch zur Laufzeit, zum Beispiel beim Debugging, angepasst werden können (vgl. [S:MSDSL]).

#### 3.1.1.7 Debugging und Deployment

Das Debugging von dem generierten graphischen Modellierungswerkzeugs erfolgt durch den Start einer weiteren Instanz von Visual Studio mit dem automatisch gebauten und geladenen Plugin. Die mit [S:MSDSL] erzeugten Anwendungen können entweder als Plugin gebaut und in Visual Studio integriert werden, oder als Stand-alone Anwendung auf Microsoft Windows installiert werden. Im zweiten Fall wird dazu ein Microsoft Installer-Paket (.msi) erzeugt.

### 3.1.2 Eclipse Development Platform

Was als Java Integrated Development Environment bekannt geworden ist, bietet mittlerweile Unterstützung für den Bau, das Deployment und Management von Software über ihren gesamten Lebenszyklus (vgl. [S:Eclipse, Newcomers FAQ]).
Historisch ist die Eclipse Development Platform, hier kurz *Eclipse*, aus der *Visual Age* Entwicklungsumgebung der Firma IBM durch die Freigabe von Quellcode hervorgegangen (vgl. [Daum05, S. 9]). Unterdessen haben sich die rechtlichen Rahmenbedingungen geklärt, so tritt die Open-Source Community um Eclipse nach außen unter dem Namen *Eclipse Foundation* auf. Die eigentliche Entwicklungsarbeit wird von den so genannten *Commiters* ausgeführt, und die dabei entstehenden Software-Projekte werden unter der *Eclipse Public License* [EPL1.1] veröffentlicht. Diese Lizenz regelt unter anderem, wie die Software für proprietäre Produkte verwendet werden darf. Zum Beispiel ist der *ActiveBPEL Designer* der Firma ActiveEndpoints (siehe [S:ActiveBPEL]) eine Eclipse-basierte Anwendung, jedoch nicht Open-Source.

Die Java-basierte Eclipse IDE hat im Laufe der Zeit zahlreiche Änderungen in der Architektur erfahren. So wurde etwa mit der Version 3.0 die gesamte Plattform neu organisiert. Wesentlich dabei war, dass mit dieser Umstellung die Einsatzmöglichkeiten dieser Plattform enorm erweitert wurden. Die eigentlichen Programmfunktionen sind in Plugins[9] gekapselt, die von dem Eclipse-Kern geladen werden. Plugins sind Komponenten, die den Eclipse-Kern oder auch andere Plugins mit der von ihnen angebotenen Funktionalität erweitern. Dazu wird das Plugin an bestimmte Schnittstellen angeschlossen (*Extensions*). Es kann auch eigene Schnittstellen definieren, an denen es selbst mit zusätzlicher Funktionalität erweitert werden kann (*ExtensionPoints*), siehe auch [S:Extensions]. Eine Eclipse-basierte Anwendung besteht daher aus einer Menge von Plugins. Deshalb bestehen auch die hier vorgestellten Frameworks aus einem oder mehreren Plugins: „Everything is a Plugin."

Die zahlreichen Eclipse-Projekte, die zum Teil aufeinander aufbauen, werden regelmäßig zu einem Release zusammengefasst; zurzeit ist die Version 3.3 (*Europa*) aktuell. Für die Entwicklung von eigenen Plugins empfiehlt es sich, den jeweils aktuellen Versionsstand der verwendeten Komponenten, sprich Plugins, einzusetzen, da diese Releases nur jährlich stattfinden. Das nächste Release erscheint voraussichtlich im Juli 2008 mit dem Namen *Ganymede*, ebenso wie die Vorgänger *Callisto* und *Europa* nach den Galiläischen Monden des Jupiters benannt.

[9] Englisch: (to) plug – anschließen.

#### 3.1.2.1 EMF

Das *Eclipse Modeling Framework* (EMF) ist eine Eclipse-basierte Plattform zur Modellierung und Weiterverarbeitung von strukturierten Datenmodellen. Wesentlich hervorzuheben sind die vielfältigen Möglichkeiten, das interne Datenmodell *Ecore* aufzubauen und transformieren zu können (vgl. [S:EclipseEMF]).
Ursprünglich wurde EMF als Implementierung der Meta Object Facility (MOF) der Object Management Group (OMG) begonnen. Mittlerweile stellt es nach [MDGW04, S. 4] durch eine einfachere Benutzbarkeit sogar eine Verbesserung von MOF 2.0 dar. Laut [S:EclipseEMF] sind die Unterschiede eher gering und liegen zumeist im Naming. Das Framework ist der Model Driven Architecture (MDA) zuzuordnen, die im Moment von der OMG standardisiert wird. Der Ansatz bei MDA liegt hauptsächlich darin, durch Fokussierung auf das (Meta-)Modell den gesamten Software-Lebenszyklus zu steuern. Das Metamodell wird unter der Verwendung von *Mappings* (Abbildungen) dazu benutzt, Software-Artefakte zu generieren, die das eigentliche System implementieren. Dazu können unter Umständen mehrere Transformationsschritte notwendig sein (vgl. [MDGW04, S. 4]).
Die eigentliche Betrachtung ist in die drei für EMF wesentlichen Use-Cases unterteilt: Wie kommt das Datenmodell in das Framework hinein, wie kann es dort bearbeitet werden und welche Möglichkeiten werden für die Weiterverarbeitung (Export) geboten?
Für den initialen Import werden die Formate EMOF (Essential MOF, siehe [MOF2.0]), Ecore (das EMF-eigene Format), Rational Rose Class Diagram, XSD (XML Schema Definitions), WSDL (Web Service Description Language), Annotated Java Classes (die EMF auch generieren kann) sowie XMI (XML Metadata Interchange) unterstützt.
Nach dem Import liegen zwei Dateien vor, eine *.ecore* Datei für die interne Repräsentation in XMI, sowie eine *.genmodel* Datei zur Steuerung der Codegenerierung. Änderungen an der in der *.ecore* Datei enthaltenen Modellbeschreibung werden automatisch in die *.genmodel* Datei übernommen. Diese können in einer Baumstruktur komfortabel bearbeitet werden (siehe Abbildung 63: Datenmodellbearbeitung in EMF).
EMF bietet mehrere vordefinierte Arten der Codegenerierung: Die wohl wichtigste ist die Generierung des *Model Code*. Dabei wird das Datenmodell in Java-Klassen transformiert, die das Datenmodell für Java-Anwendungen (wie zum Beispiel Plugins) nutzbar machen. Man muss beachten, dass in den erzeugten Java-Klassen auch Klassen aus dem EMF Framework eingebunden sind, die für die Laufzeitunterstützung (zum Beispiel für die Modellserialisierung oder Modellvalidierung) benötigt werden. Eine genaue Beschreibung der erzeugten Paket-Struktur wird in [Schu07, S.32] als UML-Klassendiagramm dargestellt.
Die weiteren Arten der Codegenerierung sind als Zusatz zum *Model Code* gedacht. Mit *Test Code* werden *JUnit Test Cases* (siehe [S:JUnit]) zum *Model Code* erzeugt. Mit *Edit Code* werden zusätzliche Klassen erzeugt, die für den Einsatz in Eclipse nützlich sind, wie zum Beispiel *ItemProviders*. Der *Editor Code* dient als Vorbereitung für die Implementierung als graphisches Modellierungswerkzeug. Um einen voll einsatzfähigen graphischen Editor zu erhalten ist jedoch noch viel Anpassungs- und Implementierungsauswand nötig (vgl. [Kapl06], [Schu07]). Mit EMF alleine kann dies nur bedingt bewerkstelligt werden, in der Regel wird zusätzlich das Framework [S:EclipseGEF] dazu eingesetzt.
Eine Alternative zu der vordefinierten Codegenerierung wird in [MDGW04, S. 79 ff.] umfassend beschrieben. Dort wird gezeigt, wie die *Java Emitter Templates* (JET), die in EMF zur Codegenerierung eingesetzt werden, angepasst werden können. [ScSc05] gehen noch weiter und demonstrieren anschaulich, wie sich mithilfe von JET beispielsweise HTML-Seiten aus dem Ecore Modell generieren lassen. Die Möglichkeiten an dieser Stelle sind also lediglich auf die Zeit beschränkt, die zur Entwicklung der Templates zur Verfügung steht.

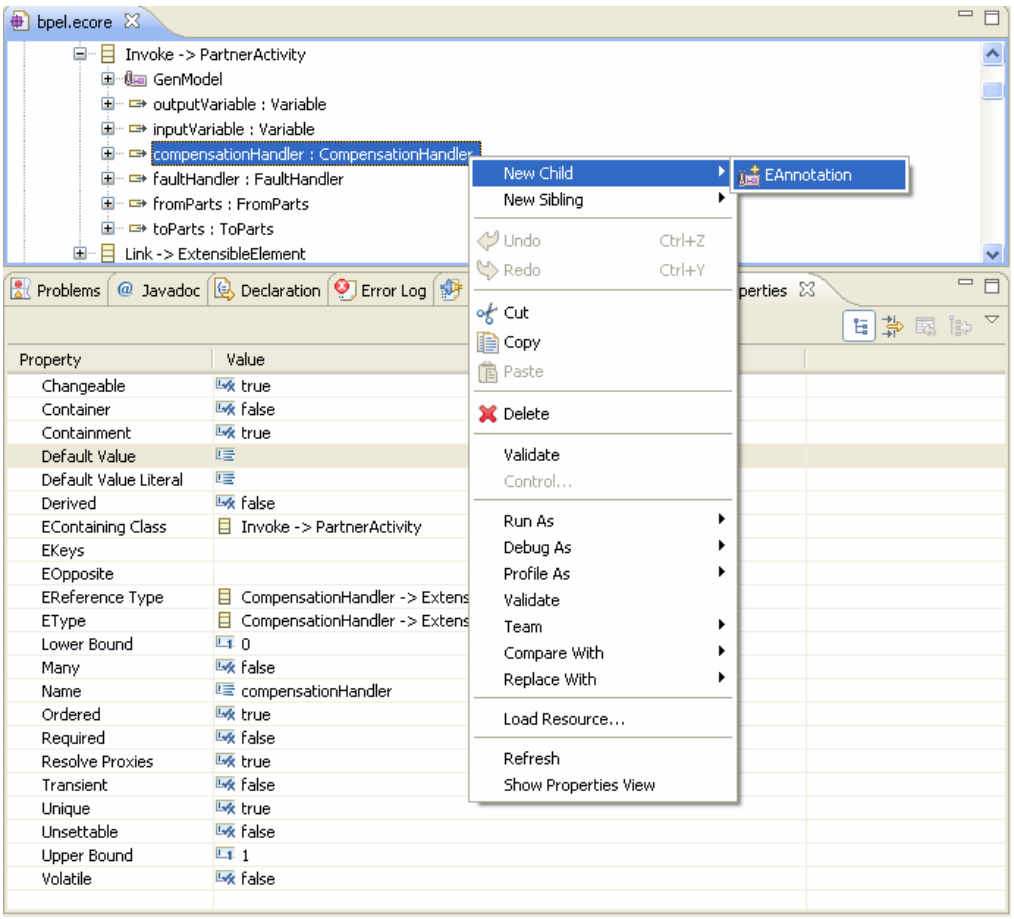


**Abbildung 63: Datenmodellbearbeitung in EMF**

### 3.1.2.2 GEF

Das zuvor vorgestellte Framework EMF liefert ein strukturiertes Datenmodell. Instanzen dieses Modells können mithilfe des Graphical Editing Framework (GEF) graphisch dargestellt und bearbeitet werden. Wenn beide Frameworks zusammen eingesetzt werden, lässt sich damit das *Model-View-Controler (MVC)* Entwurfsmuster realisieren. Das Modell wird von EMF bereitgestellt, die Darstellung, sowie der Controler werden in GEF umgesetzt. Eine saubere Trennung, sowie eine rein generische Verbindung des Modells mit der Darstellung ist allerdings überaus komplex und nur unter erheblichem Aufwand realisierbar (vgl. [Schu07, S. 29 ff., 68]).

Mit GEF können graphische Modellierungswerkzeuge für nahezu jedes Modell gebaut werden. Ein Beispiel für eine Anwendung die mit den Frameworks EMF und GEF erstellt wurde, wird in 3.2.3 vorgestellt. Mit einem solchen Werkzeug können einfache Bearbeitungsschritte wie das Ändern von Elementattributen und auch erweiterte Funktionalität wie das Verbinden oder Verschachteln (Nesten) von Elementen ermöglicht werden. Gängige Funktionen wie Drag & Drop, Copy & Paste, Undo / Redo, (Kontext-)Menüs, Toolbars, Editorpaletten und Zooming werden ebenso von dem Framework zur Verfügung gestellt (vgl. [MDGW04, S 88]). Die Anbindung dieser Funktionen an das Modell hat allerdings einen nicht zu vernachlässigenden Aufwand (vgl. [Kapl06, S. 37 ff.]).

Grundlegende Elemente in der Entwicklung einer GEF-basierten Anwendung sind die so genannten *EditParts*. Diese Elemente definieren wie die einzelnen Konstrukte einer Modellinstanz graphisch abgebildet werden sollen. Daher wird für jedes abzubildende Element

des Metamodells ein solches Artefakt benötigt. Man unterscheidet zwischen drei Arten von *EditParts*. In [Kapl06] und [Schu07] (siehe auch 3.2.4) wurden *GraphicalEditParts* für die Abbildung von BPEL-Aktivitäten, *ConnectionEditParts* für die Abbildung von `<link>`s in `<flow>` Aktivitäten und *TreeEditParts* für weitere Konstrukte wie `PartnerLinks` und `Variables` verwendet (vgl. [Schu07, S. 28]).

In [Daum05, S. 350] werden die Frameworks zur graphischen Darstellung, denen sich das Graphical Editing Frameworks (GEF) wiederum bedient, prägnant beschrieben: „Das GEF setzt nicht direkt auf dem *SWT* (Anm.: Standard Widget Toolkit, Java Framework für graphische Benutzeroberflächen) auf, sondern auf einer höheren Grafikschicht – der *Draw2D*-Schicht. Dessen Bausteine realisieren einen leichtgewichtigen Überbau über das *SWT*.“ In [Lee03] wird anschaulich dargestellt, wie sich mit *Draw2D* UML-Diagramme graphisch darstellen lassen. Die Beschreibung von [MDGW04, S. 93] über *Draw2D figures*, „they can have arbitrary, nonrectangular shapes and can be nested in order to compose complex scenes or custom controls” bestätigt ebenso, dass die Möglichkeiten dieses Frameworks (siehe auch [S:Draw2d]) für den Bau eines graphischen Modellierungswerkzeugs ausreichen.

GEF ist in der Anbindung des Modells nicht auf EMF angewiesen, dafür können ebenso andere Frameworks oder eine eigene Implementierung herangezogen werden. Die gemeinsame Nutzung hat sich in der Entwicklung von Modellierungswerkzeugen in [S:Eclipse] jedoch etabliert und wird häufig praktiziert. In vielerlei Hinsicht ist die eigentliche Realisierung der Verbindung von EMF und GEF überaus anspruchsvoll und erfordert ein tiefgehendes Wissen der Materie. Aus diesem Grund werden im Folgenden zwei Eclipse-zugehörige Projekte vorgestellt, [S:EclipseGMF] und [S:GEMS], die eine Integration der beiden Frameworks EMF und GEF zum Ziel haben.

#### 3.1.2.3 GMF

Das Project *Graphical Modeling Framework* (GMF) wurde durch die häufig kombinierte Verwendung von EMF und GEF zur Erstellung von graphischen Modellierungswerkzeugen inspiriert. Die beiden Frameworks werden weitgehend generisch verwendet, dies erfolgt durch verschiedene Stufen von Modelltransformationen (Code-Generierung und Code-Transformation).

Die folgende Abbildung aus [S:EclipseGMF] stellt die wesentlichen Abläufe und Modelle dar, die während der Erstellung einer GMF-basierten Anwendung eingesetzt, beziehungsweise erzeugt werden. Der hier dargestellte Ablauf ist dabei in der BPMN-Notation (siehe 2.5) visualisiert und wird nachfolgend näher erläutert. Noch eine Anmerkung zu dem Diagramm, das erste Gateway ist hier fehlerhaft als *XOR* modelliert, sinngemäß sollte es ein *AND Gateway* sein.

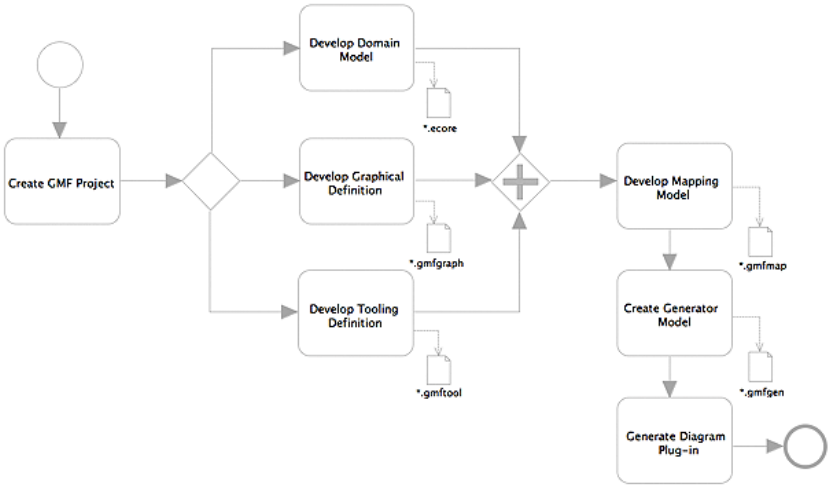


**Abbildung 64: Abläufe und Modelle in der Entwicklung mit GMF**

Das *Domain Model* (.ecore) wird mit EMF erstellt (siehe 3.1.2.1). Wie in der alleinigen Anwendung von EMF wird aus diesem Modell ein EMF *Generator Model* (.genmodel) erzeugt, das die notwendige Codegenerierung aus dem *Domain Model* steuert. Wie in 3.1.2.1 beschrieben, werden in diesem Generierungsschritt der *Model Code* und *Edit Code* erzeugt.
Die Funktionalität von EMF wird mit dem GMF-eigenen graphischen *Ecore Diagram Editor* erweitert. Dieser Editor (siehe Abbildung 65:) stellt das Ecore Modell graphisch dar und bietet die gleiche Funktionalität wie der EMF *Ecore Editor* in der Baumstruktur. Für die graphische Darstellung werden zusätzliche Daten wie Koordinaten in einer *.ecore_diagram* Datei abgelegt (vgl. [S:EclipseWiki, GMF Tutorial I]). Anfängliche Tests mit dem Editor haben gezeigt, dass Änderungen an dem Modell bidirektional zwischen der *.ecore* und der *.ecore_diagram* Datei synchronisiert werden (für den Versionsstand siehe [S:EclipseGMF]).

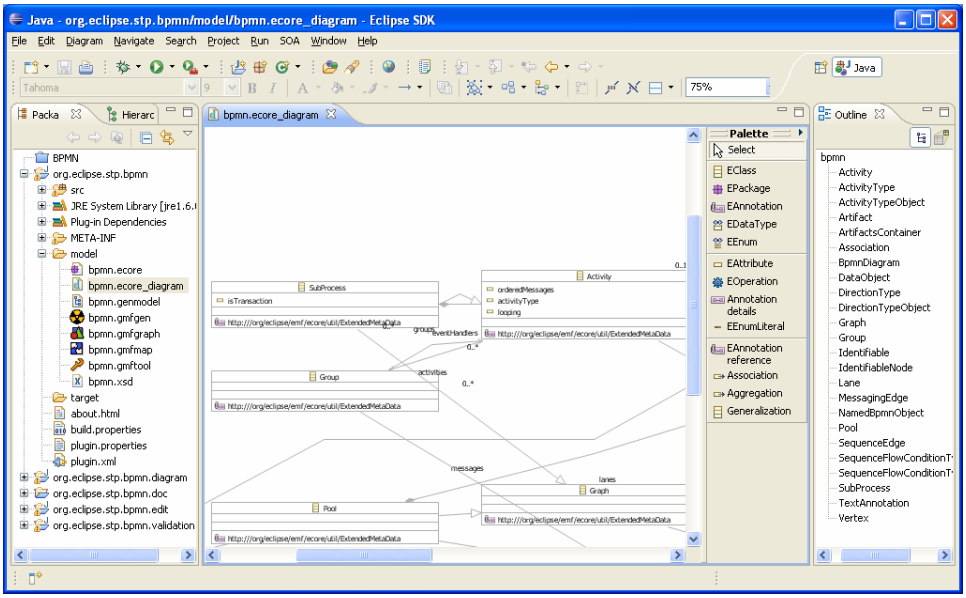


**Abbildung 65: GMF Ecore Diagram Editor**

Das *Graphical Definition Model* (.gmfgraph) wird in der Regel initial mit einem Wizard aus dem *Ecore* Modell erzeugt. Dieses Modell beinhaltet all diejenigen Informationen, die in Bezug zu den graphischen Elementen stehen, die später von dem eingebundenen GEF zur Laufzeit verwendet werden (*figures*, *nodes*, *connections*, *compartments* und *diagram label*). In dieses Modell können weitere Elemente importiert werden; das erlaubt den Aufbau von graphischen Bibliotheken und unterstützt die Wiederverwendbarkeit damit erheblich (vgl. [S:EclipseWiki, GMF Tutorial I]).

Das *Tooling Definition Model* (.gmftool) spezifiziert die Toolpalette, die in dem zu Modellierungswerkzeug zur Verfügung stehen soll. Außer der Toolpalette können das Hauptmenü, Kontextmenüs, Pop-Ups und einige weitere Artefakte angegeben werden (vgl. [S:EclipseWiki, GMF Tutorial I]).

Der nächste Schritt in der Entwicklung des Modellierungswerkzeugs (siehe auch Abbildung 64:) ist die Verbindung der drei bisher erstellten Modelle. Das *Mapping Definition Model* wird initial mithilfe eines Wizards erstellt. Die Elemente aus dem Ecore *Domain Model* werden ihrer (graphischen) Entsprechung aus dem *Graphical Definition Model* zugewiesen und an die jeweiligen Stellen in dem *Tooling Definition Model* angebunden (vgl. [S:EclipseGMF]). Selbst unter Zuhilfenahme des Wizards stellt dies für komplexe Modelle keine triviale Aufgabe dar.

Das im GMF Entwicklungsplan letzte Modell wird anschließend aus dem zuvor erzeugten *Mapping Definition Model* mit einem weiteren Wizard generiert. Das *Generator Model* ist vergleichbar mit dem .genmodel in EMF (siehe 3.1.2.1). Es wird zur Steuerung der finalen Codegenerierung des Editors verwendet, die wie in EMF mit JET erfolgt. Aus diesem *Generator Model* heraus wird letztlich das graphische Modellierungswerkzeug (in Form von Eclipse Plugins) erstellt. Die fertige Anwendung bietet neben Persistenzunterstützung und der grundlegenden Editorfunktionalität zusätzliche Features wie den Export in verschiedene Grafikformate, Ausdruck, Font und Farboptionen für Texte, Zooming und ein Übersichtsfenster.

In [S:EclipseWiki] wird an mehreren Beispielen gezeigt, wie die Erstellung eines graphischen Modellierungswerkzeugs mit [S:EclipseGMF] abläuft. In diesen Beispielen wird deutlich, dass es mit GMF möglich ist, einen Großteil der erwünschten Basisfunktionalität zu generieren, die von einem EMF / GEF basierten Editor erwartet wird. Wenn allerdings spezielle Anpassungen an dem über mehrere Etappen generierten Editor durchgeführt werden sollen, so müssen diese gezielt und äußerst behutsam vorgenommen werden, denn der eigentliche Fokus liegt auf dem Modell, in GMF sogar auf mehreren Modellen. Die manuelle Anpassung von generiertem Code hat ihre Tücken (vgl. [Schu07, 33 f., 68f.]). Diese sollte soweit wie möglich vermieden oder zumindest gut dokumentiert werden.
Spezielle Anforderungen mit GMF umsetzen zu können erfordert daher ein profundes Verständnis der ablaufenden Modelltransformationen, die in GMF hauptsächlich mithilfe von *JET* stattfinden (siehe 3.1.2.2). Es ist schwierig abzuschätzen, wie viel Einarbeitungszeit notwendig ist, um die genauen Abläufe zu durchschauen, die in [S:EclipseGMF] „under the cover“ ablaufen. Ein grundlegendes Verständnis der Frameworks EMF und GEF ist dafür bereits eine absolute Voraussetzung. Dass mit GMF hervorragende Anwendungen zur graphischen Modellierung gebaut werden können, wird in 3.2.1 gezeigt.

#### 3.1.2.4 GEMS

Ein weiteres Projekt zur Verbindung der Frameworks [S:EclipseEMF] und [S:EclipseGEF], sowie nach Angaben der Entwickler (vgl. [WWNS07, S. 18]) bald auch [S:EclipseGMF] ist das *Generic Eclipse Modeling System (GEMS)*. Dieses Projekt ist ein Unterprojekt der *Eclipse Generative Modeling Technologies* (GMT) und hat die Zielsetzung, den Entwicklungsaufwand von graphischen Modellierungstools in überschaubaren Grenzen zu halten. In [WWNS07, S. 18] beschreiben die Entwickler, dass vor allem in kleinen Organisationen das Know-how im Umgang mit EMF, GEF und GMF fehlt oder sich der notwendige Entwicklungsaufwand für die sehr speziellen Problembereiche nicht rechnet: „Trotz Eclipse-basierter Frameworks sind graphische Modellierungswerkzeuge nicht einfach zu entwickeln und lohnen sich daher nur für Domänen und Produkte mit einer ausreichenden Rendite.“
Eine kurze Anmerkung: Die Autoren unterscheiden in ihrer Beschreibung des Frameworks in [WWNS07] nicht zwischen einem Metamodell (vgl. 3.1) und einem Datenmodell.

Um ein graphisches Modellierungswerkzeug mit [S:GEMS] zu erstellen, wird zuerst ein Metamodell der zu modellierenden Domain-Specific Language (DSL) erstellt. Dies findet entweder in dem mitgelieferten graphischen Editor statt, oder es wird [S:EclipseEMF] (siehe dazu 3.1.2.1) erstellt. Der graphische Editor basiert in seiner Funktionalität auf UML Klassendiagrammen, bildet diese jedoch nicht vollständig ab. Es können Klassen, Attribute, Vererbung, Assoziation, Aggregation, Komposition und Multiplizität abgebildet werden. Verbindungsobjekte, die in dem späteren Editor zum Verbinden der Entitäten nutzbar sein sollen, sind ebenfalls modellierbar und können mit den bereits erstellten Klassen in Verbindung gesetzt werden (vgl. [WWNS07, S. 19], [S:GEMS]).

Das erstellte Metamodell wird danach von [S:GEMS] transformiert. Zuerst wird ein *EMF Ecore* Modell generiert, aus dem wiederum die Repräsenation in Java-Klassen erzeugt wird. Diese werden mit den ebenfalls generierten Wrapper-Klassen an das GEMS-Laufzeit-Framework angebunden. „Die GEMS-Laufzeitumgebung liefert eine auf GEF (und bald GMF) basierende Schicht mit weitergehenden Features, wie CSS Styles auf Modellelemente, Remote Update und automatisierte Modellierungsvorschläge mit Model Intelligence zur Wahrung der Constraints“ (siehe [WWNS07, S. 20]). Als Resultat der Transformation entsteht ein graphisches Modellierungswerkzeug in Form mehrerer Eclipse Plugins. Die folgenden Abbildungen zeigen die bei der Generierung durch [S:GEMS] erzeugten Artefakte (aus [WWNS07, S. 21]) und die Bausteine des GEMS-Frameworks (aus [WWNS07, S. 23]).

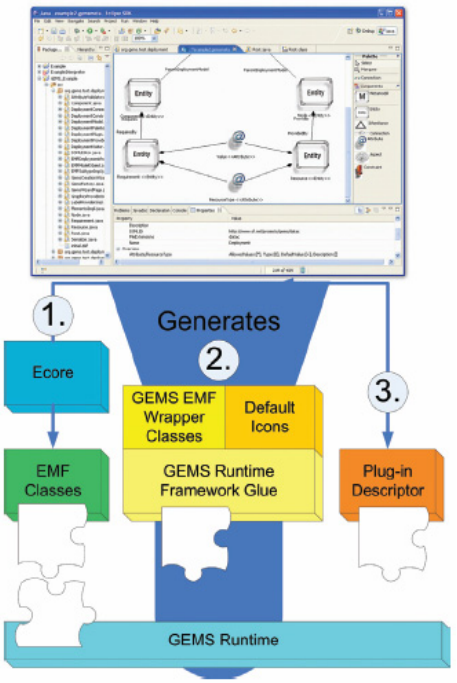


**Abbildung 66: GEMS Artefakte**

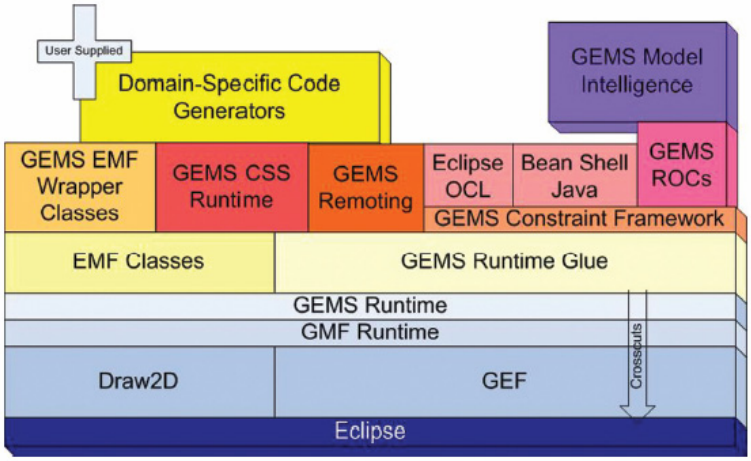


**Abbildung 67: Bausteine des GEMS-Frameworks**

Die Erstellung eines graphischen Modellierungswerkzeuges ist mit GEMS sogar möglich „ohne dabei notwendiger Weise Code zu schreiben“ (siehe [WWNS07, S. 18]). Die Entwickler betonen zudem, dass die Visualisierung vielseitig anpassbar ist: „So ist es beispielsweise möglich, die Default-Icons der Metamodell-Palette und der Modelle zu ändern. Durch eine Kombination von CSS mit dem GEMS-Tags-Mechanismus ist es möglich, komplexe domänenspezifische Visualisierungsmechanismen zu implementieren“ (siehe [WWNS07, S. 24]).

Für eine erweiterte Nutzung des entstandenen Modellierungswerkzeugs enthält das generierte Plugin *ExtensionPoints* (siehe 3.1.2), an denen die Modellinstanz für eine Traversierung zugänglich gemacht wird. Auf diese Weise können Funktionen zur Codegenerierung und Modellvalidierung in Form von Eclipse Plugins angeschlossen werden.

### 3.1.3 Oryx Framework

Vier Bachelor-Absolventen des Hasso-Plattner-Instituts haben innerhalb eines Jahres ein Web-basiertes Werkzeug zur grafischen Modellierung von Geschäftsprozessen entwickelt, das im Juni 2007 unter dem Namen Oryx veröffentlicht wurde (siehe [S:Oryx]). Durch die erweiterbare Architektur und den generischen Aufbau hat dieses Werkzeug jedoch eher den Charakter eines Frameworks und wird daher in eben diesem Kontext behandelt. In diesem Projekt wurde als Notation für die Modellierung der Geschäftsprozesse BPMN eingesetzt. Wie in [Pola07, S. 6] beschrieben, können auch andere Notationen mit diesem Framework realisiert werden. Beispielsweise wurden im Rahmen des Projekts auch Petri-Netze mit dem Framework implementiert, es existiert zudem eine Implementierung für die Modellierung von BPEL (vgl. [Cola07]). Die in der Entwicklungszeit erarbeiteten Bachelorarbeiten [Czuc07a], [Pete07], [Pola07] und [Tsch07] dokumentieren die Architektur, die Implementierung, die Erweiterungsmöglichkeiten und die Anpassbarkeit dieses Frameworks. Die in dieser Dokumentation beschriebenen Konzepte und Ansätze werden nun vorgestellt.

#### 3.1.3.1 Architektur

Das Framework [S:Oryx] besteht auf der Clientseite aus einer Browser-basierten Anwendung, die in der Interpretersprache JavaScript realisiert wurde. In [Czuz07b] wird die Architektur der Clientseite gezeigt:

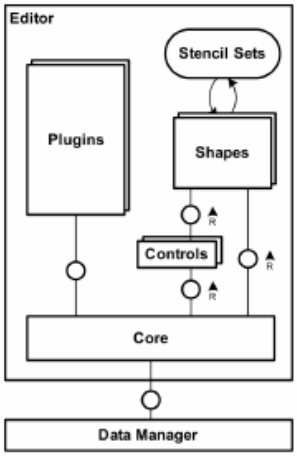


**Abbildung 68: Architektur der Clientseite**

Besonderer Wert wurde auf die Erweiterbarkeit und ein generisches Design gelegt. Der Editor bietet dazu eine eigene API, um Funktionen als Plugin zu integrieren (siehe dazu [Tsch07, S. 32 ff.], [Pete07, S. 51 ff.]). Ähnlich wie in [S:Eclipse] (siehe 3.1.2) können Erweiterungen nahtlos in die Toolbar integriert werden. Auch die Editor-Oberfläche kann auf diese Weise erweitert werden. Als Beispiele für Plugins werden in [Tsch07, S. 41] unter anderem *Drag & Drop*, *Resize*, *Zooming* und eine *Property View* zur Darstellung der Eigenschaften von Modellelementen genannt.
Das Metamodell wird als „*Stencil Set*“ (siehe 3.1.3.3) bezeichnet und von dem Editor erst zur Laufzeit geladen. Dieser generische Ansatz steht in markantem Gegensatz zu [S:MSDSL] und den [S:Eclipse] Frameworks. Er wird durch Verwendung der Interpretersprache JavaScript und die Objektbeschreibungssprache [S:JSON] auf einfache Weise ermöglicht.

Der Editor baut auf mehrere Frameworks auf, die einem [S:Oryx] Modellierungswerkzeug das Look & Feel einer Desktopanwendung ermöglichen. Die JavaScript Bibliothek [S:EXTJS] bietet erweiterte Funktionalität und Komponenten für grafische Benutzeroberflächen (vgl. [Tsch07, S. 19]). Das Framework [S:PrototypeJS] erweitert die JavaScript-Sprache mit zusätzlichen Datentypen. „So werden z.B. die JavaScript-Objekte wie Array oder Hash erweitert und zusätzlich neue Objekte wie Enumeration oder AJAX eingeführt“ (siehe [Tsch07, S. 19]).

Der *Data Manager* wird in [Tsch07, S. 17] als Datenschicht beschrieben. Diese stellt die Verbindung zwischen dem Editor und dem Modell her. Die einzelnen Konstrukte eines Modells werden als RDF-Tripel (Subjekt, Prädikat, Objekt, siehe dazu [S:RDF]) in das Document Object Model (DOM) der im Browser angezeigten Webseite eingebettet. Genauer gesagt wird dazu Embedded RDF verwendet [S:eRDF], eine für diesen Zweck geeignete Teilmenge von RDF. Diese Schicht regelt zudem den Datenaustausch mit der Serverseite, um Modellinformationen auszutauschen.

Die Serverseite hat zwei Funktionen. Zum einen liefert sie die eigentliche Anwendung in einer [S:XHTML] Seite aus und stellt den JavaScript-Code für die Ausführung des Editors bereit. Das Metamodell des Editors (siehe dazu 3.1.3.3) wird von einem *Stencil Repository Server* gestellt.
Die zweite Funktion der Serverseite ist die erweiterte Modellverarbeitung. Beispielsweise wird für die Speicherung eines Modells aus dem clientseitigen JavaScript heraus eine Kommunikation aufgebaut, mit der Modellinformationen durch einen *XMLHttpRequest*[10] ausgetauscht werden. Der Server kann auch andere Funktionalitäten implementieren, die mit JavaScript nicht ohne weiteres umsetzbar sind, wie beispielsweise eine Modellvalidierung. Diese ist aus Performancegründen nicht implementiert: „the browser’s JavaScript environment is single threaded“, siehe dazu [Pete07, S. 49].

#### 3.1.3.2 Entwicklungsumgebung

Um auf Basis des [S:Oryx] Frameworks ein graphisches Modellierungswerkzeug zu bauen, wird von den Entwicklern ein umfangreiches Szenario empfohlen: Als Entwicklungsumgebung kommt [S:Eclipse] zum Einsatz (siehe 3.1.2), das durch [S:Aptana] für die Entwicklung in JavaScript erweitert wird. Für die JavaScript Entwicklung werden die Bibliotheken [S:PrototypeJS] und [S:EXTJS] verwendet. Als Webserver lässt sich [S:ApacheHTTP] direkt an diese Umgebung anbinden. Die aufwendigen Schritte zur Zusammenführung von Code und das Deployment werden am besten mit [S:Ant] automatisiert. Das Testing findet mit einem Mozilla Firefox Browser ab der Version 2.0 statt. Clientseitiges Debugging wird mit dem JavaScript Fehlerprotokoll im Browser ermöglicht. Für die dynamische Erzeugung von Webseiten wird serverseitig [S:PHP] eingesetzt (vgl. [Tsch07, S. 9 ff.], [S:Oryx]). Zur Entwicklung von graphischen Repräsentationen in [S:SVG] (siehe 3.1.3.3.1) werden in [Pola07, S. 11] außerdem die Applikationen [S:Paint.NET] und [S:Inkscape] empfohlen.

#### 3.1.3.3 Stencil Sets

„Ein Stencil Set ist die Beschreibung einer Menge von graphischen Objekten mit ihren graphischen Repräsentationen, Eigenschaften und Regeln“ (siehe [Pola07, S.6]). Darin wird also das Metamodell der zu modellierenden Sprache festgelegt.

##### 3.1.3.3.1 Graphische Darstellung

Für die Beschreibung der graphischen Darstellung eines Stencil werden *Scalable Vector Graphics* (siehe [S:SVG]) verwendet. SVG ist ein Standard des World Wide Web Consortiums (W3C) zur Beschreibung zweidimensionaler Vektorgrafiken in der XML-Syntax. In dem Browser Mozilla Firefox wird dieser Standard ab der Version 2.0 ohne zusätzliche Plugins unterstützt.

[10] XMLHttpRequest ist eine Browsererweiterung. Dadurch kann man mit Skriptsprachen wie JavaScript Anfragen mittels des HTTP-Protokolls an einen Webserver übermitteln. Die Antwort von dem Webserver kann ohne ein vollständiges Neuladen in die Darstellung der Webseite dynamisch integriert werden.

Dieser Browser wird daher für die Ausführung von [S:Oryx] auf der Clientseite empfohlen. Das folgende Code-Listing aus [Pola07, S. 6] zeigt beispielhaft eine solche Beschreibung.

```
<svg
   xmlns="http://www.w3.org/2000/svg" xmlns:oryx="http://www.b3mn.org/oryx"
   width="40" height="40" version="1.0">
   <defs></defs>
   <oryx:magnets>
      <oryx:magnet oryx:cx="16" oryx:cy="16" oryx:default="yes" />
   </oryx:magnets>
   <g pointer-events="fill">
      <path id="frame" fill="white" stroke-width="1" stroke="black"
         d="M 1,16 L 16,1 L 31,16 L 16,31 L 1,16" />
      <circle cx="16" cy="16" r="7.5" stroke="black" fill="none" stroke-width="2.5"/>
   </g>
</svg>
```

**Listing 40: Beschreibung der graphischen Darstellung mit SVG**

#### 3.1.3.3.2 Eigenschaften

Die Eigenschaften eines Stencils werden unter Verwendung der JavaScript Object Notation (siehe [S:JSON]) definiert. JSON ist eine Teilmenge von JavaScript und bietet die Darstellung von Objekten[11] als Name-Wert Paare. In [Pola07, S. 7] wird eine derartige Stencil Definition gezeigt. In dieser Definition wird die graphische Beschreibung in SVG eingebunden (`view`), zusätzlich ist noch ein Piktogramm angegeben (`icon`), welches in dem Modellierungswerkzeug in der Toolpalette dargestellt wird. Die Regeln, die für dieses Stencil gelten, werden durch die Gruppenzugehörigkeit (`roles`) referenziert.

```
{
   "type": "node",
   "id":"OR Gateway",
   "title":"OR Gateway",
   "groups":["Gateways"],
   "description":"A decision point.",
   "view":"gateway/node.gateway.or.svg",
   "icon":"new gateway or.png",
   „roles": [
      // Verschiedene Rollen, die das Stencil einnehmen kann
   ],
   "properties": [
      {
         "id":"id",
         "type":"String",
         "title":"Id",
         "value":"",
         "description":"",
         "readonly":false,
         "optional":false,
         "refToView":"",
         "length":"30"
      },
      // weitere Properties
   ]
}
```

**Listing 41: Beschreibung der Eigenschaften eines Stencils**

[11] In diesem Zusammenhang wird JSON eigentlich zur Definition von Klassen verwendet.

#### 3.1.3.3.3 Regeln

In [S:Oryx] werden die Relationen der einzelnen Objekte untereinander durch Regeln definiert. Ein Stencil Set enthält eine Menge von Regeln, die für diejenigen Stencils gelten, die eine bestimmte Rolle erfüllen. Beispielsweise hat ein BPMN *Task* mehrere mögliche Rollen, er kann die Quelle eine *Sequence Flow* Verbindung sein, er kann aber auch das Ziel sein. Ein *Start-Event* erfüllt hingegen nur die Rolle der Quelle einer Verbindung. In [Pola07, S. 17] wird das Konzept deutlich gemacht: „Die Stencil Set Spezifikation arbeitet grundlegend so, dass sie alles verbietet, was nicht explizit erlaubt ist. Folglich müssen also Regeln definiert werden, die sagen, was erlaubt ist. Es gibt keine Anti-Regeln, die etwas explizit verbieten."

Der Aufbau des Regelsatzes, der für die Implementierung eines Metamodells zur Verfügung steht, wird in [Pete07, S. 39] gelistet:

```
{
   "title":"Workflow Nets",
   "namespace":"http://www.example.org/workflownets",
   "description":"Simple stencil set for Workflow Nets.",
   "stencils": [/*...*/],
   "rules":
      {
         "connectionRules": [/*...*/],
         "cardinalityRules": [/*...*/],
         "containmentRules": [/*...*/],
         "canConnect": function(args) { return args.result; },
         "canContain": function(args) { return args.result; }
      }
}
```

**Listing 42: Regeln in einem Stencil Set**

[Pete07, S. 39 ff.] unterscheidet drei grundsätzliche Arten von Regeln. Mit `connectionRules` wird definiert, wie einzelne Arten von Elementen verbunden werden können, wie beispielsweise eine *Activity* und ein *Gateway*. Mit `cardinalityRules` kann festgelegt werden, wie oft ein Stencil in einem Diagramm vorkommen darf. Mit diesen Regeln wird auch definiert, wie viele eingehende, beziehungsweise ausgehende Verbindungen erlaubt sind. Ein Beispiel zu *Start-Event* (siehe 2.5.1) aus [Pola07, S. 19] kann das verdeutlichen:

```
{
   "role":"Startevents all",
   maximumOccurrence:undefined,
   outgoingEdges: [
      {
         role:"SequenceFlow",
         maximum:1,
      }
   ]
}
```

**Listing 43: cardinalityRules in einem Stencil Set**

Mit den `containmentRules` wird festgelegt, welches Objekt in einem anderen enthalten sein kann, wie beispielsweise ein *Task* in einem *Sub-Process*.

In [Pete07, S. 44] wird ein Erweiterungsmechanismus gezeigt, mit dem sich auch komplexere Regelsysteme abbilden lassen. Wenn die Regelüberprüfung ergibt, dass ein Schritt in der Modellierung zulässig ist, so kann dieser dennoch nach einer erweiterten Prüfung abgelehnt werden. Mit den Funktionen `canConnect` und `canContain` lassen sich mit JavaScript erweiterte Restriktionen implementieren.

#### 3.1.3.4 Implementierung eines Stencil Sets

In [Pola07] ist die Implementierung des Stencil Sets für BPMN dokumentiert. Dadurch steht zusätzlich zu dem eigentlichen Framework, dessen Leistungsfähigkeit dadurch unter Beweis gestellt wurde, auch ein graphisches Modellierungswerkzeug für BPMN zur Verfügung. Sowohl der Quellcode für das BPMN Stencil Set, als auch der Quellcode für das gesamte Framework sind frei zugänglich (siehe [S:Oryx]). Die Anpassung dieses Werkzeugs wird in 3.2.2 gezeigt. Ein weiteres Stencil Set wurde in [Cola07] vorgestellt, mit dem ein auf [S:Oryx] basierender BPEL Editor realisiert wurde.

Der Umfang und der Aufwand für die Erstellung eines Stencil Sets für BPMN, sowie im Vergleich für Petri-Netze sind in [Pola07, S. 63] beschrieben:

*„43 SVG Dateien enthalten die Informationen zum Aussehen von Stencils, 33 Icons wurden gezeichnet, und alle Elemente in die BPMN Stencil Set Definitionsdatei (bpmn.json) eingefügt, Rollen und – darauf aufbauend – Regeln zum Verbinden, zu Enthaltenseinsbeziehungen und Kardinalitätsanforderungen definiert, mehrere hundert Properties zu den Stencils definiert, was die BPMN Stencil Set Spezifikationsdatei mit einer Gesamtlänge von derzeit 6525 Zeilen nicht gerade einfach und übersichtlich macht. [...] Des Weiteren sollte der Aufwand für den Designer zum Erstellen eines neuen Stencil Sets getestet werden. Die Implementierung eines einfachen Petrinetz Stencil Sets hat 1 Arbeitsstunde benötigt, um die drei grundlegenden Elemente Kante, Stelle und Transition in ihrer SVG Repräsentation, den Icons und der Stencil Set Definitionsdatei inklusive Regelwerk und einfachen Eigenschaften zu erzeugen.“* [Pola07, S. 63]

## 3.2 Open-Source Anwendungen

In diesem Abschnitt werden Open-Source Anwendungen vorgestellt, mit denen sich der geforderte graphische Editor durch Anpassung und Erweiterung realisieren lassen könnte. Wie bei der Betrachtung der Frameworks fand auch an dieser Stelle eine Vorauswahl statt, es wurden bewusst Anwendungen ausgewählt, die eine breite Unterstützung in der Open-Source Community finden. Darüber hinaus sollten sie einen Teil des Anwendungsbereichs (siehe 2.4, 2.5) bereits abdecken. Bei dieser Vorauswahl stellte sich heraus, dass es sowohl auf Basis des [S:GEMS]-Frameworks als auch auf Basis des [S:MSDSL]-Frameworks bisher keine derartige Open-Source Anwendung gibt.
Im Bereich der graphischen Modellierung von BPMN wird die [S:EclipseGMF]-basierte Anwendung [S:EclipseBPMN] und die [S:Oryx]-basierte Anwendung [S:BPEL4Chor] gezeigt. Im Bereich der graphischen Modellierung von BPEL wird das schon mehrere Jahre bestehende *BPEL Project*, beziehungsweise die Anwendung [S:EclipseBPEL] beschrieben. Diese Anwendung basiert auf den Frameworks [S:EclipseEMF] und [S:EclipseGEF]. Eine weitere Möglichkeit stellt die Anpassung des in [Kapl06] erstellten und in [Schu07] erweiterten graphischen BPEL Editors dar. Dieser Editor wurde auf Basis der Frameworks [S:EclipseEMF] und [S:EclipseGEF] gebaut und wird in diesem Abschnitt zuletzt vorgestellt.

### 3.2.1 STP BPMN Modeler

Der *SOA Tools Platform BPMN Modeler* [S:EclipseBPMN] ist ein auf GMF basierendes graphisches Modellierungswerkzeug für Diagramme in der BPMN 1.0 Notation. Es deckt einen Teilbereich des *SOA Tools Platform Project* ab. Dieses übergeordnete Projekt hat sich dem Bau zahlreicher Werkzeuge und Frameworks (zum Beispiel ein Editor für WS-Policy) verschrieben, die die Entwicklung von Software im Bereich serviceorientierter Architekturen (SOA) unterstützen sollen (vgl. [S:EclipseSTP]).
Der *STP BPMN Modeler* wurde Ende 2006 von dem BPM Hersteller *Intalio* an die Eclipse Foundation (siehe 3.1.2) gespendet (donated) und ist dadurch heute als Open-Source Werkzeug unter der Lizenz [EPL1.0] verfügbar. Diese Lizenz ermöglicht eine Verwendung von Open-Source innerhalb von proprietären Produkten. Durch diese Konstellation kann im *Intalio BPMN Designer* (siehe [S:Intalio]) der *STP BPMN Modeler* [S:EclipseBPMN], sowie dessen Weiterentwicklung als Kernstück verwendet werden, ohne die eigene Anpassung und Erweiterung offen legen zu müssen. Die wohl wichtigste Erweiterung durch Intalio ist die Integration in sein Business Process Management System (BPMS), das mehrere Open-Source Frameworks, wie beispielsweise [S:ApacheAxis2], [S:Geronimo] und [S:ApacheODE] beinhaltet (vgl. [S:Intalio]).
Der *STP BPMN Modeler* ist in erster Linie ein Modellierungswerkzeug für Prozesse unter Verwendung der BPMN-Notation[12]. Intalio beschreibt seine Erweiterung als „single tool that is used by business analysts, software engineers and system administrators for supporting the modeling of business-level processes, their binding onto external systems and user interfaces, and their deployment" (siehe [S:Intalio]). Diese Einsatzmöglichkeiten werden auch in [S:EclipseBPMN] beschrieben, sind allerdings nicht Teil der Kernfunktionalität.
Das in [S:EclipseBPMN] eingesetzte EMF-Datenmodell beschreibt die BPMN Spezifikation scheinbar nicht vollständig: „The STP BPMN Modeler uses a light and flexible object model. It strives to achieve the look and feel of the BPMN visual notation rather than force a schema that fully describes a specification." Diese Tatsache wird anschließend vorteilhaft ausgelegt: „The simplicity of the object model minimizes the impact when the specifications evolve. It also keeps the size of the generated code maintainable" (vgl. [S:EclipseBPMN]). Eine im Rahmen dieser Arbeit durchgeführte Evaluation hat ergeben, dass dieses Modellierungswerkzeug die Version

[12] Ausgeschrieben steht der Begriff BPMN-Notation für „Business Process Modeling Notation – Notation". BPMN ist nun einmal eine Notation mit dem Namen BPMN. Ebenso ist BPEL eine Sprache (Language), die Abkürzung „the BPEL-Language" hat dieselbe Problematik. Für den Inhalt dieser Arbeit hat diese sprachliche Unschönheit aber keine tragende Bedeutung.

[BPMN1.0] abbildet, jedoch die Attribute der Konstrukte nicht voll unterstützt. In Hinsicht auf die Modellierung beliebiger Abläufe wird auch keine volle Unterstützung von [BPMN1.0] geboten, beziehungsweise wird die freie Modellierung in BPMN bewusst eingeschränkt.
In [Inta07] werden die Arbeitsgebiete beschrieben, die im Entwicklungsprozess von *STP BPMN Modeler* gerade im Zentrum der Bemühungen stehen, nämlich der Export von BPMN-Diagrammen zu BPEL und WSDL Code sowie deren Import.

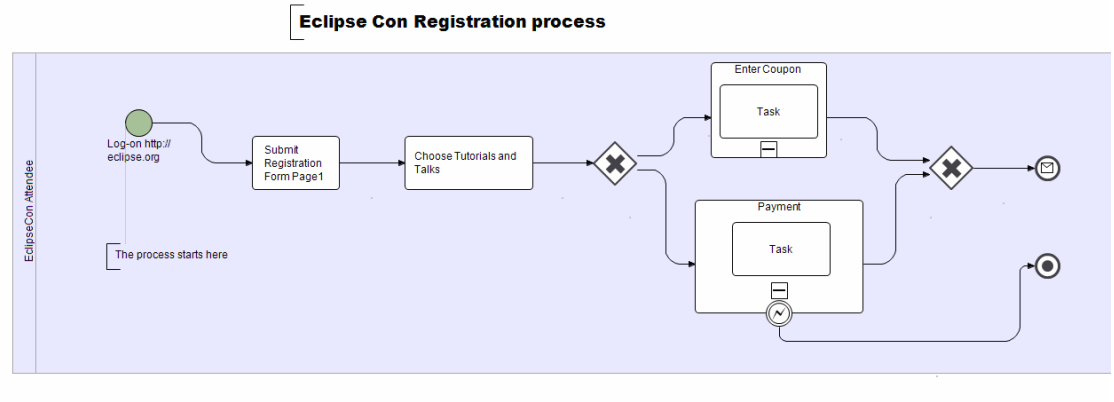


**Abbildung 69: Modellierung mit dem STP BPMN Modeler**

Diese Abbildung aus [S:EclipseWiki] zeigt beispielhaft die graphische Modellierung eines BPMN-Diagramms mit dem STP BPMN Modeler. Das *Message-End-Event* deutet auf den Versionsstand [BPMN1.0] hin, in dem nicht zwischen *Catching* und *Throwing Events* unterschieden wurde.

### 3.2.2 BPEL4Chor Oryx Editor

In [Pfit07, S. 83 ff.] wird beschrieben, wie unter Verwendung des [S:Oryx] Frameworks ein graphisches Werkzeug gebaut wurde, mit dem die Interaktion zwischen mehreren Partnern (Choreographie) in BPMN-Notation modelliert werden kann.

*"BPEL4Chor is an extension of BPEL for expressing choreographies and was first introduced in [DKLW07]. It is based on the Abstract Process Profile for Observable Behavior described in [BPEL07] and adds an interconnection layer on top of this profile."* [Pfit07, S. 10][13]

Durch die Erweiterung von [S:Oryx] (vgl. [Pfit07, S. 90 ff.]) kann ein mit [S:BPEL4Chor] erstelltes Modell durch eine serverseitige Transformation unter Einsatz des in 2.6.1.3 beschriebenen Verfahrens zur BPEL-Mustererkennung in BPEL4Chor (siehe [DKLW07]) übersetzt werden. Dazu wurde das bereits implementierte BPMN Stencil Set (siehe 3.1.3.4) angepasst[14]. Diese Umsetzung (vgl. [S:BPEL4Chor]) demonstriert sowohl die Flexibilität und Erweiterbarkeit des [S:Oryx] Frameworks, als auch die praktische Anwendbarkeit eines bestimmten Transformationsverfahrens von BPMN zu BPEL. Es wurden jedoch auch Grenzen in dem derzeitigen Entwicklungsstand identifiziert:

[13] Für die Beschreibung von Abstract Process siehe 2.4.5.10, [Schu07, S. 26 f.] und [BPEL2.0, S. 147 ff.].
[14] Diese Anpassung umfasst sowohl die Einschränkung als auch die Erweiterung von BPMN. Die resultierende Sprache wird als BPMN$^{+}$ bezeichnet.

*“Since the Oryx editor is still under development, there may be implemented some improvements in the future. First of all a verification after the diagram was modeled should be realized. In this way also the element attributes can be taken into account for the verification. Moreover, a better verification mechanism could ensure that there can be modeled only diagrams that are transformable to BPEL4Chor. For instance there may be implemented a check for the soundness and safeness of the diagram and that the diagram does not contain any arbitrary cycles.”* [Pfit07, S. 96]

Die festgestellte Problematik ist gundlegend für die Verbindung von BPMN und BPEL und wird in der Analyse und Diskussion der Transformationsverfahren in 4.1 eingehend behandelt.

Die folgende Abbildung zeigt eine Choreographie mehrerer Partner in der BPMN-Notation. In [Silv07a] beschreibt Marlon Dumas (siehe auch 2.6.1.3) diesen Editor unter der Referenzierung genau dieses Prozesses als “fairly complete AND produces readable BPMN-to-BPEL code.”

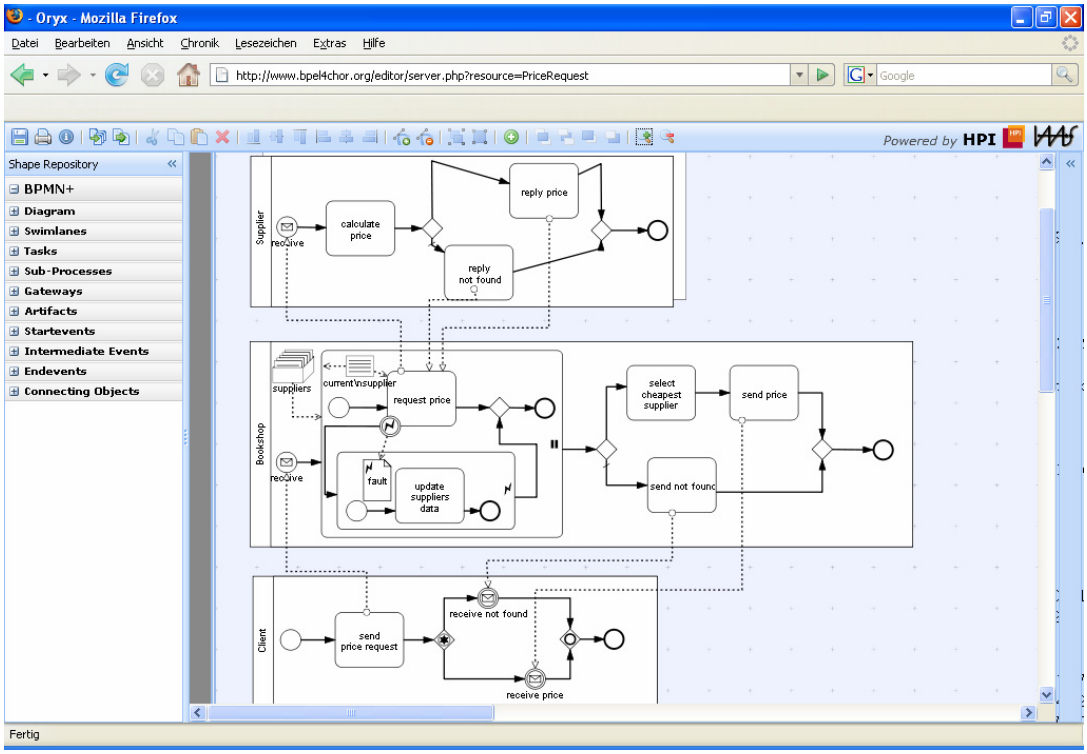


**Abbildung 70: Modellierung von BPEL4Chor in Oryx**

### 3.2.3 Eclipse BPEL Designer

Das *BPEL Project* hat sich mit dem auf EMF und GEF basierenden *Eclipse BPEL Designer* das Ziel gesetzt, Eclipse zu einem vollwertigen Framework für BPEL auszubauen. Dazu gehören nach [S:EclipseBPEL] die Definition, das Authoring (Erstellung), die Bearbeitung, das Deployment, das Testen sowie das Debuggen von BPEL-Prozessen. Um diese Ziele zu erreichen werden weitere Eclipse-basierte Frameworks wie die *Data Tools Platform* (siehe [S:EclipseDTP]) und die *Web Tools Platform* (siehe [S:EclipseWTP]) in die Software integriert. Die Komponenten, welche diese gewünschte Funktionalität realisieren sollen (vgl. [S:EclipseBPEL]) sind hier in Kürze beschrieben.

Der *Designer* ist das GEF basierte graphische Modellierungswerkzeug zur Erstellung und Bearbeitung von BPEL Prozessen. Das diesem Werkzeug zugrunde liegende *Model* wird als EMF-Modell zur Verfügung gestellt, das die Spezifikation [BPEL2.0] abbildet. Die Komponente *Validator* operiert auf einer Instanz des EMF-Modells, das mit dem Designer erstellt wurde. Es validiert einen BPEL-Prozess gegen die Beschreibung des Metamodells (siehe 3.1) in der Spezifikation [BPEL2.0] und erstellt dazu Warnungen und Fehlermeldungen. Das *Runtime Framework* ist eine erweiterbare Komponente zur Anbindung von BPEL-Engines, die das Deployment und die automatische Ausführung ermöglicht. Das *Debug Framework* soll das Debuggen von BPEL-Prozessen möglicherweise durch Simulation oder auch durch Anbindung an die BPEL-Engine zur Laufzeit zulassen.
Nach [S:EclipseBPEL] soll die Implementierung zahlreiche Möglichkeiten zur Erweiterung durch Drittthersteller bieten. In der gegebenen Architektur liegt dafür die zur Verfügungstellung von `ExtensionPoints` (siehe 3.1.2) nahe. Erweiterungsmöglichkeiten dieser Art sind unter anderem für die Erweiterung der Werkzeugpalette im Designer und für die Anbindung eines XPath Editors bereits verfügbar. Weitere Schnittstellen sind für die Unterstützung neuer Aktivitätstypen (siehe 2.4.1.3), die Erweiterung bestehender Konstrukte (siehe 2.4.5.6) und für ein herstellerspezifisches Branding geplant oder bereits in Arbeit (vgl. [S:EclipseBPEL, Milestoneplan]). Das aktuelle Release ist Milestone 3 (0.3.0) und deutet darauf hin, dass das Framework zurzeit noch keine vollständige Marktreife erreicht hat. Die Spezifikation [BPEL2.0] wird zudem noch nicht in vollem Umfang umgesetzt, die Erstellung von abstrakten Prozessen nicht unterstützt und die graphische Oberfläche bietet keine universelle Unterstützung der `extensionActivity` (siehe 2.4.1.3). Eine gute Stabilität und Benutzbarkeit hat eine im Rahmen dieser Arbeit durchgeführte Evaluation dennoch ergeben. Die Erweiterbarkeit von [S:EclipseBPEL] wurde durch die Anpassung „BPEL with Explicit Data Flow“ bereits von [Fern07] demonstriert. Diese Arbeit stellt darüber hinaus eine ausführliche Code-Dokumentation zu [S:EclipseBPEL] dar.

Die folgende Abbildung aus [S:EclipseBPEL] zeigt beispielhaft die graphische Modellierung eines BPEL-Prozesses mit dem Eclipse BPEL Designer:

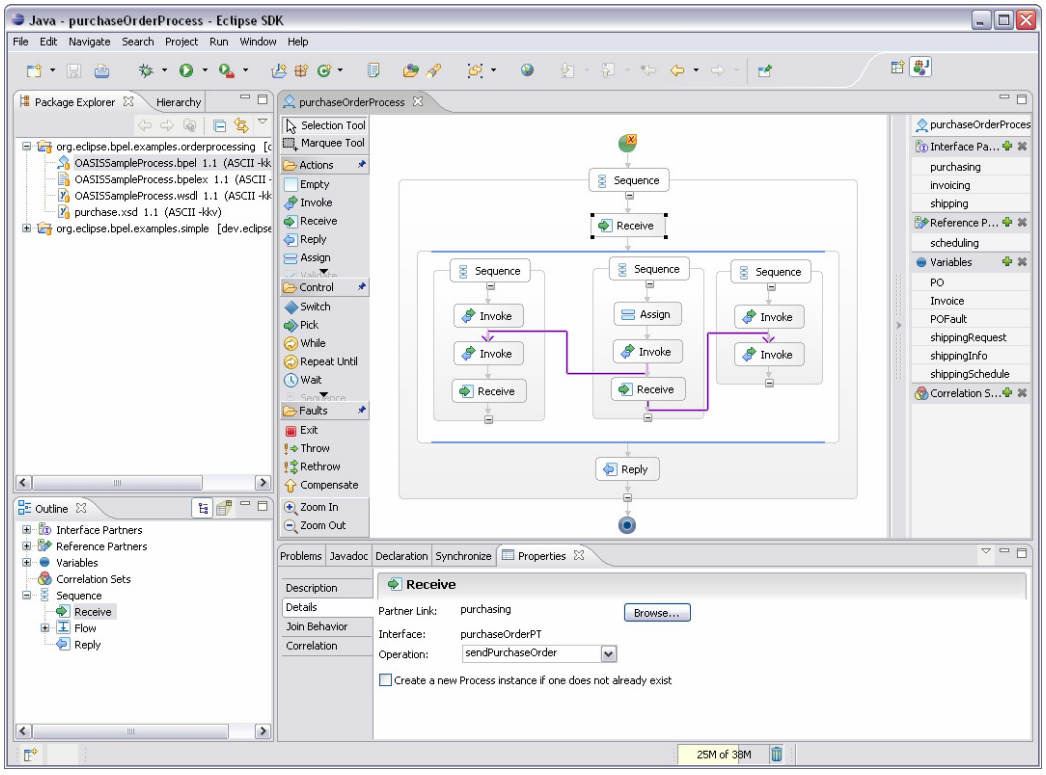


**Abbildung 71: Modellierung mit dem Eclipse BPEL Designer**

### 3.2.4 Graphisches BPEL Modellierungstool

In der Zeit, als die frei verfügbaren graphischen Modellierungstools für BPEL noch nicht die heutige Marktreife hatten, wurde in [Kapl06] zu Forschungszwecken ein eigenes Werkzeug entwickelt. Es basiert auf den Frameworks EMF (siehe 3.1.2.1) und GEF (siehe 3.1.2.2) und hat unter anderem weiterführende Konzepte wie *Templates* zur Wiederverwendung von Prozessteilen in BPEL umgesetzt (vgl. [Kapl06, S. 1, 9 f.]). Das Werkzeug erlaubt ebenso wie [S:EclipseBPEL] die graphische Modellierung von BPEL Prozessen und auch die Generierung von BPEL-Code aus dem erstellten Modell.
Durch die Standardisierung von BPEL durch OASIS, die im April 2007 mit der [BPEL2.0]-Spezifikation vorerst abgeschlossen wurde, war dieses Werkzeug jedoch nicht mehr compliant mit dem nun aktuellen Standard. In [Schu07] wurden daher die Standards [BPEL1.1] und [BPEL2.0] gegenübergestellt und die Unterschiede herausgearbeitet. Die identifizierten Änderungen wurden daraufhin in das Werkzeug übernommen, zum Beispiel wurden die neuen Aktivitäten `<forEach>` (siehe 2.4.2.2) und `<extensionActivity>` (siehe 2.4.1.3) in das Tool überführt und für die graphische Modellierung nutzbar gemacht (vgl. [Schu07, S. 61 ff.]). Die Umstellung auf [BPEL2.0] gelang aufgrund der zahlreichen feinen Änderungen sowie bereits zuvor bestehender Implementierungslücken leider nur unvollständig (vgl. [Schu07, S. 66]).
In [Schu07, S. 35] werden verschiedene Ansätze beschrieben, wie ein Mapping von BPEL zu BPMN realisiert werden könnte und darüber hinaus, welcher dieser Ansätze am besten zu der Architektur dieses Werkzeugs passt, die in [Schu07, 28 ff] ausführlich dokumentiert ist.

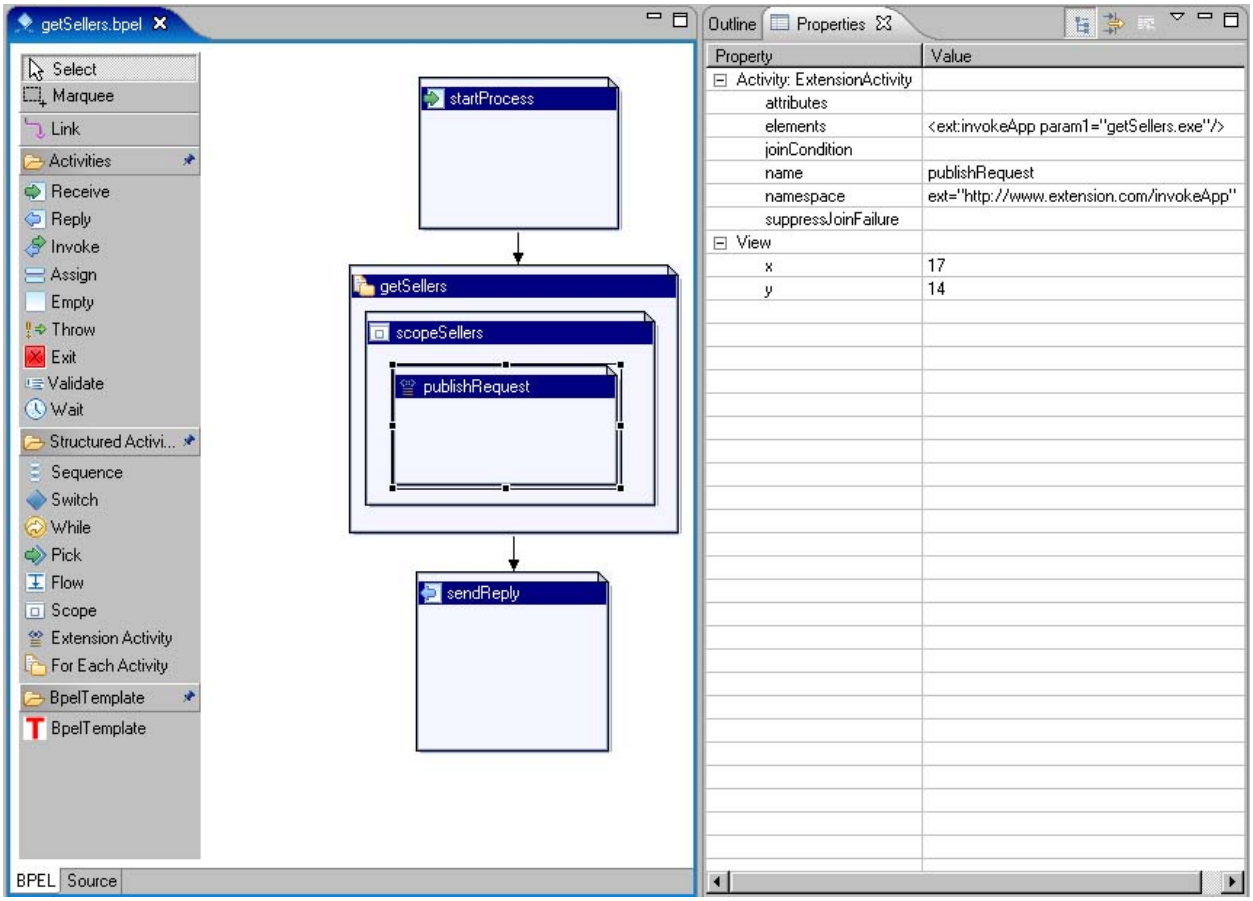


**Abbildung 72: Graphisches BPEL-Modellierungstool**

## 3.3 Auswahl und Begründung

In den wissenschaftlichen Arbeiten zum Thema *Model Driven Development* (MDD) werden überwiegend die Frameworks [S:EclipseEMF] und [S:EclipseGEF] sowie neuerdings auch [S:EclipseGMF] vorgestellt; das Framework [S:GEMS] führt bisher eher ein Schattendasein. Die Überlegungen, [S:MSDSL] für den Bau eines graphischen Modellierungswerkzeugs einer DSL einzusetzen, wie beispielsweise in [Rath06], finden selten statt. Diese werden bisweilen – wie auch in [Rath06] geschehen – verworfen und stattdessen ein auf [S:EclipseEMF] und [S:EclipseGEF] basierender Ansatz realisiert.
Ein generischer und dazu Web-basierter Ansatz wie in [S:Oryx] ist zwar im Moment noch nicht allzu verbreitet, gewinnt jedoch durch eine zunehmende Verlagerung von Anwendungen in das Internet[15] immer mehr an Bedeutung.
Um für die in 1.3 dargestellten Anforderungen die optimale Technik auszuwählen, werden im Folgenden die Frameworks [S:MSDSL] und [S:Eclipse] gegenübergestellt. Eine angemessene Gegenüberstellung[16] dieser MDD-Ansätze mit einem generischen und Web-basierten Ansatz bedeutet eine große Herausforderung und kann im Rahmen dieser Arbeit nicht geleistet werden.

### 3.3.1 Gegenüberstellung von DSL Tools und Eclipse

Eine detaillierte Gegenüberstellung von [S:MSDSL] und [S:EclipseGMF] wird in [Ozgu07, S.42 ff.] geleistet. Diese kann bei der Entscheidung für eines der vorliegenden Frameworks jedoch nur eine eingeschränkte Hilfestellung geben, denn beide Frameworks bieten nach [Ozgu07] die geforderte Funktionalität zur Erstellung eines graphischen Modellierungswerkzeuges. Zwar werden die in [S:MSDSL] zum Einsatz kommenden proprietären Formate als nachteilig gegenüber offener Standards wie beispielsweise *Ecore* oder *EMOF* (siehe [MOF2.0]) in [S:EclipseEMF] beschrieben, dafür bietet [S:MSDSL] mit seiner nahtlosen Integration in die Entwicklungsumgebung *Visual Studio* eine wesentlich stabilere Plattform – die allerdings, wie auch das zu entwicklende Modellierungswerkzeug, auf Microsoft Windows basiert. Letztendlich lautet das in [Ozgu07, S. 49 f.] gezogene Fazit, dass beide Frameworks überaus mächtig in ihren Einsatzmöglichkeiten sind, viel Einarbeitungszeit benötigen und noch nicht voll ausgereift sind.

Dass sowohl mit [S:EclipseEMF] und [S:EclipseGEF] anspruchsvolle Werkzeuge gebaut werden können, wurde in 3.2.3 und ebenso für [S:EclipseGMF] in 3.2.1 gezeigt. Für [S:GEMS] existieren derartige Werkzeuge nicht im Bereich von BPEL oder BPMN. Für das Framework [S:MSDSL] konnte gar kein derartig komplexes, frei zugängliches Werkzeug vorgestellt werden. Es ist jedoch wahrscheinlich, dass die *Model Designer Templates* (siehe 3.1.1.2) bei dem Bau von derartigen Werkzeugen entstanden sind. Das *Component Diagrams Template* könnte im Zusammenhang der Toolentwicklung für das *Windows Communication Foundation (WCF)* Framework entwickelt worden sein.

[15] Beispiele für Anwendungen sind Outlook Webaccess, Google Writely, Google Spreadsheets, Adobe Remix, Adobe Photoshop. Beispiele für unterstützende Technologien sind die unterschiedlich ausgeprägte JavaScript Unterstützung in den Browsern, Macromedia Flash, Macromedia Flex sowie Microsoft Silverlight.

[16] Eine angemessene Gegenüberstellung muss nicht nur die angebotene Funktionalität und deren Grenzen evaluieren, sondern in diesem Fall im besonderen nichtfunktionale Eigenschaften wie Performance, Lastverteilung, Stabilität und Fehlertoleranz mit einbeziehen.

In der folgenden Tabelle sind einige Eckpunkte des Frameworks [S:MSDSL] und der [S:Eclipse]-basierten Frameworks zur Erinnerung knapp gegenübergestellt:

| | **[S:MSDSL]** | **[S:Eclipse]** |
|---|---|---|
| **Eingesetzte Technologien** | | |
| Entwicklungsumgebung | Microsoft Visual Studio | Eclipse |
| Modelltransformation | Text Templates | Java Emitter Templates (JET) |
| Programmiersprache | Visual C# | Java |
| Datenmodell | Domain Model (Microsoft) | Ecore |
| Modellvalidierung | Custom Code | EMF Validation Framework |
| Graphische Werkzeuge | Domain Model Designer | EMF Ecore Modeler, GMF Ecore Diagram Editor |
| **Deployment** | | |
| Integration in IDE | Visual Studio Plugin | Eclipse Plugin |
| Stand-alone Anwendung | MSI | Eclipse RCP[17] |
| **Betriebssystem** | | |
| Entwicklung | Alle .NET[18] unterstützenden Plattformen | Alle Java[19]-unterstützenden Plattformen |
| Laufzeit | Alle .NET unterstützenden Plattformen | Alle Java-unterstützenden Plattformen |
| **Sonstiges** | | |
| Starthilfe | Model Designer Templates | Open-Source Anwendungen |

**Tabelle 3: Vergleich von DSL Tools und Eclipse Frameworks**

Diese Tabelle und die Ausführungen in 3.1.1 und 3.1.2 lassen sich in der Art interpretieren, dass beide Frameworks einen vergleichbaren Technologiestack aufbauen.

[17] Eclipse ist auch eine Plattform für das Deployment als Rich-Client Anwendungen. Um eine Anwendung nicht zusammengefasst als Feature oder als lose Sammlung von Plugins vertreiben zu müssen, können diese in Verbindung mit einem produktspezifischen Branding zusammen mit einer Distribution von [S:Eclipse] und einem eigenen Installer angeboten werden. Dieses Vorgehen wird zum Beispiel bei [S:ActiveBPEL] und [S:Intalio] praktiziert.

[18] Eine andere Implementierung von Common Language Infrastructure (CLI) basierten Frameworks wie [S:Mono] wird in [S:MSDSL] nicht erwähnt. Es werden die .NET Frameworks ab der Version 1.1 unterstützt.

[19] Die vorgestellten Frameworks setzen zurzeit das Java Runtime Environment (JRE) in der Version 1.5 (J2SE Version 5.0) ein.

### 3.3.2 Gegenüberstellung der Open Source Anwendungen

Das in 3.2.1 vorgestellte BPMN-Modellierungstool [S:EclipseBPMN] erlaubt die graphische Modellierung von Prozessen mit der (eingeschränkten) BPMN-Notation, bietet jedoch noch keine Funktionalität zur Übersetzung von BPMN in BPEL. Wie in 3.2.1 beschrieben, wird gerade an dieser Schnittstelle gearbeitet.

Das in 3.2.2 vorgestellte [S:BPEL4Chor] erlaubt die graphische Modellierung BPMN$^{+}$ und implementiert die Transformation eines Modells in BPEL4Chor mit dem Verfahren aus 2.6.1.3. Für das zu Grunde liegende Framework [S:Oryx] ist zudem eine Anpassung für die Modellierung von [BPEL2.0] verfügbar welche in [Cola07] beschrieben wird.

Eine aktive Entwicklergemeinschaft (vgl. [S:BPELnews]) hat mit [S:EclipseBPEL] ein graphisches Modellierungswerkzeug geschaffen (siehe 3.2.3), das unter anderem über BPEL 2.0 Compliance und Modellvalidierung verfügt. Eine schlüssige Dokumentation der Architektur wurde zudem in [Fern07] geleistet. Eine Schnittstelle zu BPMN ist laut dem Milestoneplan in [S:EclipseBPEL] im Moment weder in Planung noch in Arbeit.

Der in 3.2.4 vorgestellte graphische BPEL Editor wurde in [Kapl07] zu Studienzwecken erstellt und in [Schu07] ausführlich dokumentiert. Den dadurch gewonnenen Kenntnissen über die Architektur dieses Werkzeugs steht mit [S:EclipseBPEL] allerdings ein hochentwickeltes Werkzeug gegenüber. Dieses erfüllt gehobene Ansprüche an Erweiterbarkeit und Wartbarkeit besser (vgl. [Schu07, S. 68 f.]).

### 3.3.3 Auswahl durch Ausschlussverfahren

Die Modellierungswerkzeuge [S:EclipseBPMN], [S:BPEL4Chor] und [S:EclipseBPEL] bieten bereits zum heutigen Zeitpunkt eine größere Funktionalität als ein Werkzeug, das im Rahmen dieser Arbeit mit [S:Eclipse] erstellt werden könnte (vgl. [Kapl06], [Schu07]). Nach [Ozgu07, S. 42 f., 49 f.] ist mit großer Wahrscheinlichkeit anzunehmen, dass die Erstellung mit [S:MSDSL] einen vergleichbar großen Aufwand bedeutet. Das Framework [S:MSDSL] kommt daher nicht zum Einsatz, obwohl es für den Bau eines graphischen Modellierungswerkzeuges ebenso wie [S:Eclipse] geeignet wäre. Aus demselben Grund wird das Framework [S:GEMS] nicht eingesetzt werden. Für das Framework [S:Oryx] existieren bereits Erweiterungen und Anpassungen für BPMN und BPEL, die einer Neuentwicklung mit diesem Framework vorgezogen werden.

Um eine Entscheidung zwischen [S:EclipseBPMN], einer bestehenden [S:Oryx] Anpassung und [S:EclipseBPEL] treffen zu können, muss zuerst das Konzept definiert werden, mit dem die Verbindung von BPMN und BPEL realisiert werden soll. Dies geschieht im folgenden Kapitel.

# 4 Graphische Modellierung von BPEL Prozessen

## 4.1 Kritische Analyse der bestehenden Ansätze

### 4.1.1 Einschränkung von BPMN

Der erste Gesichtspunkt, nach dem die in 2.6 vorgestellten Ansätze analysiert werden, ist die Einschränkung der Ausdruckskraft von BPMN, um eine Transformation zu ermöglichen. Dazu wird zuerst ein Beispiel vorgeführt, welches die Notwendigkeit einer Einschränkung mehr als nahe legt. Anschließend werden die Einschränkungen im Verfahren von Ouyang et al. (siehe 2.6.1.3) zur Transformation von BPMN in BPEL betrachtet. Das Verfahren von Mendling et al. (siehe 2.6.1.2) ist im Vergleich dazu bei weitem nicht so präzise und fortgeschritten und bringt daher für eine Analyse keinen Mehrwert. Danach werden diese Einschränkungen aus der Sicht des Anwenders, beziehungsweise anhand der Implementierung in Modellierungswerkzeugen betrachtet.

#### 4.1.1.1 Ein Beispiel freier Modellierung in BPMN

Die folgende Abbildung zeigt einen Prozess in der BPMN-Notation, modelliert mit dem [S:Oryx]-Framework. Dieser Prozess soll die Mächtigkeit der BPMN-Notation demonstrieren, die in kurzer Prägnanz einen überaus komplexen Sachverhalt anschaulich darstellen kann.

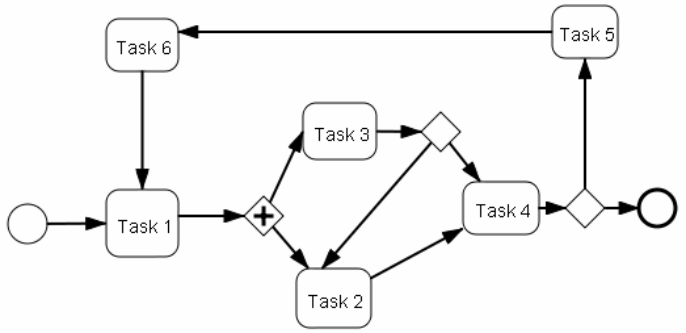


**Abbildung 73: Möglichkeiten der Modellierung in BPMN**

In dieser Abbildung wird ein Prozess dargestellt, der sowohl Gebrauch von Parallelität und von „Uncontrolled Flow“ (siehe 2.2.3) macht, als auch von bedingten Sprüngen mit einem *XOR Gateway* (siehe 2.5.3.1.1). Eine grobe Beschreibung dieses Prozesses zeigt die Ausdruckskraft von BPMN: Nach der Ausführung von „Task 1“ findet eine Aufteilung in zwei parallele Abläufe (Threads) statt. Diese Threads werden allerdings in „Task 4“ weder zusammengefasst noch synchronisiert, sondern sie haben dann lediglich denselben Ablaufpfad. Ein bedingter Sprung nach der Ausführung „Task 4“ kann die iterative Ausführung des Prozesses bewirken. Die Anzahl bestehender Threads erhöht sich nach jeder Ausführung von „Task 1“ um einen Thread. Eine Abgrenzung von Gültigkeitsbereichen sowie eine Berücksichtigung möglicher Race Conditions sind nicht gegeben. Für die Schwierigkeiten der Überführung von singlethreaded Code in multithreaded Code vgl. [Tane01, S. 97].

#### 4.1.1.2 Einschränkungen nach Ouyang et al.

Der in 4.1.1.1 gezeigte Prozess setzt mehrere Strukturen ein, die in dem in 2.6.1.3 beschriebenen Verfahren von vorn herein ausgeschlossen sind. Dies betrifft zum einen den in 2.2.3 beschriebenen *Uncontrolled Flow*. Mit derartigen Strukturen lassen sich in BPMN Prozesse modellieren, die in BPEL nicht abbildbar sind (vgl. Pattern #08, Multi-Merge, 2.3.3), daher wurden diese Strukturen von der Verwendung ausgeschlossen. Verzweigungen und Zusammenführungen von Pfaden sind nach Ouyang et al. nur an *Gateways* zulässig.

In BPMN können prinzipiell beliebige Abläufe modelliert werden, auch dies wird in BPEL nicht unterstützt (vgl. Pattern #10, Arbitrary Cycles, 2.3.3). In dem Verfahren der *Event-Action Rule-Based Translation* (siehe 2.6.1.3.6) wird gezeigt, wie sich bestimmte Arten von Sprüngen dennoch realisieren lassen, jedoch auf Kosten der *Readability* (Verständlichkeit) des erzeugten Codes (siehe 2.6.1.3.1).
Die Verwendung von *OR Gateways* (siehe 2.5.3.2) wird von dieser Gruppe gerade erforscht und ist zum derzeitigen Stand von dem Verfahren (vgl. [ODHA07]) nicht berücksichtigt. Die Übersetzung dieses Gateways ist mit seiner bedingten Parallelität besonders im Zusammenhang mit beliebigen Abläufen „challenging" (vgl. [ODHA07, S. 20]). Eine maßgebliche Schwierigkeit ist die Verwendung als *OR-Join*, bei dem auf alle Tokens gewartet wird, die von jedem der angelegten Eingänge eintreffen können.

#### 4.1.1.3 Einschränkungen in den Modellierungswerkzeugen

In [S:Intalio] (siehe 3.2.1) lassen sich die einzelnen Elemente von BPMN zwar frei platzieren, jedoch nicht beliebig verbinden. In [S:Oryx] hingegen ist die Modellierung „fully supporting BPMN" (siehe 3.1.3, [S:Oryx]). Die Einschränkungen, die Modellierungswerkzeuge ihrem Benutzer auferlegen, sind in der weiteren Verarbeitung der Modelle begründet. Während [S:Intalio] eine Anbindung zu der BPEL-Engine [S:ApacheODE] bereitstellt und daher keine beliebigen Abläufe modelliert werden können, dient [S:Oryx] zunächst nur der Modellierung. Die Erweiterung von [S:Oryx] (siehe 3.2.2, [S:BPEL4Chor]) hat gezeigt, dass Einschränkungen von BPMN erforderlich sind sobald eine Transformation in BPEL gewährleistet sein soll.
Das Verfahren *Event-Action Rule-Based Translation* (siehe 2.6.1.3.6), ebenso wie das Verfahren *Process Rewrite* (siehe 2.6.3.1), ermöglichen unter nur geringen Einschränkungen die Modellierung mit BPMN. Sie haben allerdings den Nachteil, dass der generierte BPEL-Code nur noch schwer mit dem eigentlichen Modell in Verbindung gebracht werden kann. In den überschaubaren Prozessen, an denen diese Verfahren demonstriert werden, ist dies nicht so offensichtlich und wird nur eingeschränkt angesprochen.

#### 4.1.1.4 Schlussfolgerungen

Welche Möglichkeiten bleiben also für ein graphisches BPMN Modellierungswerkzeug? Zum einen die nahezu uneingeschränkte Modellierung in BPMN (jedoch nie die vollständige, vgl. 4.1.1.1) und die Erzeugung von schlecht verständlichem (vgl. 2.6.1.3.6), kaum wartbarem und nur sehr eingeschränkt anpassbarem BPEL-Code (vgl. 2.6.3.1). Die andere, praktikable Möglichkeit (siehe [S:Intalio]) ist die Einschränkung von BPMN und die Erzeugung von verständlichem, wartbarem und gut anpassbarem BPEL-Code. Der Nachteil bei diesem Ansatz liegt nicht so offensichtlich auf der Hand und wird nun erörtert.

BPEL wird in der Spezifikation von BPMN als „unintuitive format" sowie als „hard to understand by the business analysts and managers" beschrieben (siehe [BPMN1.1, S. 11]), BPMN hingegen wird als „visually intuitive" verstanden (siehe [Whit04, S.1]). Ohne das Wort „intuitive" zu verwenden, argumentieren Ouyang et al. ähnlich.

*"There are over 20 execution engines supporting BPEL. Many of them come with an associated graphical editing tool. However, the notation supported by these tools directly reflects the underlying code, thus forcing users to reason in terms of BPEL constructs (e.g., block-structured activities and syntactically restricted links). Current practice suggests that the level of abstraction of BPEL is unsuitable for business process analysts and designers. Instead, these user categories rely on languages perceived as "higher-level" such as BPMN and various UML diagrams, thus justifying the need for mapping languages such as BPMN onto BPEL."*
[ODHA07, S. 3, hervorgehoben]

Was bewirken nun die Einschränkungen von BPMN? Zum einen ermöglichen sie eine weitgehend direkte Transformation in BPEL-Code. Das Verfahren von Mendling et al. (siehe 2.6.1.2) ist dafür ein Musterbeispiel, denn in diesem Verfahren werden lediglich Strukturen erlaubt, die direkt BPEL-Code reflektieren. Allerdings wird der Benutzer – trotz der Argu-

mentation von Ouyang et al., genau dies verhindern zu wollen (siehe oben) – durch Einschränkungen immer mehr dazu gezwungen, in den Strukturen einer Programmiersprache zu modellieren, beziehungsweise in diesen Strukturen zu denken. Ist ein derartig eingeschränktes BPMN-Modellierungswerkzeug im Sinne von [Whit04] noch immer „intuitive“? Eine eigene Evaluation von [S:Intalio] ergab, dass die Modellierung mit eingeschränktem BPMN eher mühsam verläuft. Der Modellierer muss nicht nur die Konzepte der Modellierungssprache verstehen, sondern auch die durch das Transformationsverfahren auferlegten Einschränkungen. Für eine erfolgreiche Modellierung muss dann ein Verständnis geschaffen werden, wieso ein Sachverhalt auf die „BPEL-orientierte“ Weise umgesetzt werden muss und nicht auf die „intuitive“ Art.

### 4.1.2 Modelltransformationen

Die Übersetzung eines BPMN-Diagramms in BPEL-Code wird als Modelltransformation bezeichnet. Diese umfasst außer einer Abbildung der unterschiedlichen Sprachkonstrukte auch eine Übersetzung zwischen den unterschiedlichen Sprachsemantiken. Wenn eine Modelltransformation eingesetzt wird, so sind bestimmte Aspekte zu beachten. Die Qualität einer Modelltransformation zeigt sich an dem generierten Zielcode. Dieser Gesichtspunkt wird nachfolgend unter dem Abschnitt „Readability“ betrachtet. Ein weiteres Kriterium stellt die Validierung eines Modells dar, die sowohl vor als auch nach der Transformation durchgeführt werden kann. Zuletzt spielt die semantische Äquivalenz der originalen und der transformierten Prozesse eine erhebliche Rolle.

#### 4.1.2.1 Readability

Der in 2.6.1.3.1 eingeführte Begriff der *Readability*, der Verständlichkeit des Zielcodes für Menschen, spielt bei der Modelltransformation eine wesentliche Rolle. Eine Transformation beliebiger Abläufe mit der *Event-Action Rule-Based Translation* (siehe 2.6.1.3.6) generiert BPEL-Code, der die Eigenschaft der Readability nicht aufweist. Der Begriff wurde durch diesen Missstand geprägt. Die absolute Notwendigkeit der Readability wird in [ODHA07, S. 2] beschrieben:

*“BPEL definitions produced by the translation are likely to require refinement (e.g., to specify partner links and data manipulation expressions) as well as testing and debugging.”*

Mehrere Werkzeuge, die die Modellierung in BPMN und die Modelltransformation in BPEL unterstützen, erlauben bereits die Eingabe von `partnerLinks` und XPath-Ausdrücken. Das Testing und Debugging läuft allerdings in einer BPEL-Engine fern von dem BPMN Modell ab. Weitere Argumente für die Readability sind das Monitoring, die dynamische Anpassung von Prozessen zur Laufzeit, Verifikation sowie die Verständlichkeit der Protokollierung der Ausführung (Audit-Trail, vgl. [LeRo00, S. 45, 105]).

#### 4.1.2.2 Modellvalidierung

Eine Modellvalidierung überprüft ein gegebenes Modell auf Korrektheit. Dabei werden gemäß der Spezifikation des Metamodells[20] Warnungen, beziehungsweise Fehlermeldungen erzeugt. Die Spezifikationen [BPMN1.1] und [BPEL2.0] sind, wie hinreichend deutlich gemacht wurde, grundlegend verschieden. Dadurch ist auch die Modellvalidierung von BPMN und BPEL grundlegend verschieden.
Modelltransformationen erschweren die Validierung und somit auch die Modellierung erheblich. Vor der Transformation von BPMN in BPEL kann nur eine Aussage darüber getroffen werden, ob es sich um ein Modell handelt, das gegen ein (eingeschränktes) BPMN valide und damit transformierbar ist. Erst nach der Transformation kann eine Aussage darüber getroffen werden, ob es sich bei dem transformierten Modell um ein Modell handelt, das gegen die BPEL-Spezifikation valide ist. Die eigentliche Schwierigkeit besteht in diesem Ablauf darin,

[20] Der Begriff des Metamodells ist in 3.1 definiert.

Validierungsfehler in BPEL auf das originale Modell in BPMN zurückzuführen und beheben zu können, da die Transformationsverfahren die Prozesstopologie teilweise verändern (vgl. 2.6.1.3.4, 2.6.3.1).

#### 4.1.2.3 Besonderheiten der Metamodelle

Die *Flattening* Strategie (siehe 2.6.2.2.1) von Mendling et al. beschreibt eine Transformation von BPEL zu BPMN, die die Dead-Path-Elimination (DPE, siehe 2.4.2.1.4) in BPEL nicht berücksichtigt. Die Transformation ist daher nicht korrekt, beziehungsweise noch unvollständig. Das Konzept der DPE existiert in BPMN in der Tat nicht. In den Transformationsverfahren von BPMN zu BPEL wird der DPE in soweit Rechnung getragen, dass für eine Transformation in einen BPEL-`<flow>` nur solche Strukturen in BPMN erlaubt sind, die keine DPE erfordern. In [ODHA07, S. 11] werden derartige Strukturen als „sound“ bezeichnet (vgl. 2.6.1.3.1). Die DPE ist ein herausragendes Beispiel der Besonderheit von einem der Metamodelle. Die Spezifikationen von BPMN und BPEL beinhalten jedoch eine ganze Reihe feiner Besonderheiten, die eine semantisch vollständig korrekte Transformation erschweren.

#### 4.1.2.4 Weiterer Lebenszyklus nach der Modelltransformation

Die in 2.6 beschriebenen Verfahren machen nur geringfügige Annahmen über den weiteren Lebenszyklus eines Modells nach einer Transformation. In [ODHA07, S. 2] wird beschrieben, dass der BPEL-Code, der bei einer Transformation erzeugt wird, höchstwahrscheinlich überarbeitet werden muss, „e.g., to specify partner links and data manipulation expressions.“ Solange eine derartige Überarbeitung nach einer Transformation stattfinden muss, sind Mechanismen zum Schutz vorgenommener Anpassungen notwendig. Die in 3.1 vorgestellten Frameworks zum Bau graphischer Editoren setzen beispielsweise einen Mechanismus in Form von *Preservation-Tags* (siehe 3.1.1.6) bei der Modelltransformation ein. Das Zurückführen nicht-trivialer Anpassungen in dem erzeugten Code in das originale Modell (Round Trip, vgl. auch 2.6.3) stellt allerdings auch diese überaus mächtigen Frameworks vor eine bisher ungelöste Aufgabe.

*„We had what became known as the roundtripping problem. A process model created in BPMN or comparable flowcharting notation could not be easily kept in sync with the executable BPEL design throughout the implementation lifecycle. Essentially, you couldn't update the process model from the BPEL. Once the toothpaste was out of the tube, you weren't going to get it back in."* [Silv07a]

#### 4.1.2.5 Schlussfolgerungen

Die Readability des erzeugten BPEL-Codes ist für die Wartbarkeit unerlässlich, daher sollten nur Modelltransformationsverfahren zum Einsatz kommen, die diese gewährleisten. Dies ist nach dem heutigen Stand der Spezifikationen nur unter der Einschränkung von BPMN möglich. Die Modellvalidierung der Zielsprache sollte so früh wie möglich stattfinden. Wenn sie erst nach einer Modelltransformation durchgeführt wird, so sind Fehler und Probleme zum Teil nur schwer auf entsprechende Strukturen des originalen Modells zurückzuführen. Sowohl das Metamodell von BPMN als auch das Metamodell von BPEL haben Besonderheiten, die in einer Transformation berücksichtigt werden müssen. Nicht-triviale Anpassungen an einem transformierten Modell können bisher nicht in das originale Modell zurückgeführt werden. Infolgedessen müssen Anpassungen und Verfeinerungen trotz möglicher Schutzmechansimen überprüft und gegebenenfalls wiederholt durchgeführt werden, sobald eine erneute Modelltransformation durchgeführt wird. Alternativ müsste ein Modellierungswerkzeug dazu in der Lage sein, alle Konstrukte der Zielsprache zur Modellierung anzubieten, damit keine Anpassung des transformierten Modells erforderlich ist.

### 4.1.3 Umstrittene Verwendung von Goto

Der letzte Aspekt, unter dem die bestehenden Ansätze analysiert werden, erscheint auf den ersten Blick möglicherweise weit hergeholt. Eine eingehende Beschäftigung mit beliebigen Abläufen in BPMN und der *Event-Action Rule-Based Translation* (siehe 2.6.1.3.6) zeigt jedoch Parallelen zu dem `Goto`-Ausdruck auf, auf die nun eingegangen wird. Zuerst wird die schon lange zurückliegende Diskussion um die Verwendung des `Goto`-Ausdrucks wiederbelebt und anschließend die Verbindung zu den Sprachen BPMN und BPEL hergestellt.

#### 4.1.3.1 Goto Ausdrücke werden als schädlich eingeschätzt

Edsger W. Dijkstra hat im Jahre 1968 in einem offenen Brief („Go To Statement Considered Harmful“) an die Association for Computing Machinery (ACM) gefordert, den damals überaus gebräuchlichen `Goto` Ausdruck aus höheren Programmiersprachen zu verbannen (vgl. [Dijk68]). Stattdessen sollen Programmstrukturen wie `if / then / else`, `while / do`, `repeat / until`, `switch / case` verwendet werden. Diese Forderung wird mit der Schwierigkeit begründet, einen gegebenen Ablauf anhand von Variablenwerten nachvollziehen zu können.

*“The unbridled (Anm.: ungezügelt) use of the go to statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress.”* [Dijk68, S. 1]

In dem Aufsatz „the humble programmer“ (Anm.: humble – demütig), den Dijkstra bei der Verleihung des Turing Award im Jahre 1972 verlesen hat, fand er noch deutlichere Worte über die Wartbarkeit von Programmen, die Gebrauch von `Goto`-Ausdrücken machen (siehe [Dijk72, S. 11]).

A study of program structure has revealed that programs –even alternative programs for the same task and with the same mathematical content– can differ tremendously in their intellectual manageability. A number of rules have been discovered, violation of which will either seriously impair or totally destroy the intellectual manageability of the program. These rules are of two kinds. Those of the first kind are easily imposed mechanically, viz. by a suitably chosen programming language. Examples are the exclusion of goto-statements and of procedures with more than one output parameter.

**Abbildung 74: Zitat aus „the humble programmer”**

#### 4.1.3.2 Programme mit und ohne Goto Ausdrücke

Ouyang et al. weisen bei der Beschreibung ihrer Transformationsverfahren auf die Erkenntnisse hin, die zum Teil aufgrund der Forderungen[21] nach der Verbannung der `Goto`-Ausdrücke gewonnen wurden.

*„Research into structured programming in the 60s and 70s led to techniques for translating unstructured flowcharts into structured ones.“* [ODHA07, S. 19]

Die in dieser Aussage angesprochenen Techniken referenzieren zum Beispiel ein von Ashcroft und Manna in [AsMa72] beschriebenes Vorgehen. Darin wird die prinzipielle Ersetzbarkeit von `Goto`-Ausdrücken durch `While`-Schleifen beschrieben. Die weitere Diskussion um das Für und Wider dieser Ausdrücke fand Anhänger auf beiden Seiten, wie Donald E. Knuth in „Structured Programming with go to statements“ beschreibt (siehe [Knut74, S. 4]):

**A consideration of several different examples sheds new light on the problem of creating reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without go to statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code. The discussion brings out opposing points of view about whether or not go to statements should be abolished; some merit is found on both sides of this question.**

**Abbildung 75: Donald E. Knuth über die Kontroverse von Goto Ausdrücken**

Diese Diskussion hat nunmehr über 40 Jahre angedauert und noch immer kein Ende gefunden. Manche Programmiersprachen, die prominenteste darunter ist Java, haben den `Goto`-Ausdruck verbannt. Die meisten Sprachen, dazu gehören unter anderen C, C++, C# und Ada95, lassen eine Verwendung weiterhin zu und überlassen den sorgsamen Umgang dem Programmierer. Die allgemeine Umformung eines Programms mit `Goto`-Ausdrücken in ein Programm mit `While`-Ausdrücken ist heute in Büchern zur theoretischen Informatik im Rahmen der Berechenbarkeitstheorie ein gängiges Thema (vgl. [Scho01, S. 104 ff.]).

#### 4.1.3.3 Verbindung zu BPMN und BPEL

Die *XOR-Gateways* (siehe 2.5.3.1.1) in BPMN lassen eine Modellierung von bedingten Sprüngen im Ablauf zu und sind durchaus mit dem `Goto`-Ausdruck vergleichbar. Mithilfe von *XOR-Gateways* lassen sich die Programmstrukturen `if / then / else`, `while / do`, `repeat / until` und `switch / case` darstellen (vgl. [ODHA07, S. 9]). Sofern keine Einschränkung von BPMN vorgenommen wird, lassen sich mit *XOR-Gateways* (ebenso wie mit einem `Goto`-Ausdruck) beliebige Abläufe (Arbitrary Cycles) modellieren. Eine Möglichkeit zur Transformation eines Modells mit beliebigen Abläufen in eine Sprache, die keine `Goto`-Ausdrücke erlaubt, stellt das Umschreiben zu einer `While`-Schleife dar, wie in 2.6.3.1 in ähnlicher Form von Gao demonstriert. Allerdings haben Ouyang et al. bei dem Umschreiben eine Grenze ausgemacht:

*“However, these techniques are no longer applicable when AND-splits and AND-joins are introduced.”* [ODHA07, S. 19]

[21] Außer Dijkstra wurde diese Forderung unter anderem von D. E. Knuth und R. W. Flloyd gestellt. Nach [Knut74, S. 4] geht die Diskussion erstmals auf P. Naur im Jahre 1963 zurück.

*AND-splits* und *AND-joins* (siehe 2.5.3.4) dienen der Erzeugung und der Synchronisation von parallelen Abläufen (Threads). In der damaligen Diskussion um die Verwendung von `Goto`-Ausdrücken war Parallelität kein Diskussionspunkt. Ohne die vergangene Diskussion neu aufzugreifen, wurde in [ODBH05] ein Verfahren vorgestellt, das eine Transformation von BPMN in BPEL unter der Berücksichtigung von bedingten Sprüngen in parallelen Threads durchführt. Das Verfahren wird durch das BPEL `<eventHandlers>` Konstrukt (siehe 2.4.5.12) ermöglicht:

*"When the message constituting an event arrives, the `<scope>` activity specified in the corresponding event handler is executed. Business processes are enabled to receive such messages concurrently with the normal activity of the scope to which the event handler is attached, as well as concurrently with other event handler instances. This allows such events to occur at arbitrary times and an arbitrary number of times while the corresponding scope (which may be the entire business process instance) is active."* [BPEL2.0, S. 140]

Die `While`-Schleife zur Vermeidung des `Goto`-Ausdrucks wurde also durch Multithreading erweitert, um der Anforderung der bedingten Sprünge in parallelen Abläufen nachzukommen.

### 4.1.4 Fazit

Die Analyse der bestehenden Verfahren hat gezeigt, dass mit BPMN überaus komplexe Abläufe darstellbar sind. Sie hat auch gezeigt, dass sich diese Strukturen, wie beispielsweise bedingte Sprünge in parallelen Abläufen, in nicht weniger komplexen Code transformieren lassen. Wenn diese Strukturen zur Verwendung zugelassen werden, so bleibt der „sorgsame Umgang" damit dem Modellierer überlassen, der sich der schweren Verständlichkeit und Wartbarkeit des erzeugten Codes bewusst sein muss. Es spielt dabei keine Rolle, ob der Modellierer ein technischer Entwickler, Business Analyst oder Manager ist. Ein tiefes Verständnis der Konsequenzen einer bestimmten Modellierungsweise muss vorhanden sein, um kritische Prozesse angemessen umsetzen zu können.
Eine Alternative zum sorgsamen Umgang mit den überaus komplexen Strukturen und `Goto`-ähnlichen Ausdrücken durch den Modellierer ist eine Einschränkung von BPMN, die deren Verwendung verbietet, beziehungsweise wie in der *strukturierten Programmierung* konkrete Strukturen statt dessen nahe legt. Dadurch lassen sich bestimme Teile in einem BPMN-Diagramm ziemlich direkt auf BPEL-Konstrukte abbilden, wofür ein Mustererkennungsverfahren dient. Der auf diese Weise erzeugte Code ist gut verständlich und ebenso gut wartbar. Diese Alternative hat jedoch den Nachteil, dass der Modellierer implizit dazu gezwungen ist, in den „terms of BPEL" zu arbeiten, statt von der „intuitive" Modellierung in BPMN profitieren zu können. Die Modellierung von Arbitrary Cycles wird ausgeschlossen; stattdessen werden `if / then / else` und `while`-Strukturen modelliert, jedoch nur implizit und unter Verwendung von BPMN Sprachelementen wie *XOR-Gateways*.
Die Analyse hat auch ergeben, dass durch Modelltransformationen zusätzliche Anforderungen an Modellierungswerkzeuge gestellt werden. Ein Management für den Lebenszyklus eines Modells wird notwendig, um Änderungen an einem Prozess reibungslos durchführen zu können. Ein derartiges Management erfordert entweder Lösungsansätze auf die bei einem Round Trip identifizierten Probleme, oder eine umfassende, wenn nicht sogar vollständige Unterstützung des Metamodells der Zielsprache der Transformation.

## 4.2 Alternative Herangehensweisen

BPEL wird in den in 1.3 beschriebenen Anforderungen als das der Modellierung zu Grunde liegende Metamodell sowie als Zielsprache für die Codegenerierung vorgegeben. Als Standard für die graphische Modellierung soll BPMN verwendet werden. Die Anforderung, für Menschen verständlichen (readable) BPEL-Code zu erzeugen, war nicht explizit angegeben. Die Ausführungen zur Readability in 2.6.1.3.1, 2.6.1.3.6 und 4.1.2.1 erlauben jedoch ohne Zweifel, dies als Anforderung mit aufzunehmen. Daraus ergeben sich zwei verschiedene Ansätze, die sich mit den in 3.3.3 ausgewählten Werkzeugen realisieren lassen. Diese werden im Folgenden beschrieben. Anschließend findet eine Auswahl durch die Gegenüberstellung von Vorteilen und Nachteilen statt.

### 4.2.1 Zwei mögliche Ansätze

Für den ersten Ansatz müssten bestimmte Erweiterungen und Einschränkungen an BPMN vorgenommen werden. Die graphische Modellierung findet mit einem BPMN-Modellierungswerkzeug statt, wobei BPEL-Code durch eine Modelltransformation erzeugt wird. Die Erweiterungen von BPMN sind unter Umständen notwendig, um alle Konstrukte des BPEL-Metamodells modellieren zu können. Die Einschränkungen von BPMN sind notwendig, damit nur Abläufe modelliert werden können, die dem BPEL-Metamodell entsprechen. Im Fazit der Analyse der bestehenden Ansätze zur Verbindung von BPMN und BPEL (siehe 4.1.4) wurde festgestellt, dass bestimmte Modelle in BPMN zu unverständlichem Code in BPEL führen. Die Einschränkung von BPMN ist eine Möglichkeit, die Erstellung derartiger Modelle zu verhindern. Für die Erzeugung von BPEL-Code müssten allerdings mehrere Modelltransformationen (vgl. 2.6.1.3) durchgeführt werden.

Auch für den zweiten Ansatz müssten bestimmte Erweiterungen und Einschränkungen an BPMN vorgenommen werden. Die graphische Modellierung findet jedoch mit einem BPEL-Modellierungswerkzeug statt, in dem die graphischen Elemente die BPEL-Konstrukte direkt abbilden und die Erzeugung des BPEL-Codes relativ direkt stattfindet. Die Darstellung aller Konstrukte des BPEL-Metamodells erfordert unter Umständen die Erweiterung von BPMN. Die Einschränkung von BPMN besteht in diesem Ansatz im Wesentlichen darin, dass BPMN-Elemente wie *Gateways* nicht frei platzierbar, sondern nur noch in Kombination (Split/Join, Fork/Merge) verwendbar sind. Eine Modelltransformation findet in diesem Sinne nicht statt. Die gegebenen BPEL-Konstrukte werden lediglich in der BPMN-Notation visualisiert.

Es wäre auch ein hybrider Ansatz denkbar, in dem BPMN um die strukturierten Elemente erweitert wird, die im zweiten Ansatz skizziert sind. Genau genommen ist dies allerdings kein eigenständiger Ansatz, sondern lediglich eine Verfeinerung des ersten Ansatzes.

### 4.2.2 Gegenüberstellung und Auswahl

Beide Ansätze erfordern sowohl die Erweiterung, als auch die Einschränkung von BPMN. Durch die Vermeidung der Modellierung beliebiger Abläufe kann der erste Ansatz dafür verwendet werden, aus einem bestehenden Modell einen für Menschen verständlichen BPEL-Code zu erzeugen (vgl. 3.2.2). Allerdings hat der erste gegenüber dem zweiten Ansatz durch diese Einschränkung keinen echten Mehrwert, da die Modellierung deswegen in beiden Ansätzen auf die in BPEL vorhandenen Strukturen beschränkt ist. Die Modelltransformation kann dem ersten Ansatz als nachteilig angekreidet werden, da diese eine Modellvalidierung erschwert (vgl. dazu 4.1.2.2, 4.1.2.5). Um Anpassungen an dem erzeugten Code nicht zu erfordern (vgl. 4.1.2.4, 4.1.2.5), muss in beiden Ansätzen das BPEL-Metamodell vollständig abgebildet werden. Die Anpassung eines BPEL-Modellierungswerkzeuges wie im zweiten Ansatz ist dafür von Vorteil, da dieses das Metamodell in der Regel bereits implementiert. Obwohl beide Ansätze BPMN als standardisierte Darstellung von Abläufen verwenden, können sie die Kluft zwischen Business und IT damit nicht schließen (vgl. 4.1.1.4). Sie ermöglichen jedoch eine standardisierte Darstellung von Prozessen mit allen damit verbundenen Vorteilen.

Die festgestellten Vorteile und Nachteile lassen keine eindeutige Aussage zu, ob einer der beiden Ansätze „besser“ geeignet ist. Der erste Ansatz wurde bereits in ähnlicher Form in [S:BPEL4Chor] umgesetzt (vgl. 3.2.2, [Pfit07]). Nach eingehender Überprüfung existiert für den zweiten Ansatz bisher keine quelloffene Umsetzung.
Eine Realisierung des zweiten Ansatzes könnte daher in vielerlei Hinsicht Erkenntnisgewinn bringen. Es könnte eine Evaluation vorgenommen werden mit dem Ziel der Klärung, bei welcher der beiden Herangehensweisen die Modellierung als verständlicher und „intuitiver“ im Sinne der Beschreibung in 4.1.1.4 wahrgenommen wird. Ebenso kann sie Aufschluss darüber geben, ob sich gegebene Abläufe dadurch wirklich einfacher kommunizieren lassen. Die Veröffentlichungen in dem Bereich der Visualisierung von BPEL durch BPMN (vgl. 2.6.2) sind bisher eher knapp und der Diskurs darüber gering. Dies hat zur Folge, dass jeder Hersteller eines BPEL-Modellierungswerkzeuges eine „eigene“ Visualisierung in BPMN bereitstellen müsste. Das Verfolgen dieses Ansatzes könnte eine Diskussionsgrundlage erbringen, wie eine standardisierte Darstellung von BPEL in BPMN konkret und im Detail aussehen könnte. Aus diesen Gründen wird in dieser Arbeit der zweite Ansatz theoretisch konzipiert und praktisch umgesetzt.

### 4.2.3 Auswahl der Technik

Für die Realisierung des graphischen Editors wird nach dem in 4.2.1 skizzierten und in 4.2.2 ausgewählten zweiten Ansatz ein BPEL-Modellierungswerkzeug angepasst. Die Open-Source Anwendung [S:EclipseBPEL] bietet von den durch Ausschlussverfahren verbleibenden Werkzeugen (vgl. 3.3.3) die größtmögliche Funktionalität für die graphische Modellierung und Visualisierung von BPEL-Prozessen. Diese Anwendung wird daher nach den in 1.3 beschriebenen Anforderungen angepasst und entsprechend erweitert. Dabei können möglicherweise Code-Teile für die Visualisierung von BPMN-Elementen aus [S:EclipseBPMN] verwendet werden. Sowohl [S:EclipseBPEL] als auch [S:EclipseBPMN] stehen unter der Eclipse Public License (siehe [EPL1.0]). Das geplante Vorgehen ist daher lizenzrechtlich zulässig. Die entstehende Anwendung wird, wie in [EPL1.0] beschrieben, ebenso unter die Lizenz [EPL1.0] gestellt.

## 4.3 Definition der graphischen Abbildung

In die Überlegungen, die in diesem Abschnitt präsentiert werden, sind mehrere unterschiedliche Ansätze aus verschiedenen Arbeiten eingeflossen, die zu den Themengebieten BPEL, BPMN und Workflow Patterns erstellt wurden. Beispielsweise stellt das in [BPMN1.1, S. 145 ff.] vorgestellte Mapping von BPMN zu BPEL bereits eine solide Arbeitsgrundlage dar. Ein tatsächlich implementierter Algorithmus mit anderen Ansätzen für ein Mapping wurde in [ODHA07] spezifiziert. Die Definition und Erforschung von Workflow Patterns in [AHKB03] und [RHAM06], sowie deren Anwendung auf BPMN in [WADH05] und auf BPEL in [WADH03], [Muly05] und [Kram06] liefern weitere Ansätze und Hilfestellungen für die Definition einer graphischen Abbildung von BPEL in BPMN. Welche dieser Arbeiten im Detail und an welcher Stelle der hier vorgestellten Visualisierung zum Tragen kommen, wird durch entsprechende Literaturverweise und Fundstellen gekennzeichnet.

### 4.3.1 Nähere Beschreibung des Ansatzes

Der Ansatz, der für die graphische Modellierung von BPEL-Prozessen unter Verwendung der BPMN-Notation verfolgt wird, sieht eine direkte Abbildung von BPEL-Konstrukten in BPMN-Strukturen vor. Dabei kommen keine Modelltransformationen zum Einsatz, sondern es wird lediglich eine bestimmte Visualisierung definiert. Eine angemessene Implementierung schafft die Möglichkeit, diese Visualisierung an- und auszuschalten. Auf diese Weise kann man ein Modellierungswerkzeug nicht nur auf die vertraute Art und Weise verwenden, sondern bei Bedarf auch in den BPMN-Visualisierungsmodus wechseln. Somit kann ein Prozessdesigner, der im Umgang mit einem bestimmten Tool geübt ist, dieses weiterhin auf die vertraute Art einsetzen. Die bestehenden graphischen BPEL-Modellierungswerkzeuge wie „WebSphere Integration Developer“, „Oracle BPEL Designer“ oder „ActiveBPEL Designer“ haben allerdings allesamt eine eigene Darstellungsform. Wenn sich alle an der Prozessmodellierung beteiligten Personen mit dieser Notation gut auskennen, ergibt sich dadurch kein Problem. Soll ein modellierter Prozess allerdings an Außenstehende oder mit dieser herstellerspezifischen Notation nicht vertraute Personen kommuniziert werden, so wird in den BPMN-Visualisierungsmodus gewechselt, der eine Art „Lingua Franca“ darstellt.
Die Definition der Visualisierung wird für jedes Konstrukt separat vorgenommen. Die einzelnen Visualisierungsfunktionen benötigen lediglich den Zugriff auf das eigentliche Konstrukt und auf die darin genesteten Konstrukte. Es wird also keine Annahme in Bezug auf den äußeren Verwendungskontext gemacht. Dies ist ein durchgängiges Prinzip in dem hier beschriebenen Ansatz.
Die eigentliche Modellierung orientiert sich nicht an den BPMN-Elementen, sondern an BPEL-Konstrukten. Daher können manche der dargestellten BPMN-Elemente nicht beliebig verbunden, platziert und verschoben werden, wie es in einem BPMN Modellierungswerkzeug gestattet wird. *Gateways* können beispielsweise nicht frei platziert werden. Wenn Abläufe modelliert werden, die semantisch ein *Gateway* erfordern, so wird dieses entsprechend dargestellt, sprich gezeichnet. Nachfolgend werden zwei Beispiele gezeigt, die den Ansatz illustrieren. Sie setzen Kenntnisse von Verfahren zur Visualisierung von Modellen voraus, die in 3.1.2.1 (EMF) und 3.1.2.2 (GEF) beschrieben sind. Kenntnisse von [S:Draw2d] (vgl. beispielsweise [Lee03]) sind von Vorteil, jedoch nicht zwingend erforderlich. Beispielhaft wird eine Abbildung der Aktivitäten `<invoke>` und `<if>` gezeigt. In diesen Beispielen sind bestimmte Schwierigkeiten ausgeklammert, auf die erst bei der eigentlichen Definition der Abbildung eingegangen wird.

#### 4.3.1.1 Beispiel 1 – <invoke>

Die Aktivität `<invoke>` (siehe 2.4.1.5) erlaubt es, in-line einen Compensation Handler (siehe 2.4.5.3) anzugeben:

```
<invoke name="Do" . . .>
   <compensationHandler>
      <invoke name="Undo" . . . />
   </compensationHandler>
</invoke>
```

**Listing 44: Beispiel 1 – <invoke> mit in-line Compensation Handler**

Eine `<invoke>` Aktivität ohne einen solchen Handler kann als *Task* (siehe 2.5.2.1) dargestellt und mit einem zusätzlichen *Marker* versehen werden, um die Art des *Tasks* näher zu spezifizieren. Bei einem in-line definierten `<compensationHandler>` legt die Spezifikation (vgl. [BPMN1.1, S. 24]) nahe, den *Task* mit einer *Compensation Association* (siehe 2.5.1.5, 2.5.5.2) zu dem kompensierenden *Task* darzustellen. Es handelt sich in BPEL jedoch noch immer um ein Konstrukt mit einer inneren Aktivität. Eine direkte Visualisierung in einem einfachen BPEL-Modellierungswerkzeug (siehe 3.2.4) ohne die Verwendung von BPMN könnte in etwa so aussehen, wie in der folgenden linken Abbildung dargestellt. Die Hervorhebung in der rechten Abbildung zeigt, dass bei dieser Darstellung bereits zwei Visualisierungsfunktionen involviert sind, die von `<invoke>` (rot) und die des `<compensationHandler>` (grün).

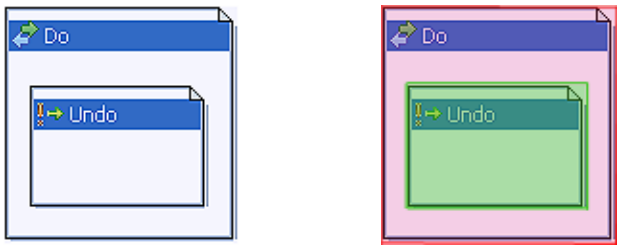


**Abbildung 76: Beispiel 1 – Direkte Visualisierung**

Ohne die Funktionalität des Werkzeugs zu verändern, kann die Visualisierungsfunktion der Konstrukte einen völlig anderen Eindruck von demselben Sachverhalt vermitteln. Sobald der `<compensationHandler>` („Undo“) in die `<invoke>` Aktivität („Do“) eingefügt wird, zeichnet die Visualisierungsfunktion des `<invoke>` Konstrukts die Aktivität auf eine andere Weise. Der äußere Rahmen wird nicht mehr gezeichnet. In die innere Fläche werden zwei Aktivitäten gezeichnet, die mit einer *Association Connection* verbunden sind. Eine der beiden gezeichneten Aktivitäten („Do“) existiert nur virtuell, denn „Do“ ist ja bereits das umgebende Konstrukt. Die andere Aktivität („Undo“) existiert substanziell. Daher wird sie von der eigenen Visualisierungsfunktion gezeichnet, nämlich von der des `<compensationHandler>` Konstrukts. Die virtuelle Aktivität kann daher nicht beliebig verbunden, platziert und verschoben werden. Die Darstellung von `<invoke>` mit dem in-line definierten `<compensationHandler>` könnte unter Verwendung der BPMN-Notation dann so aussehen:

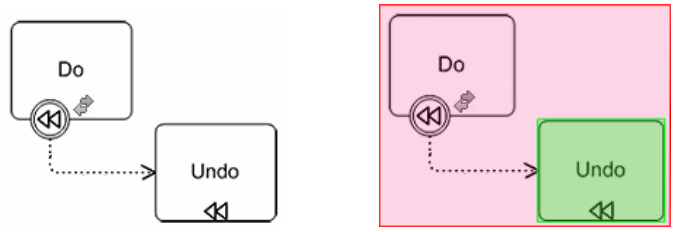


**Abbildung 77: Beispiel 1 – Visualisierung in BPMN**

#### 4.3.1.2 Beispiel 2 – <if>

Als zweites Beispiel wird die `<if>` Aktivität (siehe 2.4.2.3) betrachtet, mit der Verzweigungen dargestellt werden. Das folgende Listing zeigt eine `<if>` Aktivität. Wenn die Bedingung „b1“ erfüllt ist, wird „Invoke“ ausgeführt. Andernfalls wird der `<elseif>` Zweig geprüft. Ist die Bedingung „b2“ erfüllt, so wird die Aktivität „Receive“ gestartet. Für den Fall, dass keine der beiden Bedingungen erfüllt ist, werden keine Aktivitäten ausgeführt.

```
<if name="If">
  <condition>b1</condition>
  <invoke name="Invoke"/>
  <elseif>
     <condition>b2</condition>
     <receive name="Receive"/>
  </elseif>
  <else/>
</if>
```

**Listing 45: Beispiel 2 – Bedingte Verzweigungen mit <if>**

Das Modellierungswerkzeug [S:EclipseBPEL] (siehe 3.2.3) unterstützt die graphische Modellierung mit dieser Aktivität vollständig. Zu der eigentlichen `<if>`-Klausel, zu jedem `<elseif>` und zu `<else>` wird ein separater Zweig visualisiert, der durch Pfeile verbunden ist. Die Visualisierungsfunktion bezieht die Anzahl der genesteten Verzweigungen mit ein. Die in diesen Verzweigungen genesteten Aktivitäten haben wiederum eine eigene Visualisierungsfunktion, die für das Zeichnen der Aktivität zuständig ist. Diese Implementierung verfolgt denselben Gedanken und zeigt dadurch, dass der in dieser Arbeit verfolgte Ansatz mit den Technologien [S:EclipseEMF], [S:EclipseGEF] und [S:Draw2d] realisierbar ist. Die nachfolgende linke Abbildung zeigt die Visualisierung des obigen Codes in [S:EclipseBPEL], die rechte Abbildung beinhaltet die Hervorhebung der involvierten Visualisierungsfunktionen.

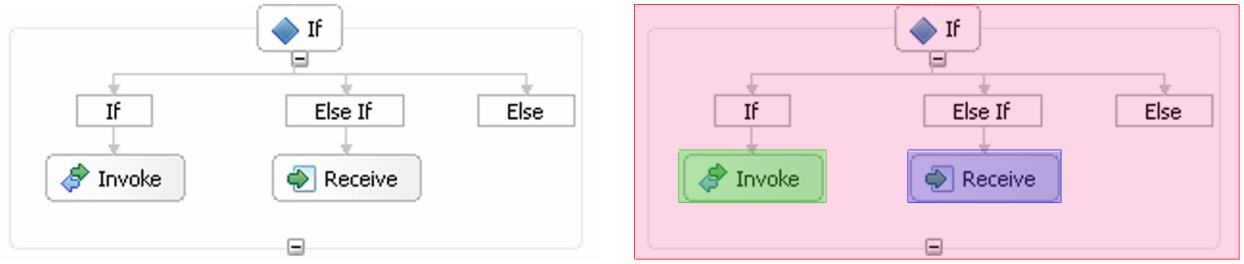


**Abbildung 78: Beispiel 2 – Visualisierung in [S:EclipseBPEL]**

In [ODHA07, S. 9] wird ein Pattern gezeigt, mit dem ein bestimmtes Muster in BPMN in ein BPEL `<if>`-Konstrukt transformiert werden kann. Dieses Pattern kann somit auch zur Visualisierung eines `<if>`-Konstrukts herangezogen werden, ohne dazu eine Modelltransformation zu erfordern. Ausgehend von der obigen graphischen Darstellung von `<if>`, müsste die Visualisierungsfunktion wie folgt geändert werden:
Der äußere Rahmen wird nicht mehr gezeichnet. An die Stelle des Rechtecks mit der blauen Raute („If“) wird ein *XOR-Gateway* (siehe 2.5.3.1.1) gezeichnet, welches mit den einzelnen Verzweigungen mit *Conditional Sequence Flow Connections* (Pfeile, siehe 2.5.4.1) verbunden ist. Der `<else>`-Zweig wird mit einer *Default Sequence Flow Connection* (siehe 2.5.4) gezeichnet. Die Verzweigungen werden mit einem auf der gegenüberliegenden Seite gezeichneten weiteren *XOR-Gateway* wieder zusammengeführt.

Es handelt sich also nicht um eine freie Modellierung in BPMN, denn die *Gateways* sind nicht beweglich und nur „in Kombination“ platzierbar. Die Kombination ist durch die Abbildung des BPEL-Konstrukts definiert. Diese Einschränkung kann als unvorteilhaft für die Modellierung ausgelegt werden. Sie hat jedoch auch gewisse Vorteile, denn dadurch ist die Erstellung von Modellen, die zu unverständlichem BPEL-Code führen würden, per se ausgeschlossen. Ebenso sind keine Modelltransformationen erforderlich, die ihrerseits mehrere Nachteile haben (vgl. 4.1.2, 4.1.4). Die nachfolgende linke Abbildung aus [ODHA07, S. 9] zeigt, wie die oben gelistete `<if>`-Aktivität visualisiert werden könnte. Die rechte Abbildung zeigt die Integration mehrerer Visualisierungsfunktionen, die rote Hervorhebung zeigt das `root`-Konstrukt.

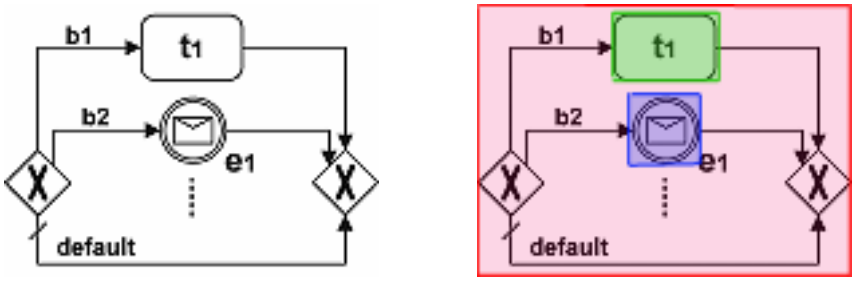


**Abbildung 79: Beispiel 2 – Visualisierung in BPMN**

### 4.3.2 Einschränkungen und Erweiterungen

In Abschnitt 2.2.3 wurde das Metamodell von BPMN beschrieben. Dort wurde aufgezeigt, dass BPMN nicht nur eine graphische Notation zur Darstellung von Geschäftsprozessen ist, sondern auch eine überaus komplexe Semantik beinhaltet. Diese Semantik muss bei einer Abbildung berücksichtigt werden, um eine eindeutige Interpretation derselben zu gewährleisten. Die Beschreibung eines Ansatzes zur Transformation von BPEL in BPMN in 2.6.2.2.1 und dessen Analyse in 4.1.2.3 werfen für die Abbildung jedoch ein Problem auf. Das Konzept der Dead-Path-Elimination (DPE) in BPEL-`<flow>` Konstrukten ist nicht ohne weiteres auf das Tokenkonzept in BPMN abbildbar. Wegen dieses Konzepts wird bei der Modellierung von bedingten Übergängen in BPMN (*Conditional Sequence Flow*, vgl. 2.5.4.1) nahe gelegt, einen Standardübergang zu definieren (*Default Sequence Flow*, vgl. 2.5.4.2), um damit eine Verklemmung (Deadlock) auszuschließen (vgl. [BPMN1.1, S. 26]). Für dieses Problem bestehen zwei mögliche Lösungen.

Die erste Lösung wäre, die DPE für die Modellierung auszuschließen. Durch Setzen des `<process>`-Attributs `suppressJoinFailure` auf `yes` wird eine mögliche Verklemmung dann durch einen Fault Handler behoben und das Tokenkonzept in BPMN nicht verletzt (vgl. [BPEL2.0, S. 108]).

Die zweite Lösung stellt die Erweiterung von BPMN um das Konzept der DPE dar. Für die Modellierung von BPEL-Prozessen ist die DPE ein wichtiger Bestandteil. Aus diesem Grund und aus der in 1.3 beschriebenen Anforderung, BPEL als Metamodell für die Modellierung einzusetzen, wird die Erweiterung von BPMN der Einschränkung von BPEL vorgezogen. Diese Erweiterung (siehe 4.4.2.1.3) ist bei der Abbildung der Konstrukte und deren Interpretation zu beachten.

Aber wie lässt sich BPMN um die DPE erweitern, und welche weiteren Anforderungen aus der Spezifikation sind davon betroffen? Ein *Conditional Sequence Flow*, der bei der Verwendung ohne einen *Default Sequence Flow* eine Verklemmung (Deadlock oder „Eternal Wait“) zur Folge haben kann, ist durch die Einführung der DPE per Definition von der Verklemmung ausgeschlossen. Die Auswirkungen der Erweiterung werden unter anderem bei der Abbildung des `<flow>` Konstrukts (siehe 4.4.2.1) näher erläutert. Diese führt für expliziten Tokenfluss eine neue Art von Tokens ein, so genannte „Anti-Tokens“, denn die Erweiterung von BPMN mit der DPE stellt eine explizite Verletzung der Spezifikation dar, wenn von implizitem Tokenfluss Gebrauch gemacht wird:

*“There MUST NOT be any implicit flow during the course of normal Sequence Flow (i.e., there should always be either Sequence Flow or a graphical indicator, such as an Intermediate Event to show all the potential paths of Tokens). An example of implicit flow is when a Token arrives at a Gateway, but none of the Gates are valid, the Token would then (implicitly) pass to the end of the Process, which occurs with some modeling notations.”* [BPMN1.1, S. 37]

Eine Einschränkung, die weniger die theoretische Definition der Abbildung betrifft, ist durch die Unterstützung der Technik und die Zeitbeschränkung gegeben. Das Werkzeug [S:EclipseBPEL] befindet sich zurzeit noch in Entwicklung und hat noch keine volle Unterstützung für die graphische Modellierung von [BPEL2.0]-Prozessen. Beispiele für die Unvollständigkeit sind die noch nicht umgesetzte Unterstützung abstrakter Prozesse, Extensions und Extension Activities.
Die Erweiterung des BPEL-Modellierungswerkzeuges aus 3.2.4 für eine BPEL 2.0 Compliance in [Schu07] hat gezeigt, dass Anpassungen an einem Modellierungswerkzeug eine komplexe und zeitaufwendige Aufgabe darstellen können. Eine vollständige Implementierung aller Visualisierungsfunktionen, die die Abbildungen realisieren, und damit ein Nachweis, dass all die konzipierten Abbildungen tatsächlich in diesem Werkzeug mit den damit verbundenen Technologien umsetzbar sind, kann nicht erbracht werden.
Aufgrund dessen jedoch die Definition der Abbildung auf diejenigen Konstrukte zu beschränken, zu denen ein Nachweis der Implementierung erbracht wurde, würde den praktischen Nutzen dieser Diplomarbeit erheblich schmälern. In 4.2.2 wurde aufgezeigt, dass ein Diskurs über eine standardisierte Verwendung des Standards BPMN in der graphischen Darstellung von BPEL bisher nicht in ausreichendem Maße geführt wird. Einen ersten Ansatz stellt eine mögliche Definition aller Abbildungen dar. Wenn diese Definition unter Verwendung der Erkenntnisse anerkannter Wissenschaftler und unter Berücksichtigung der Spezifikationen [BPMN1.1] und [BPEL2.0] stattfindet, so ist sie wissenschaftlich fundiert und diskussionsfähig. Nach dieser Argumentation stellt die Definition der Abbildungen unabhängig von einer vollständigen Implementierung einen eigenen Wert dar. Sie wird daher im folgenden Abschnitt vorgenommen. Eine Realisierung findet demnach ohne Anspruch auf die vollständige Umsetzung aller definierten Abbildungen in Kapitel 5 statt.

## 4.4 Abbildung der Konstrukte von BPEL 2.0

Die Reihenfolge der Definitionen orientiert sich an der Darstellung der BPEL-Konstrukte in Abschnitt 2.4. In den einzelnen Definitionen können Querverweise auftreten, um doppelte Definitionen zu vermeiden. Beispielsweise referenziert die Definition der `<invoke>` Aktivität die Definition des `<compensationHandler>`, da dieser innerhalb des Konstrukts verwendet werden kann. Die Visualisierungsfunktionen beziehen für eine optimale Erweiterbarkeit und Flexibilität den äußeren Kontext nicht mit ein (modular und composable). Dies bewirkt, dass die Abbildung einer Aktivität universell einsetzbar ist. Für ein besseres Verständnis sind in komplexen Abbildungen zusätzlich zu den möglichen Visualisierungen in BPMN auch hervorgehobene Visualisierungen aufgeführt. Das eigentliche Konstrukt wird mit der Farbe Rot hervorgehoben. Andere Farben referenzieren die Visualisierungsfunktionen genesteter Konstrukte. Ein Teil der Abbildungen ist unter der Voraussetzung definiert, dass die referenzierten Transformationsverfahren und Mappings symmetrisch sind. Wenn beispielsweise ein *Terminate-Event* eindeutig auf eine `<exit>`-Aktivität abgebildet wird, so gilt der Umkehrschluss unter Voraussetzung der Symmetrieeigenschaft.

### 4.4.1 Basic Activities

#### 4.4.1.1 <assign>

In [BPMN1.1, S. 35] wird beschrieben, dass *Assignments* Attribute von *Flow Objects* sind, dazu gehören *Events, Activities* und *Gateways*. Von der Semantik sind weder *Events* noch *Gateways* passend. Es bleiben daher ein *Task* (siehe 2.5.2.1) oder ein *Sub-Process* (siehe 2.5.2.2), um eine `<assign>` Aktivität darzustellen. In 2.4.1.1 wird zu `<assign>` angemerkt, dass darin beliebig viele Zuweisungen in Form von `<copy>` Konstrukten oder Erweiterungen enthalten sein können. Demnach bleiben zwei mögliche Visualisierungen.

1. Visualisierung als *Task*, optional mit einem *Marker* (linke Abbildung).
2. Visualisierung als *Sub-Process* (mittlere Abbildung), optional mit einem zusätzlichen *Marker*. Die enthaltenen `<copy>` Konstrukte und Erweiterungen in der *expanded* Darstellung (siehe 2.5.2.2, rechte Abbildung) werden als Sequenz von *Tasks* angezeigt, die mit *Sequence Flow Connections* (siehe 2.5.4) verbunden sind. Allerdings ist die Semantik von `<assign>` in Bezug auf die `<copy>`-Operationen atomar (vgl. 2.4.1.1), daher wird ein weiterer *Task* („persist“) in diese Sequenz eingefügt (dargestellt), der die Persistierung der `<copy>`-Operationen repräsentiert und die „alles-oder-nichts“ Semantik von `<assign>` unterstreicht. Die Darstellung von einem *Start-Event* und einem *End-Event* sind für diese Abbildung nicht erforderlich (vgl. [BPMN1.1, S. 37]). Die anderen Abbildungen in dieser Arbeit setzen den Einsatz dieser *Events* im Rahmen von *Sub-Processes* allerdings konsequent ein. Für eine durchgängig konsistente Abbildung müssten daher in der nachfolgenden Abbildung ein *Start-Event* mit „copy 1“ und ein *End-Event* mit „persist“ verbunden werden. Die Semantik ändert sich dadurch nicht.


**Abbildung 80: <assign> als Task (links) oder als Sub-Process (mitte, rechts)**

#### 4.4.1.2 <empty>

Die `<empty>` Aktivität (siehe 2.4.1.2) ist ein atomares Konstrukt, zu dem die Beschreibung der *Task Activity* in BPMN passt:

*"A Task is an atomic activity that is included within a Process. A Task is used when the work in the Process is not broken down to a finer level of Process Model detail."* [BPMN1.1, S. 65]

Eine `<empty>` Aktivität wird daher als *Task* visualisiert. Für *Tasks* gilt die Beschreibung in 2.5.2.1. Das bedeutet, sie können einen optionalen *Marker* und eine Beschriftung tragen. Die *Marker* für *Loop*, *Multi-Instance* und *Compensation* werden nicht verwendet. Selbst wenn die Aktivität im Rahmen von Compensation Handling (siehe auch 4.4.5.3) eingesetzt wird, wie beispielsweise in ähnlicher Form in 2.4.1.2, so hat ihre Visualisierungsfunktion aus Gründen der Erweiterbarkeit und Flexibilität keine Kenntnis über den Kontext, in dem sie eingesetzt wird.

#### 4.4.1.3 <extensionActivity>

Eine `<extensionActivity>` (siehe 2.4.1.3) enthält ein Unterkonstrukt, das eine Aktivität darstellt. Dieses kann atomar, aber auch komplex sein, abhängig davon, wie diese Erweiterung definiert ist. In der Verwendung als atomare Aktivität bietet sich dieselbe Argumentation wie bei der `<empty>`-Aktivität an (siehe 4.4.1.2), also die Visualisierung als *Task*. Bei einer komplexen `<extensionActivity>` muss die Visualisierung vom Hersteller definiert werden. Eine entsprechende Implementierung (siehe Kapitel 5) kann dies ermöglichen.

#### 4.4.1.4 <exit>

In der BPMN-Spezifikation wird in dem Mapping zu *BPEL4WS* (siehe [BPEL1.1]) vorgeschlagen, das *Terminate-End-Event* (siehe 2.5.1.9) auf die `<terminate>` Aktivität in BPEL abzubilden (vgl. [BPMN1.1, S. 149]). In der Standardisierung durch OASIS wurde diese Aktivität in `<exit>` (siehe 2.4.1.4) umbenannt. Die Beschreibung der Semantik von `<exit>` und dem *Terminate-Event* stimmen überein. Daher ist das *Terminate-End-Event* vorläufig die geeignete Abbildung. Die Verwendung eines *End-Event* anstatt eines *Intermediate-Event* birgt allerdings gewisse Schwierigkeiten bei der Integration in Structured Activities, auf die in 4.4.1.9 eingegangen wird. Um eine vollständige Korrektheit der Abbildung zu gewährleisten, wird BPMN durch ein *Throwing Intermediate Terminate Event* erweitert. Es hat dieselbe Semantik wie das *Terminate-End-Event*, kann jedoch ausgehende *Sequence Flow* Verbindungen haben. Da ein Prozess bei diesem Ereignis unmittelbar beendet wird, werden diese Verbindungen ohnehin nicht weiterverfolgt und die Semantik bleibt daher dieselbe.

**Abbildung 81: Darstellung von <exit>**

#### 4.4.1.5 <invoke>

In 4.3.1.1 wurde die Abbildung des `<invoke>`-Konstrukts (siehe 2.4.1.5) als Beispiel für den Ansatz in dieser Arbeit angeführt. Bei dieser beispielhaften Visualisierung wurde allerdings eine bestimmte Schwierigkeit aus Vereinfachungsgründen ausgeklammert. Die `<invoke>`-Aktivität kann in einen nahezu beliebigen Kontext eingebettet sein, in den sie mittels *Sequence Flow Connections* integriert ist. Für die Visualisierung dieser *Connections* ist es von erheblichem Vorteil, wenn ein bestimmter Punkt an einem äußeren Rand definiert werden kann, von dem derartige *Connections* ausgehen können. Diese Anforderung wurde in der beispielhaften Abbildung nicht berücksichtigt. Denn sie bietet keinen geeigneten äußeren Rand für ausgehende *Sequence Flow* Verbindungen. Die nachfolgend definierte Abbildung dieses Konstrukts geht auf diese Schwierigkeit ein.
Das Mapping in der BPMN-Spezifikation bildet einen *Task*, dessen *TaskType*-Attribut den Wert *Service* trägt, auf eine `<invoke>`-Aktivität ab. Allerdings kann eine `<invoke>`-Aktivität Konstrukte für eine Fehlerbehandlung sowie zur Kompensation beinhalten (vgl. 2.4.1.5). Die umgekehrte Abbildung, also von `<invoke>` auf einen *Task,* ist daher nur dann angemessen, wenn diese Konstrukte nicht in `<invoke>` enthalten sind. Die Visualisierung von einem `<compensationHandler>` oder von einem `<faultHandlers>`-Konstrukt kann durchaus größeren Raum einnehmen. Wenn ein `<invoke>` unter Berücksichtigung der Visualisierung dieser Handler einen geeigneten äußeren Rand für ausgehende und für eingehende *Sequence Flow* Verbindungen bieten soll, so muss dieses Konstrukt als *Sub-Process* dargestellt werden. Die Handler können innerhalb dieses *Sub-Process* dargestellt und graphisch von der Visualisierungsfunktion von `<invoke>` integriert werden.
Die zur Fehlerbehandlung definierbaren Konstrukte `<catch>` und `<catchAll>` (siehe 2.4.5.1, 2.4.5.2) sind in einem `<invoke>`-Konstrukt im Gegensatz zu einem `<process>` oder `<scope>` nicht in ein `<faultHandlers>`-Konstrukt eingebettet, sondern direkt in die `<invoke>`-Aktivität genestet (vgl. [BPEL2.0, S. 85]). Dieser Umstand erfordert eine besondere Implementierung, um dennoch die Visualisierungsfunktion von `<faultHandlers>` nutzen zu können. Die Visualisierungsfunktion von `<compensationHandler>` ist hingegen nativ und ohne besondere Implementierung integrierbar.
In einer Darstellung als *Sub-Process* sind ein explizites *Start-Event* und *End-Event* nicht zwingend erforderlich (vgl. [BPMN1.1, S. 37]). Die Abbildungen in dieser Arbeit setzen diese *Events* im Zusammenhang mit *Sub-Processes* generell ein. Konsequenterweise werden sie in der Abbildung von `<invoke>` auf einen *Sub-Process* ebenso eingesetzt.
Nach dieser Argumentation wird eine `<invoke>`-Aktivität auf einen einfachen *Task* (siehe 2.5.2.1) abgebildet, sofern weder Konstrukte zur Fehlerbehandlung noch zur Kompensation darin definiert sind. Sind diese angegeben, so wird das Konstrukt auf einen *Sub-Process* abgebildet, wie die nachfolgende Darstellung zeigt. In die Visualisierung sind die Funktionen von `<invoke>` (rot), `<faultHandlers>` (grün) und `<compensationHandler>` (blau) eingebunden.

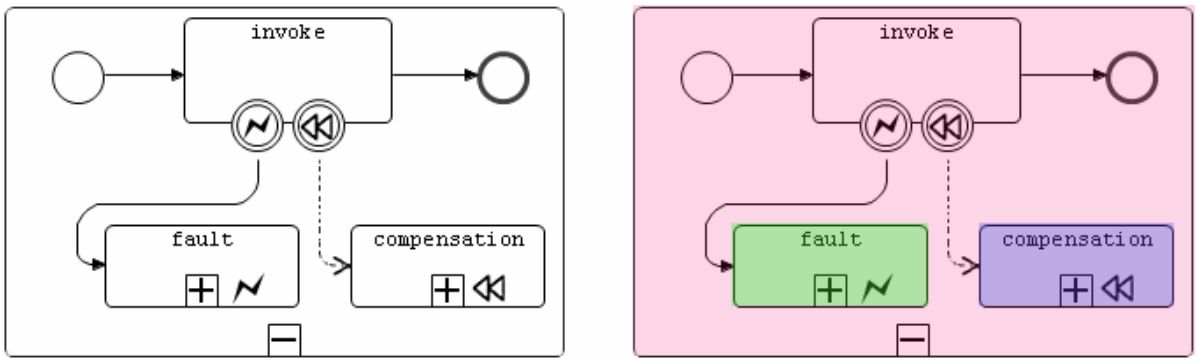


**Abbildung 82: Abbildung einer <invoke>-Aktivität mit genesteten Handlern**

#### 4.4.1.6 <receive>

In dem Mapping, das in [BPMN1.1] spezifiziert ist, wird ein *Catching Message-Event* (siehe 2.5.1.1) auf die `<receive>` Aktivität (siehe 2.4.1.6) abgebildet. Wenn es sich genau genommen um ein *Start-Event* handelt, so wird das Attribut `createInstance` auf `yes` gesetzt. Handelt es sich um ein *Intermediate-Event*, ist der Wert dieses Attributs `no` (vgl. [BPMN1.1, S. 148, 150]). Es erscheint einleuchtend, diese Abbildung auch in umgekehrter Richtung entsprechend zu definieren. Eine `<receive>` Aktivität wird auf ein *Catching Message-Event* abgebildet. Wenn der Wert des Attributs `createInstance` auf `yes` gesetzt ist, wird es als *Start-Event* (siehe 2.5.1) gezeichnet. Ist der Wert `no`, wird es als *Intermediate-Event,* also mit doppelter Umrandung gezeichnet.
Damit diese Abbildung in jedem Kontext korrekt ist, müssen allerdings zwei Probleme angesprochen werden. In 2.5.4.1 wird *der Conditional Sequence Flow* beschrieben. Dabei ist angegeben, dass *Events* nicht der Ursprung einer *Conditional Sequence Flow* Verbindung sein dürfen. Diese Verbindung entspricht der Semantik eines `<link>` (siehe 2.4.2.1.1) mit einer `<transitionCondition>` (siehe 2.4.2.1.2) in einem BPEL-`<flow>` (siehe 2.4.2.1). Um der Spezifikation gerecht zu werden, müsste die `<receive>`-Aktivität in diesem Zusammenhang als *Task* dargestellt oder als *Message-Event* mit einem nachfolgenden *XOR-Gateway* abgebildet werden, was derselben Semantik wie eine ausgehende *Conditional Sequence Flow* Verbindung entspricht. Eine in jedem Fall korrekte Abbildung wäre die Darstellung als *Task*.
Eine Begründung dafür, dass ein *Event* keine ausgehende *Conditional Sequence Flow* Verbindung haben darf, sucht man in der Spezifikation von *Events* und *Sequence Flow Connections* vergebens. Es kann jedoch anhand des Tokenkonzepts (siehe 2.2.3) erklärt werden. In der gemeinsamen Verwendung beider Konstrukte bleiben nämlich *Events* möglicherweise unbehandelt, da ein Token „stecken bleiben kann“. Durch die Erweiterung von BPMN um die Dead-Path-Elimination (vgl. 4.3.2) können diese Situationen allerdings nicht auftreten, denn *Conditional Sequence Flow* Verbindungen werden nur in einem `<flow>` für die freie Modellierung zugelassen. Infolgedessen kann die Anforderung, keine *Conditional Sequence Flow* Verbindungen mit einem *Event* als Quelle zuzulassen, durch die Erweiterung von BPMN mit der DPE (siehe auch 4.3.2 und 4.4.2.1) als nichtig betrachtet werden.
Daher wird eine `<receive>` Aktivität auf ein *Catching Message-Event* abgebildet. Unabhängig von dem Wert des Attributs `createInstance` wird es als *Intermediate-Event* (siehe 2.5.1) gezeichnet. Eine Unterscheidung, ob die `<receive>`-Aktivität zum Start von Prozessen eingesetzt wird oder nicht, ist unpraktikabel. In der Abbildung der Aktivität `<pick>` wird dieser Umstand näher erläutert (siehe 4.4.2.4).

**Abbildung 83: Darstellung von <receive>**

#### 4.4.1.7 <reply>

Für die Aktivität `<reply>` (siehe 2.4.1.7) wird die Abbildung analog zu der Definition und Argumentation der `<receive>` Aktivität (siehe 4.4.1.6) vorgenommen. Das Mapping in [BPMN1.1, S. 150] gibt an, dass ein *Throwing Message-Event* (siehe 2.5.1.1) entweder auf eine `<reply>` oder eine `<invoke>` Aktivität abgebildet wird. Das genaue Mapping bezieht die Information mit ein, ob eine Beziehung zu einer zuvor eingegangenen Nachricht besteht. Ist dies der Fall, dann wird das *Event* in ein `<reply>` abgebildet, ansonsten in ein `<invoke>`. Eine `<reply>`-Aktivität wird daher in ein *Throwing Intermediate Message-Event* abgebildet.

**Abbildung 84: Darstellung von <reply>**

#### 4.4.1.8 <rethrow>

Die Abbildung von <rethrow> (siehe 2.4.1.8) baut auf die Definition der Abbildung der <throw>-Aktivität auf (siehe 4.4.1.9). Diese wird auf ein *Throwing Intermediate Error-Event* (siehe [BPMN1.0, S. 45]) abgebildet. Die Bedeutung von <throw> und <rethrow> unterscheidet sich eigentlich nicht. Beide Aktivitäten dienen dazu, einen Fehler zu generieren, der an den übergeordneten Fault Handler übergeben wird. Allerdings darf <throw> nicht innerhalb eines Fault Handlers verwendet werden. Dafür ist die Aktivität <rethrow> bestimmt.

**Abbildung 85: Darstellung von <rethrow>**

#### 4.4.1.9 <throw>

In [BPMN1.1, S. 149] wird das *Throwing End Error-Event* (siehe 2.5.1.3) auf die BPEL-Aktivität <throw> (siehe 2.4.1.9) abgebildet. Beide Konstrukte dienen der Generierung eines Fehlers, der an einen Fault Handler (siehe 2.4.5.7, 4.4.5.7) übergeben wird. Unter Voraussetzung der Symmetrie dieses Mappings, die angesichts der gleichen Bedeutung unterstellt werden kann, wird vorläufig definiert, dass eine <throw>-Aktivität mit einem *Throwing End Error-Event* dargestellt werden kann. Die Semantik von BPMN gibt vor, dass ein *End-Event* keine ausgehenden *Sequence Flow* Verbindungen haben darf (vgl. [BPMN1.1, S. 40]). Dies erschwert allerdings die Einbindung in andere Konstrukte wie <pick>, <if> oder <flow>, die mit *Gateways* ein geschlossenes System bilden. In der Vorversion (vgl. [BPMN1.0, S. 45]) war dieses *Event* auch in einer *Throwing Intermediate*-Version verfügbar, dieses wurde in der Version [BPMN1.1] jedoch entfernt. Ein *Throwing Intermediate Error-Event* wäre eine semantisch korrekte und saubere Lösung, die keine Sonderbehandlung in den umgebenden Konstrukten erfordert. Es wird die unerlaubte Verwendung, sprich Erweiterung, des in [BPMN1.0] gezeigten und in [BPMN1.1] entfernten *Throwing Intermediate Error-Event* einer Sonderbehandlung vorgezogen.

**Abbildung 86: Darstellung von <throw>**

#### 4.4.1.10 <wait>

In [BPMN1.1, S. 151] wird ein *Intermediate Timer-Event* (siehe 2.5.1.2) auf eine <wait> Aktivität (siehe 2.4.1.10) abgebildet. Zu *Timer-Events* werden auch noch andere Mappings definiert. Dies liegt darin begründet, dass ein *Timer-Event* auch einen Prozess starten kann oder eine Ausnahmebehandlung bewirkt, wenn es an eine *Activity* angeheftet wird. Das Anheften eines *Events* an eine Aktivität in der Modellierung ist eine Funktionalität, die allein durch die graphische Visualisierung nicht möglich ist. Wenn die <wait> Aktivität als *Intermediate Timer-Event* gezeichnet wird, so ist sie dennoch eine normale Aktivität. Sie kann daher in einer <sequence>, einem <flow> oder einer anderen Structured Activity verwendet werden. Sie kann jedoch nicht dazu verwendet werden, einen Prozess zu starten oder andere Aktivitäten durch Ausnahmebehandlung bei einer Zeitüberschreitung zu erweitern. Die graphische Visualisierungsfunktion erweitert BPEL nicht. Bei der Verwendung von <wait> in einem <flow> zeigen sich dieselben Einschränkungen wie in der Abbildung von <receive> (vgl. 4.4.1.6). Nach derselben Argumentation ist die Verwendung eines *Events* zur Darstellung jedoch zulässig. Die Aktivität <wait> wird somit auf ein *Intermediate Timer-Event* abgebildet.

**Abbildung 87: Darstellung von <wait>**

### 4.4.2 Structured Activities

#### 4.4.2.1 <flow>

Eine Abbildung der `<flow>` Aktivität (siehe 2.4.2.1) ist anhand der zur Verfügung stehenden Literatur nicht vollständig ableitbar. Die *Flattening* Strategie (vgl. 2.6.2.2.1) ist zu unpräzise und bezieht zudem wesentliche Elemente eines BPEL-`<flow>` nicht mit ein. In diesem Verfahren wird sowohl vor als auch nach einer Aktivität ein *Parallel Gateway* eingefügt (*AND*, siehe 2.5.3.4). `<link>`s werden in *Sequence Flow* Verbindungen transformiert und mit den *Gateways* verknüpft. Die Spezifikation [BPMN1.1] liefert für diese Aktivität ebenso kein geeignetes Mapping (vgl. 2.6.1.1.2).
In umgekehrter Richtung, also von BPMN zu BPEL, gehen die Transformationsverfahren von einer wie in der *Flattening* Strategie erzeugten Struktur als Basis aus. In 2.6.1.3.5 wird mit der *Generalised Flow-Pattern-Based Translation* nahezu derselbe Ansatz in umgekehrter Richtung verfolgt. Es wird eine Struktur vorausgesetzt, die nur aus (reduzierten) *Tasks* und *Parallel Gateways* besteht. Zusätzlich dürfen mehrere *Sequence Flow* Verbindungen nur zwischen *Gateways* bestehen, denn „uncontrolled flow“ (siehe 2.2.3) ist ausgeschlossen (vgl. 2.6.1.3.2).

Daher muss ein eigener Ansatz vorgestellt werden, der einen BPEL-`<flow>` vollständig in BPMN abbildet. Dieser Ansatz muss alle Konstrukte eines BPEL-`<flow>` einbeziehen:

- Eine `<transitionCondition>` (siehe 2.4.2.1.2) beschreibt einen bedingten Übergang, der in BPMN mit einer *Conditional Sequence Flow* Verbindung (siehe 2.5.4.1), einem *XOR-Gateway* (siehe 2.5.3.1.1) oder einem *OR-Gateway* (siehe 2.5.3.2) dargestellt werden kann.
- Eine `<joinCondition>` (siehe 2.4.2.1.3) beschreibt eine Bedingung für die Ausführung einer Aktivität. Dazu werden die eingehenden `<link>`s betrachtet. Ist sie nicht definiert, so wird implizit eine OR-Verknüpfung angenommen. Eine Auswertung findet statt, sobald alle eingehenden `<link>`s evaluiert sind. Dies lässt sich durch ein *OR-Gateway* (siehe 2.5.3.2) beschreiben. Wenn eine `<joinCondition>` definiert ist, kann diese einen beliebigen logischen Ausdruck darstellen, der durch ein *Complex Gateway* dargestellt werden kann.
- `<link>`s (siehe 2.4.2.1.1) können die Grenzen von nicht-wiederholenden Structured Activities überschreiten. Der in 4.3.1 beschriebene Ansatz sieht vor, dass aus Gründen der Erweiterbarkeit und Flexibilität die Visualisierungsfunktionen keine Kenntnis über den Kontext haben, in dem sie eingesetzt werden. Die Schwierigkeit besteht also in dem Zeichnen von `<link>`s über die Grenzen von Visualisierungsfunktionen hinweg. Es wäre erforderlich, dass eine Visualisierungsfunktion Annahmen über eine andere Funktion macht. Das *Link-Event* (siehe 2.5.1.7) stellt in dieser Situation auf den ersten Blick (und nur auf den ersten Blick) den Ausweg dar. Damit können *Sequence Flow* Verbindungen symbolisch verbunden werden. Diese strikte Trennung der Visualisierungsfunktionen hat jedoch auch einen Nachteil. Die Abbildungen der Structured Activities müssen ihrerseits eine Visualisierung von `<link>` Elementen definieren. Diese ist aber bei weitem nicht so komplex wie die von `<flow>`. Zudem lässt eine geeignete Implementierung möglicherweise eine Wiederverwendung zu. Das *Link-Event* ist allerdings bei einer näheren Betrachtung der Abbildung von `<scope>` problematisch (siehe 4.4.3.3). Darin wird festgestellt, dass ein `<scope>` mit einem *Sub-Process* abgebildet werden muss. Aktivitäten in einem `<scope>` können allerdings sowohl eingehende als auch ausgehende `<link>`s beinhalten, die die Grenzen des `<scope>` und somit die des *Sub-Process* überschreiten. Diese Konstellation ist nach [BPMN1.1, S. 30] nicht zulässig. In [BPMN1.1, S. 129] wird gezeigt, wie sich dieses Problem mit einem *Signal-Event* lösen lässt und laut der Spezifikation das Pattern #18 – Milestone – beschreibt (vgl. 2.3.3). Ein *Signal-Event* ist daher einem *Link-Event* vorzuziehen.

#### 4.4.2.1.1 Visualisierungsfunktion

Der Ablauf der Abbildung einer `<flow>`-Aktivität wird im Folgenden in Form zweier BPMN-Diagramme beschrieben. Anschließend wird die Visualisierung in Pseudocode definiert, welcher danach erläutert wird.

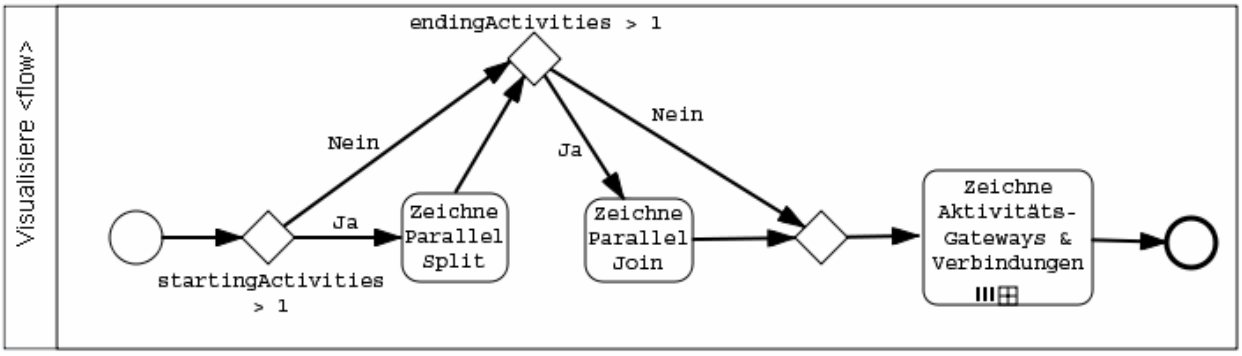


**Abbildung 88: Ablauf der Visualisierung von <flow>**

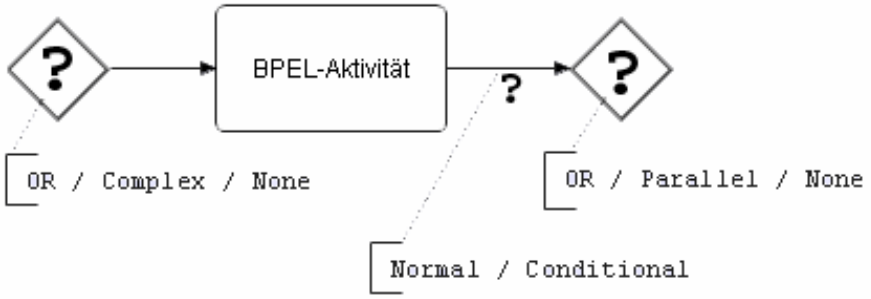


**Abbildung 89: Eingangs-Gateway, Flow Connection und Ausgangs-Gateway**

```
//Funktion zur Visualisierung eines <flow>
//Erläuterungen:
//„direkt genestet" bedeutet, dass sich eine Aktivität eine Ebene unter dem <flow>-
//Element in XML Darstellung befindet. Beispielsweise ist eine Aktivität, die in
//einer <pick> Aktivität enthalten ist, im <flow> nicht „direkt genestet"

startingActivities := Anzahl der im <flow> direkt genesteten
                      Aktivitäten ohne eingehende <link>s
endingActivities := Anzahl der im <flow> direkt genesteten
                    Aktivitäten ohne ausgehende <link>s


//Zeichne den äußeren Rahmen um den <flow>
if (startingActivities > 1) then
   Zeichne Parallel Gateway an den Anfang des <flow>
end if

if (endingActivities > 1) then
   Zeichne Parallel Gateway an das Ende des <flow>
end if

//Integration der inneren Aktivitäten in den <flow>
for each (direkt genestete Aktivität in <flow>) in Reverse
   //"in Reverse" bedeutet die umgekehrte Richtung der Kontrollabhängigkeit
   //vgl. Control Dependency, [BPEL2.0, S. 133]

   int incomingArcs := Anzahl der <link>s, die in die Aktivität eingehen
   int outgoingArcs := Anzahl der <link>s, die von der Aktivität ausgehen
```

```
int outgoingInternalArcs := Anzahl der <link>s, die von der Aktivität ausgehen
                            und als Ziel eine direkt genestete Aktivität haben
int outgoingExternalArcs := Anzahl der <link>s, die von der Aktivität ausgehen
                            und als Ziel eine Aktivität haben, die nicht direkt
                            genestet ist, sich also unterhalb einer Structured
                            Activity befindet
boolean hasJoinCondition := true, wenn <joinCondition> explizit angegeben,
                            sonst false
boolean hasTransitionCondition //Wird im Programmablauf berechnet; Ist true, wenn
                               //mindestens einer der ausgehenden <link>s eine
                               //<transitionCondition> definiert hat

//Bestimmung des Eingangs-Gateways der Aktivität
if (incomingArcs = 0) then
   //Erste Aktivität in einem <flow>, Kein Eingangs-Gateway
   if (startingActivities > 1) then
      Verbinde Aktivität direkt mit dem Anfang des <flow>, Ziel ist diese Aktivität
   else
      //Start-Aktivität
      Platziere Aktivität direkt an den Anfang des <flow>
   end if
elseif (incomingArcs = 1) then
   //Nur ein eingehender <link>, kein Eingangs-Gateway
elseif (hasJoinCondition = true) then
   //Eingangs-Gateway steht fest: Complex (Join)
   Zeichne Complex-Gateway als Eingangs-Gateway
else
   //Eingangs-Gateway steht fest: OR (Join)
   Zeichne OR-Gateway als Eingangs-Gateway
end if

//Bestimmung des Ausgangs-Gateways
if (outgoingArcs = 0) then
   //Letzte Aktivität in einem <flow>, kein Ausgangs-Gateway
   if (endingActivities > 1) then
      Verbinde Aktivität direkt mit dem Ende des <flow>, Quelle ist diese Aktivität
   else
      //End-Aktivität
      Platziere Aktivität direkt an das Ende des <flow>
   end if
elseif (outgoingArcs > 1) then
   //Prüfe, ob mindestens eine <transitionCondition> existiert
   hasTransitionCondition = false
   for each (<link> in ausgehende <link>s)
      if (link hat <transitionCondition>) then
         hasTransitionCondition = true
         exit for
      end if
   next
   if (hasTransitionCondition = true) then
      //Ausgangs-Gateway steht fest: OR (Split)
      Zeichne OR-Gateway als Ausgangs-Gateway
   else
      //Ausgangs-Gateway steht fest: Parallel (Split)
      Zeichne Parallel-Gateway als Ausgangs-Gateway
   end if
else
   //Ein ausgehender <link>, dieser wird später gezeichnet
end if

//Zeichnen der eingehenden externen Verbindungen
//Eingehende <link>s von Aktivitäten, die sich innerhalb einer direkt genesteten
//Structured Activity (<if>, <pick>, <sequence>, ...) befinden. <link>s von
//diesen Aktivitäten werden als externe <link>s bezeichnet.
//Realisiert wird dies mit einem Signal-Event, um volle Flexibilität und
//Erweiterbarkeit zu gewährleisten. Dadurch muss eine genestete Structured
//Activity nicht Context-Aware sein und ist beliebig austauschbar
//Ein Signal-Event hat für diese Situation die korrekte Semantik
for each (externer eingehender <link>) in <link>s
   Zeichne ein Catching Intermediate Signal-Event und verbinde es mit dem Eingang
   dieser Aktivität, Quelle ist das Event, die Verbindung ist Normaler Sequence
   Flow
next

//Zeichnen der ausgehenden externen Verbindungen
//Ausgehende <link>s zu Aktivitäten innerhalb einer Structured Activity (s.o.)
if (outgoingExternalArcs = 1) then
   //Ein ausgehender <link>, entweder normaler oder Conditional Sequence Flow
   if (<link> hat <transitionCondition>) then
```

```
         Zeichne ein Throwing End Signal-Event und verbinde es mit Conditional
         Sequence Flow mit dem Ausgang dieser Aktivität, Quelle ist diese Aktivität
      else
         Zeichne ein Throwing End Signal-Event und verbinde es mit normalem Sequence
         Flow mit dem Ausgang dieser Aktivität, Quelle ist diese Aktivität
      end if
   else
      for each (grenzüberschreitender ausgehender <link>) in <link>s
         Zeichne ein Throwing End Signal-Event und verbinde es mit dem Ausgang dieser
         Aktivität, Quelle ist diese Aktivität;
      next
   end if

   //Zeichnen ausgehender <link>s von Aktivitäten auf der ersten Ebene
   if (outgoingInternalArcs = 1) then
      //Ein ausgehender <link>, entweder normaler oder Conditional Sequence Flow
      if (<link> hat <transitionCondition>) then
         Zeichne <link> als Conditional Sequence Flow, vom Ausgang dieser Aktivität
         zum Eingang des Ziels, Quelle ist diese Aktivität
      else
         Zeichne <link> als normalen Sequence Flow, vom Ausgang dieser Aktivität zum
         Eingang des Ziels, Quelle ist diese Aktivität
      end if
   else
      for each (gewöhnlicher <link>) in <link>s
         Zeichne <link> vom Ausgang (Gateway) dieser Aktivität zum Eingang des Ziels,
         Quelle ist diese Aktivität
      next
   end if

next //Nächste Aktivität im <flow> in umgekehrter Reihenfolge der Kontrollabhängigkeit
```

**Listing 46: Pseudocode der Visualisierung von <flow>**

#### 4.4.2.1.2 Erläuterung der Visualisierungsfunktion

Die Visualisierungsfunktion ist so aufgebaut, dass sie eine optimale Lesbarkeit in der BPMN-Darstellung ermöglicht. Dazu wurde die geradlinige Herangehensweise Stück für Stück verfeinert.

- Auf geradlinige Weise lässt sich ein `<flow>` dadurch abbilden, dass vor und nach jeder Aktivität ein *OR-Gateway* gezeichnet wird. Quelle und Ziel der `<link>`s (*Sequence Flow Connections*) sind jeweils die Eingangs- und Ausgangs-Gateways der Aktivität. Um dieses Konstrukt werden *Parallel Gateways* gezeichnet, welche die *Gateways* der Aktivitäten ohne eingehende, beziehungsweise ohne ausgehende `<link>`s verbinden. Dies macht einen `<flow>` allerdings recht unübersichtlich. In jedem Fall wird ein *Gateway* gezeichnet, selbst wenn nur ein einziger `<link>` ein- oder ausgeht.

- Als erste Verfeinerung wird bei Aktivitäten nur dann ein eingehendes *Gateway* gezeichnet, wenn mehr als ein eingehender `<link>` existiert. Analog gilt dies für das ausgehende *Gateway*. Dadurch können manche Aktivitäten ohne ein dazwischen liegendes Gateway verbunden werden. Zum Zeitpunkt der Herstellung der Verbindung sollte bereits bekannt sein, ob sich vor einer Aktivität ein Gateway befindet. Daher verläuft die Schleife in umgekehrter Reihenfolge der Kontrollabhängigkeit.

- Wenn an einer Aktivität nur ein ausgehender `<link>` besteht, so kann eine Fallunterscheidung optisch nützlich sein. Wenn auf dem `<link>` eine `<transitionCondition>` definiert wurde, so eignet sich die Darstellung als *Conditional Sequence Flow* Verbindung (siehe 2.5.4.1) anstatt einer normalen *Sequence Flow* Verbindung.

- Die nächste Verfeinerung besteht darin, dass die „umklammernden“ *Parallel Gateways* nur dann benötigt werden, wenn es mehr als eine Aktivität ohne eingehende beziehungsweise ohne ausgehende `<link>`s gibt.
- Als weitere Verbesserung kann der Einsatz von *Complex-Gateways* bezeichnet werden. Sie dienen als Eingangs-*Gateway* der optischen Unterscheidung, ob eine `<joinCondition>` gesetzt wurde oder nicht. Sie stellen semantisch zudem die Situation einer expliziten `<joinCondition>` angemessener dar als ein *OR-Gateway.* Denn bei diesem wird lediglich auf mögliche Tokens gewartet. Eine `<joinCondition>` kann hingegen auch erfüllt sein, wenn alle Pfade als `false` ausgewertet sind.
- Auch bei den Ausgangs-*Gateways* kann noch eine optisch bessere Lesbarkeit und Eindeutigkeit des modellierten Ablaufs erzielt werden. *Parallel Gateways* werden statt der *OR-Gateways* in derjenigen Situation gezeichnet, in der mehrere `<link>`s von einer Aktivität ausgehen und keiner der ausgehenden `<link>`s eine `<transitionCondition>` besitzt. Damit ist eindeutig unterscheidbar, ob es sich um bedingte oder absolute Parallelität handelt. Denn graphisch sind *Conditional Sequence Flow* Verbindungen von normalen *Sequence Flow* Verbindungen – wenn sie von einem *Gateway* ausgehen – nicht unterscheidbar (vgl. 2.5.4.1).
- Die letzte Anpassung ist auf den ersten Blick keine offensichtliche Verbesserung. `<link>`s zu Aktivitäten, die in Structured Activities enthalten sind, werden durch ein *Signal-Event* angebunden. Der erste Gedanke könnte sein, dass dies unübersichtlich wirkt. Eine direkte Verbindung wäre besser lesbar. Es gibt jedoch zwei Gegenbeispiele, die zeigen, dass dies eine adäquate Vorgehensweise ist. Die Abbildung der Aktivität `<if>` (siehe 4.3.1.2, 4.4.2.3) sieht vor, die in den Zweigen enthaltenen Aktivitäten verbunden mit einer *Sequence Flow* Verbindung darzustellen. Angenommen, eine dieser Aktivitäten hat einen (genau einen) eingehenden `<link>`. Eine direkte Herangehensweise würde dann eine *Sequence Flow* Verbindung von der externen zu dieser enthaltenen Aktivität darstellen, wohlgemerkt ohne *Gateway*. Es ist schließlich nur ein einziger `<link>` vorhanden. Wenn die `<if>` Aktivität nun eine weitere *Sequence Flow* Verbindung zu dieser Aktivität zeichnet, so stellt diese Situation einen „uncontrolled flow“ dar, der eine völlig andere Semantik hat (vgl. 2.2.3). Daher müsste zur Korrektheit stets ein *Gateway* vor eine solche Aktivität gezeichnet werden. Dies geschieht in den Visualisierungsfunktionen der Structured Activities jedoch nur, wenn es in deren innerem Kontext angemessen ist. Für eine bestmögliche Flexibilität und Erweiterbarkeit bleibt es den Visualisierungsfunktionen der Konstrukte überlassen, wie sich ein `<link>` am besten integrieren lässt. Dies kann als *Catching* oder *Throwing Signal-Event* sein, es könnte möglicherweise auch eine völlig andere Darstellung gewählt werden. Das zweite Gegenbeispiel ist die Verwendung von *Sub-Processes* in der Abbildung von Konstrukten. Eine *Sequence Flow* Verbindung darf die Grenzen eines *Sub-Process* nicht überschreiten. Dadurch sind die Visualisierungsfunktionen erheblich eingeschränkt. Die Abbildung von `<scope>`s (siehe 4.4.3.3) wäre unter Berücksichtigung von darin definierten Handlern mit dieser Einschränkung sogar unmöglich oder falsch.

#### 4.4.2.1.3 Dead-Path-Elimination und Anti-Tokens

Durch <link>s in <flow>s in Verbindung mit der Dead-Path-Elimination (DPE, siehe 2.4.2.1.4) kann impliziter Fluss von Tokens (siehe für Tokens 2.2.3) entstehen. Um darzustellen, wo in einem Prozess dieser implizite Fluss auftreten kann, könnten *Sequence Flow* Verbindungen, die <link>s repräsentieren, auf besondere Weise dargestellt werden. Beispielsweise könnte ein *Marker* oder eine bestimmte Farbe zur Hervorhebung verwendet werden. Die Spezifikation würde beide Arten der Hervorhebung erlauben (vgl. [BPMN1.1, S. 2, 29, 101 f.]).
BPMN verbietet allerdings den impliziten Fluss von Tokens (vgl. [BPMN1.1, S. 37]). Wenn die Erweiterung von BPMN durch die DPE dies berücksichtigen soll, ist die Einführung einer speziellen Token-Art möglicherweise eine Lösung. Dieser Abschnitt stellt explizit eine rein theoretische Betrachtung dieses Problems dar. Wenn impliziter Fluss von Tokens angenommen wird, hat sie keine Relevanz.
Ohne die Einführung zusätzlicher *Gateways* kann mit einer speziellen Token-Art eine Unterscheidung getroffen werden, ob eine *Activity* ausgeführt werden soll oder nicht. Der hier gewählte Name „Anti-Token“ soll ausdrücken, dass ein bestimmter Gegenpol notwendig ist, um dieses Token in ein normales Token zu überführen. Im Gegensatz zu einem normalen Token wird eine *Activity* oder ein *Event*, in die ein Anti-Token eingeht, nicht ausgeführt. Das Anti-Token überspringt diese *Activity*. Ein Anti-Token repräsentiert damit einen toten Pfad.
Der Wirkungskreis der DPE ist allerdings ziemlich groß, dies muss in einer semantischen Beschreibung von Anti-Tokens berücksichtigt werden. Denn <link>s können die Grenzen einer Structured Activity überschreiten (vgl. [BPEL2.0, S. 110]). Wenn eine Aktivität, beispielsweise genestet in einer <sequence>, einen eingehenden <link> hat und der Status der <joinCondition> zu false ausgewertet wird, so wird diese Aktivität übersprungen. Alle ausgehenden <link>s der Aktivität werden auf false gesetzt:

*“When a target activity is not performed due to the value of the <joinCondition> (implicit or explicit) being false, its outgoing links MUST be assigned a false status according to the rules of section 11.6.2, Link Semantics. This has the effect of propagating false link status transitively along entire paths formed by successive links until a join condition is reached that evaluates to true. This approach is called Dead-Path Elimination (DPE).”* [BPEL2.0, S. 108]

In [BPEL2.0, S. 110 f.] wird an einem Beispiel gezeigt, dass nur die ausgehenden <link>s von der DPE betroffen sind, innerhalb der Structured Activity findet die Ausführung weiter statt. Der Sachverhalt wird allerdings noch komplexer:

*“If, during the performance of structured activity A, the semantics of A dictate that activity B nested within A will not be performed as part of the execution of A, then the status of all outgoing links from B MUST be set to false. [...] An example of where this rule applies is that of an activity within an <if> activity’s branch whose <condition> is false.”* [BPEL2.0, S. 107]

Um diese implizite Semantik der DPE mit einem expliziten Tokenfluss darzustellen, könnten wie Eingangs erwähnt „Anti-Tokens“ eingeführt werden. Diese entstehen direkt an einer Aktivität bei einer nicht erfüllten <transitionCondition> oder <joinCondition>. Die Beschreibung der DPE gibt vor, dass sukzessive ganze Pfade von der Ausführung ausgenommen werden (die reflexive transitive Hülle) bis eine <joinCondition> erreicht ist, die zu true ausgewertet werden kann. An dem *Gateway*, das die <joinCondition> repräsentiert, können die Anti-Tokens daher neutralisiert und in normale Tokens überführt werden.
Anti-Tokens entstehen aber auch an bedingten Verzweigungen, wie beispielsweise in nicht betretenen <if>-Zweigen, in denen Aktivitäten mit ausgehenden <link>s enthalten sind. Daher müsste für jeden der nicht betretenen Zweige ein Anti-Token erzeugt werden. Diese Anti-Tokens werden durch das umgebende Ausgangs-*Gateway* des <if>-Konstrukts (siehe 4.4.2.3) wieder zusammengefasst. Es bleibt allerdings zunächst ein offenes Problem:
Eine Aktivität in einer Structured Activity mit einer zu false ausgewerteten <joinCondition> wird übersprungen. Innerhalb der Structured Activity ist der weitere Flow dadurch jedoch nicht

betroffen, das Anti-Token dürfte nur solche Pfade entlang fließen, die durch <link>s bestehen. In den anderen Pfaden, die von den Konstrukten implizit eingeführt werden, wie beispielsweise mit *Sequence Flow Connections* in einer <sequence> (vgl. 4.4.2.6), müsste ein normales Token fließen. Dieses Problem ließe sich dadurch beheben, dass stets vor eine Aktivität mit eingehenden <link>s ein *Complex-Gateway* gezeichnet wird. Nach einer Aktivität, die über einen <link> verfügt, egal ob eingehend oder ausgehend, müsste ein *OR-Gateway* gezeichnet werden, welches den Fluss von Tokens und Anti-Tokens steuert. Dies würde sich auf die Darstellung aller Konstrukte auswirken, in denen <link>s verwendet werden können. Die Möglichkeiten und Vorteile der Verwendung verschiedener Gateways, wie in <flow> (siehe oben) und <if> (siehe 4.4.2.3) gezeigt wurden, wären nicht mehr gegeben. Dies schließt die theoretische Betrachtung der Erweiterung von BPMN durch die DPE ab.
Abhängig von der Art und Weise, wie BPMN mit der DPE erweitert wird, ergeben sich unterschiedliche Möglichkeiten zur Darstellung. Diese unterscheiden sich optisch in der Art der *Gateways*, die Aktivitäten mit <link>s vor- und nachgestellt werden. Die beschriebene Visualisierung könnte zusätzlich in einen *Sub-Process* mit einem *Start*- und einem *End-Event* eingebettet werden, dies ist jedoch aufgrund der in dieser Arbeit vorgestellten Beschaffenheit möglicher umgebender Konstrukte nicht zwingend erforderlich.

#### 4.4.2.2 <forEach>

Die Aktivität <forEach> (siehe 2.4.2.2) stellt eine neue Errungenschaft der Standardisierung von BPEL durch OASIS dar. Daher konnte sie in dem Mapping in [BPMN1.1] noch nicht nativ verwendet werden. Das Mapping emuliert jedoch eine for-Schleife in BPEL mithilfe einer <while>-Aktivität. Dies findet im Rahmen von dem Mapping eines *Multi-Instance Loops* statt (siehe 2.5.2.1, 2.5.2.2), der für einen *Task* oder einen *Sub-Process* definierbar ist (vgl. [BPMN1.1, S. 173]). Die Semantik ist hierbei die mehrfache, parallele Ausführung der *Activity*. Eine sequentielle Ausführung wird in BPMN allerdings mit einem *Loop-Marker* dargestellt, der auch für while-Schleifen verwendet wird (vgl. [BPMN1.1, S. 132]).
Die <forEach>-Aktivität kann sowohl sequentiell als auch parallel ausgeführt werden, und zusätzlich kann eine angegebene <completionCondition> einen vorzeitigen Schleifenabbruch herbeiführen. Die Visualisierung der Bedingung für den vorzeitigen Abbruch erfordert die Abbildung in einen *Sub-Process*. Dieser trägt einen *Multi-Instance Marker* zur Kennzeichnung dieser <forEach>-Schleife. Ein zusätzlicher *Marker* wäre denkbar, um eine sequentielle Ausführung darzustellen, beispielsweise ein waagerechter Strich ( - ). Dies ist nach der BPMN Spezifikation allerdings nicht korrekt, da bei sequentieller Ausführung nur ein *Loop-Marker* angebracht werden darf. Damit wiederum wäre ein sequentielles <forEach>-Konstrukt optisch nicht mehr von einem <while> unterscheidbar. Es drängt sich die Frage auf, ob es in diesem Fall sinnvoll ist, trotz dieses Nachteils konform zum Standard zu bleiben.
Der *Sub-Proess* beinhaltet ein *Start-Event* und ein *End-Event*, die mit der genesteten <scope>-Aktivität verbunden sind. Die Darstellung der <completionCondition> erfolgt, sofern dieses Element vorhanden ist, mit einem *Catching Intermediate Conditional Event* (siehe 2.5.1.6). Dieses wird an den Rand des <scope> Bereichs gezeichnet und ist mit dem *End-Event* mit einer *Sequence Flow Connection* verbunden. Die nachfolgende Abbildung zeigt die Visualisierung von <forEach> mit bestehender <completionCondition>.

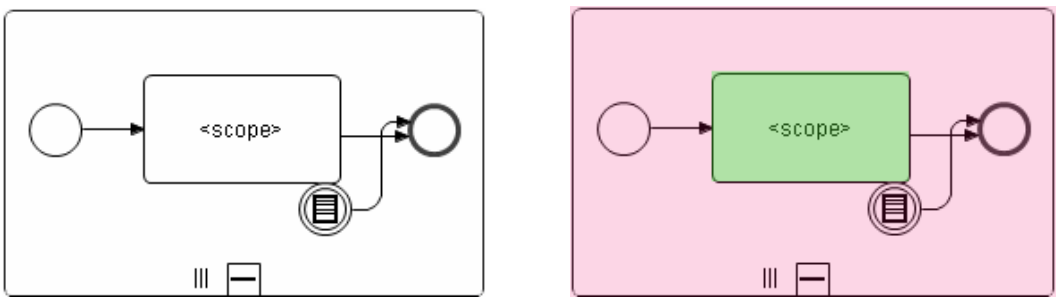


**Abbildung 90: Abbildung von <forEach>**

#### 4.4.2.3 <if>

Die <if>-Aktivität (siehe 2.4.2.3) wurde zu Anfang dieses Abschnittes als zweites Beispiel dargestellt (vgl. 4.3.1.2). In diesem wurde die Problematik der Visualisierung von <link>s in einem <flow> (siehe dazu 4.4.2.1) allerdings ausgeklammert. Auf diese Problematik wird nun, stellvertretend für alle nicht-wiederholenden Structured Activities, eingegangen, denn nur diese sind davon betroffen (vgl. [BPEL2.0, S. 104]).
Nach [ODHA07, S. 9] kann ein bestimmtes Muster in eine <if> Aktivität transformiert werden. Dieses besteht aus zwei *XOR-Gateways*, welche „dazwischen" enthaltene Aktivitäten mit *Sequence Flow Connections* verbinden. Jeder Pfad wird in ein <if> / <elseif> abgebildet, ein *Default Sequence Flow* stellt den <else>-Zweig dar. Dieses Muster lässt allerdings keine in die Struktur eingehenden oder aus der Struktur ausgehenden Verbindungen zu (vgl. [ODHA07, S. 10]). Für die Darstellung einer <if>-Aktivität muss dieses Muster erweitert werden, um möglicherweise vorhandene <link>s zu berücksichtigen.
Es finden zwei Arten der Sonderbehandlung von inneren Aktivitäten statt. Die erste ist erforderlich, wenn die Aktivität eingehende <link>s besitzt, die andere wenn sie ausgehende <link>s besitzt. Die Semantik eines <link>s innerhalb einer Structured Activity ist die eines Synchronisierungspunktes in parallelen Abläufen (vgl. auch [BPMN1.1, S. 129]). <link>s, die Aktivitäten innerhalb derselben Structured Activity als Ziel haben, müssen nicht visualisiert werden. Diese Abhängigkeit wird durch den sequentiellen Ablauf bereits impliziert und implizite Zyklen werden in [BPEL2.0, S. 133 f.] ausgeschlossen.

Die Definition dieser Abbildung erfolgt mit derselben Vorgehensweise wie bei der Definition der Abbildung von <flow>. Die Fragestellung wird durch einen Ausschnitt eines BPMN-Diagramms veranschaulicht, die Visualisierungsfunktion wird in Pseudocode definiert und dieser wird anschließend erläutert. Dieser Algorithmus unterscheidet sich jedoch in manchen Punkten von dem Algorithmus in <flow>. Die Unterschiede sind durch die implizite Einführung von *Sequence Flow* Verbindungen zur Darstellung eines Konstrukts sowie durch die sequentielle Semantik innerhalb des Konstrukts begründet.

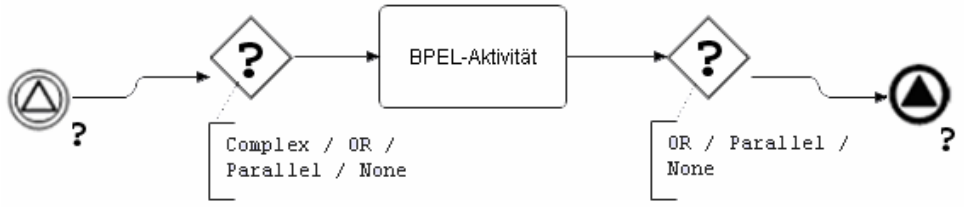


**Abbildung 91: Eingangs-Gateway, Ausgangs-Gateway und Signal-Events**

```
//Funktion zur Visualisierung einer <if>-Aktivität
//Äußerer Rahmen des <if> Konstrukts
Zeichne XOR-Gateway an den Anfang der <if>-Aktivität
Zeichne XOR-Gateway an das Ende der <if>-Aktivität

int incomingExternalArcs //eingehende externe <link>s (siehe Beschreibung unten)
int outgoingExternalArcs //ausgehende externe <link>s
boolean hasJoinCondition //gibt bei einer Aktivität an, ob eine <joinCondition>
                         //gesetzt ist
Boolean hasExternalTransitionCondition //gibt bei einer Aktivität an, ob sie
                                       //mind. einen ausgehenden externen <link>
                                       //beinhaltet, der eine <transitionCondtion>
                                       //definiert

//Bestimmung der Ein- und Ausgänge für jeden Zweig
for each (genestete Aktivität, beziehungsweise <elseif>-Zweige und <else>-Zweig)
   //Jeder Zweig wird als separater Pfad abgebildet, der mit dem Eingangs-Gateway
   //und dem Ausgangs-Gateway (XOR) des <if>-Konstrukts mit einer
   //Sequence Flow Connection verbunden ist
   //Wenn eine Aktivität mindestens einen eingehenden externen <link> hat, benötigt
   //sie ein Eingangs-Gateway. Analog gilt dies für ausgehende externe <link>s.
   //"externer <link>" bedeutet, dass sich die Quelle (bzw. Das Ziel) nicht in dieser
   //Structured Activity befindet. Interne <link>s können ohne Beschränkung der
```

```
   //Allgemeinheit ignoriert werden.

   incomingExternalArcs := Anzahl der in die Aktivität eingehenden externen <link>s
   outgoingExternalArcs := Anzahl der von der Aktivität ausgehenden externen <link>s
   hasJoinCondition := <joinCondition> an der Aktivität explizit gesetzt
   hasExternalTransitionCondition := true, wenn mind. ein externer <link> eine
                                     <transitionCondition> definiert, sonst false

   //Bestimme Eingang in die Aktivität
   if (incomingExternalArcs = 0) then
      //Kein Gateway erforderlich
      Verbinde Eingangs-Gateway von <if> direkt mit der Aktivität,
      Ziel ist die Aktivität. Wenn sich die Aktivität im <else>-Zweig befindet,
      dann zeichne eine Default Sequence Flow Verbindung, ansonsten eine Normale
   elseif(hasJoinCondition = true) then
      //Complex Gateway zeigt, dass eine besondere Bedingung zur Ausführung gilt
      Zeichne ein Complex-Gateway vor die Aktivität und verbinde das Eingangs-
      Gateway von <if> mit diesem Gateway. Wenn sich die Aktivität im <else>-Zweig
      befindet, dann zeichne eine Default Sequence Flow Verbindung, andernfalls
      wird eine Normale Sequence Flow Verbindung gezeichnet
   elseif(incomingExternalArcs > 0) then
      //Ein oder mehr eingehende <link>s; Durch die implizite Verknüpfung mit
      //dem Eingangs-Gateway von <if> (das XOR-Gateway) muss uncontrolled flow
      //ausgeschlossen werden, <joinCondition> ist nicht explizit gesetzt
      Zeichne ein OR-Gateway vor die Aktivität und Verbinde das Eingangs-
      Gateway von <if> mit diesem Gateway. Ist die Aktivität im <else>-Zweig, so
      wird Default Sequence Flow angezeigt, andernfalls Normaler Sequence Flow
   end if
   //Zeichnen der Catching Signal-Events für den Eingang der Aktivität
   for each (eingehender externer <link>)
      Zeichne ein Catching Signal-Event und verbinde es mit dem Eingangs-Gateway
      dieser Aktivität mit einer Sequence Flow Connection, Ziel ist das Eingang-
      Gateway
   next

   //Bestimme Ausgang der Aktivität
   if (outgoingExternalArcs = 0) then
      //Kein Gateway erforderlich
      Verbinde die Aktivität direkt mit dem Ausgangs-Gateway von <if>; Ziel ist das
      Ausgangs-Gateway von <if>
   elseif (hasExternalTransitionCondition = true) then
      //Bedingte Parallelität wird angezeigt
      Zeichne ein OR-Gateway nach der Aktivität und verbinde dieses Gateway mit
      Dem Ausgangs-Gateway von <if>
   else
      //Absolute Parallelität wird angezeigt
      Zeichne ein Parallel-Gateway nach der Aktivität und verbinde dieses Gateway mit
      Dem Ausgangs-Gateway von <if>
   end if
   //Zeichnen der Throwing Signal-Events für den Ausgang der Aktivität
   for each (ausgehender externer <link>)
      Zeichne ein Throwing Signal-Event und verbinde es mit dem Ausgangs-Gateway
      dieser Aktivität mit einer Sequence Flow Connection, Ziel ist das Signal-Event
   next

next //weitere Zweige

//Sonderfall: Kein <else>-Zweig angegeben, Default Sequence Flow zeichnen
if (<else>-Zweig ist nicht angegeben) then
   Zeichne eine Default Sequence Flow Verbindung vom Eingangs-Gateway von <if> direkt
   zu dem Ausgangs-Gateway von <if>
end if
```

**Listing 47: Pseudocode der Visualisierung von <if>**

### Erläuterung der Visualisierungsfunktion

Die direkte Herangehensweise nach dem Pattern von Ouyang et al. (siehe [ODHA07, S. 9]) ist eine Abbildung mit umgebenden *XOR-Gateways*, die die inneren Aktivitäten verbinden. In dem Verfahren von Mendling et al. zur Transformation von BPEL in BPMN (siehe 2.6.2.2.1) wird dieser Ansatz auch verfolgt. Allerdings wird die Semantik von `<link>`s und „uncontrolled flow“ außer Acht gelassen. Die Beachtung dieser Semantik erfordert in bestimmten Fällen die Einführung zusätzlicher *Gateways* vor und nach einer Aktivität. Die Sonderbehandlung „externer `<link>`s“ mit *Signal-Events* ist auf den Anspruch der Flexibilität und Erweiterbarkeit

sowie auf die Verwendbarkeit von *Sub-Processes* für die Abbildung von <scope>s zurückzuführen.

Bei der Einführung zusätzlicher Gateways wurden die folgenden Fälle behandelt:

- Wenn kein <link> in die Aktivität eingeht, so ist am Eingang kein zusätzliches Gateway erforderlich. Es reicht allerdings bereits ein einziger <link> aus, um dies zu ändern. Die Visualisierung des <if>-Konstrukts führt zur Darstellung des sequentiellen Ablaufs eigene *Sequence Flow* Verbindungen ein. Damit nicht der „Eindruck" von „uncontrolled flow" entsteht, sind zusätzliche Gateways erforderlich.
- Bestehen also eingehende <link>s, so kann grundsätzlich ein *OR-Gateway* vor die Aktivität gesetzt werden, um einen Synchronisierungspunkt anzuzeigen.
- Dieser Synchronisierungspunkt kann in seiner Aussagekraft noch verbessert werden. Wenn zu der Aktivität eine <joinCondition> definiert ist, kann die dadurch implizierte Semantik mit einem *Complex-Gateway* optisch eindeutig dargestellt werden.
- Ausgehende <link>s erfordern ebenso ein zusätzliches Gateway. Dazu ist grundsätzlich ein *OR-Gateway* geeignet, da sowohl bedingte als auch absolute Parallelität dadurch beschrieben werden kann.
- Dieses Gateway ist allerdings nicht immer eindeutig. Eine Verfeinerung besteht darin, dass ein *Parallel-Gateway* verwendet wird, wenn auf keinem der ausgehenden externen <link>s eine <transitionCondition> definiert ist.

Dieses Vorgehen schließt die Definition der Abbildung der <if> Aktivität ab. Der Algorithmus zur erweiterten Darstellung zusätzlicher *Gateways* wird analog in anderen Abbildungen eingesetzt. Dies betrifft alle Konstrukte, in denen <link>s verwendet werden können. Dazu gehören außer der <if>-Aktivität die Konstrukte <pick> und <sequence>. Zudem sind in <faultHandlers> und <terminationHandler> ausgehende <link>s erlaubt. In allen anderen komplexen Konstrukten muss eine <flow> Aktivität genestet sein, um darin <link>s verwenden zu dürfen. Da ein <link> die Grenze seines <flow> nicht überschreiten darf, ergeben sich keine weiteren Anforderungen (vgl. [BPEL2.0, S. 104 ff.]).

#### 4.4.2.4 <pick>

Ouyang et al. definieren in [ODHA07, S. 9 f.] ein Muster in BPMN, das auf eine <pick>-Aktivität (siehe 2.4.2.4) abgebildet wird. Das Muster besteht im Eingang aus einem *Exclusive Event-Based Gateway* (siehe 2.5.3.1.2). Ein *Intermediate Message-Event* (siehe 2.5.1.1) wird auf einen <onMessage>-Zweig abgebildet und das <onAlarm> Konstrukt geht aus einem *Intermediate Timer-Event* hervor. Das Muster wird durch ein *XOR-Gateway* abgeschlossen und mit diesem werden die Aktivitäten verbunden die auf die *Events* folgen.
Dieses Pattern kann sehr gut für die Definition der Abbildung von <pick> herangezogen werden. Eine Erweiterung wäre denkbar. Die Aktivität <pick> kann zur Instanziierung von Prozessen eingesetzt werden. Abhängig von dem Wert des Attributs createInstance ergäbe sich in BPMN eine unterschiedliche Abbildung. Ist der Wert yes, so entfiele die Darstellung des Eingangs-*Gateways*. Jeder <onMessage>-Zweig wäre durch ein *Starting Message-Event* eingeleitet. Dieses wäre mit der Aktivität verbunden, die in dem Zweig enthalten ist. Die Zweige würden durch eine *Sequence Flow* Verbindung der Aktivität mit dem *XOR* Ausgangs-*Gateway* wieder zusammengeführt. Diese semantisch passende Herangehensweise ist durch das Tokenkonzept in BPMN jedoch zum Scheitern verurteilt. Denn das auf diese Weise generierte Token müsste explizit von einem *End-Event* (siehe 2.5.1) konsumiert werden. Das *End-Event* müsste von der Aktivität, in der das <pick> direkt genestet ist, an das Ende hinzugefügt werden. Das umgebende Konstrukt kann entweder eine <sequence> oder eine <flow> Aktivität sein. Allerdings kann diese wiederum genestet sein, und aus einem *End-Event* darf keine *Sequence Flow* Verbindung ausgehen. Zudem führen <scope>s weitere Schwierigkeiten ein (siehe 4.4.3.3). Dieselbe Problematik führt dazu, dass auch eine <receive>-Aktivität nicht auf ein *Start-Event* abgebildet wird (vgl. 4.4.1.6). Das nachfolgende Code-Listing zeigt, dass dieser Ansatz wenig praktikabel ist, da unklar ist, welche Funktion ein *End-Event* zeichnet.

```
<scope name=”start”>
   <sequence name=”scope1”>
      <pick createInstance=”yes” ../>
      ...
   </sequence >
   <sequence name=”seq2”>
      ...
   </sequence>
</sequence>
```

**Listing 48: Mehrfach genestete Start-Aktivität**

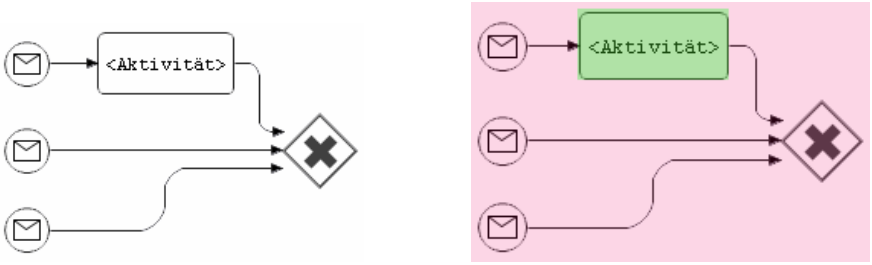


**Abbildung 92: Unpraktikable Abbildung von <pick> als startende Aktivität**

Daher ergibt sich unabhängig von dem Attribut `createInstance` die Abbildung nach dem Pattern, das Ouyang et al. definiert haben. Die Aktivität kann allerdings auch innerhalb eines `<flow>` eingesetzt werden, weshalb die Sonderbehandlung von `<link>`s notwendig wird. Diese ist im Detail in 4.4.2.3 beschrieben. Die Darstellung der `<onMessage>`-Zweige muss mit einem *Intermediate Message-Event* geschehen. Im Gegensatz zu den `<onMessage>`-Zweigen könnte die Visualisierung der `<onAlarm>`-Zweige möglicherweise in eine eigenständige Visualisierungsfunktion ausgelagert werden, da sie auch in einem anderen Kontext (`<eventHandlers>`) verwendet wird, siehe dazu die Diskussion in 4.4.5.14. Das nachfolgende Code-Listing soll die somit definierte Abbildung an einem Beispiel verdeutlichen. Darin wird zudem ein externer `<link>` in die Darstellung einbezogen.

```
<pick createInstance=”no”>
   <onMessage ...
      <invoke ...
         <targets>
            <target linkName=”from-somewhere-to-here”/>
         </targets>
      </invoke>
   </onMessage>
   <onAlarm>
      <for>’P3DT10H’</for>
      <scope ...
   </onAlarm>
</pick>
```

**Listing 49: <pick>-Aktivität in BPEL-Code mit externem <link>**

In die Visualisierung sind in diesem Beispiel drei Funktionen involviert. Die der <pick>-Aktivität (rot), des <onAlarm>-Konstrukts als eigenständige Visualisierungsfunktion (grün) und der im <onMessage>-Zweig genesteten Aktivität (blau). Die Praktikabilität der Auslagerung von <onAlarm> in eine eigenständige Visualisierungsfunktion wird in 4.4.5.14 eingehender betrachtet.

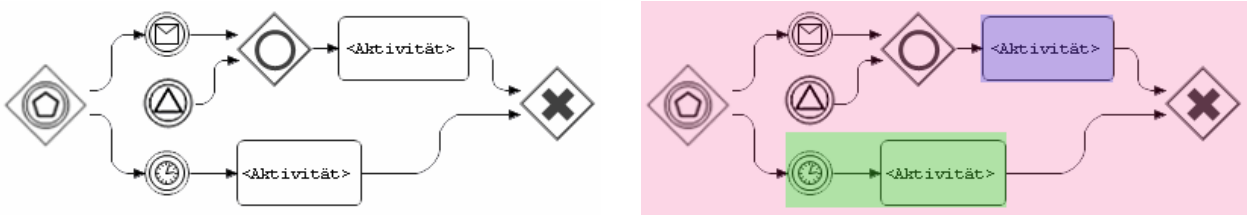


**Abbildung 93: Praktikable Abbildung von <pick> mit eingehendem <link>**

### 4.4.2.5 <repeatUntil>

*„The <repeatUntil> activity provides for repeated execution of a contained activity. The contained activity is executed until the given Boolean <condition> becomes true"* [BPEL2.0, S. 100]. Diese Anforderung kann nach dem REPEAT-Pattern (vgl. [ODHA07, S. 9]) in eine komplexe BPMN-Struktur überführt werden. Die in <repeatUntil> genestete Aktivität wird von zwei *XOR-Gateways* umgeben. Mithilfe einer *Default Sequence Flow* Verbindung kann eine Schleife erzeugt werden, die die Semantik einer <repeatUntil>-Aktivität hat. Wenn die Abbruchbedingung erfüllt ist wird die Schleife verlassen. Ansonsten wird der *Default* Pfad gewählt und die Schleife so wiederholt.


**Abbildung 94: Abbildung von <repeatUntil> mit dem REPEAT-Pattern**

Eine alternative Darstellung kann unter der Verwendung eines *Sub-Process* stattfinden. Dieser beschreibt die Charakteristik einer <repeatUntil>-Aktivität mit einem zusätzlichen *Loop-Marker* ebenso gut. Dazu kann die zweite Definition der Abbildung von <while> in 4.4.2.7 verglichen werden.

#### 4.4.2.6 <sequence>

Das SEQUENCE-Pattern in [ODHA07, S. 9] bildet eine Sequenz von *Activities*, die mit *Sequence Flow Connections* verbunden sind, auf eine `<sequence>`-Aktivität (siehe 2.4.2.6) in BPEL ab. Dieses Muster kann auf die Gegenrichtung übertragen werden, wie es auch in der *Flattening* Strategie (vgl. 2.6.2.2.1) vorgeschlagen wird. Da eine `<sequence>`-Aktivität in einem `<flow>`-Konstrukt verwendet werden kann, sind jedoch eingehende und ausgehende `<link>`s möglich. Die *Flattening* Strategie ist in diesem Punkt unvollständig. Die `<link>`s müssen mit demselben Algorithmus behandelt werden, wie bei der Abbildung der `<if>`-Aktivität demonstriert wurde (siehe 4.4.2.3). Das nachfolgende Code-Listing zeigt eine `<sequence>` Aktivität. Anschließend wird diese Aktivität graphisch abgebildet und die involvierten Visualisierungsfunktionen hervorgehoben. Der ausgehende `<link>` wird, da keine `<transitionCondition>` definiert ist, mit einem *Parallel Gateway* und einem *Throwing End Signal-Event* dargestellt.

```
<sequence>
   <assign ... />
   <invoke ...
      <sources>
         <source linkName="from-here-to-farAway" />
      </sources>
   </invoke>
   <assign ... />
</sequence>
```

**Listing 50: <sequence>-Aktivität mit ausgehendem <link>**

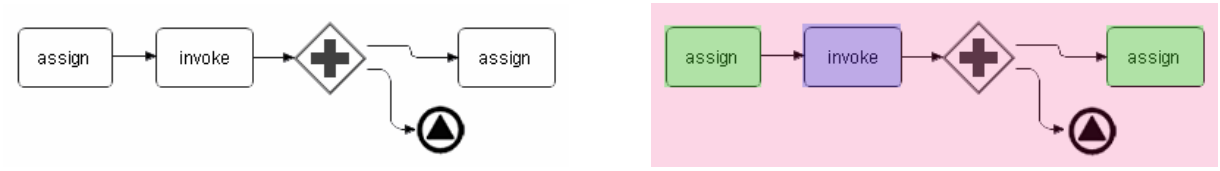


**Abbildung 95: Abbildung der <sequence>-Aktivität mit ausgehendem <link>**

#### 4.4.2.7 <while>

Ouyang et al. haben ein Muster in einem BPMN-Diagramm identifiziert, das sich direkt auf die BPEL-Aktivität `<while>` abbilden lässt (vgl. [ODHA07, S.9]). Das Muster besteht aus einer *Activity*, die von zwei *XOR-Gateways* umgeben ist. Dieses wird nun an einem Code-Beispiel, einer graphischen Abbildung (aus [ODHA07, S.9]) und einer Hervorhebung der dabei involvierten Visualisierungsfunktionen analysiert.

```
<while name="loop" condition="b1">
   <invoke name="t1" ... />
</while>
```

**Listing 51: Beispiel für <while>**


**Abbildung 96: Mögliche Abbildung von <while> (links) und Hervorhebung (rechts)**

Eine Schwierigkeit bei dieser Abbildung ist der *Default Sequence Flow*, der von dem rechten *XOR-Gateway* ausgeht. Dieser kann durch die Visualisierungsfunktion nur im Ansatz gezeichnet werden. Analog zur Abbildung von `<repeatUntil>` (siehe 4.4.2.5) kann als Alternative eine Abbildung auf einen *Sub-Process* (siehe 2.5.2.2) stattfinden. *Sub-Processes* wurden in den Patterns von Ouyang et al. nicht betrachtet, da sie auf ein komplexes Muster wie das obige abbildbar sind. Um eine Schleife mit einem *Sub-Process* darzustellen, wird ein *Loop-Marker* angezeigt. Wie ein Beispiel-Diagramm in [BPMN1.1, S. 139] zeigt, wird in der *expanded* Darstellung ein Ablauf visualisiert, der keine explizite Schleife enthält. Die Schleife besteht also nur implizit, symbolisiert durch den *Loop-Marker*. Die nachfolgende Abbildung zeigt diese Form der Darstellung. Zur Unterscheidung von `<repeatUntil>` könnte ein zusätzlicher *Marker*, oder wie in [BPMN1.1, S. 139] gezeigt, eine *Annotation* eingeführt werden.

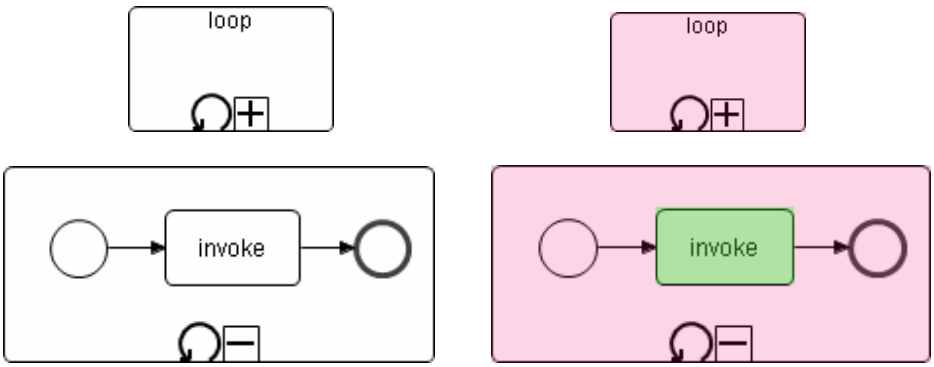


**Abbildung 97: Abbildung von <while> als Sub-Process**

### 4.4.3 Scopes

#### 4.4.3.1 <compensate>

Die Spezifikation zu BPMN (vgl. [BPMN1.1, S. 149, 153]) bildet in dem Mapping zu BPEL sowohl ein *Throwing Intermediate Compensation Event* als auch ein *Throwing End Compensation Event* (siehe 2.5.1.5) auf die BPEL-Aktivität `<compensate>` (siehe 2.4.3.1) ab. Der semantische Unterschied dieser *Events* besteht darin, dass der *Sequence Flow* nach einem *Intermediate Event* weiterläuft, an einem *End-Event* jedoch aufhört. Für die Abbildung ist daher zu klären, wie sich die `<compensate>`-Aktivität auf den Kontrollfluss auswirkt. In einem Beispielprozess in [BPEL2.0, S. 132] wird im Rahmen der Fehlerbehandlung gezeigt, dass auf eine `<compensate>`-Aktivität noch weitere Aktivitäten folgen können. Daher wird diese Aktivität auf das *Throwing Intermediate Compensation Event* abgebildet. Die nachfolgende Abbildung illustriert dieses *Event*.

**Abbildung 98: Darstellung von <compensate>**

#### 4.4.3.2 <compensateScope>

Das Mapping in [BPMN1.1] macht keinen Gebrauch von der `<compensateScope>` Aktivität (siehe 2.4.3.2), da diese Aktivität erst durch OASIS in BPEL eingebracht wurde (vgl. [Schu07, S. 21 f.]). Während `<compensate>` eine Kompensation auf allen inneren `<scope>`s startet, kann mit dieser Aktivität der zu kompensierende `<scope>` gezielt festgelegt werden. Eine Unterscheidung dieser beiden Aktivitäten ist in BPMN nicht vorgesehen. Die Aktivität wird daher ebenso wie die `<compensate>`-Aktivität auf das *Throwing Intermediate Compensation Event* abgebildet (siehe 4.4.3.1).

#### 4.4.3.3 <scope>

*"A <scope> provides the context which influences the execution behavior of its enclosed activities. This behavioral context includes variables, partner links, message exchanges, correlation sets, event handlers, fault handlers, a compensation handler, and a termination handler."* [BPEL2.0, S. 115]

Für jedes Konstrukt in dem oben genannten Kontext des `<scope>`-Konstrukts (siehe 2.4.3.3) ist eine eigenständige Visualisierungsfunktion definiert. Die Abbildung von `<scope>` hat daher lediglich die Aufgabe, diese Bausteine zusammenzufügen. Die Abbildung und Integration des Kontexts (`<variables>`, `<partnerLinks>`, `<messageExchanges>` und `<correlationSets>`) und der Handler verläuft nahezu analog zu der Abbildung des `<process>`-Konstrukts (vgl. dazu 4.4.5.10). Der Kontext wird unter Verwendung der jeweiligen Visualisierungsfunktionen dargestellt. Eine Gruppierung der einzelnen Artefakte in eine *Group* (siehe 2.5.5.6) erhöht die Übersichtlichkeit. Die `root`-Aktivität des `<scope>` wird in einen *Sub-Process* gezeichnet und mit einem *Start*- und einem *End-Event* verbunden. Die einzelnen Handler sind mit einem entsprechenden *Intermediate Event*, das an diesen *Sub-Process* angeheftet ist, verbunden und liegen außerhalb des *Sub-Process*. Eine Ausnahme stellt das `<eventHandlers>`-Konstrukt dar. Dieses wird, nach der in 4.4.5.12 geleisteten Begründung, ohne angeheftetes *Intermediate Event* innerhalb des *Sub-Process* platziert.

Um den Gültigkeitsbereich des Kontexts zu verdeutlichen, sollte um den gesamten `<scope>` ein Rahmen gezeichnet werden. Wenn dieser Rahmen als weiterer *Sub-Process* gezeichnet wird, entsteht allerdings ein Konflikt mit der BPMN-Spezifikation. Genauer gesagt besteht dieser Konflikt bereits durch den inneren *Sub-Process* an dem die Handler angebracht sind. In [BPMN1.1, S. 30] wird gefordert, dass eine *Sequence Flow Connection* – und somit die direkte Abbildung eines `<link>` - die Grenze eines *Sub-Process* nicht überschreiten darf. In [BPEL2.0, S. 133 f.] wird mit der „Peer-Scope Dependency" allerdings demonstriert, dass `<link>`s die Grenzen von `<scope>`s überschreiten dürfen, solange dadurch keine zyklischen Kontrollabhängigkeiten entstehen. Der äußere Rahmen um einen `<scope>` könnte mit einer *Group* gezeichnet werden. Aber welche Alternative gibt es für den inneren *Sub-Process*?

Eine *Group* dient lediglich der Darstellung, *Intermediate Events* können daran nicht angeheftet werden. An einen *Task* können diese *Events* angebracht werden, diese *Activity* ist allerdings atomar, kann also keine weiteren Konstrukte beinhalten. Die *Intermediate Events*, die für die Abbildung der Fault-, Compensation- und Termination-Handler notwendig sind, müssen aber allesamt an eine *Activity* angeheftet werden. Die Verwendung des inneren *Sub-Process* ist daher unumgänglich.

Die einzige noch nicht betrachtete „freie" Variable ist die Abbildung des `<link>`. Eine geradlinige, direkte Definition zur Visualisierung eines `<link>`s ist die Darstellung mit einer *Sequence Flow* Verbindung. Diese Verbindung kann durch zwei *Intermediate Link-Events* (siehe 2.5.1.7) als nicht-durchgängig gezeichnete *Sequence Flow* Verbindung dargestellt werden. Dennoch überschreitet sie auf diese Weise, wenn auch nur implizit, unter Umständen die Grenze eines *Sub-Process*. Eine Neuerung in [BPMN1.1] bietet für dieses Problem einen Ausweg. In [BPMN1.1, S. 129] wird beschrieben, wie der Ablauf über die Grenzen eines *Sub-Process* hinweg gesteuert werden kann *(„Controlling Flow Across Processes"*). Dazu dient ein *Signal-Event*, welches direkt in den Ablauf integrierbar ist. Daher müssen `<link>`s, die die Grenze eines *Sub-Process* überschreiten, mit einem *Signal-Event* abgebildet werden. Konsequenterweise kann dieses Vorgehen bei allen `<link>`s praktiziert werden, die nicht direkt durch eine *Sequence Flow* Verbindung dargestellt werden können. Für eingehende `<link>`s ist dies genauer gesagt ein *Catching Intermediate Signal-Event*, für ausgehende `<link>`s ein *Throwing End Signal-Event*. Der ausgehende `<link>` wird durch ein *End-Event* und nicht durch ein *Intermediate-Event* dargestellt, da keine weitere *Sequence Flow* Verbindung von diesem *Event* ausgeht.

Der letzte Gesichtspunkt bei der Abbildung des `<scope>`-Konstrukts ist, dass eine Definition von `<compensationHandler>`, `<terminationHandler>` und `<faultHandlers>` in dem BPEL-XML nicht zwingend erforderlich ist. Wenn diese nicht angegeben sind, wird nach [BPEL2.0, S. 132] im Ausnahmefall per Default stets die Aktivität `<compensate>` durch die BPEL-Engine aufgerufen.

Wenn diese Ausnahme ein Fault ist, wird anschließend auch die Aktivität `<rethrow>` gestartet. Die Ausführungen zur Abbildung von `<process>` (vgl. 4.4.5.10) haben nahegelegt, diesen impliziten Sachverhalt explizit darzustellen. Dies müsste durch eine geeignete Implementierung sichergestellt werden.
Nach obiger Argumentation ist die Visualisierung eines `<scope>`s vollständig definiert. Die nachfolgende Abbildung zeigt die Visualisierung eines `<scope>` mit innerem und äußerem *Sub-Process* (rot) mit einer genesteten `root`-Aktivität (grün), einem `<eventHandlers>`-Konstrukt (blau), `<faultHandlers>` (gelb), `<compensationHandler>` (magenta), `<terminationHandler>` (cyan) und den Artefakten des Kontexts (weiß).

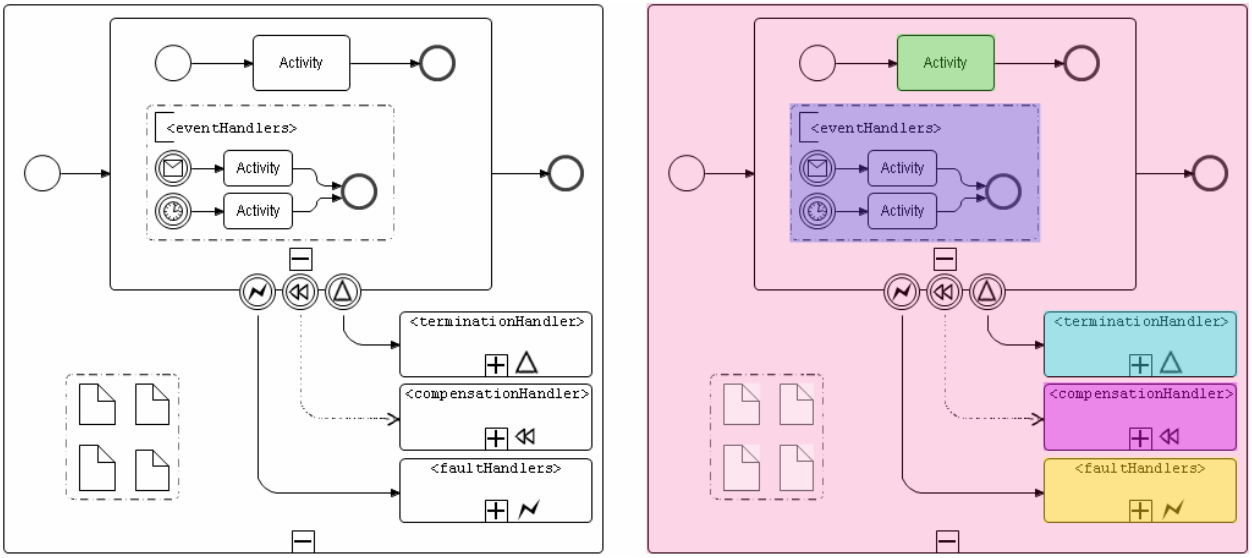


**Abbildung 99: Abbildung eines <scope>**

### 4.4.4 Variables

#### 4.4.4.1 <variable>

In 2.5.5.5 wurde ein BPMN-Artefakt vorgestellt: das Data Object. Damit können Input- und Outputdaten von Aktivitäten oder gar der Datenfluss modelliert werden, aber nur auf nicht-standardisierte Art und Weise. Ein `<variable>`-Konstrukt (siehe 2.4.4.1) wird direkt auf Prozessebene oder in einem `<scope>` definiert und daher durch deren Visualisierungsfunktionen eingebettet. Ohne Bezug auf den äußeren Kontext zu nehmen, erscheint die Abbildung auf ein *Data Object* daher als eine gute Lösung. Mit einem *Marker* könnte visualisiert werden, ob das `<variable>`-Konstrukt durch einen WSDL-Nachrichtentyp oder ein XML-Schema definiert ist.


**Abbildung 100: Darstellung von <variable>**

#### 4.4.4.2 <validate>

Die `<validate>`-Aktivität (siehe 2.4.4.2) ist eine einfache Aktivität zur Validierung von Variablen gegen ihre Definition. Mithilfe des Attributs `variables` können mehrere Variablen angegeben werden, für die eine Validierung duchgeführt werden soll. Die Definition des *Tasks* in [BPMN1.1, S. 65] könnte für eine Abbildung herangezogen werden: *"A Task is used when the work in the Process is not broken down to a finer level of Process Model detail."* Eine Abbildung könnte aber auch die Visualisierungsfunktion von `<variable>` einbeziehen und diese für jede in dem Attribut enthaltene Variable aufrufen. Die Artefakte müssen innerhalb der Figur gezeichnet

werden. Als Verbindungselement dient eine *Association Connection* (siehe 2.5.5.2). Nachteilig an dieser Herangehensweise ist, dass zusätzlich ein Parsing stattfinden müsste, denn die Variablen liegen nicht direkt als `<variable>` Elemente vor (vgl. [BPEL2.0, S. 46]).

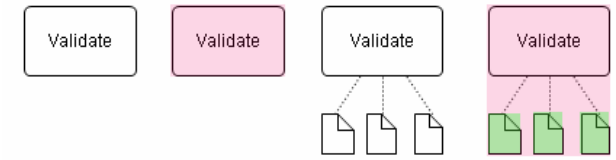


**Abbildung 101: Mögliche Abbildungen von <validate>**

### 4.4.5 Weitere Konstrukte

#### 4.4.5.1 <catch>

Das Konstrukt `<catch>` (siehe 2.4.5.1) besitzt keine eigenständige Visualisierungsfunktion und wird von seinem umgebenden Konstrukt abgebildet. Dies ist entweder ein `<faultHandlers>`-Konstrukt (siehe dazu 4.4.5.7) oder eine `<invoke>`-Aktivität (siehe hierfür 4.4.1.5).

#### 4.4.5.2 <catchAll>

Analog zum Konstrukt `<catch>` (siehe 4.4.5.1) wird das Konstrukt `<catchAll>` (siehe 2.4.5.2) von seinem umgebenden Konstrukt abgebildet.

#### 4.4.5.3 <compensationHandler>

Wenn man den Ausführungen in [BPMN1.1, S. 154] folgt, so wird ein *Task* in einen `<compensationHandler>` (siehe 2.4.5.3) übersetzt, wenn er mit einem an einer *Activity* angebrachten *Catching Intermediate Compensation-Event* (siehe 2.5.1.5) verbunden ist. In umgekehrter Richtung kann daraus gefolgert werden, dass ein `<compensationHandler>` als *Task* dargestellt wird der mit einem derartigen *Event* verbunden ist. Ein `<compensationHandler>` kann allerdings auch Structured Activities beinhalten. Daher ist die Darstellung als *Sub-Process* angebracht; die Semantik ändert sich durch diesen Umstand nicht. Auf dem *Sub-Process* wird ein *Compensation-Marker* angebracht der den Zweck der *Activity* kennzeichnet. Eine Beachtung von eingehenden oder ausgehenden `<link>`s ist nicht erforderlich (vgl. [BPEL2.0 S. 202]). Die Integration in ein Prozessdiagramm mit einer *Compensation Association* (siehe 2.5.5.2) erfolgt durch die umgebende Aktivität. Dies kann sowohl ein `<scope>`-, ein `<process>`- als auch ein `<invoke>`-Konstrukt sein. Die nachfolgende Abbildung zeigt die Visualisierung als *Sub-Process*.

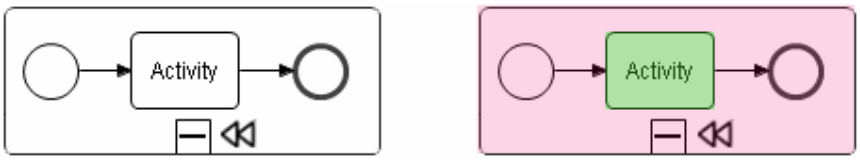


**Abbildung 102: Darstellung eines <compensationHandler> als Sub-Process**

#### 4.4.5.4 <correlationSets>

Für das BPEL-Konstrukt `<correlationSets>` (siehe 2.4.5.4) gibt es keine eindeutige Entsprechung in BPMN. Das einzige spezifizierte Artefakt, das sich zur Darstellung eignen würde, wäre das *Data Object* (siehe 2.5.5.5). Dieses Artefakt wurde bereits für die Abbildung von `<variable>` als Möglichkeit aufgezeigt (siehe 4.4.4.1). Das `<correlationSets>`-Konstrukt kann im Gegensatz zu `<variable>` allerdings aus mehreren Elementen (`<correlationSet>s`) bestehen. Für diesen Fall ist das *Group* Artefakt (siehe 2.5.5.6) als äußerer Rahmen geeignet. Für die Darstellung der inneren Elemente kann alternativ zum *Data Object* eine beliebige andere Darstellung verwendet werden. Die Spezifikation sieht dies sogar ausdrücklich vor:

*"In general, BPMN will not standardize many modeling Artifacts. These will mainly be up to modelers and modeling tool vendors to create for their own purposes."* [BPMN1.1, S. 95]

Es dürfen allerdings keine *Flow Objects* (*Events, Activities, Gateways*) zur Darstellung eingesetzt werden. Eine weitere Möglichkeit wäre, `<correlationSets>` gänzlich nicht graphisch abzubilden.

#### 4.4.5.5 <documentation>

Eine Abbildung des `<documentation>`-Elements (siehe 2.4.5.5) ist in [BPMN1.1, S. 185] klar definiert:
*"As a general rule, Artifacts do not map to BPEL4WS elements. They provide detailed information about how data will interact with the Flow Objects and Flow of Processes. However, Text Annotations can map to the documentation element of BPM execution languages."*

Dieses BPEL-Element kann an jedem Konstrukt verwendet werden. An welcher Stelle die Visualisierungsfunktion dieses Elements eingebunden werden könnte, beziehungsweise ob der Inhalt der Dokumentation überhaupt angezeigt wird, bleibt der Visualisierungsfunktion des übergeordneten Konstrukts überlassen. Die Einführung einer globalen Variable zum Ein- und Ausschalten der Anzeige von Dokumentationen wäre eine Möglichkeit, dies durch den Anwender zu steuern.

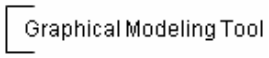


**Abbildung 103: Abbildung von <documentation>**

#### 4.4.5.6 <extensions>

Analog zu der Argumentation zur Abbildung von `<correlationSets>` (vgl. 4.4.5.4) sind für `<extensions>` (siehe 2.4.5.6) mehrere Alternativen vorhanden. Eine Visualisierungsfunktion kann dieses Konstrukt entweder als *Data Object*, als neue Art von Artefakt oder überhaupt nicht darstellen, da es nicht in den Ablauf (in BPMN als *Flow Object)* integriert ist (vgl. [BPMN1.1, S. 95]).

#### 4.4.5.7 <faultHandlers>

Das `<faultHandlers>`-Konstrukt (siehe 2.4.5.7) ist eine Art Behälter für `<catch>` und `<catchAll>` Konstrukte (siehe 2.4.5.1 und 2.4.5.2). In [BPEL2.0, S. 129] wird beschrieben, wie eine Fehlerbehandlung im Detail abläuft. Ein `<catch>`-Konstrukt wird genau dann ausgeführt, wenn es zu einem bestimmten Fehler passend ist. Dazu muss sowohl der in dem `<catch>`-Konstrukt angegebene Fehlername (`faultName`) als auch das dem Fehler zugehörige Datenformat (`faultVariable`) übereinstimmen. Wenn kein passendes `<catch>`-Konstrukt vorhanden ist, wird das Default-Konstrukt `<catchAll>` ausgeführt. Wenn allerdings auch dieses nicht vorhanden ist, dann kommt ein Default Fault Handler zum Einsatz, der in [BPEL2.0, S. 132] definiert ist:

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

**Listing 52: Default Fault Handler**

Die Integration eines `<faultHandlers>` in ein Prozessdiagramm erfolgt durch das umgebende Konstrukt, also durch den `<process>` oder einen `<scope>`. Dazu wird ein *Catching Intermediate Error-Event* an eine *Activity* angeheftet und mit einer *Sequence Flow Connection* mit dem

<faultHandlers> verbunden, wie beispielsweise in [BPMN1.1, S. 157] dargestellt. In [BPMN1.1, S. 152] wird jedes dieser angehefteten *Error-Events* in ein <catch>-Konstrukt übersetzt. Die Verwendung mehrerer Events, um mehrere Fehlerbehandlungsroutinen darzustellen, ist jedoch nicht zwingend erforderlich:

*"For an Intermediate Event attached to the boundary of an Activity: If the Trigger is an Error, then the ErrorCode MAY be entered. This Event "catches" the error. If there is no ErrorCode, then any error SHALL trigger the Event. If there is an ErrorCode, then only an error that matches the ErrorCode SHALL trigger the Event."* [BPMN1.1, S. 51]

Es ist daher ausreichend und zudem optisch überschaubarer, ein einziges *Error-Event* anzuheften und eine Fallunterscheidung mit einem *XOR-Gateway* einzuführen. Die Darstellung findet als *Sub-Process* statt, in dem die Zweige zur Fehlerbehandlung mit einem *Gateway* und einem *End-Event* verbunden sind. Die Visualisierungsfunktion muss den in 4.4.2.3 beschriebenen Algorithmus zur Sonderbehandlung von <link>s teilweise einbeziehen, da in einem <faultHandlers> zumindest ausgehende <link>s erlaubt sind (vgl. [BPEL2.0, S. 105]). Ein <catchAll>-Zweig wird mit einer *Default Sequence Flow Connection* verbunden. Wenn ein solcher <catchAll>-Zweig nicht vorhanden ist, kommt der oben beschriebene „Default Fault Handler" zum Einsatz. Dieser kann unter Zuhilfenahme der Visualisierungsfunktionen von <compensate> und <rethrow> visualisiert werden. Für eine eindeutige Abbildung des BPEL-Metamodells ist dies sogar ein absolutes Muss. Die nachfolgende Abbildung zeigt die Visualisierung eines <faultHandlers> (rot) mit einer Aktivität in einem <catch>-Zweig (grün) und einer Aktivität in einem expliziten <catchAll>-Zweig (blau).

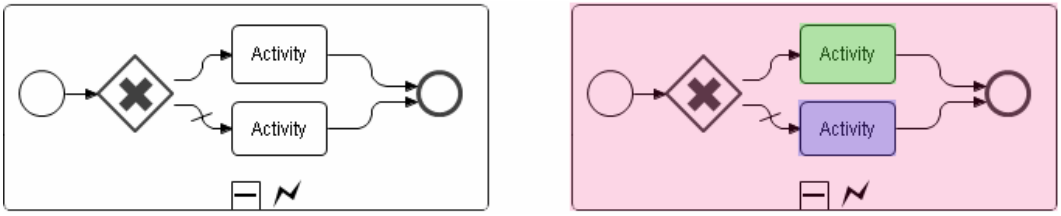


**Abbildung 104: Abbildung eines <faultHandlers>-Konstrukts**

#### 4.4.5.8 <import>

Die Abbildung eines <import>-Konstrukts (siehe 2.4.5.8) kann analog zu der Abbildung von <correlationSets> relativ beliebig vorgenommen werden (vgl. 4.4.5.4), weil dieses Konstrukt nicht in den Kontrollfluss integriert ist.

#### 4.4.5.9 <partnerLinks>

Die Definition der Abbildung von <partnerLinks> (siehe 2.4.5.9) kann analog zur Abbildung von <correlationSets> (vgl. 4.4.5.4) vorgenommen werden. Eine Alternative dazu stellt die Verwendung von *Pools* (siehe 2.5.5.3) dar, die in [BPMN1.1, S. 145] angedeutet wird. Das Konstrukt <partner> (siehe [BPEL1.1, S. 35]) wurde in der Standardisierung durch OASIS aus BPEL entfernt (vgl. [Schu07, S. 7]). Es diente zur Zusammenfassung von <partnerLink>s die von einem einzelnen Business Partner erwartet werden. In der alternativen Darstellung muss daher jeder einzelne <partnerLink> als separater Pool abgebildet werden, denn zur Modellierungszeit kann ohne das <partner>-Konstrukt nicht bestimmt werden, ob ein Business Partner (und somit ein *Pool*) verschiedene Dienste anbietet. Ein *Pool* repräsentiert demzufolge einen <partnerLink>, über den Nachrichten ausgetauscht werden können. Dies betrifft die Konstrukte <receive>, <reply>, <invoke>, <pick>, <eventHandlers> und möglicherweise <extensionActivity>. Eine Verbindung dieser Aktivitäten mit einem derartigen *Pool* erfolgt mit einer *Message-Flow Connection* (siehe 2.5.5.1) mit der entsprechenden Pfeilrichtung. Diese Visualisierung nimmt in größeren Prozessen allerdings viel Raum ein und kann die Übersichtlichkeit der Darstellung verschlechtern, sie sollte daher flexibel aktivierbar sein.

#### 4.4.5.10 <process>

Das <process>-Konstrukt wurde in 2.4.5.10 als „äußere Klammer“ um einen BPEL-Prozess beschrieben. Als solche beinhaltet sie den root-context, bestehend aus <extensions>, <import>s, <partnerLinks>, <messageExchanges>, <variables> und <correlationSets>. Zur Darstellung dieser Konstrukte sind eigenständige Visualisierungsfunktionen definiert. Für eine optimale Darstellung empfiehlt sich eine Gruppierung dieser Artefakte durch eine *Group* (siehe 2.5.5.6). Außerdem werden in einem <process> globale <eventHandlers> und <faultHandlers> spezifiziert. Die Darstellung des <eventHandlers>-Konstrukts muss nach der Argumentation in 4.4.5.12 innerhalb eines Bereichs platziert werden, für den der <faultHandlers> des <process>es gültig ist. Dies ist durch einen *Sub-Process* möglich, in den das <eventHandlers>-Konstrukt, sofern vorhanden, gezeichnet wird. Ebenso wird die eigentliche root-Aktivität in diesem *Sub-Process* gezeichnet, verbunden mit einem *Start-* und einem *End-Event*. Die Visualisierung des <faultHandlers> erfolgt außerhalb dieses *Sub-Process*. Der <faultHandlers> wird durch ein *Catching Intermediate Error-Event*, welches an den *Sub-Process* angeheftet ist, verbunden und ist somit sowohl für die root-Aktivität als auch für <eventHandlers> gültig.
Wenn das <faultHandlers>-Konstrukt in einem <process> nicht definiert ist, wird durch eine BPEL-Engine ein „Default Fault Handler“ ausgeführt, wie in [BPEL2.0, S. 132] beschrieben. Übertragen auf einen <process> wird in einem Fehlerfall ohne definierten <faultHandlers> die Aktivität <compensate> ausgeführt. Die in [BPEL2.0, S. 132] angegebene nachfolgende Aktivität <rethrow> spielt auf <process>-Ebene keine Rolle. Es erscheint daher sinnvoll diesen impliziten Sachverhalt explizit darzustellen, um eine exakte Abbildung der gegebenen Semantik zu erzielen. Aus diesem Grund ist die Visualisierungsfunktion von <compensate> (siehe 4.4.3.1) in diesem speziellen Fall einzubeziehen. Der äußere Rahmen der Abbildung eines <process>-Konstrukts ist durch das Mapping in der BPMN-Spezifikation ableitbar:

- *“There can be one or more Business Processes within a Business Process Diagram, each within a separate Pool”.* [BPMN1.1, S. 146]
- *“Each Pool in the diagram, if it is a “white box” that contains process elements, will map to an individual BPEL4WS process.”* [BPMN1.1, S. 145]
- *“Lanes do not have any specific Mapping to Execution Languages. They are designed to help organize and communicate how activities are grouped in a business process.“* [BPMN1.1, S. 185]

In einem *Business Process Diagram (BPD)* können mehrere Prozesse dargestellt werden; jeder davon ist in einem Pool (siehe 2.5.5.3) enthalten. Eine „white box“ (vgl. [BPMN1.1, S. 89]) ist vergleichbar mit einem ausführbaren Prozess, im Gegensatz zu einer „black box“ (vgl. [BPMN1.1, S. 104]), die einen abstrakten Prozess repräsentiert (siehe 2.4.5.10). Eine Abbildung mehrerer BPEL-Prozesse in ein *BPD* mit mehreren *Pools* findet im Rahmen dieser Arbeit nicht statt. Auf die Möglichkeiten, die sich in diesem Zusammenhang ergeben, wird im Ausblick in Abschnitt 6.2 eingegangen. Ein *Pool* ohne *Lanes* (siehe 2.5.5.4) ist gleichwertig wie ein *Pool* mit einer einzigen *Lane* (vgl. [BPMN1.1, S. 19]). *Lanes* haben in BPEL keine Entsprechung (siehe oben) und kommen daher bei der Abbildung nicht zum Einsatz. Die Füllfarbe des *Pools* ist unerheblich (vgl. [BPMN1.1, S. 29]).
Die Abbildung des <process>-Konstrukts ist somit vollständig definiert. Die nachfolgende Abbildung zeigt die Visualisierung eines <process> (rot), mit einer root-Aktivität (grün), <eventHandlers> (blau), einem expliziten <faultHandlers>-Konstrukt (gelb) und den Artefakten des root-context (weiß).

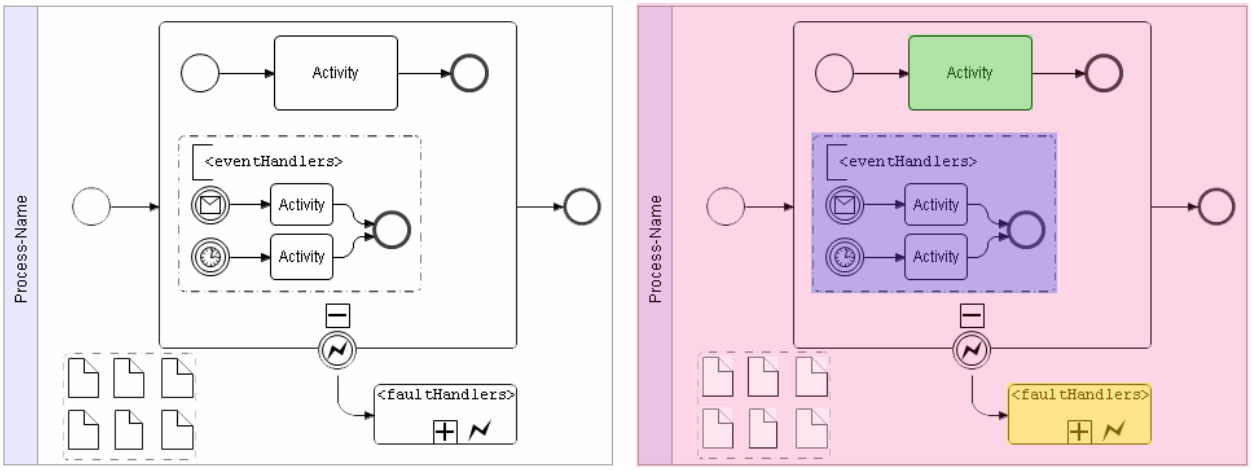


**Abbildung 105: Abbildung des <process>-Konstrukts**

#### 4.4.5.11 <terminationHandler>

Das `<terminationHandler>`-Konstrukt (siehe 2.4.5.11) wurde bei der Standardisierung durch OASIS eingeführt. Es dient dazu, bei einem forcierten Prozessende eine geringfügige Kontrolle über dessen Ablauf zu bekommen (vgl. [Schu07, S. 22]). Dieses komplexe Konstrukt kann nur direkt an einem `<scope>` (siehe 2.4.3.3) definiert werden. Das `Termination`-Ereignis kann allerdings auch außerhalb dieses `<scope>`s ausgelöst werden. Durch dieses Ereignis wird der reguläre Ablauf unterbrochen (vgl. [BPEL2.0, S. 135 f.]), in BPMN wird dieser Vorgang als „Exception Flow" bezeichnet (vgl. [BPMN1.1, S. 23, 131 f.]). Um einen *Exception Flow* in BPMN zu modellieren, wird ein *Intermediate Catching Event* an eine Aktivität angebracht. Der mit diesem *Event* verbundene *Task* oder *Sub-Process* führt die zugehörige Ausnahmebehandlung durch. Die Zugehörigkeit wird mit demselben *Marker* visualisiert, den auch das angehängte *Event* trägt. Ein *angeheftetes Signal-Event* verdeutlicht am besten, dass das Ereignis die normale Ausführung des `<scope>` in einen *Exception Flow* überführt und auch außerhalb des `<scope>` ausgelöst werden kann:

*„A BPMN Signal is similar to a signal flare that shot into the sky for anyone who might be interested to notice and then react."* [BPMN1.1, S. 46].

In der Abbildung von `<link>`s (siehe 4.4.2.1) wurde das *Signal-Event* ebenso verwendet, allerdings in einem anderen Zusammenhang, nämlich zur Darstellung von Kontrollabhängigkeiten über die Grenzen eines *Sub-Process* hinweg, innerhalb des *Normal Flow* (vgl. [BPMN1.1, S. 108 ff.]). Eine Verwechslungsgefahr kann dadurch ausgeschlossen werden, dass *Signal* im Zusammenhang des `<terminationHandler>` nur als angeheftetes *Event* oder als *Marker* verwendet wird. Aufgrund dieser Argumentation wird ein `<terminationHandler>` auf einen *Sub-Process* mit einem *Signal-Marker* abgebildet. In dieser Abbildung muss die Sonderbehandlung von `<link>`s einbezogen werden (vgl. 4.4.2.3), da bei `<terminationHandler>`s ausgehende `<link>`s erlaubt sind (vgl. [BPEL2.0, S. 202]). Die Anbindung an ein *Intermediate Catching Signal-Event* mit einer *Sequence Flow Connection* wird von der Visualisierungsfunktion des umgebenden `<scope>` vorgenommen (siehe dazu 4.4.3.3). Die nachfolgende Abbildung zeigt die Visualisierung eines `<terminationHandler>`s als *Sub-Process* mit einem *Catching Signal Marker*.

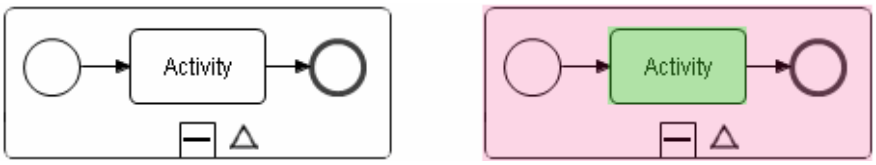


**Abbildung 106: Visualisierung eines <terminationHandler>**

#### 4.4.5.12 <eventHandlers>

Das <eventHandlers>-Konstrukt (siehe 2.4.5.12) dient der Behandlung von Ereignissen. Der Handler ist aktiv, sobald der <process> oder <scope>, in dem er genestet ist, gestartet und noch nicht beendet wurde. In 2.6.1.3.6 und 4.1.3.3 wurde beschrieben, dass mit diesem Konstrukt mehrere Ereignisse parallel verarbeitet werden können. Die Ereignisse, die von einem <eventHandlers> behandelt werden, sind entweder der Eingang einer bestimmten Nachricht (<onEvent>, siehe 2.4.5.12.1) oder ein zeitliches Muster (<onAlarm>, siehe 2.4.5.12.2):

*„There are two types of events. First, events can be inbound messages that correspond to a WSDL operation. Second, events can be alarms, that go off after user-set times. [...] Event handlers are considered a part of the normal behavior of the scope, unlike FCT-handlers*[22]*“* [BPEL2.0, S. 137]

Die Platzierung der Abbildung in einem <process> oder <scope> muss von deren Visualisierungsfunktion vorgenommen werden (siehe dazu 4.4.5.10 und 4.4.3.3). Entscheidend bei der Platzierung ist das Verhältnis, das zwischen einem <eventHandlers>- und einem <faultHandlers>-Konstrukt besteht:

*“When a fault occurs within the inbound message operation specified in <onEvent> itself [...] or its associated scope, the fault MUST be propagated to the associated*[23] *scope first. If unhandled, the fault will be propagated to the ancestor*[24] *scopes chain.”* [BPEL2.0, S. 143]

*“The behavior of a fault handler for a scope C begins by disabling the scope’s event handlers and implicitly terminating all activities enclosed within C that are currently active (including all running event handler instances).”* [BPEL2.0, S. 135]

Das bedeutet, dass ein <faultHandlers> einem <eventHandlers> quasi übergeordnet ist. Die Definitionen der Abbildung von <process> und <scope>s tragen diesem Umstand Rechnung, indem sie den <eventHandlers> innerhalb des Bereichs platzieren, der mit dem <faultHandlers>-Konstrukt des <scope> oder <process> über ein *Error-Event* verbunden ist.
Zusammen mit dem Mapping in [BPMN1.1, S. 151] ist die vollständige Definition der Abbildung möglich. In diesem Mapping wird ein *Catching Intermediate Message-Event*, welches keine eingehende *Sequence Flow* Verbindung hat, in ein <onMessage>-Element in einem <eventHandlers> übersetzt (Anm.: das <onMessage>-Element in einem <eventHandlers> wurde in der Standardisierung durch OASIS in <onEvent> umbenannt). Analog wurde ein *Catching Intermediate Timer-Event* ohne eingehende *Sequence Flow* Verbindung in ein <onAlarm>-Element übersetzt. Die in einem <eventHandlers> genesteten Konstrukte (<onEvent> und <onAlarm>) können somit durch ein *Intermediate-Event* ohne eingehende *Sequence Flow* Verbindung dargestellt werden. Für eine optimale Lesbarkeit kann um diese Abbildung eine *Group* (siehe 2.5.5.6), zur Zusammenführung der Zweige ein *End-Event* und zur Beschreibung eine *Annotation* gezeichnet werden. Eine Sonderbehandlung von <link>s ist in einem <eventHandlers> nicht erforderlich (vgl. [BPEL2.0, S. 104]). Die nachfolgende Abbildung zeigt die Visualisierung eines <eventHandlers>-Konstrukts (rot) mit einer eigenständigen <onEvent>-Funktion (grün) und einer eigenständigen <onAlarm>-Funktion (blau).

[22] FCT: Fault, Compensation and Termination (vgl. [BPEL2.0, S. 132]).
[23] Ein „associated scope“ ist derjenige scope, der innerhalb eines <onEvent>-Konstrukts definiert ist*:* *„The child activity within an event handler MUST be a <scope> activity.”* (siehe [BPEL2.0, S. 137]).
[24] Ein „ancestor scope“ ist derjenige scope, in dem das <eventHandlers>-Konstrukt direkt genestet ist.

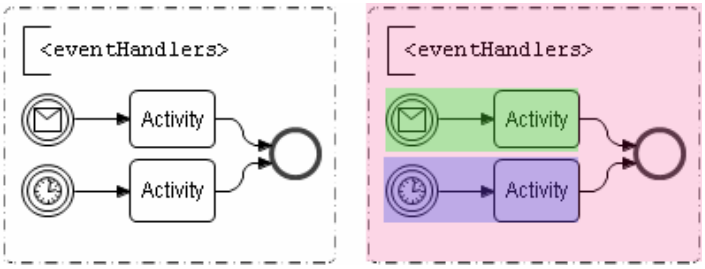


**Abbildung 107: Abbildung von <eventHandlers>**

#### 4.4.5.13 <onEvent>

Wie in der Argumentation der Abbildung eines `<eventHandlers>` (siehe 4.4.5.12) festgestellt wurde, wird ein `<onEvent>`-Konstrukt (siehe 2.4.5.12.1) mit einem *Catching Intermediate Message-Event* ohne eingehende *Sequence Flow* Verbindung dargestellt. Eine *Sequence Flow Connection* verbindet dieses *Event* wird mit der Visualisierung des `<scope>`s, der in dem `<onEvent>` genestetet ist (*„The child activity within an event handler MUST be a <scope> activity“*, [BPEL2.0, S. 137]). Theoretisch ist für dieses Konstrukt keine eigenständige Funktion erforderlich. Die Visualisierung könnte ebenso gut direkt von der `<eventHandlers>`-Funktion vorgenommen werden, siehe dazu auch die Ausführungen zu `<onAlarm>` in 4.4.5.14. Die nachfolgende Abbildung zeigt die eigenständige Visualisierungsfunktion von `<onEvent>` (rot).


**Abbildung 108: Abbildung von <onEvent>**

#### 4.4.5.14 <onAlarm>

Für eine Darstellung des `<onAlarm>`-Konstrukts (siehe 2.4.5.12.2) wurde in der Abbildung von `<eventHandlers>` (siehe dazu 4.4.5.12) festgestellt, dass ein *Catching Intermediate Timer-Event* zur Darstellung eines `<onAlarm>`-Konstrukts eingesetzt werden kann. Dieses *Event* wird durch eine *Sequence Flow Connection* mit der in `<onAlarm>` genesteten Aktivität verbunden. In den Ausführungen zur Abbildung der Aktivität `<pick>` (siehe 4.4.2.4) wurde das `<onAlarm>`-Konstrukt als eigenständige Visualisierungsfunktion referenziert, wegen möglicher Synergien mit dem `<eventHandlers>`-Konstrukt. Allerdings bestehen trotz der Namensgleichheit gewisse Unterschiede in der Verwendung eines `<onAlarm>`-Konstrukts in einem `<pick>` und in einem `<eventHandlers>`. Die Angabe unterschiedlicher Elemente (`<repeatEvery>` bei `<eventHandlers>`) ist für die Visualisierungsfunktion noch nicht entscheidend. Auch, dass ein `<onAlarm>` in einem `<eventHandlers>` nur einen `<scope>` als genestete Aktivität haben darf, ist durch Einbindung von Visualisierungsfunktionen kein Problem. Die Verwendung in `<pick>` erlaubt allerdings eingehende und ausgehende `<link>`s. Deren Visualisierung durch *Signal-Events* wurde in 4.4.2.3 beschrieben. Diese Funktionalität müsste durch die Visualisierungsfunktion von `<onAlarm>` umgesetzt werden. Allerdings ist die Integration in `<pick>` dadurch weitaus weniger flexibel, da in dessen Visualisierungsfunktion implizite *Sequence Flow* Verbindungen eingesetzt werden. Nach diesen Überlegungen erscheint es als zu feingranular, für `<onAlarm>` eine eigene Visualisierungsfunktion zu definieren. Die Visualisierung erfolgt daher in den jeweiligen Visualisierungsfunktionen der Konstrukte `<pick>` und `<eventHandlers>` mit einem *Catching Intermediate Timer-Event* und einer *Sequence Flow* Verbindung von dem *Event* zu der Visualisierung der in `<onAlarm>` genesteten Aktivität.

# 5 Praktische Umsetzung

Die prototypische Umsetzung des hier vorgestellten Ansatzes wurde abschnittsweise durchgeführt. Unter Zuhilfenahme der Dokumentationen [S:Eclipse], [S:Draw2d], [S:EclipseGEF], [S:EclipseEMF], [S:Extensions], [Fern07], [Lee03] und [MDGW04] wurde die Anwendung [S:EclipseBPEL] wie folgt angepasst und erweitert:

- Zuerst wurden die Zeichenfunktionen einzelner Konstrukte mit einer Verzweigung erweitert, in der Art „`if DEFAULT_VISUALIZATION then ... else ...`". Für eine Änderung dieser Variablen zur Laufzeit wurde die graphische Oberfläche erweitert. Das `<process>`-Kontextmenü wurde mit einer Action erweitert die ein Umschalten zwischen der Default-Visualisierung und der BPMN-Visualisierung ermöglicht. Die Funktionalität von [S:EclipseBPEL], Prozesse auch in horizontaler Ausrichtung modellieren zu können, war für diese Anpassung ungemein vorteilhaft.
- Zur Visualisierung in BPMN war dieses Vorgehen für viele Konstrukte bereits ausreichend. Für eine Modellierung in (strukturiertem) BPMN genügte sie allerdings noch nicht. Außer den Zeichenfunktionen mussten für die Anforderung der Modellierung in BPMN unter anderem die *EditParts* und *Figures* (siehe 3.1.2.2) sowie deren *Adapters* angepasst und erweitert werden. Da in [S:EclipseBPEL] in weiten Teilen Gebrauch von dem Vererbungskonzept in Java gemacht wird, ist ein einfaches Umschreiben einzelner Funktionen und Klassen oder die Verzweigung mit `if` (siehe oben) nicht ohne weiteres möglich.
- Um eine saubere und tief greifende Änderung der Architektur der Modellierungsoberfläche zu ermöglichen und dennoch die bereits bestehende Oberfläche zu bewahren, war eine Duplizierung des Plugins `org.eclipse.bpel.ui` notwendig (siehe [Fern07, S. 90 ff.]). Eine Integration dieses Duplikats, sprich Plugins, beziehungsweise seiner Anpassung ist durch die [S:Extensions]-Technologie in Eclipse möglich. Dieses Duplikat konnte ohne die Einschränkungen angepasst werden die von der bestehenden Implementierung auferlegt waren. Auf diese Weise bestehen zwei unterschiedliche Visualisierungen zu einem einzigen Modell. Prinzipiell können durch diesen Mechanismus auch mehrere Visualisierungspakete eingebunden werden, zum Beispiel zur Unterstützung und flexiblen Integration von [BPMN2.0] oder für eine unternehmensspezifische Notation. Die Integration in die Editoroberfläche erfolgt als weitere Registerkarte (Tab).
- Eine Problematik, die in der Implementierung aus Zeitgründen nicht behandelt wurde, ist die Berücksichtigung visualisierungsspezifischer Koordinaten und Längen. Die Modellierung eines `<flow>` erlaubt die freie Platzierung von Konstrukten innerhalb des `<flow>`. Dazu müssen die Koordinaten dieser genesteten Konstrukte gespeichert werden. Die Visualisierungsfunktion von `<flow>` (siehe 4.4.2.1) berücksichtigt die Größe der genesteten Konstrukte und hat zusätzlich eigene Offsets, beispielsweise wenn umgebende *Gateways* gezeichnet werden. Dies erfordert unterschiedliche Koordinaten für die Platzierung der Konstrukte und deswegen die Erweiterung des Datenmodells mit einer entsprechenden Datenstruktur zur Verwaltung. Wenn mehrere graphische Notationen unterstützt werden sollen bietet sich dazu die Verwendung einer Listenstruktur an.

Der angepasste Source Code sowie die gewonnenen Erkenntnisse bei der Implementierung werden nach Veröffentlichung dieser Arbeit an das [S:EclipseBPEL]-Projekt übergeben. Die nachfolgende Abbildung zeigt einen modellierten BPEL-Prozess in der ursprünglichen Darstellung bevor zur BPMN Visualisierung umgeschaltet wird. Anschließend wird die Visualisierung in BPMN und die weitere Modellierung und Bearbeitung dieses Prozesses anhand des Kontext-Menüs der `<reply>`-Aktivität in dieser Ansicht gezeigt.

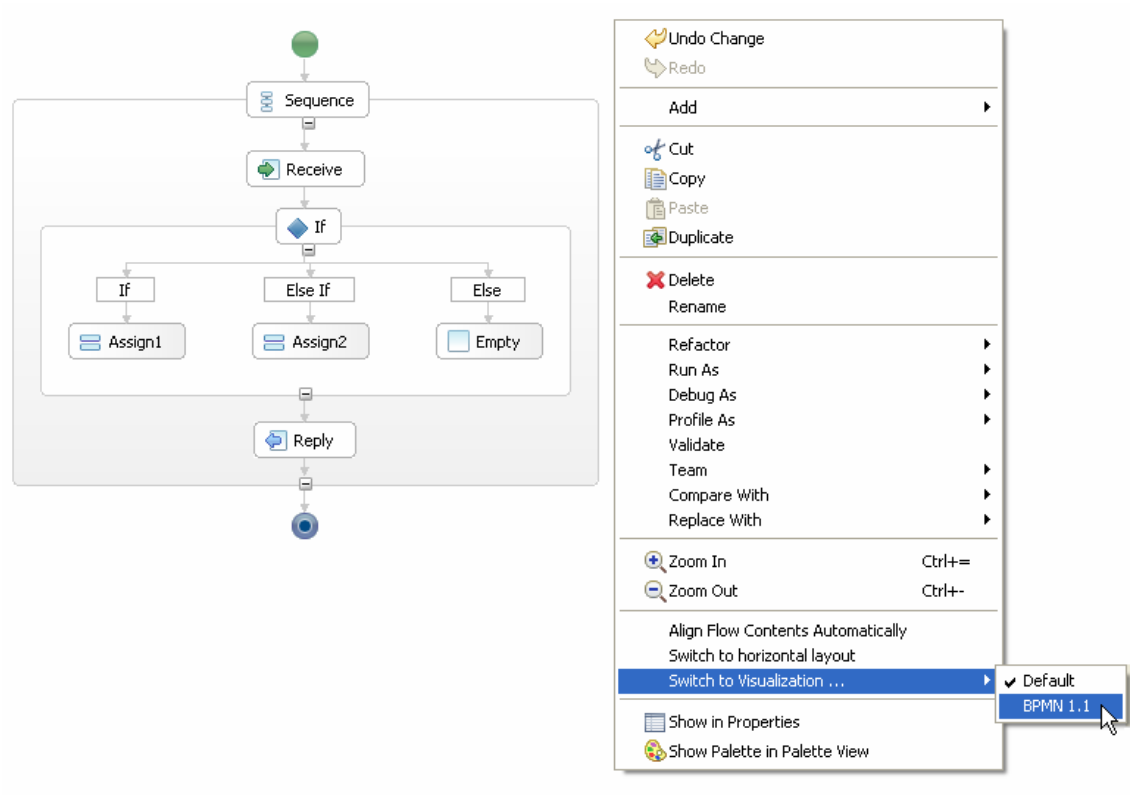


**Abbildung 109: BPEL-Prozess in der Default-Visualisierung in Eclipse BPEL Designer**

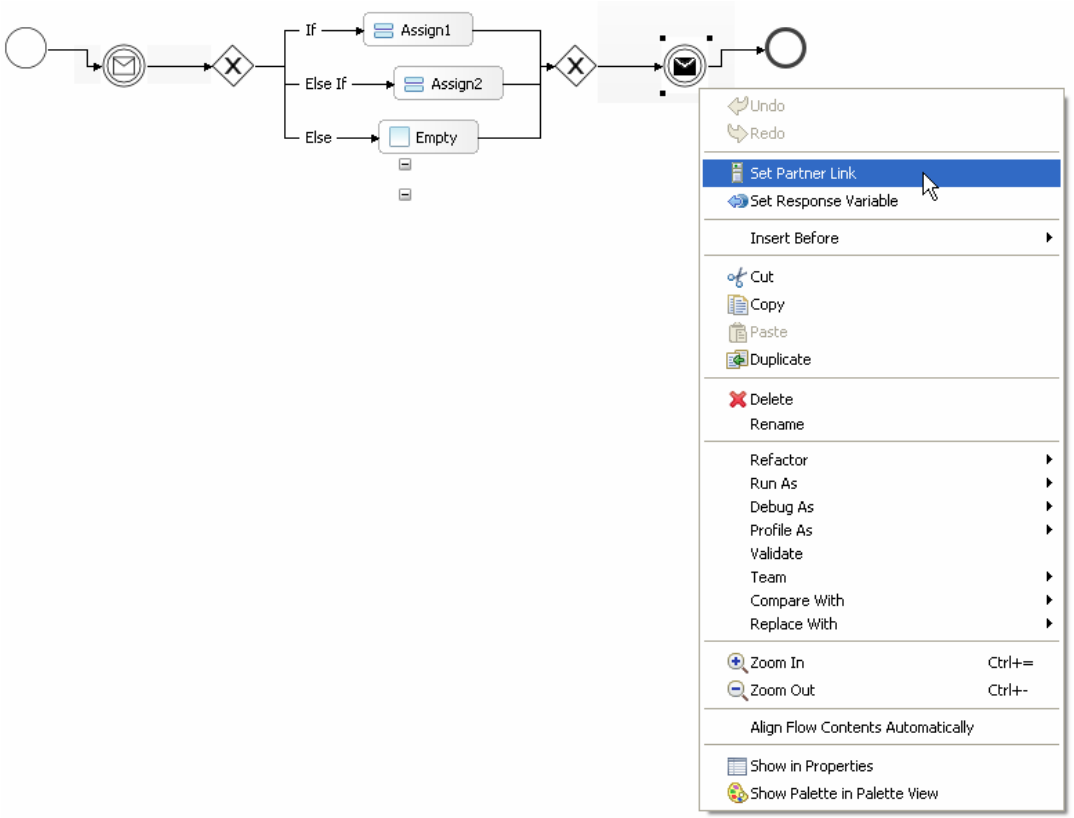


**Abbildung 110: BPEL-Prozess in der BPMN-Visualisierung in Eclipse BPEL Designer**

# 6 Zusammenfassung und Ausblick

## 6.1 Zusammenfassung

In dieser Arbeit wurde eine umfassende Darstellung und Gegenüberstellung der Standards BPMN und BPEL vorgenommen. Dazu gehört unter anderem die Vorstellung, Diskussion und Analyse bestehender Verfahren zu deren Verbindung. Diese Analyse hat zu einem alternativen Ansatz geführt der ohne Modelltransformationen im eigentlichen Sinne auskommt. Die Transformation besteht in der alternativen Herangehensweise lediglich darin, eine besondere Visualisierung eines Modells vorzunehmen. Das der Visualisierung sowie der Code-Erzeugung zu Grunde liegende Metamodell ist in diesem Ansatz BPEL. Die Schwierigkeiten, die sich hingegen bei Modelltransformationen von BPMN zu BPEL ergeben, werden dadurch vermieden.
Der wesentliche Vorteil dieses Ansatzes liegt in der ausschließlichen Erzeugung valider Modelle. Denn jedes Modell, das mit dem graphischen Modellierungswerkzeug erstellt werden kann, lässt die direkte Generierung von verständlichem BPEL-Code zu. Mechanismen zur Modellverifikation können direkt auf den modellierten Prozess auf Objektebene zugreifen, da keine Modelltransformation stattfinden muss. Ein weiterer Vorteil besteht darin, dass sowohl die Modellierung in BPEL als auch die Darstellung in BPMN in einem einzigen Werkzeug realisierbar sind. Dies trägt wesentlich dazu bei, bestehende Prozesse mit einer standardisierten graphischen Notation kommunizieren zu können.
Die Nachteile dieser Herangehensweise sind ebenso in den Ansätzen mit einer Modelltransformation vorhanden, wenn auch auf eine nicht so offensichtliche Art und Weise. Die ausschließliche Verwendung von strukturierten BPEL-Konstrukten unterbindet eine freie Modellierung mit BPMN-Elementen. Dadurch findet die Modellierung noch immer in der Sprache BPEL mit der damit verbundenen strukturierten Denkweise statt, die nach Aussage von Stephen A. White wenig intuitiv ist. BPEL bietet mit dem `<flow>`-Konstrukt und `<link>`s aber auch die Möglichkeit zur freien Modellierung eines Prozesses in Form eines Graphen. Diese Funktionalität reicht bereits für viele Anforderungen aus. Bei einem `<flow>` besteht allerdings die Beschränkung, dass nur azyklische Graphen modelliert werden dürfen, was für die Durchführbarkeit der Dead-Path-Elimination zwingend erforderlich ist. Ein Zyklus wird daher durch ein strukturiertes wiederholendes BPEL-Konstrukt wie `<while>` oder `<repeatUntil>` modelliert. Durch diese kombinierte Verwendung lassen sich überaus komplexe Prozesse (graphisch) modellieren. Das Argument mangelnder intuitiver Bedienbarkeit kann daher entkräftet werden.
Die zahlreichen Einschränkungen in der freien Modellierung mit BPMN, die bei den Verfahren mit Modelltransformationen bestehen, haben denselben Effekt. Um verständlichen und lesbaren Code zu erzeugen, werden beliebige Abläufe (Arbitrary Cycles) verboten. Die Patterns zur Transformation (also die Mustererkennung) beschreiben im Wesentlichen die in BPEL vorhandenen Structured Activities. Damit wird ebenso eine strukturierte (wenig intuitive?) Denkweise forciert. Wenn die Erkenntnisse der letzten 40 Jahre über die Erstellung wartbarer Programme ignoriert werden, kann die Business-IT Gap nicht überbrückt werden! Eine Einschränkung und Anpassung von BPMN ist daher für einen praktischen Einsatz unumgänglich. Ebenso ist ein sehr gutes Verständnis des Prozessmodellierers über die Abläufe in der Maschine erforderlich wenn ein Modell direkt in ausführbaren Code transformiert soll.
Ein weiterer Nachteil ist der mangelnde Nachweis der Korrektheit der definierten Abbildung. Bei manchen der bestehenden Verfahren zur Transformation wurde in dieser Arbeit die Unkorrektheit oder Unvollständigkeit nachgewiesen. Ein derartiger Nachweis ist allerdings erheblich leichter zu erbringen als der Nachweis der Korrektheit. Um diesem Missstand entgegen zu wirken, wurde bei der Definition des Ansatzes in dieser Arbeit großer Wert darauf gelegt, Fundstellennachweise zur Korrektheit zu liefern. In diesem Sinne wurden die Definitionen und Beschreibungen der Spezifikationen herangezogen und daraus verwendete Passagen genau gekennzeichnet. Eine vollständige Korrektheit ist dadurch nicht gewährleistet, wohl aber eine solide und fundierte Diskussionsgrundlage für eine weitere Betrachtung.
Die Abbildung von BPEL auf BPMN unter der Verwendung von BPEL als Metamodell hat sowohl die Einschränkung als auch die Erweiterung von BPMN zur Folge. Im Gegensatz zu den

anderen Ansätzen ist die Abbildung zumindest für eine der beiden Sprachen, nämlich für BPEL vollständig, da eine Einschränkung von BPEL nicht stattfand. Die eher unwesentlichen Erweiterungen von BPMN waren die Einführung eines *Throwing Intermediate Terminate-Events* sowie die Wiedereinführung des *Throwing Intermediate Error-Events*. Diese sind notwendig, um ausgehende *Sequence Flow* Verbindungen aus diesen *Events* zu ermöglichen. Ebenso wurde es gestattet, aus *Events* ausgehende *Conditional Sequence Flow* Verbindungen zu verwenden. Die wesentliche Erweiterung war die Einführung des Konzepts der Dead-Path-Elimination. Durch diese Erweiterung werden entweder impliziter Tokenfluss oder als mögliche Alternative eine andere Art von Tokens in BPMN eingeführt.
Die Einschränkungen von BPMN sind durch die Definition der Abbildung der BPEL-Konstrukte gegeben. Die Verwendung weiterer BPMN-Elemente ist nur mit einer alternativen Abbildung oder der Einführung von Modelltransformationen zu erreichen. Wegen der erheblichen Nachteile von Modelltransformationen wurde diese Möglichkeit nicht näher betrachtet. Die Abbildung der Handler in BPEL hat eine mögliche Erweiterung der Visualisierung aufgezeigt. Diese kann dazu eingesetzt werden, implizite Sachverhalte, wie beispielsweise Default Fault Handler, explizit darzustellen. Ein BPEL-Prozess wird durch diesen Umstand nicht erweitert, aber seine Semantik wird vollständiger wiedergegeben.

Für die Realisierung der graphischen Modellierung von BPEL-Prozessen unter Verwendung der BPMN-Notation wurden zahlreiche Frameworks und Open Source Anwendungen betrachtet. Jedes dieser Frameworks hat sich als für den Bau eines graphischen Modellierungswerkzeugs geeignet herausgestellt. Allerdings erfordert eine komplette Neuentwicklung erheblichen Entwicklungsaufwand. Dieser kann durch Erweiterung und Anpassung einer Open Source Anwendung eingespart werden, sofern diese Anwendung den Ansprüchen an ein graphisches Modellierungswerkzeug genügt. Aus diesem Grund wurden Open Source Anwendungen ausgewählt, die mit Hilfe eines der betrachteten Frameworks gebaut wurden. Dadurch ist sichergestellt, dass die zu Grunde liegende Technologie den gestellten Anforderungen an Flexibilität und Erweiterbarkeit gerecht wird.
Die Definition des Ansatzes zur Abbildung von BPEL auf BPMN hat die nötige Hilfestellung dazu gegeben, eine dieser Anwendungen auszuwählen. In diesem Sinne wurde der „Eclipse BPEL Designer“ aufgrund seiner fortgeschrittenen Unterstützung der graphischen Modellierung von BPEL-Prozessen herangezogen. Ein Teil der definierten Abbildungen wurde in dieser Anwendung prototypisch umgesetzt. Dabei wurde großer Wert auf Erweiterbarkeit und Flexibilität der Lösung gelegt. Die Umsetzung verwendet daher das Plugin-Konzept in Eclipse sowie eine Integration der Visualisierungsfunktionen mittels „Extensions“ und „ExtensionPoints“. Der erarbeitete Code wird nach der Veröffentlichung dieser Arbeit zur möglichen Verwendung in der Open Source Anwendung an das Eclipse BPEL Projekt übergeben.

## 6.2 Ausblick

Bevor ein Blick in die Zukunft geworfen wird, lohnt sich eine Betrachtung des hier vorgestellten Ansatzes unter dem Gesichtspunkt der optimalen Darstellung. Die Abbildung eines `<flow>` in Verbindung mit der eines `<scope>` hat gezeigt, dass sich nicht alle `<link>`s mit einer durchgängigen *Sequence Flow* Verbindung darstellen lassen. Nur wenn ein `<link>` die Grenzen eines *Sub-Process* nicht überschreitet, ist dies möglich. Eine optimale Darstellung von `<link>`s setzt allerdings eine hohe Integration der Visualisierungsfunktionen untereinander voraus. Diese hohe Integration wurde in dieser Arbeit nicht verfolgt. Sie bewirkt wohl eine bessere Darstellung, allerdings auf Kosten der Flexibilität und der Erweiterbarkeit.
Der Eclipse BPEL Designer ermöglicht die graphische Modellierung von BPEL-Prozessen, allerdings nicht von mehreren Prozessen zugleich. Wenn innerhalb einer Oberfläche mehrere Prozesse modellierbar sind, so lässt dies erheblich mehr Spielraum für eine erweiterte Abbildung. Jeder Prozess wird auf einen separaten *Pool* abgebildet. *Message Flow Connections* könnten für die Darstellung der Kommunikation von *Tasks* und *Events* (`<partnerLink>`s von `<receive>`, `<reply>`, `<invoke>` etc.) mit Business Partnern (andere BPEL-Prozesse „oder“ Web Services) eingesetzt werden. In diesen Zusammenhang fällt zudem die Modellierung mit abstrakten Prozessen, die auf diese Weise überaus effizient möglich wäre. Denkbar wäre der Import eines abstrakten Prozesses, dessen Aktivitäten mit *Message-Flow Connections* mit den sendenden und empfangenden Aktivitäten des ausführbaren Prozesses verbunden werden.
In dieser Arbeit wurde eine Abbildung von BPEL vorgenommen, ohne auf die Erweiterungen zu dieser Sprache einzugehen. Zu den prominentesten gehören die Erweiterung von BPEL durch Sub-Prozesse (vgl. [BPEL-SPE]) und die Einbindung von Menschen als Prozessteilnehmer anstatt der ausschließlichen Verwendung von Web Services (vgl. [BPEL4People]). Für eine konsistente Modellierung von BPEL-Prozessen mit diesen Erweiterungen müsste ebenso die graphische Visualisierung in BPMN erweitert und gegebenenfalls angepasst werden.
Der Blick in die Zukunft richtet sich vornehmlich auf die Weiterentwicklung von BPMN zu „Business Process Model and Notation“, die in zukünftiger Arbeit berücksichtigt werden muss, ebenso wie auf die rasante Entwicklungsgeschwindigkeit der Web Service Standards. Die Web Service Standards, die in BPEL eingesetzt werden, haben allerdings keine nennenswerte Auswirkung auf die graphische Darstellung, die in dieser Arbeit vorgestellt wurde. In [BPELlight] wird sogar gezeigt, wie BPEL zur Prozessmodellierung unabhängig von Web Service Technologie erweitert werden kann. Durch diese Erweiterung werden zahlreiche neue Einsatzgebiete von BPEL zur Prozessbeschreibung und –automatisierung erschlossen. Die standardisierte Darstellung eines modellierten Prozesses wird dadurch umso wichtiger. Die in dieser Arbeit beschriebene Darstellung von BPEL in BPMN beruht zwar auf einem Standard, ist dadurch aber noch keine standardisierte Darstellung. Eine standardisierende Organisation wird allerdings schwer zu finden sein. OASIS hat beschlossen, dass ein Mapping von BPEL zu BPMN „out of scope“ ist, während sich die OMG mit dem Mapping nur in umgekehrter Richtung beschäftigt, zumindest war dies bisher der Fall. Es bleibt also trotz der Verwendung einer standardisierten graphischen Notation (wieder) den Herstellern überlassen, wie sie eine graphische Darstellung von BPEL-Prozessen vornehmen.

# Anhang

## Literaturverzeichnis


[AaHe02] Wil M. P. van der Aalst, Kees van Hee:
*Workflow Management – Models, Methods, and Systems*
MIT Press, 2002
http://books.google.de/books?O1xW1_Za-I0C

[AaLa07] Wil M. P. van der Aalst, Kristian B. Lassen:
*Translating Unstructured Workflow Processes to Readable BPEL: Theory and Implementation*
Information and Software Technology, 2007
http://is.tm.tue.nl/staff/wvdaalst/publications/z23.pdf

[Aals03] Wil M. P. van der Aalst:
*Patterns and XPDL: A Critical Evaluation of the XML Process Definition Language*
2003
http://www.workflowpatterns.com/documentation/documents/ce-xpdl.pdf

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju:
*Web Services – Concepts, Architectures and Applications*
Springer, 2004

[AHKB03] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartosz Kiepuszewski, Allan P. Barros:
*Workflow Patterns*
Juli 2003
http://www.workflowpatterns.com/documentation/documents/wfs-pat-2002.pdf

[AsMa72] Edward Ashcroft, Zohar Manna:
*The Translation of "Goto" Programs to "While" Programs*
Proceedings of the IFIP Congress 71, vol. 1, 250-255
Association for Computing Machinery (ACM), 1972
http://portal.acm.org/citation.cfm?id=1241535
ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/71/188/CS-TR-71-188.pdf

[Asti07] Sylvain Astier:
*TrainingKit BPMN 1.1*
November 2007
http://www.diveintobpm.org/laszlo-explorer/Pages/TrainingKit%20BPMN%201.1%20-%20Version%201.0.1.zip

[Bart05] Martin Bartonitz:
*Wachsen die BPM- und Workflow-Lager zusammen?*
BPM-Netzwerk, 2005
http://www.bpm-netzwerk.de/articles/66
http://www.bpm-netzwerk.de/articles/17
http://www.bpm-netzwerk.de/pic/xlarge/339.gif

[BeNe97] Philip A. Bernstein, Eric Newcomer:
*Principles of Transaction Processing*
Morgan Kaufmann, 1997
http://books.google.de/books?id=G-e7tvWJxZoC

[BFV07] Giso Bartels, Philipp Frank, Marco Völz:
*Vergleich von BPMN-Modellierwerkzeugen*
Universität Stuttgart, Institut für Architektur von Anwendungssystemen, Fachstudie Nr. 75
August 2007
http://elib.uni-stuttgart.de/opus/volltexte/2007/3385/pdf/FACH_0075.pdf

[BPDM] Object Management Group:
*Business Process Definition Metamodel Request For Proposal*
Januar 2003
http://www.omg.org/cgi-bin/doc?bei/03-01-06

[BPEL1.0] Francisco Burbera, Yaron Goland, Johannes Klein, Frank Leymann, Dieter Roller, Satish Thatte, Sanjiva Weerawarana:
*Business Process Execution Language for Web Services Version 1.0*
Juli 2002
ftp://www6.software.ibm.com/software/developer/library/BPEL 2.01.pdf

[BPEL1.1] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, Sanjiva Weerawarana:
*Business Process Execution Language for Web Services Version 1.1*
Mai 2003
http://www-128.ibm.com/developerworks/library/specification/BPEL 2.0/

[BPEL2.0] Alexandre Alves, Assaf Arkin, Sid Askary, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter König, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, Alex Yiu:
*OASIS Web Services Business Process Execution Language Version 2.0*
April 2007
http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[BPEL4People] Ashish Agrawal, Mike Amend, Manoj Das, Mark Ford, Chris Keller, Matthias Kloppmann, Dieter König, Frank Leymann, Ralf Müller, Gerhard Pfau Karsten Plösser, Ravi Rangaswamy, Alan Rickayzen, Michael Rowley, Patrick Schmidt, Ivana Trickovic, Alex Yiu, Matthias Zeller:
*WS-BPEL Extension for People*
Juni 2007
http://www.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/

[BPELlight] Jörg Nitzsche, Tammo van Lessen, Dimka Karastoyanova, Frank Leymann:
*BPEL$^{light}$*
5$^{th}$ International Conference on Business Process Management (BPM 2007)
ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/INPROC-2007-24/INPROC-2007-24.pdf

[BPEL-SPE] Matthias Kloppmann, Dieter König, Frank Leymann, Gerhard Pfau, Alan Rickayzen, Claus von Riegen, Patrick Schmidt, Ivana Trickovic:
*WS-BPEL Extension for Sub-processes – BPEL-SPE*
September 2005
http://www.ibm.com/developerworks/library/specification/ws-bpelsubproc/

[BPML] Assaf Arkin:
*Business process modeling language 1.0*
Juni 2002
http://www.omg.org/technology/documents/br_pm_spec_catalog.htm

[BPMN1.0] Object Management Group:
*Business Process Modeling Notation 1.0, Final Adopted Specification*
Februar 2006
http://www.bpmn.org/Documents/OMG%20Final%20Adopted%20BPMN%201-0%20Spec%2006-02-01.pdf

[BPMN1.1] Object Management Group:
*Business Process Modeling Notation 1.1, Beta 3 Specification*
Juni 2007
http://doc.omg.org/dtc/2007-06-03
http://www.bpmn.org/

[BPMN2.0] Object Management Group:
*Business Process Model and Notation (BPMN) 2.0 Request For Proposal*
Juni 2007
http://www.bpmn.org/Documents/BPMN%202-0%20RFP%2007-06-05.pdf

[BrWi05] Kai Brüssau, Oliver Widder (Hrsg.):
*Eclipse – Die Plattform*
Software & Support Verlag, 2005

[CJK07] Steve Cook, Gareth Jones, Stuart Kent:
*Domain Specific Development with Visual Studio DSL Tools*
Addison-Wesley, Juli 2007

[Cola07] Bruno M. C. Colaço:
*Web-based editor for WS-BPEL Processes*
Universität Stuttgart, Institut für Architektur von Anwendungssystemen,
Bachelor Projekt Nr. 86
Dezember 2007

[Czuc07a] Martin Czuchra:
*Oryx: Embedding Business Process Data into the Web*
Universität Potsdam, Hasso Plattner Institut,
Bachelorarbeit
Juni 2007

[Czuc07b] Martin Czuchra:
*Oryx Business Process Editor – Designing Business Processes on the Web*
Apple Worldwide Developers Conference (Poster)
Juni 2007
http://images.apple.com/science/poster/pdf/130_czuchra.pdf

[Daum05] Berthold Daum:
*Rich-Client-Entwicklung mit Eclipse 3.1*
dpunkt Verlag, 2005

[DeSh06a] John Deeb, Devesh Sharma:
*Business Processes Make a World of Difference*
Business Integration Journal , April 2006
http://www.bijonline.com/index.cfm?section=article&aid=244#

[DeSh06b] John Deeb, Devesh Sharma:
*BPEL: Good for business*
GRID today, Juli 2006
http://www.gridtoday.com/grid/717789.html

[Dijk68] Edsger W. Dijkstra:
*Go To Statement Considered Harmful*
Communications of the Association for Computing Machinery (ACM)
Vol. 11, No. 3, März 1968
http://www.acm.org/classics/oct95/

[Dijk72] Edsger W. Dijkstra:
*The humble programmer*
Turing Award Lecture, 1972
http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF

[DKLW07] Gero Decker, Oliver Kopp, Frank Leymann, Mathias Weske:
*BPEL4Chor: Extending BPEL for Modeling Choreographies*
Proceedings of the IEEE 2007 International Conference on Web Services (ICWS)
Juli 2007
http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2007-16&mod=0&engl=0&inst=IAAS

[Dubr07] Jean-Jacques Dubray:
*The Seven Fallacies of Business Process Execution*
Dezember 2007
http://www.infoq.com/articles/seven-fallacies-of-bpm

[eCla06] eClarus:
*eClarus Business Process Modeler for SOA Architects*
2005 - 2006
http://www.eclarus.com/pdf/DS-SOA-05-06-v1-0.pdf
http://www.eclarus.com/bpel_bpmn_examples.html

[EPL1.0] Eclipse Foundation:
*Eclipse Public License (EPL)*
Version 1.0
http://www.eclipse.org/org/documents/epl-v10.php

[Fern07] Javier V. Fernandez:
*BPEL with Explicit Data Flow: Model, Editor, and Partitioning Tool*
Universität Stuttgart, Institut für Architektur von Anwendungssystemen, Diplomarbeit Nr. 2616
Oktober 2007
http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2616&mod=0&engl=0&inst=IAAS

[Gao06] Yi Gao:
*BPMN-BPEL Transformation and Round Trip Engineering*
Mai 2006
http://www.eclarus.com/pdf/BPMN_BPEL_Mapping.pdf

[inno07] innoQ:
*Web Service Standards overview*
Q1 2007
http://www.innoq.com/soa/ws-standards/poster/

[Inta07] Intalio:
*From Modeling to Execution in the Enterprise using BPMN and BPEL*
EclipseCON 2007
http://www.eclipse.org/stp/bpmn/eclipsecon2007/Modeling_BPM_For_Execution_BPMN_2_BPEL.pdf

[Kapl06] Michael Kaplan:
*Graphisches BPEL Modellierungstool für parametrisierte Prozesse und Templates*
Universität Stuttgart, Institut für Architektur von Anwendungssystemen, Diplomarbeit Nr. 2439
Juni 2006

[KCHK04] Martin Keen, Jonathan Cavell, Sarah Hill, Chee K. Kee, Wendy Neave, Bradley Rumph, Hoang Tran:
*BPEL4WS Business Processes with WebSphere Business Integration: Understanding, Modeling, Migrating*
Dezember 2004
http://www.redbooks.ibm.com/abstracts/sg246381.html

[Kent08] Stuart Kent:
*Exploring the new Domain-Specific Language (DSL) Tools*
2008
http://channel9.msdn.com/Showpost.aspx?postid=246477

[Knut74] Donald E. Knuth:
*Structured Programming with "go to" Statements*
Association for Computing Machinery (ACM), 1974
http://portal.acm.org/citation.cfm?id=356640

[Kram06] Volker Kramberg:
*Pattern-based Evaluation of IBM WebSphere BPEL*
Universität Stuttgart, Institut für Architektur von Anwendungssystemen, Studienarbeit, Nr. 2124
Juli 2006
http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=STUD-2052&mod=0&engl=0

[Kunn07] Thomas Künneth:
*Einstieg in Eclipse 3.3*
Galileo Press, 2007

[Lee03] Daniel Lee:
*Display a UML Diagram using Draw2D*
August 2003
http://www.eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html

[LeRo00] Frank Leymann, Dieter Roller:
*Production Workflow – Concepts and Techniques*
Prentice Hall, 2000

[LRT03] Frank Leymann, Dieter Roller, Satish Thatte:
*Goals of the BPEL4WS Specification*
August 2003
http://xml.coverpages.org/BPEL4WS-DesignGoals.pdf

[MDGW04] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, Philipe Vanderheyden:
*Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*
Februar 2004
http://www.redbooks.ibm.com/abstracts/sg246302.html

[MLZ05] Jan Mendling, Kristian B. Lassen, Uwe Zdun:
*Transformation Strategies between Block-Oriented and Graph-Oriented Process Modelling Languages*
2005
http://wi.wu-wien.ac.at/home/mendling/publications/TR05-Strategy.pdf

[MLZ06] Jan Mendling, Kristian B. Lassen, Uwe Zdun:
*Transformation Strategies between Block-Oriented and Graph-Oriented Process Modelling Languages*
Multikonferenz Wirtschaftsinformatik 2006
http://wi.wu-wien.ac.at/home/mendling/XML4BPM2006/XML4BPM-Mendling.pdf

[MOF2.0] Object Management Group (OMG):
*Meta Object Facility (MOF) Core Specification 2.0*
Januar 2006
http://www.omg.org/docs/formal/06-01-01.pdf

[Mueh07] Michael zur Muehlen:
*BPM Conference Tutorials – Business Process Management Standards*
5th International Conference on Business Process Management
September 2007
http://bpm07.fit.qut.edu.au/program/slides/Thursday/Thursday-Tutorials/Muehlen.pdf

[Muly05] Nataliya Mulyar:
*Pattern-based Evaluation of Oracle-BPEL*
BPM Center Report BPM-05-24
2005
http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2005/BPM-05-24.pdf

[OADH06] Chun Ouyang, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede:
*Translating BPMN to BPEL*
BPM Center Report BPM-06-02
2006
http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2006/BPM-06-02.pdf

[OASI08] Organization for the Advancement of Structured Information Standards (OASIS):
*WS-BPEL Technical Committee (TC)*
http://www.oasis-open.org/committees/wsbpel/
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

[ODBH05] Chun Ouyang, Marlon Dumas, Stephan Breutel, Arthur H. M. ter Hofstede:
*Translating Standard Process Models to BPEL*
BPM Center Report BPM-05-27
2005
http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2005/BPM-05-27.pdf

[ODHA06a] Chun Ouyang, Marlon Dumas, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst:
*From BPMN Process Models to BPEL Web Services*
2006
http://wwwis.win.tue.nl/~wvdaalst/publications/p345.pdf

[ODHA06b] Chun Ouyang, Marlon Dumas, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst:
*From Business Process Models to Process-oriented Software Systems: The BPMN to BPEL Way*
BPM Center Report BPM-06-27
2006
http://eprints.qut.edu.au/archive/00005266/01/5266.pdf

[ODHA07] Chun Ouyang, Marlon Dumas, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst:
*Pattern-based Translation of BPMN Process Models to BPEL Web Services*
International Journal of Web Services Research, 2007
http://is.tm.tue.nl/staff/wvdaalst/publications/z9.pdf

[Oest06] Bernd Oestereich:
*Analyse und Design mit UML 2.1*
Oldenbourg, 2006

[OMG08] The Object Management Group (OMG):
*OMG.org*
http://www.omg.org/
http://www.omg.org/technology/documents/spec_catalog.htm
http://bmi.omg.org/

[Ozgu07] Turhan Özgür:
*Comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks for Domain-Specific Modeling in the context of the Model-Driven Development*
Blekinge Institute of Technology, Master Thesis No. MSE-2007:07
Januar 2007
http://www.bth.se/fou/cuppsats.nsf/0/6210cafdc9323e03c1257267004c1e56

[Pete07] Nicolas Peters:
*Oryx Stencil Set Specification*
Universität Potsdam, Hasso Plattner Institut,
Bachelorarbeit
Juni 2007

[Pfit07] Kerstin Pfitzner:
*Choreography Configuration for BPMN*
Universität Stuttgart, Institut für Architektur von Anwendungssystemen,
Diplomarbeit Nr. 2618
Dezember 2007
http://elib.uni-stuttgart.de/opus/volltexte/2007/3375/index.html

[Pola07] Daniel Polak:
*Oryx Stencil Set Implementation*
Universität Potsdam, Hasso Plattner Institut,
Bachelorarbeit
Juni 2007

[Pons06] PONS:
*Kompaktwörterbuch Englisch*
Ernst Klett Sprachen, 2006

[Rath06] Istvàn Ràth:
*Declarative Specification of Domain Specific Visual Languages*
Budapest University of Technology and Economics, Department of Measurement and Information Systems, Master's Thesis
Mai 2006
http://home.mit.bme.hu/~rath/pub/diploma.pdf

[ReMe06a] Jan Recker, Jan Mendling:
*On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages*
Queensland University of Technology, Vienna University of Economics and Business Administration
2006
http://citeseer.ist.psu.edu/recker06translation.html

[ReMe06b] Jan Recker, Jan Mendling:
*On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages*
EMMSAD 2006
http://wi.wu-wien.ac.at/home/mendling/talks/Emmsad2006.ppt

[RFC2119] Scott Bradner:
*Key words for use in RFCs to Indicate Requirement Levels, RFC 2119*
Harvard University, März 1997
http://www.ietf.org/rfc/rfc2119.txt

[RHAM06] Nick Russell, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, Nataliya Mulyar:
*Workflow Control-Flow Patterns: A Revised View*
BPM Center Report BPM-06-22
2006
http://www.workflowpatterns.com/documentation/documents/BPM-06-22.pdf
http://www.workflowpatterns.com/evaluations/standard/bpel.php
http://www.workflowpatterns.com/evaluations/standard/bpmn.php

[Scho00] Uwe Schöning:
*Logik für Informatiker*
Spektrum Akademischer Verlag, 5. Aufl. 2000

[Scho01] Uwe Schöning:
*Theoretische Informatik – kurzgefasst*
Spektrum Akademischer Verlag, 4. Aufl. 2001

[Schu07] David Schumm:
*A Graphical Tool for Modeling BPEL 2.0 processes*
Universität Stuttgart, Institut für Architektur von Anwendungssystemen, Studienarbeit, Nr. 2124
Oktober 2007
http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=STUD-2124&engl=0&inst=fak

[ScSc05] Philipp Schill, Ralph Schmauder:
*Codegenerierung mit dem Eclipse Modeling Framework und JET*
2005
http://www.sigs.de/publications/os/2005/01/schmauder_schill_OS_01_05.pdf

[Silv06] Bruce Silver:
*Bidirectional Interchange: The Next Step in Process Modeling*
Februar 2006
http://www.bpminstitute.org/articles/article/article/bpms-watch-bidirectional-interchange-the-next-step-in-process-modeling.html

[Silv07a] Bruce Silver:
*Roundtripping Revisited*
November 2007
http://www.brsilver.com/wordpress/2007/11/28/roundtripping-revisited/

[Silv07b] Bruce Silver:
*Dialog with Dumas on Roundtripping*
November 2007
http://www.brsilver.com/wordpress/2007/11/30/dialog-with-dumas-on-roundtripping/

[Tane01] Andrew S. Tanenbaum:
*Modern Operating Systems, 2nd Edition*
Prentice Hall, 2001
http://www.cs.vu.nl/~ast/

[Tsch07] Willi Tscheschner:
*Oryx Dokumentation*
Universität Potsdam, Hasso Plattner Institut,
Bachelorarbeit
Juni 2007

[WADH03] Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede:
*Analysis of Web Services Composition Languages: The Case of BPEL4WS*
22nd International Conference on Conceptual Modeling
2003
http://www.workflowpatterns.com/documentation/documents/bpel_er.pdf

[WADH05] Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, Nick Russell:
*Pattern-based analysis of BPMN*
Center Report BPMN-05-26
2005
http://www.workflowpatterns.com/documentation/documents/BPM-05-26.pdf

[WCLS06] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, Donald F. Ferguson:
*Web Services Platform Architecture*
Prentice Hall, 2006

[WfMC08] The Workflow Management Coalition (WfMC):
*WfMC.org*
http://www.wfmc.org/
http://www.wfmc.org/standards/referencemodel.htm
http://www.wfmc.org/standards/docs.htm#XPDL_Spec_Final

[Whit04] Stephen A. White:
*Process modeling notations and workflow patterns*
2004
http://www.bpmn.org/Documents/Notations%20and%20Workflow%20Patterns.pdf

[Whit05a] Stephen A. White:
*Using BPMN to Model a BPEL Process*
Februar 2005
http://www.bpmn.org/Documents/Mapping%20BPMN%20to%20BPEL%20Example.pdf

[Whit05b] Stephen A. White:
*BPMN Fundamentals*
Dezember 2005
http://www.omg.org/docs/pm/05-12-06.ppt

[Whit06] Stephen A. White:
*Introduction to BPMN*
Oktober 2006
http://www.bpmn.org/Documents/OMG%20BPMN%20Tutorial.pdf

[WSPE07] WSPER (Web, Service, Process, Event & Resource):
*Unofficial BPMN 1.0 Metamodel*
2007
http://www.wsper.org/bpmn10.html

[WWNS07] Egon Wuchner, Jules White, Andrey Nechyourenko, Douglas C. Schmidt:
*Das Generic Eclipse Modeling System (GEMS)*
2007
http://www.sigs.de/publications/os/2007/04/wuchner_white_OS_04_07.pdf

[W3C01] W3C:
*Web Services Description Language (WSDL) 1.1*
März 2001
http://www.w3.org/TR/wsdl

[W3C04] W3C:
*Web Services Addressing (WS-Addressing) Member Submission*
August 2004
http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/

**Softwareverzeichnis**

[S:ActiveBPEL] Active Endpoints:
*ActiveBPEL Designer*
Version: 4.1
http://www.active-endpoints.com/active-bpel-designer.htm

[S: Ant] Apache Software Foundation:
*Ant*
Version: 1.7.0
http://ant.apache.org/

[S:ApacheAxis2] Apache Software Foundation:
*Apache Axis2/Java*
Version: 1.3
http://ws.apache.org/axis2/

[S:ApacheHTTP] Apache Software Foundation:
*HTTP Server Project*
Version 2.2
http://httpd.apache.org/

[S:ApacheODE] Apache Software Foundation:
*Apache ODE (Orchestration Director Engine)*
Version: 1.1.1
http://ode.apache.org/

[S:Aptana] Aptana:
*End-to-End Ajax: IDE for Web development*
Version: Milestone 8+
http://www.aptana.com/

[S:BPELnews] Eclipse BPEL Designer:
*Newsgroup*
news.eclipse.org // eclipse.technology.bpel-designer

[S:BPELValidator] Active Endpoints:
*Active Endpoints OnDemand BPEL 2.0 Validation*
Januar 2008
http://www.activebpel.org/BPEL_Validator/

[S:BPEL1.1XSD] Microsoft, IBM, BEA, SAP, Siebel:
*BPEL 1.1 XML Schema Definitions*
März 2003
http://schemas.xmlsoap.org/ws/2003/03/business-process/

[S:BPEL2.0XSD] OASIS:
*XML Schema Definitions for executable BPEL processes*
April 2007
http://docs.oasis-open.org/wsbpel/2.0/OS/process/executable/ws-bpel_executable.xsd

[S:BPEL4Chor] BPEL4Chor:
*Choreography Extension for BPEL*
2008
http://www.bpel4chor.org/editor/

[S:CodeDOM] .NET Framework Class Library:
*Code Document Object Model (CodeDOM)*
Version: .NET Framework 2.0
http://msdn2.microsoft.com/en-us/library/f1dfsbhc.aspx

[S:CodeSmith] CodeSmith Tools:
*CodeSmith*
Version: 4.1.3
http://www.codesmithtools.com/

[S:Draw2d] Eclipse Graphical Editing Framework Project (GEF):
*Drawd2d Eclipse Plugin*
Version: 3.2
http://www.eclipse.org/gef/?project=draw2d
http://help.eclipse.org/help31/index.jsp?topic=/org.eclipse.gef.doc.isv/index.html

[S:DreamSpark] Microsoft:
*DreamSpark*
https://downloads.channel8.msdn.com/

[S:Eclipse] Eclipse Foundation:
*Eclipse Development Platform*
Version: 3.3.1.1 (Europa)
http://www.eclipse.org/
http://wiki.eclipse.org/
http://www.ibm.com/developerworks/library/os-ecov/

[S:EclipseBPEL] Eclipse BPEL Project:
*Eclipse BPEL Designer*
Version: 0.3.0
http://www.eclipse.org/bpel/
http://www.eclipse.org/bpel/users/m3.php

[S:EclipseBPMN] Eclipse SOA Tools Platform Project (STP):
*BPMN Modeler*
Version: 0.7.0
http://www.eclipse.org/stp/bpmn/

[S:EclipseDTP] Eclipse Data Tools Platform Project (DTP):
*Data Tools Platform SDK*
Version 1.5.1
http://www.eclipse.org/datatools/

[S:EclipseEMF] Eclipse Modeling Framework Project (EMF):
*Eclipse Modeling Framework*
Version: 2.3.0, build M200712211131
http://www.eclipse.org/modeling/emf/
http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html
http://www.eclipse.org/articles/Article-GEF-EMF/gef-emf.html
http://www.eclipse.org/articles/Article-GEF-editor/gef-schema-editor.html
http://www.eclipse.org/articles/Article-Using%20EMF/using-emf.html

[S:EclipseGEF] Eclipse Graphical Editing Framework Project (GEF):
*Graphical Editing Framework*
Version: 3.3.1
http://www.eclipse.org/gef/

[S:EclipseGMF] Eclipse Graphical Modeling Framework Project (GMF):
*Graphical Modeling Framework*
Version 2.0.1 (SDK & Base Package)
http://www.eclipse.org/gmf/
http://wiki.eclipse.org/GMF_Documentation_Index
http://www.eclipse.org/articles/Article-FromFrontendToCode-MDSDInPractice/article.html
http://www.eclipse.org/articles/Article-Introducing-GMF/article.html

[S:EclipseSTP] SOA Tools Platform Project (STP):
*SOA Tools Platform*
Version: 0.7.0
http://www.eclipse.org/stp/

[S:EclipseWiki] Eclipse.org Wiki:
*Eclipsepedia*
2008
http://wiki.eclipse.org/Main_Page
http://wiki.eclipse.org/GMF_GenModel
http://wiki.eclipse.org/index.php/GMF_Tutorial
http://wiki.eclipse.org/index.php/STP_BPMN_Modeler_Status
http://wiki.eclipse.org/index.php/STP_BPMN_Presentation_Hands_on_tutorial

[S:EclipseWTP] Eclipse Web Tools Platform Project (WTP):
*Web Tools Platform*
Version: 2.0.1, build 20070926042742
http://www.eclipse.org/webtools/

[S:eRDF] Talis:
*Embedded RDF (eRDF)*
Oktober 2006
http://research.talis.com/2005/erdf/wiki/

[S:Extensions] (A) Benoit Marchal:
*Working XML: Define and load extension points*
Februar 2005
http://www.ibm.com/developerworks/xml/library/x-wxxm29.html

(B) Manfred Henning, Heiko Seeberger:
*Einführung in den „Extension Point“ Mechanismus von Eclipse*
Januar 2008
http://www.sigs.de/publications/js/2008/01/hennig_seeberger_JS_01_08.pdf

(C) Martin Lippert, Gernot Neppert:
*Flexible Architekturen mit der Eclipse Extension-Point-Technologie*
Eclipse Forum 2007
http://www.martinlippert.org/events/JAX2007-FlexibleArchitekturenMitEclipse.pdf

(D) Eclipse Dokumentation:
*Platform Plug-in Developer Guide*
Version: 3.1
http://archive.eclipse.org/eclipse/downloads/drops/R-3.1-200506271435/org.eclipse.platform.doc.isv.3.1.pdf.zip

(E) Eclipse Dokumentation:
*Plug-in Development Environment Guide*
Version: 3.1
http://archive.eclipse.org/eclipse/downloads/drops/R-3.1-200506271435/org.eclipse.pde.doc.user.3.1.pdf.zip

[S:EXTJS] Ext JS:
*Extended JavaScript Library*
Version: 1.1
http://extjs.com/
http://www.extjs.com/deploy/ext/docs/

[S:GEMS] Generic Eclipse Modeling System (GEMS):
*Generic Eclipse Modeling System*
Version 3.0 RC1
http://www.eclipse.org/gmt/gems/

[S:Geronimo] Apache Software Foundation:
*Geronimo*
Version: 2.1
http://geronimo.apache.org/

[S:Inkscape] Inkscape:
*Open Source Scalable Vector Graphics Editor*
Version: 0.45.1
http://www.inkscape.org/

[S:Intalio] Intalio:
*Intalio BPMS Designer*
Version: 5.0.0
http://www.intalio.com/products/designer/

[S:JDK5] Sun Microsystems:
*Java Standard Edition (J2SE) JDK 5.0 Update 14*
Version: 1.5.0_14
http://java.sun.com/javase/downloads/index_jdk5.jsp

[S:JET] Model to Text Project (M2T):
*Java Emitter Templates (JET)*
http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html
http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html

[S:JSON] JSON:
*JavaScript Object Notation*
http://www.json.org/

[S:JUnit] JUnit Project:
*JUnit Testing Framework*
Version 4.4
http://sourceforge.net/projects/junit/

[S:MetaEdit+] MetaCASE:
*MetaEdit+*
Version: 4.5
http://www.metacase.com/

[S:Mono] Mono Project:
*Mono*
http://www.mono-project.com/Start

[S:MSDSL] Microsoft:
*Domain-Specific Language Tools (DSL Tools)*
Version: 1.0
http://msdn2.microsoft.com/en-us/teamsystem/aa718368.aspx
http://msdn2.microsoft.com/de-de/library/bb126259(en-us,VS.80).aspx
http://msdn2.microsoft.com/en-us/library/bb381702(vs.80).aspx
http://msdn2.microsoft.com/en-us/library/bb126425.aspx
http://msdn2.microsoft.com/en-us/library/aa730848(VS.80).aspx
http://msdn2.microsoft.com/en-us/library/bb126445(VS.80).aspx

[S:MSVStudio] Microsoft:
*Visual Studio*
Version: 2005 Professional
http://msdn2.microsoft.com/de-de/vstudio/aa700919.aspx

[S:Oryx] Universität Potsdam, Hasso Plattner Institut – Business Process Technology:
*Oryx*
Version: Februar 2008
http://bpt.hpi.uni-potsdam.de/Oryx/WebHome/
http://bpt.hpi.uni-potsdam.de/Oryx/NewsIntro/
http://b3mn.hpi.uni-potsdam.de/server.php
http://code.google.com/p/oryx-editor/
http://code.google.com/p/oryx-editor/source/browse/

[S:Paint.NET] Paint.NET:
*Free Software for Digital Photo Editing*
Version: 4.00
http://www.getpaint.net/

[S:PHP] The PHP Group:
*PHP: Hypertext Preprocessor*
Version: 5.2.5
http://www.php.net/docs.php

[S:PrototypeJS] Prototype JavaScript framework:
*Easy Ajax and DOM manipulation for dynamic web applications*
Version: 1.6
http://www.prototypejs.org/

[S:RDF] W3C:
*Resource Description Framework (RDF) 1.0*
Februar 2004
http://www.w3.org/RDF/

[S:Subclipse] openCollabnet:
*Subclipse – Subversion Eclipse Plugin*
Version: 1.2
http://subclipse.tigris.org/

[S:SVG] W3C:
*Scalable Vector Graphics (SVG) 1.1*
Januar 2003
http://www.w3.org/TR/2003/REC-SVG11-20030114/

[S:XHTML] W3C:
*Extensible Hypertext Markup Language 1.1*
http://www.w3.org/TR/2001/REC-xhtml11-20010531/

[S:XMLMarker] Symbol Click:
*XML Marker*
Version 1.1
http://symbolclick.com/

**Glossar**

| | |
|---|---|
| ACID | Atomicity, Consistency, Isolation, Durability |
| ACM | Association for Computing Machinery |
| AI | Application Integration |
| AJAX | Asynchronous Javascript and XML |
| aka | also known as |
| Anm. | Anmerkung |
| API | Application Programming Interface |
| ARIS | Architektur Integrierter Informationssysteme |
| AWT | Abstract Window Toolkit |
| | |
| B2B | Business-To-Business |
| BAM | Business Activity Monitoring |
| BMI | Business Modeling & Integration |
| BMP | Bitmap |
| BPD | Business Process Diagram |
| BPDM | Business Process Definition Metamodel |
| BPE | Business Process Execution |
| BPEL | Business Process Execution Language |
| BPEL4WS | Business Process Execution Language for Web Services |
| BPI | Business Process Integration |
| BPM | Business Process Management |
| BPMI | Business Process Management Initiative |
| BPML | Business Process Modeling Language |
| BPMN | Business Process Modeling Notation (aka BPMN 1.0, BPMN 1.1) |
| BPMN | Business Process Model and Notation (aka BPMN 2.0) |
| BPMS | Business Process Management System |
| BPR | Business Process Reengineering |
| | |
| CLI | Common Language Infrastructure |
| CORBA | Common Object Request Broker Architecture |
| CPN | Coloured Petri-Net |
| CVS | Concurrent Versions System |
| | |
| DBMS | Database Management System |
| DMTF | Distributed Management Task Force |
| DOM | Document Object Model |
| DPE | Dead-Path-Elimination |
| DSL | Domain-Specific Language |
| DSM | Domain-Specific Modeling |
| DTD | Document Type Definition |
| DTF | Domain Task Force |
| DTP | Data Tools Platform |
| | |
| EAI | Enterprise Application Integration |
| EIS | Enterprise Information System |
| EMF | Eclipse Modeling Framework |
| EMOF | Essential MOF |
| EPK | Ereignisgesteuerte Prozesskette |
| EPL | Eclipse Public License |
| EPR | Endpoint Reference |
| ESB | Enterprise Service Bus |

| | |
|---|---|
| f. | folgende |
| FAQ | Frequently Asked Questions |
| FCT | Fault, Compensation and Termination |
| ff. | fortfolgende |
| GEF | Graphical Editing Framework |
| GEMS | Generic Eclipse Modeling System |
| GIF | Graphics Interchange Format |
| GMF | Graphical Modeling Framework |
| GUI | Graphical User Interface |
| IDE | Integrated Development Environment |
| IDL | Interface Description Language |
| IEEE | Institute of Electrical and Electronics Engineers |
| IETF | Internet Engineering Task Force |
| ISO | International Organization for Standardization |
| JAR | Java Archive |
| JDK | Java Development Kit |
| JET | Java Emitter Template |
| JPEG | Joint Photografic Expert Group |
| JRE | Java Runtime Environment |
| JSF | Java Server Faces |
| JSON | JavaScript Object Notation |
| JVM | Java Virtual Machine |
| J2SE | Java 2 Standard Edition |
| J2EE | Java 2 Enterprise Edition |
| MDA | Model Driven Architecture |
| MDD | Model Driven Development |
| MOF | Meta-Object Facility |
| MSI | Microsoft Windows Installer Package |
| MVC | Model-View-Controler |
| NCName | Non-Colonized Name |
| OASIS | Organization for the Advancement of Structured Information Standards |
| ODE | Orchestration Director Engine |
| OMG | Object Management Group |
| OSGi | Open Service Gateway initiative |
| PDE | Plugin Development Environment |
| PDF | Portable Document Format |
| PHP | Hypertext Preprocessor |
| Qname | Qualified Name |
| RCP | Rich-Client Platform |
| RDF | Resource Description Framework |
| REST | Representational State Transfer |
| RFC | Request for Comments |
| RFP | Request for Proposal |
| S. | Seite |
| SDK | Software Development Kit |
| SOA | Service Oriented Architecture |

| | |
|---|---|
| STP | SOA Tools Platform |
| SVG | Scalable Vector Graphics |
| SVN | Subversion |
| SWT | Standard Widget Toolkit |
| | |
| TBD | To Be Defined |
| TC | Technical Committee |
| | |
| UDDI | Universal Description, Discovery and Integration |
| UML | Unified Modeling Language |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locator |
| URN | Uniform Resource Name |
| | |
| vgl. | vergleiche |
| | |
| WCF | Windows Communication Foundation |
| WfMC | Workflow Management Coalition |
| WfMS | Workflow Management System |
| WPDL | Workflow Process Definition Language |
| WS | Web Service |
| WSDL | Web Service Description Language |
| WSFL | Web Service Flow Language |
| WS-BPEL | Web Services Business Process Execution Language |
| WTP | Web Tools Platform |
| W3C | World Wide Web Consortium |
| | |
| XML | Extensible Markup Language |
| XMI | XML Metadata Interchange |
| XPath | XML Path Language |
| XPDL | XML Process Definition Language |
| XSD | XML Schema Definition |
| XSI | XML Schema Instance |
| XSL | Extensible Stylesheet Language |
| XSLT | Extensible Stylesheet Language Transformations |

## Vollständiges Inhaltsverzeichnis

**Erklärung**

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

Stuttgart, den 15.04.2008

_____

(David Schumm)