TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

Diplomarbeit

Modellgetriebene Spezifikation von BPMN und Transformation von BPMN zu BPEL mit openArchitectureWare

Ausgeführt am

Institut für Informationssysteme
Arbeitsgruppe für Verteilte Systeme
der Technischen Universität Wien

unter der Anleitung von
Univ.Ass. Dr.rer.nat. Uwe Zdun

durch

Matthias Knoll

Schwarzenbergstrasse 8/5b
A-1010 Wien
los@chello.at

Wien, am 4. Juni 2008 _____

Die *Business Process Execution Language* (BPEL), als blockbasierte Sprache, hat sich in den letzten Jahren zu einem der Standards zur Spezifizierung von Prozessen etabliert. Die graphbasierte *Business Process Modelling Notation* (BPMN) wurde für den graphischen Entwurf von Prozessen entwickelt. Im Zuge eines einheitlichen Entwicklungsprozesses ist die automatische Transformation von BPMN Modellen zu BPEL Code wünschenswert. Diese Transformation ist jedoch Aufgrund fundamentaler Unterschiede der beiden Modellierungssprachen kompliziert, da BPMN und BPEL zwei unterschiedliche Ansätze zur Beschreibung des Kontrollflusses verwenden. Im Zuge dieser Diplomarbeit sollen diese Unterschiede erläutert und unterschiedliche Methoden erarbeitet werden, die, mit vertretbaren Einschränkungen, bestimmte Klassen von BPMN Modellen transformieren können. Diese sollen kombiniert und ein Ansatz zur Transformation einer großen Bandbreite von Modellen präsentiert werden.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Problemstellung . . . 3
- 1.2 Aufbau der Arbeit . . . 4
- 1.3 Notationskonventionen . . . 4

2 Grundlagen **5**
- 2.1 XML . . . 5
- 2.2 Web Service . . . 6
  - 2.2.1 Web Services Description Language . . . 6
- 2.3 SOAP . . . 6
- 2.4 BPEL . . . 6
  - 2.4.1 Web Service Komposition . . . 6
  - 2.4.2 Ziele von BPEL . . . 8
  - 2.4.3 Aufbau eines BPEL Prozesses . . . 10
  - 2.4.4 Control link flow . . . 12
- 2.5 BPMN . . . 13
  - 2.5.1 Ziel von BPMN . . . 13
  - 2.5.2 Arten von Diagrammen . . . 13
  - 2.5.3 BPD Core Elements . . . 14
  - 2.5.4 Probleme der BPMN Spezifikation . . . 17
- 2.6 Vergleich BPMN und BPEL . . . 19
  - 2.6.1 Control-flow Patterns . . . 19
  - 2.6.2 Process Representation Paradigm Mismatch . . . 21
- 2.7 openArchitectureWare . . . 21
  - 2.7.1 Metamodell . . . 22
  - 2.7.2 Workflow Engine . . . 22
  - 2.7.3 Das Expression Framework . . . 23
  - 2.7.4 Erweiterungen des oAW Frameworks . . . 25
  - 2.7.5 Zusammenfassung der Stärken und Schwächen . . . 26

3 Transformation von BPMN nach BPEL **27**
- 3.1 Ziele der Transformation . . . 27
- 3.2 Der Transformationsansatz . . . 27
  - 3.2.1 Grundgedanke . . . 27
  - 3.2.2 Beschränkungen des Modells . . . 29
  - 3.2.3 Teilgraph . . . 30

3.2.4 Kategorien von CAB Typen . . . 32
3.2.5 Wohlstrukturierte CABs . . . 32
3.2.6 Nichtwohlstrukturierte CABs . . . 42
3.3 Prozess überarbeiten . . . 52
3.3.1 Elemente aufspalten . . . 53
3.3.2 Gateways zusammenfassen . . . 54
3.3.3 Gateways einfügen und ausbalancieren . . . 55
3.3.4 Leere Pfade verhindern . . . 56
3.3.5 Prozesse mit multiplen Startelementen überarbeiten . . . 57
3.4 Der gesamte Ablauf . . . 58

4 Implementierung in openArchitectureWare 59
4.1 Modellgetriebener Transformationsansatz . . . 60
4.2 Aufbau der Projekte . . . 62
4.3 DSL für BPMN . . . 64
4.3.1 Metamodell . . . 64
4.3.2 Syntax und Beispiele . . . 67
4.4 Der Transformations-Workflow . . . 69
4.5 Überprüfung des Eingabemodells . . . 73
4.6 Transformation des Modells . . . 75
4.7 Generierung von BPEL . . . 78
4.7.1 Allgemeiner Ablauf . . . 78
4.7.2 Ablauf bei synchronisierenden CAB Typen . . . 81
4.7.3 Ablauf bei nicht synchronisierenden CAB Typen . . . 81
4.7.4 Generierung von sonstigen Elementen . . . 83

5 Beispiele für Transformationen 84
5.1 Wohlstrukturierte Prozesse . . . 85
5.1.1 Prozess 1 - Prozess mit Exception . . . 85
5.1.2 Prozess 2 - Prozess mit multiplen Start- und Endelementen . . . 93
5.2 Nichtwohlstrukturierte Prozesse . . . 100
5.2.1 Prozess 3 - Prozess ohne Schleifen . . . 100
5.2.2 Prozess 4 - Prozess mit Schleifen . . . 106
5.3 Testfälle . . . 114
5.3.1 Wohlstrukturierte Prozesse . . . 114
5.3.2 Nichtwohlstrukturierte Prozesse . . . 118

6 Verwandte Arbeiten 119
6.1 Relevante verwandte Arbeiten . . . 119
6.2 Vergleich mit verwandten Arbeiten . . . 122

7 Conclusio 124
7.1 Resümee . . . 124
7.2 Ausblick . . . 125

A CD **126**
A.1 Inhalt der CD 126
A.2 Installation der oAW Umgebung 126
A.3 Ausführen des Transformationsprozesses 126
A.3.1 Batch-Transformation 127
A.3.2 Einzel-Transformation 127
A.3.3 VM Optionen 127

Verzeichnisse **130**
Literaturverzeichnis 130
Glossar 134
Index 135

Abbildungsverzeichnis

1.1 Process Management Lifecycle 2

2.1 Ausschnitt aus dem Web Service Protokoll Stack 5
2.2 Web Service Composition Models 7
2.3 Web Service Orchestration und Web Service Cheography 8
2.4 BPEL - Ein BPEL Prozess als Web Service 9
2.5 BPEL - Ein BPEL Prozess und die Beziehung von BPEL zu WSDL . . . 11
2.6 BPEL - Beispiel für einen `flow` mit einem `link` 12
2.7 BPMN - Ein privater Prozess 13
2.8 BPMN - Ein abstrakter Prozess 14
2.9 BPMN - Start-, Intermediate- und End-Events 15
2.10 BPMN - Task, collapsed Subprocess und expanded Subprocess 15
2.11 BPMN - Unterschiedliche Gateway Typen 16
2.12 BPMN - Unterschiedliche Typen von Sequence Flow 16
2.13 BPMN - Association Flow 17
2.14 BPMN - Venn Diagramm der meistverwendeten BPMN Elemente 18
2.15 Unterschiedliche Darstellung von Kontrollfluss in BPMN und BPEL . . . 21
2.16 oAW - Ein typischer Transformationsprozess 23

3.1 Der Transformationsansatz 28
3.2 Vom Metamodell unterstützte Prozesse 30
3.3 Kategorisierung unterschiedlicher CAB Typen 32
3.4 Zwei wohlstrukturierte CABs 32
3.5 Transformation eines Activity-Activity Flows 33
3.6 Transformation eines Gateway-Gateway Flows 34
3.7 Transformation eines Activity-Gateway Flows 34
3.8 Transformation eines Activity-Activity Ors 35
3.9 Transformation eines Gateway-Gateway Ors 35
3.10 Transformation eines Gateway-Gateway Ors mit einem *default SequenceFlow* 36
3.11 Transformation eines simplen while 36
3.12 Überarbeitung eines upwards while 37
3.13 Transformation eines upwards while 37
3.14 Überarbeitung eines xor while 38
3.15 Transformation eines data-based Xor 39
3.16 Transformation eines event-based Xor 40
3.17 Transformation eines Exception Intermediate Events 41
3.18 Transformation eines Exception Intermediate Events 2 41

3.19 Control Link Transformation - Beispielprozess vor der Transformation . . 43
3.20 Control Link Transformation - Erster Schritt 43
3.21 Control Link Transformation - Überarbeiten der Vorbedingung 43
3.22 Control Link Transformation - Zweiter Schritt 44
3.23 Control Link Transformation - Vorbedingunge des Beispielprozesses 44
3.24 Control Link Transformation - schematische Darstellung 45
3.25 eventHandler Transformation - Struktur eines generierten BPEL Prozesses 47
3.26 eventHandler Transformation - Transformation einer *Task* 48
3.27 eventHandler Transformation - Transformation eines split *ParallelGateway* 48
3.28 eventHandler Transformation - Transformation eines split *DataXor* 49
3.29 eventHandler Transformation - Transformation eines join *ParallelGateway* 49
3.30 eventHandler Transformation - Transformation eines merge *DataXor* . . . 49
3.31 eventHandler Transformation - Beispielprozess 50
3.32 eventHandler Transformation - Vorbedingungen des Beispielprozesses . . . 50
3.33 Prozess überarbeiten - Beispiel 52
3.34 Prozess überarbeiten - Elemente Aufspalten 53
3.35 Prozess überarbeiten - Beispiel für Elemente Aufspalten 53
3.36 Prozess überarbeiten - Gateways zusammenfassen 54
3.37 Prozess überarbeiten - Beispiel für Gateways zusammenfassen 55
3.38 Prozess überarbeiten - Gateways einfügen 56
3.39 Prozess überarbeiten - leere Pfade 56
3.40 Prozess überarbeiten - Start *EventXor* einfügen 57

4.1 Modell-zu-Text Transformation. 60
4.2 Modell-zu-Modell Transformation 61
4.3 Verzeichnisbäume der vier Projekte 63
4.4 BPMN Metamodell - Grund-Elemente 64
4.5 BPMN Metamodell - Gateway Typen 65
4.6 BPMN Metamodell - Task Typen 66
4.7 Ein Prozess in BPMN DSL sowie in graphischer Darstellung 67
4.8 Ablauf der oAW Workflow Komponenten 69
4.9 Die Ablauf der Xtend Transformationsfunktion. 75
4.10 Struktur der Xtend Dateien. 76
4.11 Generation von BPEL im synchronisierenden Fall 82
4.12 Generation von BPEL im nicht synchronisierenden Fall 82

5.1 Prozess 1 - BPMN und DSL 85
5.2 Prozess 1 - Die Schritte der Transformation 87
5.3 Prozess 1 - Vorbereitung der Kapselung von `Service 3` 88
5.4 Prozess 1 - graphische BPEL Darstellungdes Prozesses 92
5.5 Prozess 2 - BPMN 93
5.6 Prozess 2 - Überarbeitung vor der Transformation 95
5.7 Prozess 2 - Die Schritte der Transformation 96
5.8 Prozess 2 - graphische BPEL Darstellung des Prozesses 99

5.9 Prozess 3 - BPMN und DSL 100
5.10 Prozess 3 - Vorbedingungen der Elemente 101
5.11 Prozess 3 - Die Schritte der Transformation 102
5.12 Prozess 3 - graphische BPEL Darstellung des Prozesses 105
5.13 Prozess 4 - BPMN und DSL 106
5.14 Prozess 4 - Der Transformationsschritt 107
5.15 Prozess 4 - Vorbedingung der Elemente 107
5.16 Prozess 4 - graphische BPEL Darstellung des Prozesses 113
5.17 Wohlstrukturierte Testprozesse (a)-(f) 115
5.18 Wohlstrukturierte Testprozesse (g)-(l) 116
5.19 Wohlstrukturierte Testprozesse (m)-(r) 117
5.20 Nichtwohlstrukturierte Testprozesse (a)-(d) 118

6.1 Verwandte Arbeiten - Beispiel eines WF-Net 120
6.2 Verwandte Arbeiten - Metamodell 1 121
6.3 Verwandte Arbeiten - Metamodell 2 122

A.1 Inhalt des Package `tuwien.bpmn2bpel.test` 127
A.2 Starten der Transformation 128
A.3 Java VM Parameter für die beiden Transformationen 129

Listingsverzeichnis

2.1 Eine Xtend Workflow Komponente . . . 23
2.2 Eine Xtend Extension . . . 24
2.3 Ein Xpand Template . . . 24
2.4 Eine Check Regel . . . 25
3.1 Control Link Transformation - Der generierte BPEL Prozess . . . 46
3.2 eventHandler Transformation - Der generierte BPEL Prozess . . . 51
4.1 Ein `receive` Element. . . . 68
4.2 Transformations-Workflow: Einlesen des Modells . . . 69
4.3 Transformations-Workflow: Multiple Startelemente überarbeiten . . . 70
4.4 Transformations-Workflow: Modell überprüfen . . . 70
4.5 Transformations-Workflow: Modell überarbeiten . . . 70
4.6 Transformations-Workflow: Modell speichern . . . 71
4.7 Transformations-Workflow: Modell transformieren . . . 71
4.8 Transformations-Workflow: Modell vergleichen . . . 72
4.9 Transformations-Workflow: BPEL erzeugen . . . 72
4.10 Beispiele für elementare Check Regeln . . . 73
4.11 Beispiel für eine BPMN spezifische Check Regel . . . 74
4.12 Beispiel für eine bpmnPattern Check Regel . . . 74
4.13 Ausschnitt aus der Xtend Datei transform.ext. . . . 76
4.14 Drei `generate` Templates für unteschiedliche Elemente . . . 79
4.15 Das AROUND `generate` Template für *FlowObject* Elemente. . . . 79
4.16 Das `generate` Template für einen CAB vom Typen *Sequence* . . . 80
5.1 Prozess 1 - DSL des Prozesses vor der Transformation . . . 85
5.2 Prozess 1 - Definition einer Variablen in der BPMN DSL . . . 88
5.3 Prozess 1 - Definition eines `assign` in der BPMN DSL 1 . . . 88
5.4 Prozess 1 - Definition eines `assign` in der BPMN DSL 2 . . . 88
5.5 Prozess 1 - Der transformierte Prozess in BPMN DSL . . . 89
5.6 Prozess 1 - Quelltext des erzeugten BPEL Prozesses . . . 90
5.7 Prozess 2 - Der Prozess in BPMN DSL . . . 94
5.8 Prozess 2 - Quelltext des erzeugten BPEL Prozesses . . . 97
5.9 Prozess 2 - Die erzeugten WSDL und `partnerLinkType` Definitionen . . . 98
5.10 Prozess 3 - DSL des Prozesses vor der Transformation . . . 100
5.11 Prozess 3 - Quelltext des erzeugten BPEL Prozesses . . . 102
5.12 Prozess 4 - DSL des Prozesses vor der Transformation . . . 106
5.13 Prozess 4 - Vereinfachter BPEL Prozess . . . 108
5.14 Prozess 4 - Quelltext des erzeugten BPEL Prozesses . . . 109

1 Einleitung

> The problem companies face is how to operate in a state of perpetual change and adaption. They are no longer wondering how to make a change; they are wondering how to repeat change, over and over again.
>
> *(Howard Smith[35])*

Business Process Management
Unter Management versteht man die Verwaltung von Ressourcen und die Choreographie der operativen Aktivitäten eines Unternehmens. Management folgt einem Kreislauf von Planung, Organisation, Personalbesetzung, Überwachung und Controlling. Business Process Management (BPM) ist die Anwendung dieses Managementkreislaufes auf die Geschäftsprozesse eines Unternehmens[25].

BPM ist ein systematischer, strukturierter Ansatz zur kontinuierlichen Überwachung, Analyse und Steuerung fundamentaler Elemente eines Unternehmens. BPM stellt nicht nur die Werkzeuge und die Infrastruktur zur Verfügung die es ermöglichen Prozessmodelle zu definieren, simulieren und analysieren, sondern auch die Möglichkeit Geschäftsprozesse so zu implementieren, daß die erhaltenen Artefakte aus der Sicht von Geschäftsprozessen gesteuert werden können[46].

Business Process Management Lifecycle
Die Entwicklung und der Einsatz von Prozessen innerhalb einer Organisation erfolgt innerhalb eines Phasenmodells. Dieser BPM *life-cycle* setzt sich aus vier Phasen zusammen:

Process Design In dieser Phase wird nach einer einleitenden Analyse ein existierender oder erwünschter Prozess in einer höheren Modellierungssprache beschrieben.

Process Implementation Die im *Process Design* Schritt entwickelten Modelle werden in eine ausführbare Workflow Spezifikation transformiert. Die Vision ist es, daß die Realisierung dieses Schrittes, der einen zeitaufwendigen und komplexen Entwicklungsprozess benötigt, mehr und mehr durch einen Konfigurations- und Kombinierungs-Prozess ersetzt wird, die Modelle aus der *Process Design* Phase also nur noch konfiguriert und kombiniert werden müssen.

Process Enactment In dieser Phase kommt das in der *Process Implementation* Phase entworfene System zum Einsatz. *Process Monitoring* überwacht die Operation zur

Laufzeit und ermöglicht die Korrektur von Fehlern während der *Process Enactment* Phase.

Process Evaluation Aufzeichnungen und Metriken von erfolgreich abgeschlossenen Prozessinstanzen bilden die Basis für eine Analyse in der Evaluation Phase. Die daraus gewonnen Information werden wiederum verwendet um die Prozessbeschreibung zu überarbeiten und eine neue Iteration des Zyklus zu beginnen.

Analog zu den Software-Entwicklungsmodellen überlappen sich die vier Phasen (Wasserfallmodell) und der Prozess als ganzes ist iterativ (Spiralmodell). Abbildung 1.1 zeigt ein graphische Darstellung eines typischen BPM Phasenmodell[48].

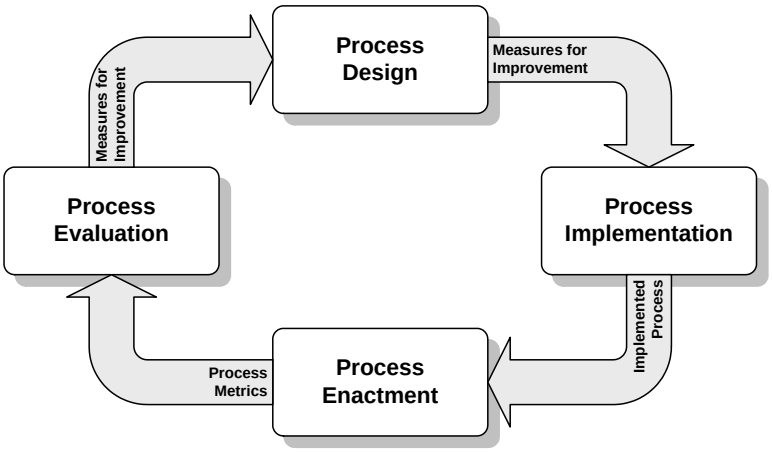


Abbildung 1.1: Ausschnitt aus dem Process Management Lifecycle[48]

Bruch zwischen den Phasen

Der Ablauf der einzelnen Phasen ist bisher jedoch nur in der Theorie dermaßen nahtlos. In der Praxis ist dieser Kreislauf an einigen Stellen unterbrochen[50]. Der für diese Arbeit relevante Bruch besteht zwischen der *Process Design* und der *Process Implementation* Phase. Oft wird der Prozess in einem ersten Schritt in einer höheren Modellierungssprache wie *Event-driven Process Chains* (EDPC), IDEF, BPMN oder UML Aktivitätsdiagrammen beschrieben. Die meisten Workflow Plattformen verwenden jedoch proprietäre Prozessbeschreibungen oder auf XML basierende Spezifikationen wie XPDL oder BPEL (siehe Abschnitt 2.4). Die Transformation zwischen Modellen unterschiedlicher Sprachen ist aber oft problematisch. Sie führt oft zu einem Verlust an Information und zu Zweideutigkeiten in den Modellen. Gleichzeitig können im Zuge der Transformation die Informationen, warum ein bestimmtes Design gewählt wurde, verlorengehen, sofern diese nicht explizit dokumentiert wurden.

Entwicklung von BPMN

Durch die Entwicklung der *Business Process Execution Language* zu einem Standard, einer XML basierten Sprache, entwickelt nur für die Beschreibung ausführbarer Prozesse, wurde dieses Problem nur offensichtlicher. Daher kam es unter Federführung der BPMI[2]

zur Entwicklung der *Business Process Modeling Notation*[43] (siehe Abschnitt 2.5). Diese Spezifikation sollte unter anderem die unterschiedlichen graphischen Notation zur Prozessmodellierung vereinen, sowie gleichzeitig die graphische Notation von BPEL werden. Diese Ziele wurden jedoch nur teilweise erreicht da durch die Rücksichtnahme auf bereits bestehende, teilweise inkompatible, graphische Notationen die Ausdrucksfähigtkeit von BPMN die von BPEL übersteigt. Gleichzeitig existiert keine formalisierte Beschreibung der Elemente sowie einer Transformation derselben nach BPEL. Beide Punkte sollen im Zuge von BPMN Version 2.0 nachgereicht werden. Bis zum Zeitpunkt dieser Arbeit lag diese jedoch noch nicht vor. Mehr zu BPMN in Abschnitt 2.5.

1.1 Problemstellung

Durch die grundsätzlichen Unterschiede zwischen BPMN und BPEL wurden sehr bald erste Transformationsalgorithmen, hauptsächlich von BPMN nach BPEL, entwickelt. Die meisten dieser Ansätze beschränken das Ausgangsmodell stark um eine gültige Transformation jedes beliebigen Modells zu gewährleisten. Die von diesen Algorithmen verwendeten Modelle haben in den meisten Fällen mit einem BPMN Prozess nur noch geringe Ähnlichkeit. So basieren sie in vielen Fällen auf Petri Netzen und bestehen nur noch aus drei Arten von Elementen: Gateways, Aktivitäten und Kanten.

Ziel dieser Diplomarbeit ist es also, die Transformation eines Prozessmodells, das auf einem BPMN kompatiblen Metamodell basiert, nach BPEL zu realisieren. Es soll also eine automatische Transformation von einem gültigen BPMN Prozess in einen ausführbaren BPEL Prozess stattfinden. Dabei werden die in der Literatur bereits beschriebenen Ansätze auf ihre Tauglichkeit untersucht und adaptiert werden, um eine möglichst große Klasse von BPMN Modellen und Elementen zu unterstützen. Gleichzeitig soll die Anzahl der Restriktionen denen das BPMN Metamodell unterliegt so klein wie möglich gehalten werden.

Zur Transformation soll ein Generator zum Einsatz kommen, der aus einer textuellen Spezifikation eines Prozesses einen ausführbaren BPEL-Prozess erzeugt. Das, dem Ausgangsmodell der Transformation zugrundeliegende Metamodell, soll dabei in der Lage sein die Semantik der *Business Process Modeling Notation* ebenso abbilden zu können wie deren Einschränkungen. Da die BPMN Spezifikation kein formalisiertes Metamodell für einen Datenaustausch zwischen BPMN unterstützenden Programmen enthält, wird dieses im Zuge der Arbeit formuliert werden. Mithilfe dieses Metamodells soll es möglich sein, realistische Prozesse zu beschreiben und diese ‚sofern sie gültig sind, in BPEL Prozesse zu transformieren. Daher ist ein weiteres Ziel eine möglichst große Bandbreite von BPMN Elementen und damit gültigen BPMN Modellen zu unterstützen.

Ein weiterer, nicht minder wichtiger Punkt ist es, daß ein Transformationsansatz gefunden wird, der einen möglichst einfachen und lesbaren BPEL Prozess erzeugt, damit die erzeugten BPEL Prozesse nach der Transformation noch manuell bearbeitet werden können.

1.2 Aufbau der Arbeit

Der weitere Aufbau der Arbeit gliedert sich wie folgt:

Kapitel 2 wird die zugrundeliegenden Technologien und Spezifikationen vorstellen, wie unter anderem BPEL und BPMN.

Kapitel 3 wird den Transformationsprozess behandeln.

Kapitel 4 wird die Implementierung des Ansatzes darstellen.

Kapitel 5 wird die Transformation anhand einiger ausgewählter Prozesse demonstrieren.

Kapitel 6 wird unterschiedliche Transformationsansätze sowohl für die Transformation von BPMN zu BPEL als auch für ähnliche Sprachen erläutern.

Kapitel 7 präsentiert die Conclusio.

1.3 Notationskonventionen

Englische Fachbegriffe werden mit jeweils deutscher Übersetzung in Klammern eingeführt. Sofern englisches Original und Übersetzung entweder sehr ähnlich sind (wie im Fall von „Prozess“ oder „process“) oder die Übersetzung allgemein verwendet wird, werden beide Varianten synonym verwendet. Bei der Beschreibung von Elementen der verschiedenen Spezifikationen und wenn die englischen Bezeichnungen im Zuge der Implementierungen verwendet werden, wird die englische Version bevorzugt.

Fachbegriffe aus Frameworks und die Namen von Elementen der Spezifikationen werden, wenn Sie in der englischer Version verwendet werden oder es der Unterscheidbarkeit dient, kursiv gesetzt. Variablen und Methodennamen werden dagegen in nicht proportionaler Schrift dargestellt. Im Anhang ist ein Index der wichtigsten englischen und deutschen Begriffe aufgeführt.

2 Grundlagen

In den folgenden Abschnitten wird kurz auf die für diese Arbeit relevanten Spezifikationen eingangen. So baut sowohl BPMN als auch die BPEL Spezifikation auf einer Reihe von bestehenden Standards auf. Die wichtigsten Spezifikationen des sogenannten *Web Service Protocoll Stack* in Bezug auf diese Arbeit werden in den Abschnitten 2.1-2.3 kurz vorgestellt. Abschnitt 2.4 stellt BPEL vor und der anschliessende Abschnitt 2.5 BPMN. Abschnitt 2.6 vergleicht die beiden Spezifikationen. Der letzte Abschnitt 2.7 dieses Kapitel beschreibt das für die Implementierung der Transformation verwendete openArchitectureWare Framework vor.

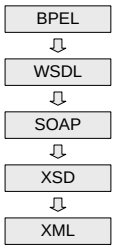


Abbildung 2.1: Ausschnitt aus dem Web Service Protokoll Stack

Leser mit Erfahrung in den entsprechenden Gebieten und Spezifikationen können die entsprechenden Abschnitte überspringen und gleich mit Kapitel 3 fortfahren.

2.1 XML

Die *Extensible Markup Language* (XML) dient zum Austausch strukturierter Daten zwischen Systemen. Ein korrektes XML Dokument muss wohlgeformt (syntaktisch korrekt) sowie gültig (semantisch korrekt) sein sofern entsprechende *Document Type Definition* (DTD) oder Schema Definitionen vorhanden sind.

XML-Schema

Ein XML Schema ist eine Beschreibung eines XML Dokumenten Typen die die Struktur und den Inhalt spezifiziert. Eine relativ einfache Schemasprache, die Document Type Definition, ist Bestandteil der XML Spezifikation. Eine weitere umfangreichere Sprache wäre die vom W3C spezifizierte XML Schema.

2.2 Web Service

Eine Menge von Technologien und Produkten kann als Web Service bezeichnet werden. Im Zuge dieser Arbeit wird unter einem Web Service die Definition der w3.org [17], die auch für BPEL relevant ist, verstanden. Diese definiert ein Web Service als ein Softwaresystem, das mit dem Ziel entwickelt wurde, plattformunabhängige Kommunikation zwischen unterschiedlichen Systemen in einem Netzwerk zu ermöglichen. Dabei sind die Schnittstellen des Systems in einer von Maschinen verarbeitbaren Form gespeichert. Die Systeme interagieren über die, in den jeweiligen Schnittstellenbeschreibungen definierten, Nachrichten.

Es handelt sich um selbständige und gekapselte Software Komponenten, deren Schnittstellen mit WSDL beschrieben sind und auf deren Funktionen mit standardisierten Protokollen zugegriffen werden kann.

2.2.1 Web Services Description Language

Die *Web Services Description Language* (WSDL) [14] ist eine auf XML basierende Sprache zur Beschreibung der Interfaces von Webservices. Durch die Verwendung von XML ist die Beschreibung des Services unabhängig von den zur Implementierung verwendeten Technologien. Mit ihr werden Datentypen, Operationen mit ihren Rückgabewerten, sowie Informationen zum Zugriff auf das Service sowie zum Deployment beschrieben.

2.3 SOAP

Ursprünglich war SOAP [16] die Abkürzung für *Simple Object Access Protocol*, inzwischen ist es zu einem eigentlichen Namen geworden. Es handelt sich um ein Protokoll zum Austausch von XML Nachrichten in einem Netzwerk. Im wesentlichen legt es die Syntax der Nachrichten fest. Die Wahl des verwendeten Transportprotokolls bleibt dem Entwickler überlassen, es wird jedoch hauptsächlich HTTP verwendet.

2.4 BPEL

Die *Business Process Execution Language* [10][8] ist eine XML Sprache zur Beschreibung von Geschäftsprozessen die auf Web Services aufbauen.

2.4.1 Web Service Komposition

Web Services mit ihrem atomaren synchronen oder asynchronen Aufrufen unterstützen die Interaktion von Geschäftspartnern und deren Prozessen. In der WSDL Spezifikation sind genau vier unterschiedliche Interaktionsmodelle für einen Web Service Endpunkt definiert. Der Endpunkt erhält eine Nachricht (*one-way*), er sendet eine Nachricht (*notification*), er erhält eine Nachricht und sendet daraufhin eine korrelierte Antwort (*request-respone*) und

als letztes Interaktionsmodell sendet der Endpunkt eine Nachricht und erhält daraufhin eine korrelierte Antwort (*solicit-respones*).

Für Geschäftsprozesse sind diese Modelle jedoch nur eingeschränkt geeignet, da diese typischerweise aus einer Serie von unterschiedlichen Operationen zwischen einer Vielzahl von Teilnehmern innerhalb einer längeren zustandsbehafteten Transaktion bestehen. Daher bestand der Bedarf nach einer formalen Spezifikation der Interaktionen von komplexen Prozessen.

Ein wichtiges Merkmal eines Web Service ist aber die Möglichkeit der Wiederverwendung für die Komposition von komplexen Anwendungen. Oft werden neue Prozesse aus einzelnen Teilaufgaben zusammengesetzt, die bereits als Web Service implementiert sind. Für diese Komposition von komplexen aus einfachen Services wurden verschiedene Spezifikationen vorgeschlagen. Sie alle legen die Reihenfolge in der Services aufgerufen werden und auch eventuelle Bedingungen, die dafür erfüllt sein müssen, fest. Die zugrundeliegenden Ansätze dieser Spezifikationen lassen sich, wie in wie in Abbildung 2.4.1 skizziert, in zwei Klassen aufteilen, in statische und dynamische Kompositionsmodelle.[13]

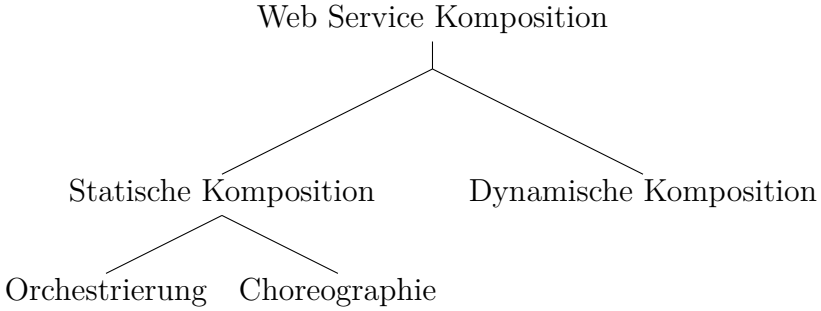


Abbildung 2.2: Web Service Composition Models[30]

Statische Komposition

Innerhalb der Klasse der statischen Komposition wird zwischen zwei Ansätzen unterschieden. Der erste, *Web Services Orchestration*, integriert vorhandene Services indem er einen zentralen Koordinator (engl. orchestrator) einführt. Dieser ist für die Aufrufe und Integration der einzelnen Services zuständig. Spezifikationen die diesen Ansatz verfolgen wären BPML und auch BPEL. Der zweite Ansatz, *Web Service Choreography*, verwendet keine zentrale Koordinierungsstelle, sondern definiert komplexe Interaktionen indem er den möglichen Nachrichtenverkehr zwischen allen Teilnehmern festlegt. Dieser Ansatz wird zum Beispiel von WS-CDL verfolgt. Abbildung 2.3 zeigt die beiden Ansätze.

Dynamische Komposition

Im Unterschied zur statischen Komposition, bei der der Informationsfluss und die Bindings der Services a priori bekannt sind, ist das Ziel dynamischer Ansätze aus vorhandenen Services, die aber nicht die gewünschte Funktionalität bieten, durch deren Neukombination die Anfrage zu erfüllen. Dynamische Ansätze benötigen daher Methoden zur Erkennung

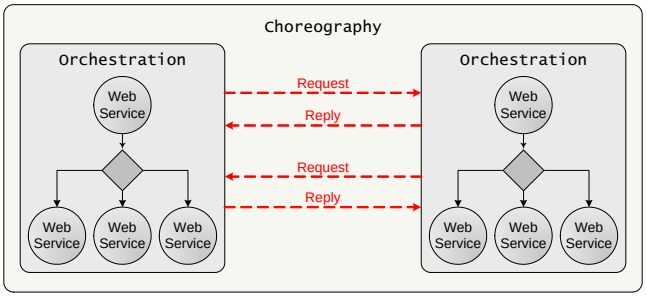


Abbildung 2.3: Web Service Orchestration und Web Service Cheography

der benötigten Funktionalitäten, zur deren Lokalisierung, zur Bewertung der jeweiligen Funktionalitäten und zur Kombination der ausgewählten Services[33] [9]. Im Feld der dynamischen Web Service Komposition existieren eine Vielzahl von unterschiedlichen Methoden und Ansätzen. In [7] werden diese in zumindestens sechs unterschiedlichen Kategorien klassifiziert. Die Anzahl begründet sich durch die unterschiedlichen Arten von Problemen, die durch die dynamische Komposition gelöst werden sollen. So können grob drei Klassen von Problemen identifiziert werden [23]:

Erfüllung von Vorbedingungen Es existiert ein Service welches die Anforderungen erfüllen würde, aber nicht alle Vorbedingungen des Services sind erfüllt.

Aufteilen von Anfragen Mehrere korrelierte Anfragen können nur von vielen Services gemeinsam beantwortet werden und nicht von einem Service alleine.

Wissenslücken Bei der Suche nach einem passenden Service sind nicht alle Fakten bekannt die zur korrekten Wahl notwendig wären.

Je nach Problem Typ oder Typen eignet sich ein anderer Ansatz besser.

2.4.2 Ziele von BPEL

Die folgenden zehn Ziele waren die Basis der ursprünglichen Arbeitsgruppe, die für die Entwicklung von BPEL maßgeblich war.[26]

1. **Webservices als Basis**

   BPEL beschreibt Prozesse die durch Web Service Aufrufe interagieren und selbst als Web Services veröffentlicht werden.

2. **XML zur Beschreibung**

   Geschäftsprozesse werden in BPEL in XML beschrieben. Die graphische Darstellung von Prozessen ist nicht Rahmen der Spezifikation, noch definiert sie eine bestimmte Methodik zum Entwurf von Prozessen.

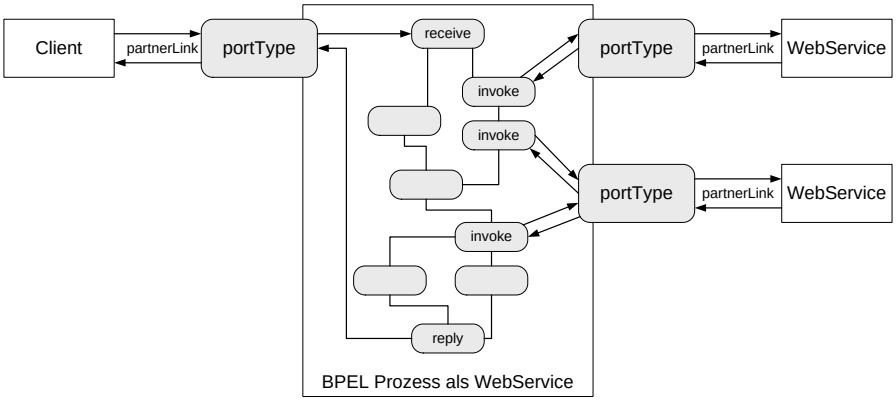


Abbildung 2.4: BPEL Prozess als Web Service

3. **Gemeinsame Konzepte für Orchestration und Choreographie**

   Es soll zwischen abstrakten und ausführbaren Prozessen unterschieden werden. Ausführbare BPEL-Prozesse können auf einem WFMS eingesetzt werden. Abstrakte Prozesse beschreiben das externe Verhalten eines Prozesses, ohne jedoch seinen internen Ablauf preiszugeben.

   Sowohl externe (abstract) als auch interne (executable) Prozesse sollen auf den gleichen Konzepten aufbauen. Damit kann kontrolliert werden, ob externes und internes Verhalten des Prozesses übereinstimmen. Darüber hinaus ist es damit möglich aus einem ursprünglich abstraktem Prozess einen ausführbaren zu erstellen, sowie umgekehrt einen ausführbaren Prozess als abstrakten zu veröffentlichen.

4. **Kontrollfluss**

   BPEL soll sowohl hierarchische als auch graphische Konzepte enthalten. Eine gemischte Verwendung soll übergangslos möglich sein um eine Fragmentierung des Standards zu vermeiden.

5. **Datenverarbeitung**

   Es soll einfache Methoden zur Bearbeitung von Kontrollfluss- sowie Prozess-Daten geben.

6. **Identifikation**

   Es soll einen flexiblen Mechanismus geben, der eine Anzahl von unterschiedlichen vom Benutzer definierten, Identifikatoren unterstützt die sich im Prozessverlauf auch ändern können. Diese werden von den Anwendungen in den Nachrichten eingebettet und nicht als Header im Protokoll. Dadurch ist die Zuordnung nicht abhängig vom verwendeten Binding.

7. **Lifecycle**

   Die Spezifikation sollen nur die Instanzierung und Beendigung von Prozessen unterstützen. Fortgeschrittene Konzepte wie Pausieren und Fortsetzen werden noch nicht unterstützt.

8. **Transaktionsmodell**

   Die Spezifikation soll ein Transaktionsmodell verwenden, das auf bewährten Konzepten, wie Kompensation und Scopes für Teile der Prozesse, aufbaut, um Fehler abfangen zu können.

9. **Modularisierung**

   Zur Modularisierung von Prozessen sollen Web Services benutzt werden. Es gibt daher nur beschränkte Unterstützung für Modularisierung und Dekomposition innerhalb von BPEL.

10. **Composition**

    BPEL soll soweit wie möglich auf bereits vorhandenen Web Service Standards modular aufbauen und diese verwenden. Nur wenn es für eine Anforderung keinen geeigneten Standard gibt, soll eine entsprechende Spezifikation entwickelt werden.

2.4.3 Aufbau eines BPEL Prozesses

Da BPEL auf XML aufbaut, besteht eine Prozessspezifikation aus einem XML Dokument, in dem der Prozess sowie alle sonstigen notwendigen Komponenten definiert sind.

Zur Beschreibung von Prozessen wird zwischen zwei Grundsätzlichen Sprachbestandteilen unterschieden. Einerseits den `basic activities` und andererseits den `structured activities`. Etwas außerhalb dieser Kategorisierung steht die `scope` Aktivität. Variablen und `partnerLink` Elemente sind weitere wichtige Bestandteile von BPEL. Abbildung 2.5 skizziert den typischen Aufbau eines BPEL Prozesses.

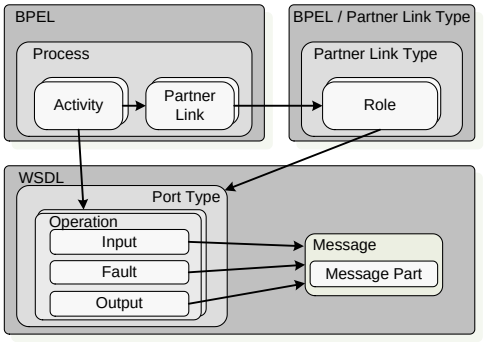
Die folgende Auflistung beschreibt die wichtigsten Aktivitäten von BPEL, für eine detailliertere Beschreibung empfiehlt sich die BPEL Spezifikation [10].

`basic activities` Dabei handelt es sich um atomare Aktivitäten eines Prozesses.

- **`assign`** Zuweisung an eine Variable des Prozesses
- **`invoke`** Aufruf eines Web Service
- **`receive / reply`** Eine Web Service Schnittstelle externen Partnern zur Verfügung stellen.
- **`throw`** Einen Fehler signalisieren.
- **`wait`** Den Prozess für eine bestimmte Zeitspanne oder bis zu einem gewissen Zeitpunkt aufhalten
- **`empty`** Eine Aktivität die nichts tut.

```
<?xml version="1.0" encoding="UTF-8"?>
<bpel:process xmlns:bpel="..." xmlns:xsd="..." name="abc"
     targetNamespace="abc" xmlns:ns1="abc">
  <!-- Import von WSDL und partnerLinkType Definitionen
       -->
  <bpel:import importType="http://schemas.xmlsoap.org/wsdl
       /" location="xy.wsdl" namespace="abc"/>
  <bpel:partnerLinks>
    <bpel:partnerLink myRole="handler" name="handlerPL"
         partnerLinkType="ns1:handlerPartnerLink"/>
  <!-- Prozessvariablen -->
  <bpel:variables>
    ...
  </bpel:variables>
  <!-- Aktivitaeten -->
  <bpel:sequence name="sequence">
    <bpel:receive name="startProzess" .../>
    ...
    <bpel:send name="endProzess" ... />
  </bpel:sequence>
</bpel:process>
```

(a) Aufbau eines BPEL Prozesses.



(b) Beziehung zwischen BPEL und WSDL

Abbildung 2.5: Beziehung zwischen BPEL und WSDL

`extensionActivity` Eine Möglichkeit für Entwickler eigene neue Aktivitäten zu definieren.

`exit` Die Prozessinstanz sofort terminieren.

`rethrow` Ermöglicht es, bereits abgefangene Fehler noch einmal auszulösen.

`structured activities` Diese Aktivitäten legen die Reihenfolge fest in der Aktivitäten abgearbeitet werden, also den Kontrollfluss des Prozesses. Sie können beliebig verschachtelt werden und ermöglichen so die Komposition komplexer Geschäftsprozesse.

`sequence` Die enthaltenen Aktivitäten werden sequentiell abgearbeitet.

`if / elseif / else` Enthalten jeweils eine sortierte Liste von Aktivitäten die mit einer Bedingung verknüpft sind.

`while / repeatUntil` Die enthaltene Aktivität wird solange ausgeführt, solange eine Bedingung erfüllt ist. Die Bedingung wird zu Beginn oder am Ende jedes Durchlaufs evaluiert.

`pick` Diese Aktivität wartet auf das Auftreten eines Events aus einer Liste von vorher definierten möglichen. Ist einer dieser Events eingetreten kann das pick nicht noch einmal ausgelöst werden.

`flow` Mit dieser Aktivität wird paralleler Kontrollfluss modelliert. Alle enthaltenen Aktivitäten werden gleichzeitig ausgeführt.

`forEach` Führt die enthaltene Aktivität eine bestimmte Anzahl von Durchläufen aus. Die Anzahl wird vor der ersten Ausführung des `pick` festgelegt und ist nicht mehr von der enthaltenen Aktivität veränderbar. Im Unterschied zu `while` oder `repatUntil` kann die enthaltene Aktivität auch gleichzeitig ausgeführt werden. Dann entspricht das `forEach` einem impliziten `flow` mit sovielen Instanzen der enthaltenen Aktivität wie Durchläufen.

scope Ein scope stellt eine Ausführungsumgebung für die enthaltene Aktivität dar.

variable Eine Variable spezifiziert die Daten eines Nachrichtenaustausches. Wenn ein Prozess eine Nachricht empfängt, wird die entsprechende Variable mit dem Inhalt der Nachricht befüllt.

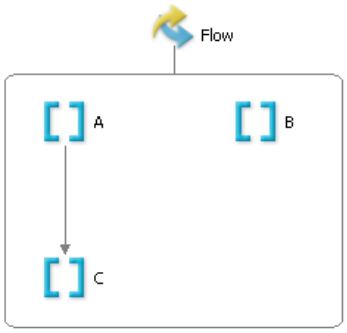
partnerLink Ein `partnerLink` kann ein beliebiges Service sein, das den Prozess aufruft oder ein Service das vom Prozess aufgerufen wird. Jeder `partnerLink` entspricht einer bestimmten Rolle im Geschäftsprozess.

2.4.4 Control link flow

Eines der Ziele von BPEL war es auch, graphische Modellierungskonzepte innerhalb einer Prozessdefinition zu ermöglichen. Innerhalb einer `flow` Aktivität ist es möglich mit Hilfe von `link` Elementen Abhängigkeiten zwischen den Aktivitäten zu modellieren. In Abbildung 2.6 ist ein einfacher `flow` mit drei `none` Aktivitäten dargestellt. So werden *A* und *B* sofort ausgeführt sobald der `flow` aktiv wird. Aktivität *C* aber erst nach *A* ausgeführt.

```
<bpel:flow>
  <bpel:links>
    <bpel:link name="L1"/>
  </bpel:links>
  <bpel:empty name="A">
    <bpel:sources>
    <bpel:source linkName="L1"/>
    </bpel:sources>
  </bpel:empty>
  <bpel:empty name="C">
    <bpel:targets>
    <bpel:target linkName="L1"/>
    </bpel:targets>
  </bpel:empty>
  <bpel:empty name="B"/>
</bpel:flow>
```

(a) BPEL Quelltext



(b) graphische Darstellung

Abbildung 2.6: Beispiel für einen `flow` mit einem `link`

Zusätzlich läßt sich noch eine `joinCondition` für Aktivitäten definieren, die Ziel mehrerer, eingehender `link` Elemente sind. Ebenso kann aber jeder `link` mit einer `transitionCondition` belegt werden. Diese ist mit einer Bedingung verknüpft und führt dazu, daß die Zielaktivität des `link` nur ausgeführt wird, wenn die Bedingung erfüllt ist.

Es lassen sich also mit Hilfe von `link`, `joinCondition` und `transitionCondition` auch Gateways innerhalb eines `flow` abbilden. Es können jedoch mit ihrer Hilfe keine Schleifen gebildet werden und ein `link` darf niemals die Grenze einer der `structured activities` für Schleifen überschreiten.

2.5 BPMN

Die *Business Process Modeling Notation* [43] ist eine graphische Notation für die Modellierung von Prozessen. Sie definiert die graphische Darstellung und die Semantik der Elemente eines *Business Process Diagram* (BPD). Jedoch ist weder ein standardisiertes Format zum Austausch von Diagrammen, noch eine formale Definition der Semantik der Elemente, Bestandteil der Spezifikation.

2.5.1 Ziel von BPMN

Das primäre Ziel der BPMN ist es eine Notation bereitzustellen, die für alle an einem Geschäftsprozess beteiligten Anwender verständlich ist. Die Notation soll vom Analysten, der den Prozess entwirft, über den implementierenden Entwickler, bis zum eigentlich Anwender, der den Prozess steuert und überwacht, klar sein. Mit ihr soll es möglich sein, den Bruch zwischen Entwurf und Implementierung des Prozesses zu überbrücken. Eine Anwendung muss drei wesentliche Punkte erfüllen um BPMN konform zu sein.

1. **Visuell Konform** - Die graphische Darstellung der Elemente ist ein wesentlicher Part der Spezifikation, daher dürfen die Elemente der Spezifikation nicht verändert werden. Ergänzungen und Erweiterungen sind zulässig sofern sie nicht in Konflikt mit bereits existierenden Elementen stehen.

2. **Semantisch Konform** - Eine konforme Implementierung muss sich an die semantischen Definitionen der einzelnen Elemente halten.

3. **Portabel** - Es muss möglich sein, Diagramme zwischen unterschiedlichen Anwendungen die BPMN unterstützen auszutauschen. Diese Anforderung wird jedoch erst gültig wenn es zur Definition eines solchen Formats gekommen ist. Bis zum Zeitpunkt dieser Arbeit existiert noch kein solches Format.

2.5.2 Arten von Diagrammen

BPMN unterscheidet zwischen drei Arten von Prozessen um den unterschiedlichen Zielgruppen und Anwendungen gerecht zu werden:

Private (interne) Prozesse Interne Prozesse eines Unternehmens die einen bestimmten Prozessablauf abbilden.

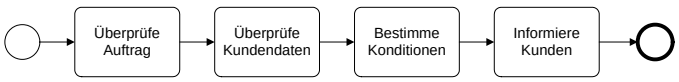


Abbildung 2.7: Ein einfacher privater Prozess

Abstrakte (öffentliche) Prozesse Abstrakte Prozesse stellen die Interaktion zwischen einem privaten Prozess und anderen Prozessen oder Teilnehmern dar. Dabei werden

nur die Aktivitäten und Kontrollflusselemente des Prozesses veröffentlicht, die für die Kommunikation mit Partnern notwendig sind. Alle anderen Aktivitäten und Elemente die nur für den internen Kontrollfluss notwendig sind, werden nicht dargestellt. Folglich stellt der abstrakte Prozess nur die Nachrichten dar, die für die externe Kommunikation mit dem Prozess notwendig sind. Ein einzelner abstrakter BPMN Prozess kann also auf einen einzelnen abstrakten BPEL Prozess abgebildet werden. Abbildung 2.8 auf Seite 14 zeigt einen abstrakten Prozess einer Bestellung.

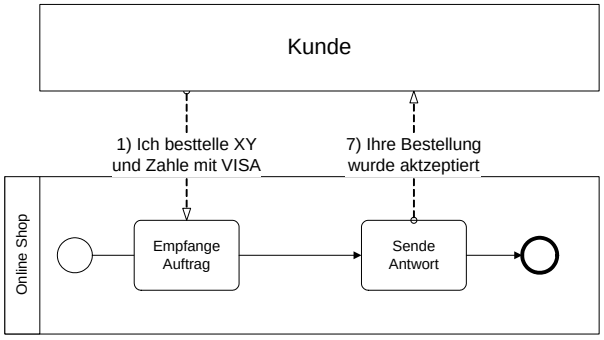


Abbildung 2.8: Ein einfacher abstrakter Prozess

Kollaborierende (globale) Prozesse Ein kollaborierender Prozess definiert die Interaktionen zwischen mehreren unterschiedlichen Unternehmen und Prozessen in einem Diagramm. Er besteht zumindestens aus zwei abstrakten Prozessen und ihren Schnittstellen.

2.5.3 BPD Core Elements

Die Entwicklung von BPMN war motiviert von dem Gedanken einen einfachen Mechanismus zur Erstellung von Modellen zu entwickeln und gleichzeitig die Komplexität von Geschäftsprozessen abbilden zu können. Um diese widersprüchlichen Anforderung handhaben zu können, wurden die graphischen Aspekte in Kategorien aufgeteilt. Durch die kleine Anzahl von Kategorien ist ein Leser eines Diagramms schnell in der Lage die grundlegenden Elemente des Diagramms zu erkennen. Zusätzliche Information kann dann je nach Element hinzugefügt werden, ohne das grundsätzliche Aussehen eines Diagramms zu verändern. Folgende Kategorien von Elementen bietet BPMN:

Flow Objects

Flow Object Elemente sind die wichtigsten graphischen Elemente um den Ablauf eines Prozesses zu modellieren.

Event Ein Event ist ein Ereignis im Prozessablauf. Es kann weiters zwischen Start, Intermediate und End Events unterschieden werden. Wie der Name bereits impliziert, stellt der Start Event den Punkt dar, an dem der Kontrollfluss seinen Ursprung nimmt. Folglich kann ein Start Event keinen eingehenden Sequence Flow besitzen.

Analog stellt der End Event einen Endpunkt des Prozesses dar und darf keinen ausgehenden Sequence Flow besitzen. Intermediate Events können zwischen den beiden im Prozess verwendet werden. Sie werden eingesetzt um zu verdeutlichen wo bestimmte Nachrichten erwartet oder gesendet werden, wo es zu zeitlichen Verzögerungen kommen kann, um den normalen Fluss durch Exception Handling zu unterbrechen und um Compensation Handling zu modellieren. Falls sie keinen eingehenden Sequence Flow besitzen, müssen sie mit einer Aktivität assoziiert sein.

Sie werden graphisch durch einen Kreis dargestellt in dessen Mitte sich je nach Eventtyp eine andere Markierung befindet. Abbildung 2.9 auf Seite 15 zeigt jeweils drei unterschiedliche Event-Typen.

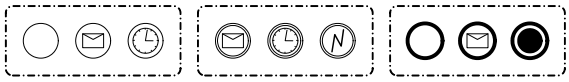


Abbildung 2.9: Start-, Intermediate- und End-Events

Activity Eine Activity ist ein Arbeitsschritt im Prozessablauf. Es wird zwischen atomaren und nicht atomaren unterschieden. Die Task Activity ist eine atomare Activity, während ein Subprocess ein Set von graphischen Objekten ist.

Sie werden durch ein abgerundetes Rechteck dargestellt wie in Abbildung 2.10.

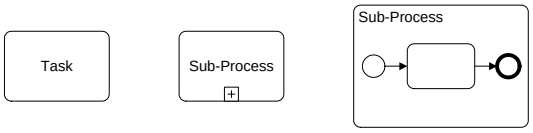


Abbildung 2.10: Task, collapsed Subprocess und expanded Subprocess

Gateway Mit einem Gateway lassen sich Aufspaltungen und Zusammenführungen des Sequence Flows kontrollieren. Ein Gateway ist eine Sammlung von Gates die jeweils mit einer Bedingung und einem Sequence Flow verbunden sind. Diese muss erfüllt sein um den zugehörigen Sequence Flow zu aktivieren. Der Typ des Gateways bestimmt wieviele dieser Gates aktiviert werden können.

Sie werden durch einen Rhombus dargestellt in dessen Mitte sich je nach Typ des Gateways unterschiedliche Markierungen befinden. Abbildung 2.11 auf Seite 16 zeigt die möglichen Typen und ihre graphische Darstellung. Sie lassen sich folgendermaßen charakterisieren:

- Exklusiv - Der Sequence Flow des in der Reihenfolge ersten aktivierten Gates wird verfolgt.
  - Data-Based - Die Bedingungen überprüfen die Werte von Variablen.
  - Event-Based - Nachrichten oder Timer führen zur Auslösung des nachfolgenden Sequence Flows.

- Inklusiv - Jeder Sequence Flow dessen Gate aktiv ist wird verfolgt.
- Parallel - Alle ausgehenden Sequence Flows werden verfolgt.
- Komplex - Jeder Sequence Flow, dessen Gate aktiv ist, wird verfolgt.

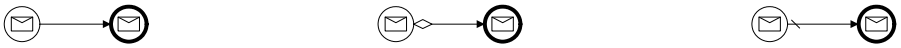
Gateways können den Fluss von sowohl zusammenführenden und/oder auseinanderführenden Sequence Flow kontrollieren. Daher kann ein Gateway sowohl mehrere eingehende wie auch ausgehende Sequence Flows gleichzeitig besitzen. Der jeweilige Typ bestimmt dann das genaue Verhalten. Detailliert wird auf die Semantik der Gateways in der BPMN Spezifikation[43, Seite 70-80] eingegangen.

Abbildung 2.11: Unterschiedliche BPMN Gateway Typen

Connecting Objects

Verbinden Flow Objects miteinander oder mit Artifacts.

Sequence Flow Ein Sequence Flow definiert die Reihenfolge in der die Aktivitäten eines Prozesses abgearbeitet werden. Graphisch wird ein Sequence Flow immer durch einen Pfeil wie in Abbildung 2.12 dargestellt. Eine Raute am Anfang eines Sequence Flow symbolisiert einen Conditional Sequence Flow, also einen Sequence Flow, der nur unter einer bestimmten Bedingung genommen wird. Ein Schräger Strich wiederum steht für einen Default Sequence Flow der genommen wird, wenn sonst kein anderer Sequence Flow des Elementes aktiviert wurde.



(a) None Sequence Flow (b) Conditional Sequence Flow (c) Default Sequence Flow

Abbildung 2.12: Die drei unterschiedlichen Typen von Sequence Flow: (a)*None*, (b)*Conditional*, (c)*Default*

So führt zum Beispiel ein normaler und ein default Sequence Flow die von derselben Aktivität ausgehen dazu, daß der Pfad, der mit dem default Sequence Flow beginnt niemals aktiv werden kann.

Message Flow Ein Message Flow stellt den Fluss von Messages zwischen zwei unterschiedlichen Teilnehmern eines Porzesses dar. Daher verbindet ein Message Flow immer zwei unterschiedliche Pools. Abbildung 2.8 auf Seite 14 zeigt ein Beispiel für den Einsatz von Message Flows.

Association Eine Association verbindet Informationen und Artifacts mit Flow Objects und Connecting Objects. Sie werden durch eine strichlierte Linie dargestellt. Es ist

auch möglich direktionale Associations zu verwenden. Damit ist es zum Beispiel möglich, Eingangs- und Ausgangsdaten zu modellieren wie in Abbildung 2.13 gezeigt.

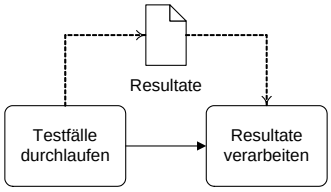


Abbildung 2.13: Directional Association Flow

Swimlanes

Werden verwendet um Elemente zu gruppieren.

Pools Ein Pool repräsentiert einen Teilnehmer an einem Prozess.

Lanes Eine Lane ist ein Teil eines Pools und wird verwendet um Aktivitäten zu kategorisieren und organisieren.

Artifacts

Können verwendet werden um zusätzliche Informationen im Prozess zu modellieren. Darunter fallen zum Beispiel Data Objects, da sie keinen direkten Einfluss auf den Sequence oder Message Flow eines Prozesses haben, aber Informationen bereitstellen über Daten die Aktivitäten benötigen um ausgeführt zu werden und / oder Daten die sie erzeugen.

2.5.4 Probleme der BPMN Spezifikation

Die BPMN Spezifikation ist aber auch mit einer Reihe von Problemen behaftet.

Fehlen einer formalen Definition Durch die rein textuelle Beschreibung und den Verzicht auf eine formale Spezifizierung der Semantik der Elemente, gibt es BPMN Elemente, deren Semantik nicht ganz eindeutig ist. So ist die Spezifikation im Zusammenhang mit Schleifen und einem Or-Merge nicht eindeutig. Laut Spezifikation synchronisiert ein Or-Merge die Sequence Flows wieder, die von einem oder mehreren Splits vor dem Merge im Prozessfluss ausgegangen sind. Ein Or Merge im Zusammenhang mit einer Schleife ist jedoch nicht beschrieben.

Austausch von Diagrammen Durch den bereits erwähnten Verzicht auf eine formale Spezifikation des Formats in dem ein BPMN Diagramm gespeichert wird, können keine Diagramme zwischen unterschiedlichen Modellierungswerkzeugen ausgetauscht werden. Verschärft wird dieses Problem dadurch, daß viele Programme nur einen Teil der Spezifikation abdecken und die komplizierteren Konzepte nicht unterstützen.

Fehlerhafte Beispiele Durch den Umfang der Spezifikation und die Anzahl der Elemente benötigt es einige Zeit sich in die Semantik und, im speziellen, in die Ausnahmen einzulesen. Ein Problem dabei stellen die teilweise fehlerhaften Beispiele von BPMN Diagrammen dar, die im Netz zu finden sind.

Freiheit beim Modellieren Die Freiheiten, die einem BPMN beim Modellieren bietet, können aber auch als Schwachpunkt gesehen werden. So ist es sehr leicht möglich, kleine Details, wie mehrere in eine Aktivität eingehende Sequence Flows, zu übersehen. Daß der Prozess an dieser Aktivität aber nicht synchronisiert wird und die nachfolgenden Aktivitäten ab dieser mehrfach ausgeführt werden können, macht gegenüber anderen Modellierungssprachen aber einen enormen Unterschied. Dieses Beispiel ist aber nur eines der harmloseren, wo eine an sich unauffällige Konstruktion im Prozess sehr weitreichende Auswirkungen hat.

Einen sehr interessanten Ansatz verfolgt die Arbeit von Muehlen [49]. In dieser werden typische BPMN Beispieldiagramme untersucht und die Verwendung der Elemente statistisch analysiert. Ein Ergebnis der Untersuchung ist, daß in einem durchschnittlichen Diagramm kaum mehr als 9 Elemente der Spezifikation genutzt und generell nur weniger als 20% der verfügbaren Elemente verwendet werden. Abbildung 2.5.4 zeigt die in der Studie ermittelten, am häufigsten verwendeten Subsets von Elementen. Die Arbeit ist jedoch nicht unumstritten was die Auswahl der verwendeten Diagramme und auch die statistischen Methoden betrifft[1].

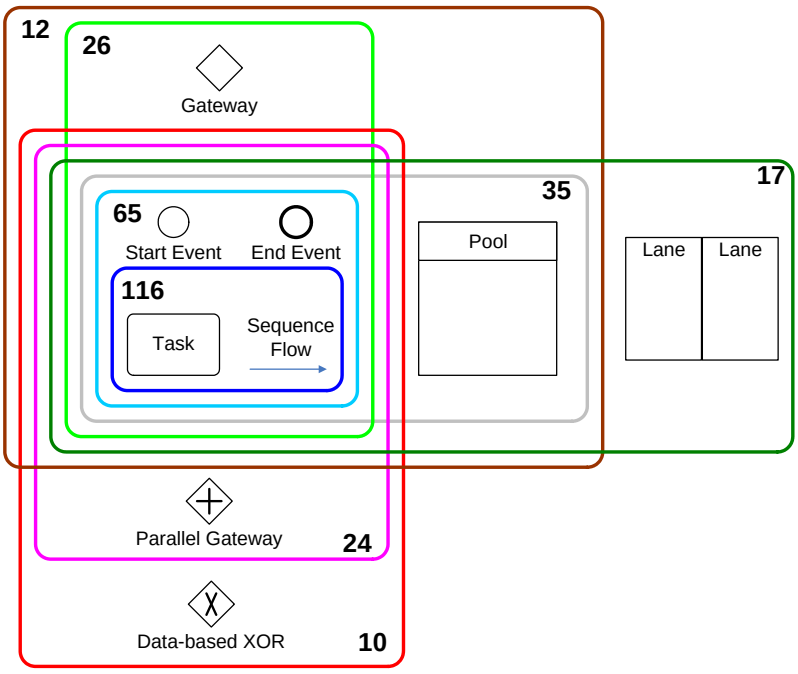


Abbildung 2.14: Venn Diagramm der meistverwendeten BPMN Elemente[49]

[1] siehe: `http://www.column2.com/2008/03/the-great-bpmn-debate/`

2.6 Vergleich BPMN und BPEL

Eine Fragestellung, die vor einer Transformation von BPMN nach BPEL geklärt werden muss, ist, ob Elemente oder Strukturen existieren die keine Entsprechung in der jeweils anderen Spezifikation haben. Decken die beiden Spezifikationen wirklich diesselbe Domäne ab, oder ist es möglich einen BPMN Prozess zu modellieren, der Aufgrund verwendeter Elemente oder Patterns nicht als BPEL Prozesse abgebildet werden kann. So finden sich einige Arbeiten zur Evaluierungen von BPMN und BPEL die sich aber entweder auf unterschiedliche Aspekte konzentrieren oder aber verschiedene, nicht vergleichbare, Methoden anwenden [21].

2.6.1 Control-flow Patterns

Der wichtigste Aspekt den es abzuklären gilt, ist ein Vergleich der Modellierungsmöglichkeiten beider Sprachen anhand der möglichen untestützten Prozess Patterns. Starke Unterschiede in der Ausdruckskraft würden eine Transformation erschweren oder gar unterbinden.

Es existieren zahlreiche Arbeiten die anhand des *Workflow Pattern Framework*[5] unterschiedliche Spezifikationen im Umfeld der Workflow Modellierung vergleichen. Dieses Framework definiert und beschreibt zahlreiche Patterns, die bei der Modellierung von Prozessen verwendet werden, aus unterschiedlichen Perspektiven, inbesonders aber in Bezug auf den Kontrollfluss, Daten und die Ausnahmebehandlung. Die Bandbreite der Patterns reicht dabei von einfachen bis zu sehr komplexen Szenarien. Sie werden in unterschiedlichen Kategorien zusammengefasst. So vergleicht White[41] BPMN Anhand dieser Muster mit UML2. Wohed[44] untersucht die Unterschiede zwischen BPEL, XLANG und WSFL. In [45] geht er detailliert auf BPMN ein und ergänzt den Vergleich noch um die Daten und Ressourcen Perspektive.

Basic Patterns

Die fünf elementaren Patterns wie *sequence*, *parallel split*, *synchronisation*, *exclusive choice* und *simple merge* werden sowohl von BPMN als auch BPEL in vollem Umfang unterstützt.

Komplexe Verzweigungs- und Synchronisations-Pattern

Unter diese Kategorie fallen komplexere branching und synchronisations Szenarien, die keine direkte Entsprechung in Modellierungssprachen haben aber häufig in realen Geschäftsprozessen vorkommen. Dabei handelt es sich um *multiple choice*, *synchronising merge*, *multiple merge*, *discriminator*. BPMN unterstützt alle vier Patterns. Die Modellierung eines *synchronizing merge* in BPMN ist jedoch nur innerhalb einer strukturierten Umgebung möglich. Auch das *discriminator* Pattern ist nicht für jede Aktivität einsetzbar. Ein *multiple merge* wird in BPEL nicht unterstützt. Eine solche Aktivität muss im Prozess dupliziert werden. Das *discriminator* Pattern wird weder direkt unterstützt, noch kann

es mit Hilfe anderer Konstrukte modelliert werden. Die anderen beiden Patterns können in BPEL modelliert werden.

Structural Patterns

Strukturelle Patterns beschreiben, ob bestimmte Beschränkungen der Formalismen in Bezug auf die Strukturierung des Prozesses beim Modellieren bestehen. Besonders relevant sind Aspekte wie die Art von unterstützten Schleifen oder ob es einer einzigen End Aktivität bedarf. Folglich heissen die beiden Patterns *arbitrary cycles* und *implicit termination*. Beide Patterns werden von BPMN unterstützt. BPEL unterstützt jedoch keine *arbitrary cycles*. Die `while` Aktivität kann nur strukturierte Schleifen, also mit einem Ein- und Ausgang, abbilden. Da die Spezifikation aber besagt, daß `link` Elemente keine Schleifen bilden dürfen ist es nicht möglich *arbitrary cycles* direkt in BPEL zu modellieren.

Multiple Instances Patterns

Sie beschreiben Szenarien in Prozessen in denen mehr als eine Instanz einer Aktivität in derselben Prozessinstanz aktiv ist. Es wird dabei zwischen vier Patterns unterschieden. a)*MI without synchronisation*, b)*MI with a priori design time knowledge*, c)*MI with a priori runtime knowledge*, d)*MI without a priori runtime knowledge*. Pattern d) wird nicht von BPMN unterstützt da es nicht möglich ist Instanzen zur Laufzeit hinzuzufügen. Die übrigen drei können abgebildet werden. Pattern a) und b) können in BPEL modelliert werden, indem für jede Instanz ein externer Prozess aufgerufen und instanziert wird. Dieser Ansatz funktioniert aber nur a priori. c) und d) lassen sich durch eine Schleife mit Zähler realisieren, was jedoch keine vollwertige Lösung darstellt.

State-Based Patterns

Sie beschreiben Situationen in denen der weitere Verlauf eines Prozesses durch den Zustand der Prozess Instanz bestimmt wird. Es handelt sich um *deferred choice*, *interleaved parallel routing* und *milestone*. Einzig *deferred choice* kann vollständig in BPMN modelliert werden. *Interleaved parallel routing* kann unter bestimmten Bedingungen dargestellt werden. Das *milestone* Pattern wird nicht unterstützt. In BPEL ist es möglich alle drei Patterns durch Hilfskonstruktionen abzubilden, die aber nur im Fall der *deferred choice* wirklich diesselbe Semantik besitzen. Zusätzlich sind diese Lösungen sehr komplex und aufwendig.

Cancellation Patterns

Sie beschreiben die Möglichkeiten der Terminierung von Prozessinstanzen oder Aktivitäten unter bestimmten, ausformulierten Bedingungen. Die Patterns werden sowohl von BPMN als auch BPEL direkt unterstützt.

Conclusio

Allgemein kann also gesagt werden, daß in den meisten Fällen eine Transformation von BPMN nach BPEL möglich ist. Ein relevantes Pattern, für das im Zuge einer Transformation aber ein Ansatz zur Transformation gefunden werden muss, ist das *arbitrary cycles* Pattern.

2.6.2 Process Representation Paradigm Mismatch

Der Vergleich der beiden Spezifikationen sollte jedoch nicht nur alleine Aufgrund der unterstützten Patterns erfolgen. Sowohl [31] als auch [21] argumentieren, daß bei einem Vergleich von Spezifikationen auch der zugrundeliegende Ansatz berücksichtigt werden sollte. Im Falle von BPMN und BPEL treffen zwei fundamental unterschiedliche Ansätze aufeinander, eine graphbasierte auf eine blockbasierte Darstellungs- bzw Modellierungsweise.

(a) graphbasierte Darstellung (b) blockbasierte Darstellung

Abbildung 2.15: graphbasierte versus blockbasierte Darstellung von Kontrollfluss.

In BPMN werden Prozesse vereinfacht durch Knoten, die miteinander über Kanten in einem Graph verbunden sind, dargestellt-wie in Abbildung 2.15(a) auf Seite 2.15(a) skizziert. Dabei gibt es nur geringe Beschränkungen der Verbindungen. BPEL bedient sich einer blockbasierten Darstellung die den Kontrollfluss eines Prozesses mit Hilfe von verschachtelten strukturierten Aktivitäten beschreibt. Nur sehr begrenzt können auch graphbasierte Konzepte eingesetzt werden solange diese nicht zyklisch sind und innerhalb bestimmter Elemente bleiben. Dadurch sind in BPMN beliebige Schleifen möglich, während BPEL nur strukturierte Schleifen, also mit einem Ein und Ausgang, erlaubt.

2.7 openArchitectureWare

openArchitectureWare (oAW) [4] ist ein modulares Generator Framework für die Modell-getriebene Softwareentwicklung, das vollständing in Java implementiert ist.

Model Driven Software Development MDSD soll eine erhebliche Steigerung der Entwicklungsgeschwindigkeit ermöglichen. Das Mittel dafür ist die automatische Erzeugung von Code aus formal eindeutigen Modellen. Darüber hinaus wird durch diesen Ansatz auch die Softwarequalität verbessert. Ein weiteres Ziel ist es, die Komplexität der Entwicklung durch Abstraktion zu mildern. Dies geschieht durch die Trennung von fachlichen und technischen Anteilen. Sogennanter "dummer" Code soll möglichst automatisch generiert werden, um die dadurch gewonnen Zeit in eine bessere Fachlogik investieren zu können.

oAW unterstützt das parsen von beliebigen Modellen und bietet eine Serie von Sprachen zur Validierung, Transformation und Generation von Code aus Modellen. Als Entwicklungsumgebung wird Eclipse verwendet, da oAW auf das Eclipse Modelling Framework (EMF) als Standard zur Beschreibung von Modellen aufbaut. Es kann jedoch auch auf Modellen, die in anderen Standards beschrieben sind, arbeiten.

2.7.1 Metamodell

Die Metamodelle, als Basis der konkreten Modelle mit denen oAW arbeitet, können dabei sowohl auf Basis von UML2 als auch EMF entwickelt werden. Zusätzlich können mit Hilfe von Xtext eigene Metamodelle beschrieben werden. Auf Xtext wird in einem späteren Abschnitts dieses Kapitels noch eingegangen.

2.7.2 Workflow Engine

Kernstück von oAW bildet eine deklarative konfigurierbare Generator Engine. Sie verwendet eine XML basierte Sprache zur Beschreibung von Generator Workflows. Ein Generator Workflow besteht aus einer Serie von Workflow Components die sequentiell in einer JVM abgearbeitet werden. Abbildung 2.16 stellt einen Transformationsprozess mit vier Komponenten graphisch dar. Es können eigene Workflow Komponenten entwickelt werden, oAW stellt aber bereits eine große Zahl von Komponenten für die am häufigsten benötigten Aufgaben bereit, die nur noch geringe Anpassungen benötigen. Falls diese Komponenten jedoch nicht ausreichen, gibt es eine Reihe von unterschiedlich komplexen Java Interfaces für den Entwurf von eigenen Workflow Komponenten.

Listing 2.1 zeigt einen Ausschnitt aus einem oAW Workflow mit der Definition eines Workflow Component. Diese Komponente definiert das verwendete Metamodell *mm* und ruft anschliessend die xTend Funktion `path::doSomething()` auf. *bpmnModel* ist eine Inputvariable der Funktion. Das Ergebnis der Funktion wird wieder unter diesem Variablennamen gespeichert. Die letzten beiden Zeilen der Komponente definieren zwei globale Variablen auf die die Funktion Zugriff haben soll.

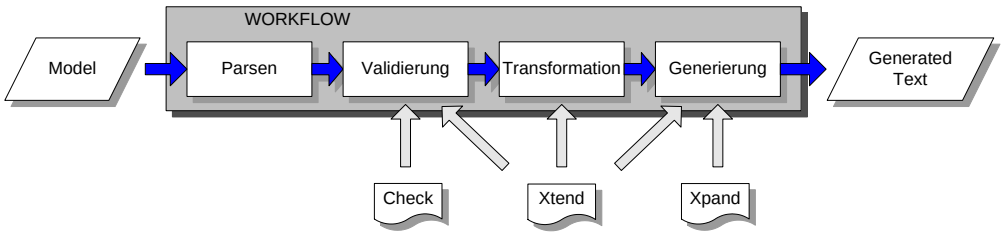


Abbildung 2.16: Vier typischen Workflow Komponenten eines oAW Transformationsprozesses

```
<component class="oaw.xtend.XtendComponent">
   <metaModel  id='mm' class='org.eclipse.m2t.type.emf.EmfRegistryMetaModel'/>
   <invoke  value="path::doSomething(bpmnModel)"/>
   <outputSlot value="bpmnModel"/>
   <globalVarDef  name="logVerbose" value='${transform.logVerbose}'/>
   <globalVarDef  name="debug" value='${transform.debug}'/>
</component>
```

Listing 2.1: Eine Xtend Workflow Komponente

2.7.3 Das Expression Framework

oAW stellt textuelle Sprachen für unterschiedliche Aufgaben im Verlauf eines MDSD Prozesses zu Verfügung. Alle Sprachen basieren auf demselben Typsystem und arbeiten mit derselben Expression Language. Daher können sie ohne Änderung der Syntax auf denselben Modellen, Metamodellen und Meta-Metamodellen operieren. Die Syntax der Expression Language von oAW ist eine Mischung von Java und OCL. Das Typsystem sowie die Expression Language werden vom Expression Framework von oAW bereitgestellt. Das Expression Framework unterstützt weiters Multimethoden. Darunter versteht man überladene Methoden, die nicht aufgrund des Types des impliziten ersten Parameter `this` eines Objektes gewählt werden, sondern aufgrund aller ihrer Parameter.

Xtend

Mit Hilfe von Xtend lassen sich Bibliotheken von unabhängigen Operationen und Erweiterungen von Metamodellen in Java oder oAW Expressions definieren. Diese können von allen anderen textuellen Sprachen des Frameworks referenziert werden. Bei Xtend handelt es sich um eine funktionale Sprache. Im Kontext von Xtend werden Methoden als Extension bezeichnet.

Eine Extension kann auf zwei unterschiedliche Arten aufgerufen werden. Einerseits wie eine Funktion, in der Form `extensionName(meinElement)` andererseits mit Hilfe der sogenannten “member syntax” `meinElement.extensionName()`. Dabei kann es sich bei `meinElement` um ein Objekt oder eine Liste handeln.

Die Extension in Listing 2.2 auf Seite 24 überprüft, ob alle ausgehenden Sequence Flows eines Elements das Zielelement erreichen können.

Listing 2.2: Eine Xtend Extension

```
/**
 * Erreichen alle ausgehenden SequenceFlows das Zielelement
 */
boolean doOutgoingSequenceFlowsReach(Element out,Element target):
   out != target &&
   out.getOutgoingSeq().size > 0 &&
   target.getIncomingSeq().size > 0 &&
   (
      getSuccessor(getElementsOfProcess(out),out).forAll(suc|
         getAllSuccessor(getElementsOfProcess(out),suc,{}).contains(target)
         )
      ||
      getSuccessor(getElementsOfProcess(out),out).forAll(suc|
         suc.id == target.id
         )
   );
```

Xpand

Die Codegenerierung erfolgt mit Hilfe von Templates. Ein Template besteht aus normalem Text, Xpand Ausdrücken und Kommentaren. Das Listing 2.3 zeigt ein Xpand Template. In Zeile 4 beginnt ein Template namens `Root` für ein Objekt vom Typen *BPD*. Anschliessend wird in Zeile 6 eine neue Datei erzeugt. In diese Datei wird `<Diagram .... >` geschrieben und anschliessend in Zeile 8 ein Template `pool` für ein gleichlautendes Element, das ein Attribut von *BPD* ist, aufgerufen. Bei den Ausdrücken zwischen « » handelt es sich um Xpand Ausdrücke.

Listing 2.3: Ein Xpand Template

```
«IMPORT bpmn»
«EXTENSION tuwien::bpmn2bpel::bpmn::generator::bpmnExpandExt»

«DEFINE Root FOR BPD»
   «IF generate()»
   «FILE fileName()-»
      <Diagram «id» name "«name»"«debugExpand("Processing BPD '"+this.id+"'",0)»
         «EXPAND pool FOREACH  pools-»
      >
   «ENDFILE»
   «ENDIF»
«ENDDEFINE»
```

Check

Mit Check bietet oAW eine Sprache zur Formulierung von Beschränkungen für Modelle. Dabei wird zwischen Warnungen und Fehlern, die zum Abbruch des entsprechenden

Workflows führen, unterschieden. Mit ihrer Hilfe sind komplexe Überprüfungen des Modells möglich.

Die Regel in Listing 2.4 ist ein Beispiel für eine typische Check Regel. In diesem Fall wird überprüft ob das ***source*** Attribut eines *SequenceFlow* auf ein gültiges *FlowObject* verweist. Ist dies nicht der Fall, wird ein Fehler gemeldet und eine Fehlermeldung ausgegeben.

Listing 2.4: Eine Check Regel

```
    context SequenceFlow ERROR "Source not a valid Flow Object" :
2       eRootContainer.eAllContents.typeSelect(FlowObject).exists(f|
            f.id == this.source
4           );
```

2.7.4 Erweiterungen des oAW Frameworks

Java Extensions

Nicht immer reichen die Ausdruckskraft und die von von der Expression Language zur Verfügung gestellten Methoden aus. Andererseits kann es auch vorkommen, dass man Methoden sowohl in der Expression Language auch als in der Implementierung benötigt. In diesen Fällen können diese Methoden zur Vereinfachung und besseren Übersicht in externe Java Klassen ausgegliedert werden. Die Methoden der Java Klassen können dann wie Expressions aufgerufen werden. Der Zugriff auf externe Resourcen und komplizierte Kontrollstrukturen sind mögliche Anwendungsgebiete.

Code-Integration und Recipes

Mit Hilfe der Codeintegration ist es möglich, den automatisch und den manuell erstellten Code miteinander zu verbinden. Dabei werden geschützte Bereiche innerhalb der generierten Dateien definiert. Zeilen innerhalb dieser geschützten Bereiche werden bei einer erneuten Generierung in neue Datei übernommen. Recipes (Rezepturen) werden bereitgestellt um den manuell hinzugefügten Code auf bestimmte Strukturregeln hin zu überprüfen. Die Prüfungen sind in Java implementiert. Einzelne Prüfungen lassen sich zu einer Checkliste zusammenfassen, die als Komponente in den Workflow eingebunden wird. Der Generator erzeugt daraus eine Recipe Datei. Diese wird in Eclipse eingebunden. Die Überprüfungen werden dann während der Entwicklung im Hintergrund durchgeführt und etwaige Probleme sofort in Eclipse angezeigt.

Cartridges

Cartridges sind Generatoren die als Blackbox in einen Workflow gehängt werden können. Sie haben eine klar definierte Schnittstelle und führen auf einem konformen Modell eine

bestimmten Operation durch. Ihr Mechanismus ermöglicht die schnelle und unkomplizierte Weitergabe und Wiederverwendung von unterschiedlichen Generatoren.

Xtext

Mit Xtext können eigene textuelle domänenspezifische Sprachen erstellt werden. Die Beschreibung der Metasprache und des Modells erfolgt in einer der EBNF ähnlichen Sprache. Xtext erzeugt daraus in einem oAW Workflow ein Metamodell, einen Parser für das Modell, ein Editor-Plugin mit syntaxhighlightning und codecompletion für Eclipse sowie Komponenten für oAW Workflows.

2.7.5 Zusammenfassung der Stärken und Schwächen

openArchitectureWare ist ein mächtiges Framework für die Umsetzung von MDSD Projekten. Die bereites vorhandene Funktionalität lässt sich durch Erweiterungen an die jeweiligen Bedürfnisse anpassen. Folgende Aspekte zählen zu den Stärken von oAW:

- Es können beliebige Metamodelle verwendet werden. Bedingung ist, daß ein Parser für oAW existiert und das Modell in Java implementiert werden kann.
- Erzeugung von Quelltext
- Einfache Syntax und Semantik. Selbst in Templates ist die Syntax aber doch so unterschiedlich, daß eine Verwechslung von Expressions mit dem zu erzeugenden Text sehr schwer möglich ist.
- Die Möglichkeit die Expression-Language mit Java Klassen zu kombinieren.
- Das Xtext Framework zur Erzeugung von Metamodellen und Editoren.
- Umfangreiche Beispiele und Dokumentation.
- Eine große Community, die rasch hilft.
- Als Framework ist es sehr flexibel und lässt sich für die unterschiedlichsten Anwendungen verwenden.

Gleichzeitig besitzt oAW auch einige Schwächen die mit den obigen Stärken einhergehen.

- Die Installation von oAW in Eclipse funktioniert nicht immer reibungslos.
- Hohe Einarbeitungszeit und eine steile Lernkurve zu Beginn des Projektes.
- Die Dokumentation hinkt in einigen Bereichen der aktuellen Version hinterher.
- Die Fehlersuche ist sehr aufwendig da nur bestimmte Teile des Transformationsprozesses debuged werden können.

3 Transformation von BPMN nach BPEL

Aufbau des Kapitels

In Abschnitt 3.1 werden die Anforderungen an die Transformation von BPMN nach BPEL präzisiert. Im darauf folgenden Abschnitt 3.2 wird die Idee der Transformation vorgestellt. Abschnitt 3.3 stellt die wichtigsten Methoden zur Vereinfachung von BPMN Prozessmodellen vor. Im letzten Abschnitt des Kapitels 3.4 wird der Ablauf der in den Abschnitten zuvor dargstellten Transformationsschritte erläutert.

3.1 Ziele der Transformation

Die Transformation von BPMN nach BPEL soll folgende Ziele erfüllen.

- Aus einem gültigen BPMN Modell soll ein ausführbarer BPEL Prozess erzeugt werden.
- Der erzeugte BPEL Prozess soll möglichst kompakt und für den Anwender lesbar sein.
- Ferner sollen dem Ausgangsmodell möglichst wenige Beschränkungen auferlegt werden.

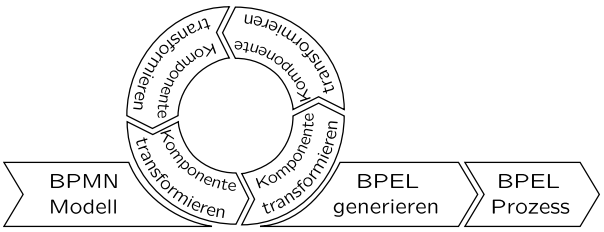
Der letzte Punkt ist von Interesse, da in der Literatur viele Ansätze zur Transformation von Graphen in eine blockbasierte Form, den Modellen relativ starke Einschränkungen auferlegen. In den meisten Fällen geschieht dies um eine Darstellung des Prozesses als Petri Netz zu ermöglichen. Dieses kann dann auf Korrektheit überprüft werden und somit auch die Korrektheit des Prozesses.

In diesem Zusammenhang wird von einem Standard Process Model (SPM) gesprochen oder der Prozess wird als Work-Flow net (WF-net) modelliert. So verwendet [39] ein WF-Net als Ausgangsbasis. In Abschnitt 3.2.2 wird noch näher auf SPMs eingegangen.

3.2 Der Transformationsansatz

3.2.1 Grundgedanke

Den Grundgedanken des gesamten Transformationsansatzes skizziert die folgende Graphik:



Um das BPMN Modell in einen lesbaren BPEL Prozess umzuwandeln und dabei die `structured activities` von BPEL zu nutzen, wird der Graph in einem ersten Schritt zerlegt. Diese Teilgraphen sind Ausschnitte des Graphen mit genau einer Start- und End-Aktivität, die einer bestimmten Struktur entsprechen. So kann zum Beispiel eine sequentielle Serie von Elemente in einem Teilgraphen durch eine BPEL `sequence` dargestellt werden, ein Teilgraph hingegen, der eine parallele Struktur enthält, durch ein BPEL `flow` Element. In einem zweiten Schritt werden die so gefundenen Komponenten des Modells, die direkt einer bestimmten BPEL Struktur entsprechen, innerhalb des BPMN Modells in einem *Subprocess* Element gekapselt. Dieses wird an Stelle der gekapselten Elemente in den Prozess eingefügt.

Diese zwei Schritte werden solange wiederholt bis nur noch ein *Subprocess* Element als Wurzelelement des Modells vorhanden ist, oder aber es nicht mehr möglich ist, im verbliebenen Prozess eine bekannte Struktur zu finden.

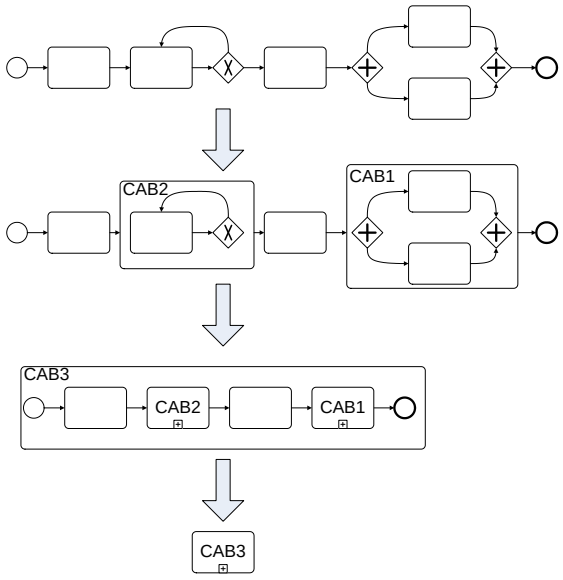


Abbildung 3.1: Schrittweise Transformation des BPD

Es findet also eine Transformation des ursprünglichen unstrukturierten BPD in ein strukturiertes BPD statt. Abbildung 3.1 demonstriert diesen Ablauf anhand eines einfachen Prozesses. Für den Fall, daß nach einem Transformationsschritt keine Strukturen mehr identifiziert werden können, werden die verbliebenen Elemente mit Hilfe zweier Methoden übersetzt die eine Transformation unstrukturierter Teilgraphen erlauben. Diese

produzieren jedoch einen schlecht lesbaren BPEL Code und werden daher erst als letztes Mittel eingesetzt. Das so erhaltene BPD wird im Anschluss rekursiv in einen BPEL Prozess übersetzt.

3.2.2 Beschränkungen des Modells

Eines der Ziele ist es, dem Ausgangsmodell möglichst wenige Beschränkungen aufzuerlegen, um eine größere Bandbreite von Prozessen modellieren und damit transformieren zu können, als es vergleichbare Ansätze vermögen. Diese meisten der vergleichbaren Transformationsansätze verwenden, sofern sie nicht direkt ein Petri Netz gebrauchen, ein sogenanntes wohlgeformtes BPD oder auch standard process model (SPM). Diese sind folgendermaßen charakterisiert:

Standard Prozess Modell

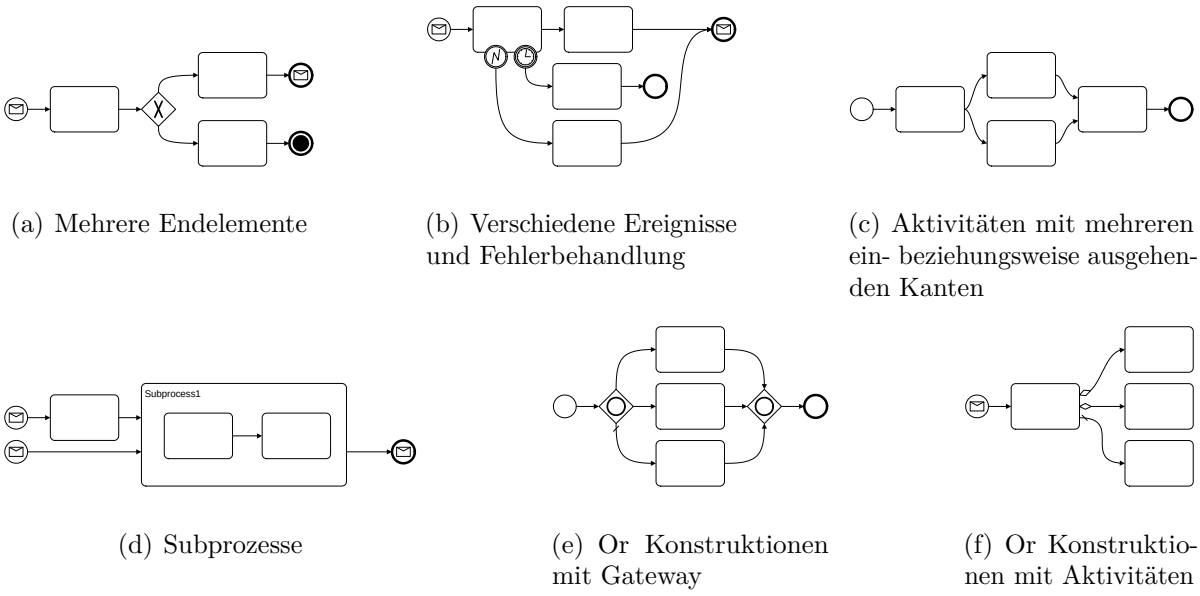
- Sie bestehen aus nur drei Arten von Elementen. Aktivitäten, Gateways (AND und XOR) sowie Kanten die diese verbinden.
- Start- bez. Endelemente haben nicht mehr als einen einen Nachfolger beziehungsweise Vorgänger.
- Aktivitäten und Events, die keine Start- oder Endevents sind besitzen nur einen Nachfolger und Vorgänger.
- Gateways müssen entweder einen Vorgänger und mehrere Nachfolger besitzen oder einen Nachfolger und mehrere Vorgänger.
- Jedes Element muss auf einem Pfad vom Startelement zum Endelement liegen. Es kann also nur ein Startelement und ein Endelement geben.

Ein solches Modell bildet relativ gut einfache Prozesse ab, komplizierte Prozesse lassen sich damit jedoch nicht modellieren und somit auch nicht transformieren. Um kompliziertere Prozesse transformieren zu können war es notwendig ein weniger restriktives Metamodell zu verwenden.

BPMN Metamodell

Daher wurden als Basis des Metamodells die Beschreibung der Elemente der BPMN Spezifikation gewählt. Es können also BPMN konforme Modelle als Ausgangsmodelle des Transformationsprozesses verwendet werden. Es existieren im wesentlichen nur die Restriktionen, die sich aus der BPMN Spezifikation für die einzelnen Elemente eines BPMN Diagrammes ergeben. Abbildung 3.2 zeigt einen Ausschnitt aus der Bandbreite der mit diesem Metamodell modellierbaren Prozesse:

Damit sind eine Reihe von Strukturen in den Prozessen modellier- und transformierbar, die in bisherigen Ansätzen nicht verwendet werden konnten.



(a) Mehrere Endelemente

(b) Verschiedene Ereignisse und Fehlerbehandlung

(c) Aktivitäten mit mehreren ein- beziehungsweise ausgehenden Kanten

(d) Subprozesse

(e) Or Konstruktionen mit Gateway

(f) Or Konstruktionen mit Aktivitäten

Abbildung 3.2: Vom Metamodell unterstützte Prozesse die in bisherigen Transformationsansätzen nicht möglich sind.

- Abbildung 3.2(a): mehrere Start- und Endelemente.
- Abbildung 3.2(b): komplexe Ausnahme- und Fehlerbehandlung.
- Abbildung 3.2(c): Aktivitäten können mehrere eingehende und ausgehende *SequenceFlow* Elemente besitzen.
- Abbildung 3.2(d): Strukturierung von Prozessen mit Hilfe von Subprozessen.
- Abbildung 3.2(e) und 3.2(f): Es ist möglich komplizierte OR Konstruktionen zu verwenden. Diese dürfen sowohl durch Gateways als auch Aktivitäten gebildet werden.

Ein Nachteil dieses Metamodells ist die schwierigere Verifikation der Prozesse auf ihre Korrektheit.

3.2.3 Teilgraph

Der Grundgedanke des Transformationsprozesses ist die Zerlegung des Prozesses in Teilgraphen die einer bestimmten Struktur entsprechen. Dieser Ansatz wird von den meisten Methoden verwendet, die versuchen Modelle mit Hilfe von BPEL `structured activities` zu transformieren. In [27] werden die Teilgraphen als *Clusterable Activity Block* bezeichnet und als schwach zusammenhängende Komponenten eines gerichteten Graphen definiert, der maximal einen Start- und einen End-Punkt hat. Weiters besteht ein CAB aus Activities, Control Nodes, jedoch ohne AND-Splits und AND-Joins, und den Verbindungen zwischen diesen Elementen.

CAB

Die Bezeichnung CAB wurde für diese Arbeit übernommen, jedoch wurden die Beschränkungen in Bezug auf die unterstützten Elemente fallengelassen. Diese waren in der ursprünglichen Definition notwendig, da das der Arbeit zugrundeliegende Metamodell auf einem Petri Netz basierte. Ein CAB im Zuge des, in dieser Diplomarbeit präsentierten Transformationsprozesses muss daher folgende Bedingungen erfüllen:

- Ein CAB darf maximal einen eingehenden Sequence Flow besitzen, daher nur ein Startelement.
- Ein CAB darf maximal einen ausgehenden Sequence Flow besitzen, daher nur ein Endelement.
- Ein CAB muss mehr als ein BPMN Element enthalten, kann sich jedoch aus einer beliebigen Menge von Elementen und Kombination derselben zusammensetzen.

Die Start- und Endelemente eines CAB sind in keiner Weise auf bestimmte Element Typen beschränkt. Dadurch sind alle Kombinationen von *FlowObject* Elementen möglich, sofern dies die BPMN Spezifikation erlaubt. Aufgrund der Verwendung eines BPMN konformen Metamodells ist die Integration des CAB in das Metamodell auch relativ einfach. Es handelt sich dabei um einen BPMN *Subprocess* für den dieselben Einschränkungen gelten.

3.2.4 Kategorien von CAB Typen

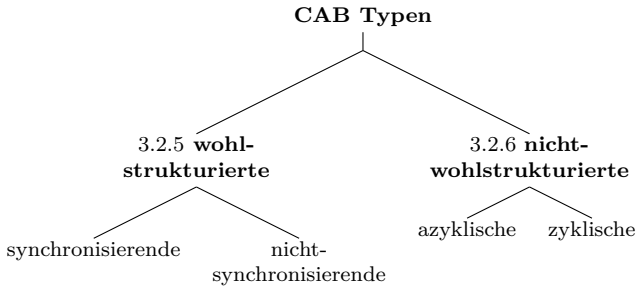
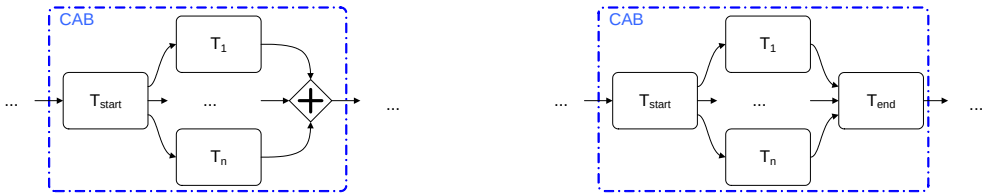


Abbildung 3.3: Kategorisierung unterschiedlicher CAB Typen

Für die Transformation wurden eine Reihe von CAB Typen identifiziert, die jeweils eine eindeutige Entsprechung in BPEL haben. Diese CAB Typen lassen sich in zwei Hauptkategorien unterteilen-wie in obiger Abbildung dargestellt.

Die erste Gruppe wären alle BPMN Strukturen, für die es eine direkte und eindeutige BPEL Entsprechung gibt. CABs dieser Kategorie werden daher in Folge als wohlstrukturiert bezeichnet. Sie lassen sich ohne großen Aufwand transformieren. Bei der zweiten Gruppe handelt es sich um Strukturen, die sich nicht direkt auf eine BPEL Struktur umlegen lassen. Sie werden als nichtwohlstrukturiert bezeichnet. Um sie in BPEL darstellen zu können sind komplexe Veränderungen notwendig.



(a) synchronisierender CAB (b) Nicht synchronisierender CAB

Abbildung 3.4: Zwei wohlstrukturierte CABs, ein synchronisierender und ein nicht synchronisierender

Eine weitere Unterteilung der wohlstrukturierten CAB Typen ergibt sich durch die unterschiedlichen Endelemente. Handelt es sich bei dem End Element nicht um ein Gateway Element, ist das CAB ein nicht synchronisierendes. Diese Unterscheidung spielt aber erst im Zuge der Erzeugung des BPEL Prozesses eine Rolle.

Auf den folgenden Seiten werden unterschiedlichen CAB Typen und ihre Entsprechungen in BPEL vorgestellt.

3.2.5 Wohlstrukturierte CABs

Folgende Komponenten zählen zu den wohlstrukturierten CABs.

Sequence

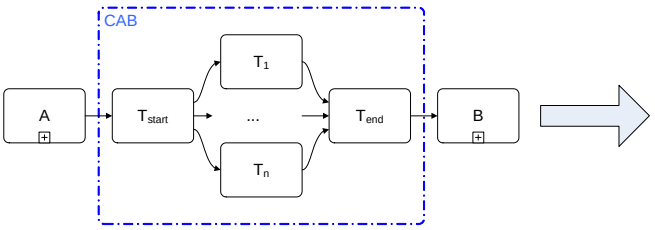
Eine einfach Struktur die dem BPEL sequence Element entspricht. Jedes Element in einem CAB des Typen *Sequence* besitzt maximal einen eingehenden und einen ausgehende *SequenceFlow*.

Flow

Eine Struktur bei der von einem Startelement mehrere parallele Pfade ausgehen die alle bei einem Element wieder zusammenkommen. Je nach Typ des Start und des Endelements können die vier Fälle unterschieden werden, die den folgenden CAB Typen entsprechen: *ActivityActivityFlow*, *ActivityGatewayFlow*, *GatewayGatewayFlow* und *GatewayActivityFlow*.

Activity–Activity und Gateway-Activity Diese Fälle führen zu einem wesentlich umfangreicheren BPEL Prozess. Da das Endelement des CABs kein Gateway ist, der den Prozess synchronisiert, sondern eine Aktivität, müssen alle nachfolgenden Elemente für jeden parallelen Pfad dupliziert werden. Daher kann die Anzahl der Elemente des transformierten Prozesses erheblich größer sein als die Anzahl der Elemente im BPMN Modell.

Abbildung 3.5 zeigt einen einfach Fall der das Problem verdeutlicht. Alle Elemente vor und nach dem *ActivityActivityFlow* CAB sind bereits in den CABs `A` und `B` zusammengefasst. Bei der Transformation eines *ActivityActivityFlow* oder *ActivityGatewayFlow* CABs nach BPEL muss nun für jeden parallelen Pfad die komplette Transformation `B` des restlichen Prozesses an den jeweiligen Pfad angehängt werden.



(a) Ein *ActivityActivityFlow* CAB

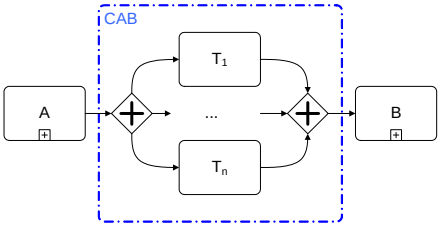
```
<sequence>
  Transformation von A
  Transformation von T_start
  <flow>
    <sequence>
      Transformation von T_1
      Transformation von T_end
      Transformation von B
    </sequence>
    <sequence>
      ...
    </sequence>
    <sequence>
      Transformation von T_n
      Transformation von T_end
      Transformation von B
    </sequence>
  </flow>
</sequence>
```

(b) BPEL Struktur

Abbildung 3.5: Ein Activity–Activity Flow und seine Entsprechung in BPEL

Activity–Gateway und Gateway-Gateway Die Struktur die dem BPEL `flow` direkt entspricht. Da im Unterschied zum vorigen Fall ein Gateway alle parallelen Pfade wieder synchronisiert, ist die Transformation des BPMN Modells direkt und es

werden keine zusätzlichen Elemente im BPEL Prozess eingeführt. In Abbildung 3.6 ist die Transformation eines *GatewayGatewayFlow* CAB dargestellt während 3.7 die eines eines *ActivityGatewayFlow* CAB zeigt.

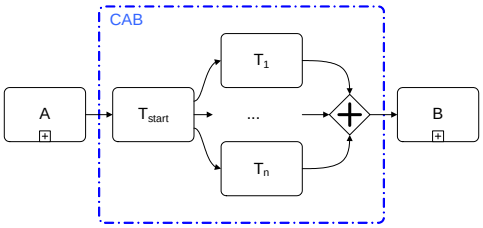


(a) Ein *GatewayGatewayFlow* CAB

```
<sequence>
  Transformation von A
  <flow>
    Transformation von T1
    ...
    Transformation vonTn
  </flow>
  Transformation von B
</sequence>
```

(b) BPEL Struktur

Abbildung 3.6: Ein Gateway–Gateway Flow und seine Entsprechung in BPEL



(a) Ein *ActivityGatewayFlow* CAB

```
<sequence>
  Transformation von A
  Transformation von Tstart
  <flow>
    <sequence>
      Transformation von T1
      Transformation von Tend
    </sequence>
    <sequence>
      ...
    </sequence>
    <sequence>
      Transformation von Tn
      Transformation von Tend
    </sequence>
  </flow>
  Transformation von B
</sequence>
```

(b) BPEL Struktur

Abbildung 3.7: Ein Activity–Gateway Flow und seine Entsprechung in BPEL

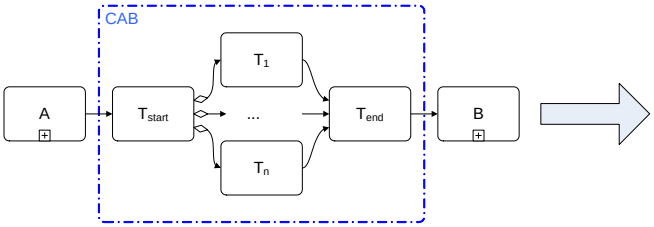
Or

Eine Struktur, die dem Flow Konstrukt sehr ähnlich ist, aber mit dem entscheidenden Unterschied, daß die vom Startelement ausgehenden Pfade mit einer Bedingung verknüpft sind. Laut BPMN Spezifikation kann in einem gültigen Prozess eine beliebige Anzahl von Pfaden des Ors ausgeführt werden, es muss jedoch mindestens einer gewählt werden. Da es keine direkte Entsprechung in einer BPEL Aktivität gibt, führt die Transformation zu zusätzlichen Aktivitäten im Prozess. Auch hier wird wieder, je nach Typ des Endelements, zwischen einem synchronisierten und einem unsynchronisiertem Fall unterschieden. Es existieren also auch in diesem Fall wieder vier unterschiedliche CAB Typen: *ActivityActivityOr*, *ActivityGatewayOr*, *GatewayActivityOr* und *GatewayGatewayOr*.

Die Verwendung eines *default SequenceFlow* führt zur Einführung einer Hilfsvariable bei der Transformation in einen BPEL Prozess wie in Abbildung 3.10 anhand eines Gateway-Gateway Ors dargestellt. Diese Variable wird zum Prozessstart initialisiert. In

jedem anderen Pfad des Ors wird nun eine Zuweisung zu dieser Variable plaziert. Ist sie nach Abarbeitung des Prozesss unverändert, wird der Default Pfad verfolgt. Bei einem Activity-Activity Or wird dasselbe Prinzip angewandt.

Activity–Activity und Gateway–Activity Die Transformation dieses CABs nach BPEL erfordert ebenso zusätzliche Elemente. Wie in Abbildung 3.8 dargestellt, muss analog zum Flow der nachfolgende Teil des Prozesses in jeder möglichen Verzweigung abgebildet werden.



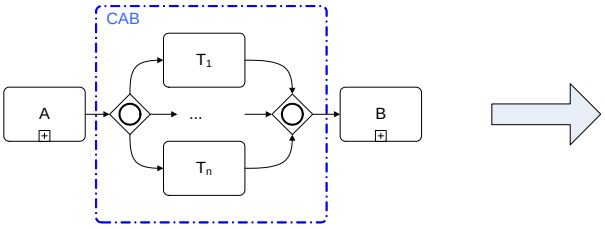
(a) Ein *ActivityActivityOr* CAB

```
<sequence>
  Transformation von A
  Transformation von T_start
  <flow>
    <if>
      <sequence>
        Transformation von T_1
        Transformation von T_end
        Transformation von B
      </sequence>
    </if>
    <if> ... </if>
    <if>
      <sequence>
        Transformation von T_n
        Transformation von T_end
        Transformation von B
      </sequence>
    </if>
  </flow>
</sequence>
```

(b) BPEL Struktur

Abbildung 3.8: Ein Activity–Activity Or und seine Entsprechung in BPEL

Activity–Gateway und Gateway–Gateway Der einfachere Fall eines Ors. Wie in Abbildung 3.9 dargestellt, synchronisieren alle Pfade wieder an einem Gateway und der Rest des Prozesses *B* muss nur einmal abgebildet werden.



(a) Ein *GatewayGatewayOr* CAB

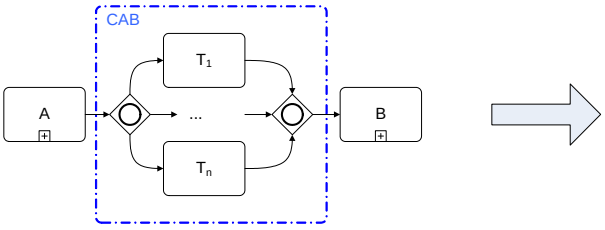
```
<sequence>
  Transformation von A
  <flow>
    <if>
      Transformation von T_1
    </if>
    <if> ... </if>
    <if>
      Transformation von T_n
    </if>
  </flow>
  Transformation von B
</sequence>
```

(b) BPEL Struktur

Abbildung 3.9: Ein Gateway-Gateway Or und seine Entsprechung in BPEL

While

Im einfachsten Fall entspricht diese Struktur genau einem BPEL `while`. Es können jedoch neben dieser direkten Entsprechung noch zwei Fälle unterschieden werden.



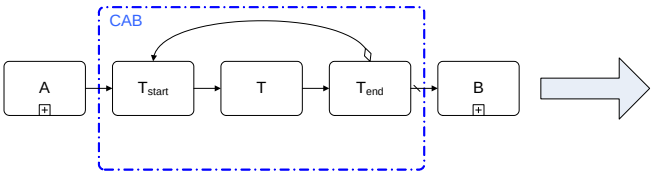
(a) Ein *GatewayGatewayOr* CAB mit*default SequenceFlow*

```
<sequence>
  Transformation von A
  //noDefault mit true initialisieren
  <flow>
    <if>
      <sequence>
        Transformation von T1
        //noDefault auf false setzen
      </sequence>
    </if>
    <if> ... </if>
    <if>//noDefault == true
      Transformation von Tn
    </if>
  </flow>
  Transformation von B
</sequence>
```

(b) BPEL Struktur

Abbildung 3.10: Ein Gateway-Gateway Or mit Default Sequence Flow und seine Entsprechung in BPEL

Simple while Der einfacheste Fall. Der Startknoten hat einen eingehenden *conditional SequenceFlow* von einem Nachfolger. Durch die Bedingung, daß es sich um einen *conditional SequenceFlow* handeln muss, ist implizit garantiert, daß keine Elemente auf dem zurückführenden Pfad liegen. Diese Unterscheidung ist wichtig da dieser Fall der nächste CAB Typ ist.



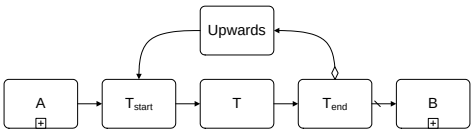
(a) Ein *While* CAB

```
<sequence>
  Transformation von A
  <while>
    <sequence>
      Transformation von Tstart
      Transformation von T1
      Transformation von Tend
    </sequence>
  </while>
  Transformation von B
</sequence>
```

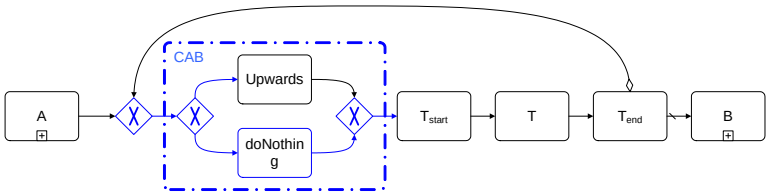
(b) BPEL Struktur

Abbildung 3.11: Ein simples while und seine Entsprechung in BPEL

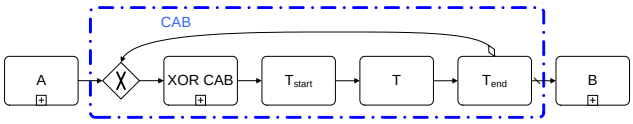
Upwards while Dabei handelt es sich um ein simple while wobei im Unterschied zu diesem, auf dem Pfad zurück zum Startelement des CABs, noch Elemente liegen. Es existiert kein eigener CAB Typ für diese Struktur sondern sie wird, wie in Abbildung 3.12 dargestellt, überarbeitet. In einem ersten Schritt wird das Element ***Upwards*** wie in 3.12(b) dargestellt, innerhalb zweier *DataXor* gekapselt. Diese Struktur wird vor das ursprüngliche Startelement der Schleife T_{start} gehängt. Diese Struktur wird im nächsten Durchlauf des Transformationsalgorithmus als Xor erkannt und transformiert. Einen weiteren Durchlauf später wird die gesamte Struktur als simples while erkannt und transformiert. Nicht abgebildet ist die zusätzlich eingeführte Hilfsvariable, die die Bedingung des kapselnden Xors ist. In Abbildung 3.13 auf Seite 37 ist nocheinmal die Struktur sowie die Entsprechung in BPEL dargestellt.



(a) Ausgangszustand - Der unveränderte Prozess mit Elementen im zurückfliessenden Sequence Flow

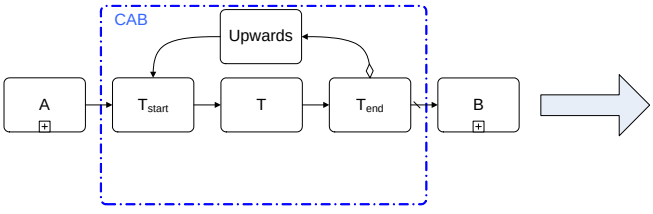


(b) Schritt 1 - Der Prozess nach der Kapselung von ***upwards*** sowie der Umleitung der *SequenceFlow* Elemente



(c) Endzustand - Das upwards while ist in ein simples while Transformiert worden

Abbildung 3.12: Die Transformationsschritte zur Vorbereitung eines upwards while auf die Transformation



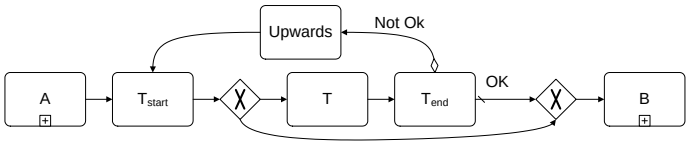
(a) Ein upwards while CAB

```
<sequence>
  Transformation von A
  //Initialisiere Variable
  <while>
    <sequence>
      <if>//Wenn NICHT erster Durchlauf
        Transformation von Upwards
      </if>
      Transformation von T_start
      Transformation von T_1
      Transformation von T_end
      //Speichere erster Durchlauf vorbei
    </sequence>
  </while>
  Transformation von B
</sequence>
```

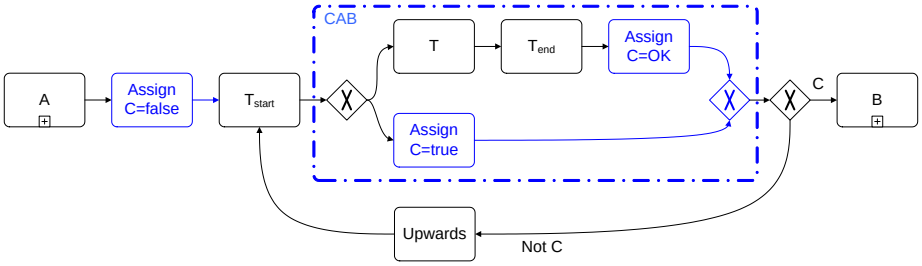
(b) BPEL Struktur

Abbildung 3.13: Ein updwards while und seine Entsprechung in BPEL

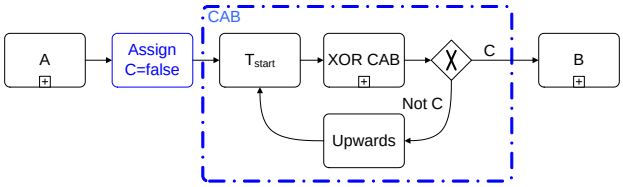
Xor While Eine Struktur, die nur mit Hilfe einer Hilfsvariablen in eine wohlstrukturierte Form übergeführt werden kann. Dadurch kann ein Prozess, der ansonsten nur aus wohlstrukturierten CABs besteht, auf eine Verwendung der Control Link Transformation verzichten. Der Einsatz der Hilfsvariable ist in Abbildung 3.14 dargestellt. In der Variable, in diesem Fall C, wird das Ergebnis der ursprünglichen Bedingung, ob die Schleife durchlaufen wird oder nicht, gespeichert. Zusätzlich werden noch drei Aktivitäten hinzugefügt die die entsprechenden Werte in C speichern. Das Ergebnis dieser Überarbeitung des Graphen ist eine strukturierte Schleife, die mit den beiden oben beschriebenen Ansätzen in einem späteren Schritt zerlegt und übersetzt werden kann.



(a) Ausgangszustand - Der unveränderte Prozess



(b) Überarbeitungsschritt - Der Prozess nach dem Einfügen einer Hilfsvariable samt zugehörigem Gateway



(c) Endzustand - Das xor while ist in ein simples while transformiert worden

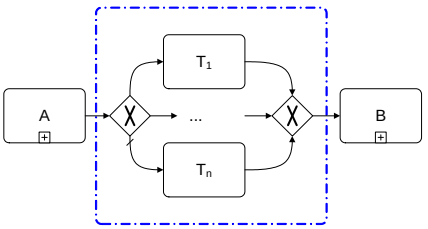
Abbildung 3.14: Die Transformationsschritte zur Vorbereitung eines xor while auf die Transformation

Xor

In BPMN wird zwischen zwei unterschiedlichen Typen von Xors unterschieden. Wird ein bestimmter Pfad aufgrund eines Zustandes einer Varibale verfolgt, handelt es sich um einen data-based Xor. Hängt der weitere Prozessverlauf jedoch von unterschiedlichen Nachrichten ab, die der Prozess an einer bestimmten Stelle empfangen kann, handelt es sich um einen event-based Xor. Die beiden entsprechenden Gateways heissen *EventXor*

sowie *DataXor*. Von beiden Gateways können mehrere *SequenceFlow* ausgehen, jedoch darf laut BPMN Spezifikation nur einer der *SequenceFlow* zur Laufzeit aktiv sein. Erfüllen mehrere der ausgehenden *SequenceFlow* Elemente die Bedingung, wird der in der Reihenfolge erste gewählt. Die anderen *SequenceFlow* Elemente, deren Bedingung ebenfalls erfüllt ist, werden ignoriert. Zusätzlich zu dieser Einteilung kann in beiden Fällen unterschieden werden, ob alle vom Gateway ausgehenden Pfade sich zu einem späteren Zeitpunkt wieder vereinen oder nicht. Alle Fälle werden in einem *XorSplitJoin* CAB gekapselt.

Data-based Xor Ein Gateway mit mehreren ausgehenden Sequence Flow die mit einer Bedingung verknüpft sind. Da nur ein Pfad zur Laufzeit genommen werden darf, entspricht diese Struktur genau den BPEL `if elseif` Strukturen. In Abbildung 3.15 ist die Transformation eines *XorSplitJoin* mit einem *default SequenceFlow* abgebildet.



(a) Ein data-based Xor CAB

```
<sequence>
  Transformation von A
  <if>
    Transformation von T1
    <elseif>
      ...
    </elseif>
    <elseif>
      ...
    </elseif>
      ...
    <else>
      Transformation von Tn
    </else>
  </if>
  Transformation von B
</sequence>
```

(b) BPEL Struktur

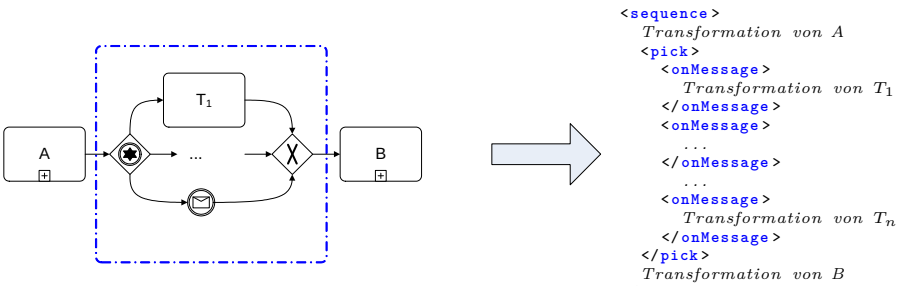
Abbildung 3.15: Ein data-based Xor und seine Entsprechung in BPEL

Event-based Xor Ein Gateway, bei dem je nach nachfolgender Nachricht ein anderer Pfad genommen wird. Der in Abbildung 3.16 abgebildete BPEL Code dient nur zur Darstellung der Struktur. Tatsächlich ist die erste Nachricht des CABs T1 im `onMessage` Element spezifiziert und zwischen den `onMessage` Tags dann die restlichen Komponenten des CABs.

Exception

Eine Exception wird in BPMN als ein *IntermediateEvent* der an einem Element hängt modelliert. Vom *IntermediateEvent* muss ein *SequenceFlow* zu einem Element führen. Dieser sogenannte Exception Flow muss nicht wieder in den Prozess münden. Es können vier Fälle unterschieden werden:

- Der Exception Flow mündet **sofort** nach dem auslösenden Element wieder in den normalen Prozessverlauf.



(a) Ein event-based Xor CAB

(b) BPEL Struktur

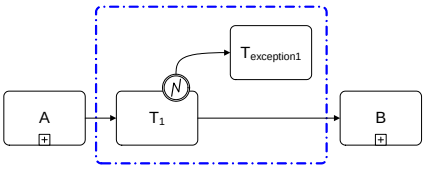
Abbildung 3.16: Ein event-based Xor und seine Entsprechung in BPEL

- Der Exception Flow und der normale Prozessverlauf haben keine gemeinsamen Elemente. Der Prozess endet also nach dem letzten Element des Exception Flow.
- Der Exception Flow mündet erst wieder zu einem **späteren** Zeitpunkt in den Prozess. Folglich benötigt die Transformation eine Hilfsvariable, ob der Prozess nach der Exception fortgesetzt wird, sowie eine weitere Variable, in der gespeichert wird welche Exception ausgelöst wurde.
- Der Exception Flow bildet eine Schleife.

Abbildung 3.17 zeigt einen sehr simplen Prozess mit der resultierenden BPEL Struktur. Die praktische Umsetzung in BPEL ist aber sehr unterschiedlich und hängt sowohl vom Typen des verwendenten *IntermediateEvent* ab, als auch von den vier oben beschriebenen Fällen.

In Abbildung 3.17 ist die Transformation eines Prozesses mit Exception Flow der wieder in den normalen Prozessverlauf mündet skizziert. Die Transformation erfolgt dabei in zwei Schritten, wobei die Abbildung nur den ersten zeigt. Dabei werden dem Ausgangsmodell (Abbildung 3.18(a)) zwei *DataXor* sowie die entsprechenden *SequenceFlow* hinzugefügt und mit diesen der Teil des Prozesses gekapselt, der im Falle einer Exception ausgelassen wird (*between*). Danach wird der *SequenceFlow* zwischen $T_{exception1}$ und B gelöscht sowie zwei *Assignment* Elemente vor T_1 und $T_{exception1}$ in den Prozess gehängt.[1] Anschliessend läuft die Transformation normal weiter und erst im nächsten Durchlauf wird die Struktur als normale Exception erkannt. Durch diese Transformation in zwei Durchläufen ist es auch möglich Strukturen zu transformieren, die mehrere Exception Flows aufweisen. Detailliert wird auf die Transformation von Exception Flow in [43, Seite 142-147 und 182-188] eingegangen.

[1] Eigentlich werden sie direkt im jeweiligen Element gespeichert aber zur Demonstration des Ansatzes werden sie als eigene Elemente in der Abbildung modelliert.

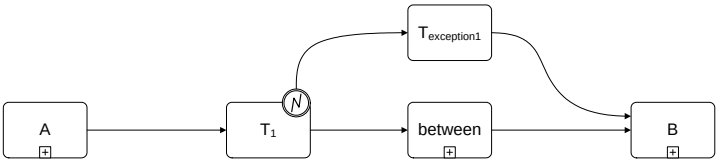


(a) Ein *ActivityScope* CAB

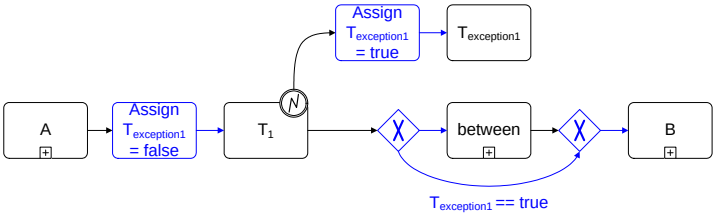
```
<sequence>
  Transformation von A
  //exceptionT1 mit false initialisieren
  //T1normalEnd auf false setzen
  <scope>
    <faultHandlers>
      <catch>
        <sequence>
          Transformation von Texception1
          //exceptionT1 auf true setzen
        </sequence>
      </catch>
    </faultHandlers>
    Transformation von T1
    //T1normalEnd auf true setzen
  </scope>
  <if>//T1normalEnd == true
    Transformation von B
  <elseif>
    <empty>
    //Prozessende
  </else>
  </if>
</sequence>
```

(b) BPEL Struktur

Abbildung 3.17: Ein Exception Intermediate Event und seine Entsprechung in BPEL



(a) Exception Flow - Der Prozess mit Exception Flow der wieder in den Prozess mündet vor der Überarbeitung



(b) Exception Flow - Der Prozess nach der Überarbeitung

Abbildung 3.18: Überarbeitung eines Exception Flows der wieder in den Prozess mündet

3.2.6 Nichtwohlstrukturierte CABs

In BPMN, als graphbasierte Sprache, können Elemente innerhalb bestimmter Regeln beliebig miteinander verbunden werden, daher kann ein BPD auch nichtwohlstrukturierte Komponenten enthalten. Für solche Komponenten sind die im vorigen Kapitel beschriebenen Strukturen nicht mehr anwendbar. Es gibt keine diesen Strukturen direkt entsprechende BPEL Konstrukte. Es ist jedoch mit Hilfe von zwei Verfahren möglich, einen Großteil dieser in BPEL abzubilden.

Nicht zyklische Graphen - Control Link Transformation

Der erste Ansatz verwendet BPEL `link` Elemente in Kombination mit einem `flow` Element. Damit ist es möglich azyklische nichtwohlstrukturierte Komponenten zu transformieren. Die Idee dabei ist es, alle Vorbedingung der Elemente zu sammeln und daraus die entsprechenden `link` Elemente zu generieren. Eine kurze Einführung in die Verwendung eines `flow` mit `link` Aktivität findet sich in Abschnitt 2.4.4.

Die Verwendung von `link` Elementen ist aber aufgrund zweier Aspekte problematisch. Wegen der death-path elimination von BPEL kann sie dazu führen, daß Deadlocks unentdeckt bleiben. Zweitens kann eine Aktivität nur ausgeführt werden, wenn der Status aller eingehenden Links klar und die `joinCondition` evaluiert worden ist. Daher kann die Ausführung einer Aktivität nur maximal eine Ausführung aller durch Links verbundenen nachfolgenden Aktivitäten auslösen. Daher muss für alle Komponenten, die mit Hilfe der Control Link Methode transformiert werden sollen, vor der Transformation sichergestellt werden, daß sie deadlock frei ist und jedes Element nur maximal einmal ausgeführt wird. Es darf also innerhalb der Struktur die übersetzt werden soll, keine Schleifen geben.

Eine Teilgraph der mit Hilfe der Control Link Methode transformiert werden soll muss zusätzlich zu den Bedingungen für CABs, die auf Seite 31 beschrieben sind, daher folgende Bedingungen erfüllen:

- Alle vom Startelement ausgehenden Pfade müssen letztlich wieder bei einem Endelement zusammenkommen. Diese Bedingung ergibt sich aus der Semantik der `flow` Elements in BPEL.
- Es darf keine Schleifen innerhalb des Graphen geben. Alle Elemente dürfen aber mehr als einen ausgehenden *SequenceFlow* besitzen, sofern diese zu unterschiedlichen Elementen führen.

Die Control Link Methode läuft in einigen Schritten ab. Abbildung 3.19 zeigt einen Ausschnitt aus einem Prozess der im folgenden verwendet wird, um die Transformationsschritte zu demonstrieren. Es handelt sich dabei um ein Minimalbeispiel um den Ansatz zu veranschaulichen. Ein umfangreicherer, nichtwohlstrukturierter Prozess würde keine zusätzlichen Informationen bieten, sondern nur schwieriger darstellbar sein.

Kapseln In einem ersten Schritt werden zwei neue *None* Activities erzeugt und der Teilgraph zwischen diesen beiden gekapselt. Wenn das Start- und Endelement des CAB

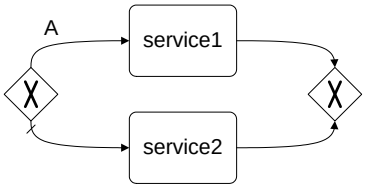


Abbildung 3.19: Der Prozess vor der Anwendung der Control Link Transformation.

ein Gateway ist, gäbe es sonst kein Element von dem die `link` Elemente ausgehen, beziehungsweise wieder zusammenkommen könnten. Abbildung 3.20 zeigt den Prozess aus Abbildung 3.19 nach der Kapselung.

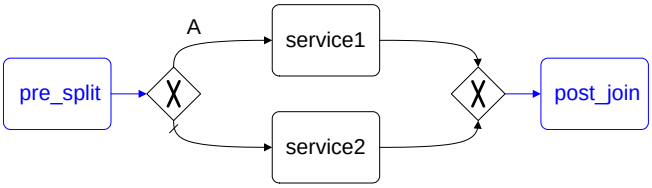


Abbildung 3.20: Kapselung der Struktur innerhalb zweier neuer *None* Elemente.

Vorbedingungen überarbeiten Der nächste Schritt ist die Überarbeitung aller Vorbedingungen. Dies ist notwendig um nach der Transformation weiterhin die BPMN Spezifikation zu erfüllen. Diese verlangt, daß die Bedingungen der einzelnen Zweige eines data-based Xor Gateways in der Reihenfolge der zugehörigen Gates geprüft werden sollen. Dabei wird nur der erste Zweig, dessen Vorbedingung erfüllt ist, weiter verfolgt. Alle anderen Zweige deren Bedingung auch erfüllt sind werden ignoriert. Um diese Anforderung nach der Transformation noch zu erfüllen müssen die Vorbedingunge aller Sequence Flows des Gateways überarbeitet werden.

Es werden die Vorbedingungen aller *SequenceFlow* explizit kombiniert und auch der *default SequenceFlow* durch einen *conditional SequenceFlow* ersetzt. $\{sf_1, ..., sf_n\}$ steht für die ausgehenden Sequence Flows und $Bed(sf_i)$ für die Vorbedingung von sf_1 . Dabei wird $Bed(sf_i)$ in der Reihenfolge von sf_1 bis sf_n überprüft, wobei sf_n der default Sequence Flow ist. Abbildung 3.21 zeigt eine formalisierte Darstellung zur Erzeugung der überarbeiteten Vorbedingung.[28]

$$NeueBed(sf_i) = \begin{cases} Bed(sf_i), & \text{für } i = 1 \\ \neg\left(Bed\left(sf_1\right) \wedge ... \wedge Bed\left(sf_{i-1}\right)\right) \wedge Bed\left(sf_i\right), & \text{für } 1 < i < \text{n} \\ \neg\left(Bed\left(sf_1\right) \wedge ... \wedge Bed\left(sf_{n-1}\right)\right), & \text{für } i = \text{n} \end{cases}$$

Abbildung 3.21: Funktion zur Überarbeitung der Vorbedingungen.

Im Fall des Prozesses aus Abbildung 3.20 muss nur eine Vorbedingung überarbeitet werden. Aus dem *default SequenceFlow* wird ein *conditional SequenceFlow* mit der

Bedingung $not(A)$. Der überarbeitete Prozess ist in Abbildung 3.22 dargestellt.

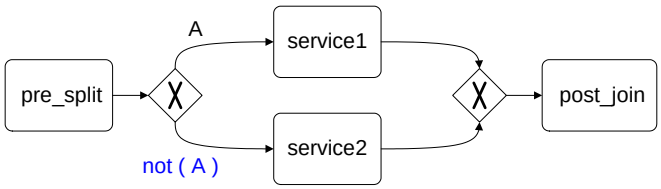


Abbildung 3.22: Überarbeitung der Vorbedingungen.

Vorbedingungen sammeln Nachdem die Vorbedingungen korrigiert worden sind werden sie nun für alle Activities der Komponente gesammelt. Dabei werden für jede Activity die Pfade aller eingehender Sequence Flows rekursiv rückwärts verfolgt, bis sie wieder eine Activity erreichen. Während dieser Wanderung rückwärts durch den Graphen werden die dabei gefundenen Vorbedingungen der passierten Sequence Flows gesammelt.

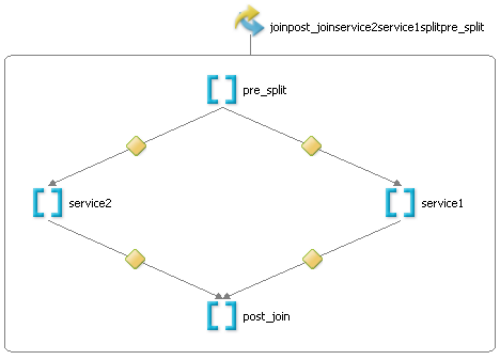
Das Ergebnis hängt dabei vom Typ des Elementes ab von dem der jeweilige Sequence Flow ausgeht. Ist der Vorgänger eine andere *Activity*, besteht die Vorbedingung nur daraus, daß dieses vor dem Ziel Element ausgelöst wird. Im Fall eines Gateways wird die Methode rekursiv mit diesem aufgerufen. Handelt es sich um einen mergenden Gateway wird die Menge der Vorbedingungen des Gateways zurückgeliefert um die Semantik des Gateways abzubilden, daß er aktiviert wird sobald einer der eingehenden Pfade aktiv ist. Ähnlich wird vorgegangen wenn es sich um einen join Gateway handelt. In diesem Fall wird das Kartesische Produkt der Vorbedingungen gebildet um sicherzustellen, daß der Gateway erst aktiv wird wenn alle eingehenden Pfade aktiv sind. Handelt es sich um einen split Gateway wird die Vorbedingung des Gateways zurückgeliefert.

Element	Vorbedingung	`transitionCondition`	`joinCondition`
pre_Split	-	-	-
service1	A	A	`true($pre_split_LINK_service1)`
service2	not (A)	not(A)	`true($pre_split_LINK_service2)`
post_Join	-	-	`true($service1_LINK_post_join) or true($service2_LINK_post_join)`

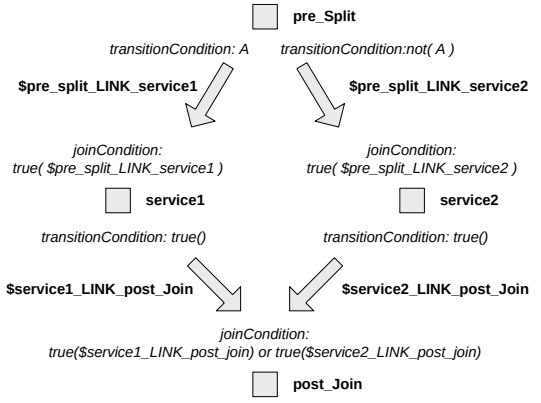
Abbildung 3.23: Die Vorbedingungen für die Elemente des Prozesses aus 3.22.

Gleichzeitig wird für jeden Pfad die Ausgangs-Aktivität gespeichert, da sie für die Erzeugung der `joinCondition` des `flow` notwendig ist. Tabelle 3.23 zeigt die in Prozess 3.22 gefundenen Vorbedingungen. Die `transitionCondition` und `joinCondition` werden anschliessend aus der Vorbedingung erzeugt. Die `transitionCondition` wird im Ausgangs Element des Links gespeichert[2]. Die gesammelten Informationen werden für die Transformation als Annotation in den Elementen gespeichert. Anschliessend wird der Teilgraph gekapselt, also in einen CAB verschoben, und dieser in den Prozes gehängt.

[2]Siehe Listing 3.1 Zeile 16 und Zeile 26. Die `transitionCondition` wird als Tag im Ausgangs Element, die `joinCondition` als Tag im Ziel Element gespeichert.



(a) Darstellung in ActiveBpel[1]



(b) Schematische Darstellung der wichtigsten Tags im BPEL Prozess.

Abbildung 3.24: Darstellung des transformierten Prozesses

BPEL erzeugen Bei der eigentlichen Transformation werden alle Aktivitäten des CABs in einem Flow gekapselt und die gesammelten Vorbedingungen und Ausgangs-Aktivitäten dazu verwendet die entsprechenden `link` Tags zu erzeugen. Den aus dem Prozess aus Abbildung 3.22 erzeugten BPEL Prozess zeigt Listing 3.1 während Abbildung 3.24(a) denselben BPEL Prozess in der graphischen Darstellung von ActiveBpel[1] zeigt. Abbildung 3.24(b) zeigt den Aufbau der `link` Tags aus Listing 3.1.

Listing 3.1: Der Prozess aus Abbildung 3.19 nach der Control Link Transformation

```
<?xml version="1.0" encoding="UTF-8"?>
<bpel:process ... >
   <bpel:import ... />
   <bpel:variables ... />
   <bpel:flow name="joinpost_joinservice2service1splitpre_split">
      <bpel:links>
         <bpel:link name="pre_split_LINK_service1"/>
         <bpel:link name="pre_split_LINK_service2"/>
         <bpel:link name="service1_LINK_post_join"/>
         <bpel:link name="service2_LINK_post_join"/>
      </bpel:links>
<!-- pre_split -->
      <bpel:empty name="pre_split">
         <bpel:sources>
            <bpel:source linkName="pre_split_LINK_service2">
               <bpel:transitionCondition>true()  and not( A )</bpel:
                   transitionCondition>
            </bpel:source>
            <bpel:source linkName="pre_split_LINK_service1">
               <bpel:transitionCondition>A</bpel:transitionCondition>
            </bpel:source>
         </bpel:sources>
      </bpel:empty>
<!-- service1 -->
      <bpel:empty name="service1">
         <bpel:targets>
            <bpel:joinCondition>true($pre_split_LINK_service1)</bpel:
                joinCondition>
            <bpel:target linkName="pre_split_LINK_service1"/>
         </bpel:targets>
         <bpel:sources>
            <bpel:source linkName="service1_LINK_post_join">
               <bpel:transitionCondition>true()</bpel:transitionCondition>
            </bpel:source>
         </bpel:sources>
      </bpel:empty>
<!-- service2 -->
      <bpel:empty name="service2">
         <bpel:targets>
            <bpel:joinCondition>true($pre_split_LINK_service2)</bpel:
                joinCondition>
            <bpel:target linkName="pre_split_LINK_service2"/>
         </bpel:targets>
         <bpel:sources>
            <bpel:source linkName="service2_LINK_post_join">
               <bpel:transitionCondition>true()</bpel:transitionCondition>
            </bpel:source>
         </bpel:sources>
      </bpel:empty>
<!-- post_join -->
      <bpel:empty name="post_join">
         <bpel:targets>
            <bpel:joinCondition>( true($service1_LINK_post_join) ) or ( true(
                $service2_LINK_post_join) )</bpel:joinCondition>
            <bpel:target linkName="service1_LINK_post_join"/>
            <bpel:target linkName="service2_LINK_post_join"/>
         </bpel:targets>
      </bpel:empty>
   </bpel:flow>
</bpel:process>
```

Zyklische Graphen - eventHandler Transformation

Mit dem im vorigen Abschnitt vorgestellten Ansatz scheitert man jedoch bei der Transformation nichtwohlstrukturierter Modelle, die Schleifen enthalten. Ouyang [29] hat für diesen Zweck eine Methode entwickelt und sie *event-action rule-based translation approach* genannt. Nach dem wichtigsten verwendeten BPEL Element kann sie auch eventHandler Transformation genannt werden. Mit Hilfe dieser Methode ist es möglich, jedes BPD in ein BPEL zu transformieren. Ein Nachteil dieses Ansatzes ist jedoch, daß der erzeugte BPEL Prozess sehr unübersichtlich und schwer nachvollziehbar ist.

Ansatz Der Grundgedanke des Ansatzes ist dem im vorigen Abschnitt beschriebenen im Grunde nicht unähnlich. In diesem Fall werden aber keine Vorbedingungen gesammelt sondern nur die Vorgänger. Der Gedanke ist, daß jedes Element innerhalb eines BPEL `eventHandler` gekapselt wird. Nach der Ausführung eines Element löst dieses einen „Event“ aus, der die `eventHandler` der nachfolgenden Elemente aktiviert. Ein „Event“ ist in diesem Kontext nur eine Nachricht, die die Fertigstellung eines Element signalisiert, und nicht ein BPMN *Event*.

Implementierung Diese Methode wird nun modifiziert, da Ouyang ein SPM als Ausgangsmodell verwendet, um alle Teilgraphen zu transformieren, die sonst nicht mehr transformierbar wären.

Wird ein ganzes Modell mit Hilfe dieses Ansatzes transformiert, besteht der eigentlich Prozess, wie in Abbildung 3.25 skizziert, nur aus dem Startelement des Prozesses und dem darauffolgenden Elementen die den „Event“ auslösen. Alle übrigen Elemente des Prozesses befinden sich in einem `eventHandler`.

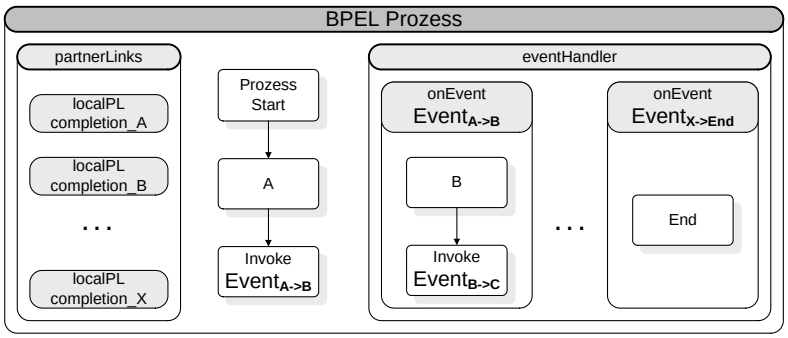
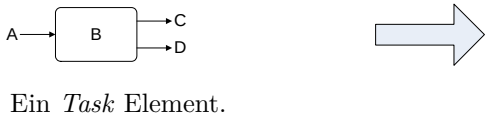


Abbildung 3.25: Die Struktur eines BPEL Prozesses der mit Hilfe der eventHandler Transformation erzeugt wurde

Abbildung 3.26 stellt die Transformation einer Task in einen BPEL `eventHandler` dar. $Event_{A \rightarrow B}$ repräsentiert dabei den *SequenceFlow* von Element `A` nach Element `B`. Dieser ”Event“ signalisiert also, daß Element `A` ausgeführt wurde und die Ausführung von Element `B` beginnen kann.

Generell kann so jedes Element in einem `eventHandler` gekapselt werden der jeweils ausgelöst wird wenn ein Element davor den entsprechenden „Event“ auslöst. Jedes



(a) Ein *Task* Element.

```
<onEvent Event_{A→B} >
  <scope>
    <sequence>
      Transformation von B
      <flow>
        <invoke Event_{B→C} />
        <invoke Event_{B→D} />
      </flow>
    </sequence>
  </scope>
</onEvent>
```

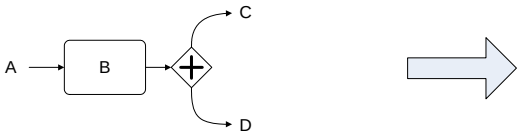
(b) BPEL Struktur.

Abbildung 3.26: Ein *Task* Element mit mehreren ausgehenden *SequenceFlow* Elementen und seine Entsprechung in BPEL

gekapselte Element löst alle seine Nachfolger aus, sofern eventuell vorhandene Bedingungen erfüllt sind. Bei einem *Task*, *Event* oder *Subprocess* Element führt der `eventHandler` zuerst die enthaltene BPEL Struktur aus, in diesem Fall mit *Transformation X* bezeichnet, und löst anschliessend alle „Events“, welche die Ausführung des Elements an seine Nachfolger mitteilen, aus.

Ein split *Gateway* ruft die nachfolgenden Elemente in der Reihenfolge auf in der sie im Gateway definiert sind (siehe Abbildung 3.27). Ein merge *Gateway* wird auf mehrere `eventHandler` so abgebildet, daß jeder von diesen Aufgrund eines ”Events“ von einem der Vorgänger ausgelöst werden kann.

Bei der Abbildung eines join *ParallelGateway* muss ein wenig anderes vorgegangen werden, da in BPEL nur ein einziger Event einen `eventHandler` auslösen kann. Wie in Abbildung 3.29 zu sehen kann er jedoch dargestellt werden, indem der erste eingehende *SequenceFlow* ($Event_{A \to C}$) des Gateway als auslösender Event verwendet wird. Der `eventHandler` kann also durch $Event_{A \to C}$ ausgelöst werden aber C wird erst ausgeführt, wenn alle übrigen, in diesem Fall nur $Event_{B \to C}$, auch empfangen worden sind.



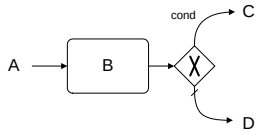
(a) Ein split *ParallelGateway* Element.

```
<onEvent Event_{A→B}>
  <scope>
    <sequence>
      Transformation B
      <flow>
        <invoke Event_{B→C} />
        <invoke Event_{B→D} />
      </flow>
    </sequence>
  </scope>
</onEvent>
```

(b) BPEL Struktur.

Abbildung 3.27: Ein *ParallelGateway* split und seine Entsprechung in BPEL

Einsatz Diese Methode kann in zwei unterschiedlichen Fällen zum Einsatz kommen. Im ersten Fall kann sie zur Transformation eines Teilabschnittes eines Prozess verwendet werden. Dann wird dieser in einem CAB gekapselt und die Transformation versucht den

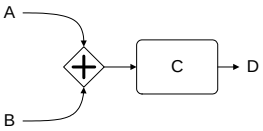


(a) Ein split *DataXor* Element.

```
<onEvent Event_{A→B}>
  <scope>
    <sequence>
      Transformation B
      <flow>
        <if> //cond
          <invoke Event_{B→C} />
        </if>
        <if> //not(cond)
          <invoke Event_{B→D} />
        </if>
      </flow>
    </sequence>
  </scope>
</onEvent>
```

(b) BPEL Struktur.

Abbildung 3.28: Ein *DataXor* split gateway und seine Entsprechung in BPEL

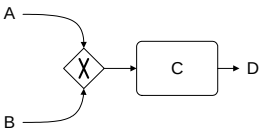


(a) Ein join *ParallelGateway* Element.

```
<onEvent Event_{A→C}>
  <scope>
    <sequence>
      <receive Event_{B→C} />
      Transformation C
      <invoke Event_{C→D} />
    </sequence>
  </scope>
</onEvent>
```

(b) BPEL Struktur.

Abbildung 3.29: Ein join *ParallelGateway* und seine Entsprechung in BPEL



(a) Ein merge *DataXor* Element.

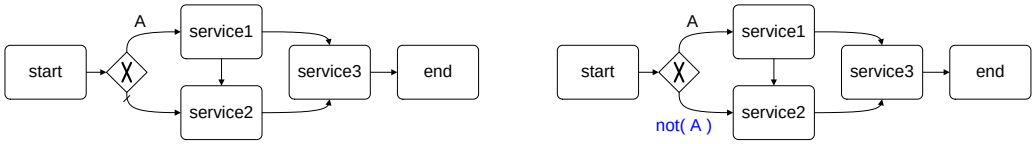
```
<onEvent Event_{A→C} >
  <scope>
    <sequence>
      Transformation C
      <invoke Event_{C→D} />
    </sequence>
  </scope>
</onEvent>

<onEvent Event_{B→C} >
  <scope>
    <sequence>
      Transformation C
      <invoke Event_{C→D} />
    </sequence>
  </scope>
</onEvent>
```

(b) BPEL Struktur.

Abbildung 3.30: Ein merge *DataXor* gateway und seine Entsprechung in BPEL.

übrigen Teil des Prozesses weiter zu zerlegen. Der zweite Fall tritt ein, wenn im gesamten Prozess keine wohlstrukturierten Strukturen zu finden sind und auch die Verwendung der Control Link Transformation aufgrund von Schleifen nicht möglich ist. In diesem Fall wird der gesamte Prozess innerhalb des CAB gekapselt. Damit ist die Transformation abgeschlossen und der Prozess ist bereit für die Generierung von BPEL.

Abbildung 3.31(a) zeigt einen einfachen Prozess, anhand dessen im Folgenden die eventHandler Transformation eines ganzen Prozesses und nicht nur einzelner Aktivitäten



(a) Originaler Beispielprozess

(b) Prozess nach Überarbeitung der Vorbedingungen

Abbildung 3.31: Beispielprozess für die eventHandler Transformation.

vorgestellt wird. Wäre nicht der *SequenceFlow* von ***service1*** nach ***service2***, würde es sich um einen wohlstrukturierten Prozess handeln. In einem ersten Schritt werden die Vorbdedingung des Prozesses überarbeitet. Nach diesem Schritt, der in Abbildung 3.31(b) dargestellt ist, werden für alle *Activity* Elemente ihre Vorgänger sowie vorhandene Vorbedingung gesammelt. Diese sind in Tabbelle 3.32 für den Prozess aufgelistet. Der Prozess besitzt also sieben mögliche Vorgänger - Nachfolger Paare. Diese Information wird als Annotation in den Elementen gespeichert.

Element	Vorgänger	Bedingung
start	-	-
service1	*start*	A
service2	*start* *service1*	not(A) -
service3	*service1* *service2*	- -
end	*service3*	-

Abbildung 3.32: Die Vorbedingungen für die Elemente des Prozesses aus 3.31(b).

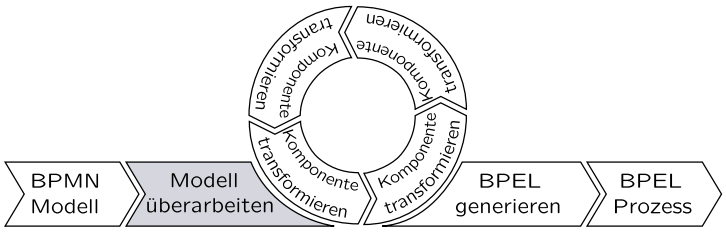
BPEL erzeugen Die Generierung von BPEL ist nur insofern aufwendig, weil für jedes Element des CABs eine WSDL Datei sowie eine Datei mit den `partnerLink` Definitionen generiert werden muss. Ansonsten werden die Elemente des CABs der Reihe nach in einem `eventHandler` generiert. Dabei werden die einzelnen `onEvent` Blöcke analog zu den Abbildungen 3.27-3.30 erzeugt.

Im Falle des Beispielprozesses werden aus den in Tabelle 3.32 dargestellten Kombinationen von Vorgänger und Nachfolger im `eventHandler` des BPEL Prozesses nun sieben `onEvent` Blöcke erzeugt:

Listing 3.2: Der aus 3.31 generierte (vereinfachte) BPEL Prozess.

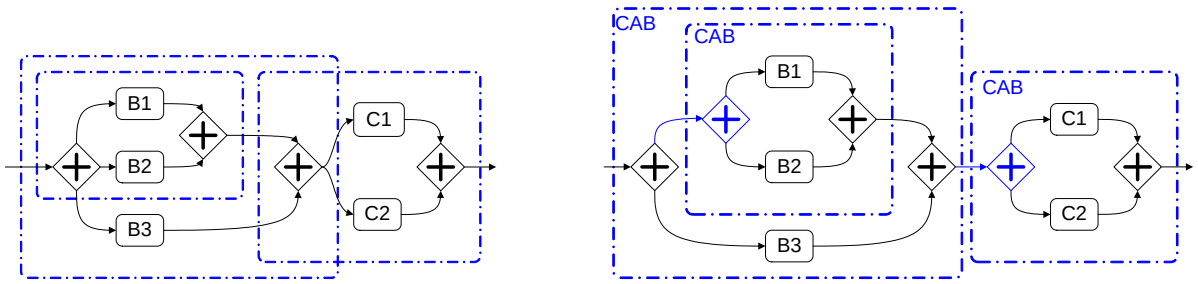
```
    <bpel:scope>
2       <bpel:eventHandlers>
          <bpel:onEvent Completion_service3_TO_end >
4           <bpel:scope>
                <bpel:empty name="end"/>
6           </bpel:scope>
          </bpel:onEvent>
8         <bpel:onEvent Completion_service1_TO_service3 >
            <bpel:scope>
10            <bpel:sequence>
                <bpel:empty name="service3"/>
12              <bpel:invoke completed_service3_TO_end />
              </bpel:sequence>
14          </bpel:scope>
          </bpel:onEvent>
16        <bpel:onEvent Completion_service2_TO_service3 >
            <bpel:scope>
18            <bpel:sequence>
                <bpel:empty name="service3_2"/>
20              <bpel:invoke completed_service3_TO_end />
              </bpel:sequence>
22          </bpel:scope>
          </bpel:onEvent>
24        <bpel:onEvent Completion_start_TO_service2 >
            <bpel:scope>
26            <bpel:sequence>
                <bpel:empty name="service2"/>
28              <bpel:invoke completed_service2_TO_service3 />
              </bpel:sequence>
30          </bpel:scope>
          </bpel:onEvent>
32        <bpel:onEvent Completion_service1_TO_service2 >
            <bpel:scope>
34            <bpel:sequence>
                <bpel:empty name="service2_2"/>
36              <bpel:invoke completed_service2_TO_service3 />
              </bpel:sequence>
38          </bpel:scope>
          </bpel:onEvent>
40        <bpel:onEvent Completion_start_TO_service1 >
            <bpel:scope>
42            <bpel:sequence>
                <bpel:empty name="service1"/>
44              <bpel:flow>
                  <bpel:invoke completed_service1_TO_service2 />
46                <bpel:invoke completed_service1_TO_service3 />
                </bpel:flow>
48            </bpel:sequence>
            </bpel:scope>
50        </bpel:onEvent>
        </bpel:eventHandlers>
52      <bpel:sequence>
          <bpel:empty name="start"/>
54        <bpel:flow>
            <bpel:if><bpel:condition>not( A )</bpel:condition>
56            <bpel:invoke completed_start_TO_service2 />
            </bpel:if>
58          <bpel:if><bpel:condition>A</bpel:condition>
              <bpel:invoke completed_start_TO_service1 />
60          </bpel:if>
          </bpel:flow>
62      </bpel:sequence>
    </bpel:scope>
64  </bpel:process>
```

3.3 Prozess überarbeiten



Ein wichtiger Teil des gesamten Transformationsansatzes besteht in der Überarbeitung des Prozesses vor dessen schrittweiser Zerlegung und Kapselung in CABs. Dieser Schritt findet nur einmal statt und geschieht, um bestimmte Konstruktionen und Strukturen die eine optimale Transformation verhindern, zu überarbeiten. Dabei handelt es sich nicht um fehlerhafte Strukturen, sondern nur um in BPMN erlaubte Arten bestimmte Sachverhalte zu modellieren, die jedoch dazu führen, daß eigentlich wohl strukturierte Teile des Modells nicht als solche erkannt werden.

Folglich wäre die Transformation auch ohne diese Veränderungen des Modelles erfolgreich, würde aber in diesen Fällen unnötigerweise auf die Transformationsansätze für nichtwohlstrukturierte CABs zurückgreifen. So könnte der Prozess in Abbildung 3.33(a) nicht ohne Verwendung des Control Link Verfahren transformiert werden, obwohl er offensichtlich wohlstrukturiert und nicht sonderlich komplex ist. Es überschneiden sich in diesem Fall mehrere wohlstrukturierte CABs. Er kann durch das Hinzufügen von jeweils zwei Gateways und Sequence Flows in eine wohlstrukturierte Form gebracht werden, wie in Abbildung 3.33(b) dargestellt.



(a) Vor der Überarbeitung

(b) Nach der Überarbeitung

Abbildung 3.33: Ein wohlstrukturierter Prozess der aber unmodifiziert (a) nicht mit Hilfe wohlstrukturierter CABs alleine transformiert werden kann.

Im folgenden werden die wichtigsten Verfahren vorgestellt, die bei diesen und ähnlichen Prozessen eine Transformation nur mit Hilfe wohlstrukturierter CABs erlauben. Ähnliche Veränderungen von Prozessen werden auch in [15] vorgestellt.

3.3.1 Elemente aufspalten

Eine einfache Modifikation die unter den folgenden Umständen notwendig ist.

Elemente im Prozess aufspalten

Wenn ein Element sowohl mehrere eingehende wie auch ausgehende Pfade besitzt, ist es nicht möglich diesem Element, in Folge daher auch seinen Vorgängern und Nachfolgern, ein CAB zuzuordnen. Somit würde bei der Transformation dieses Abschnittes des Prozesses die Control Link oder eventHandler Transformation zum Einsatz kommen. In vielen Fällen kann dies jedoch durch eine "Auftrennung" des Elementes noch verhindert werden und ermöglicht so, daß der Struktur doch noch ein CAB zugeordnet werden kann. Dieser Fall ist in Abbildung 3.34 dargestellt. Im ersten Fall werden alle eingehenden *SequenceFlow* von `A` auf ein neues *None* `Pre_A` gerichtet und dieses mit Hilfe eines neuen Sequence Flows mit `A` verbunden. Analog wird bei einem Gateway vorgegangen. In allen Fällen entspricht der Typ des neu eingefügten Elements dem des aufgespaltenen Elementes.

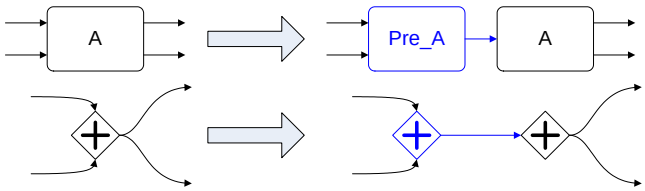
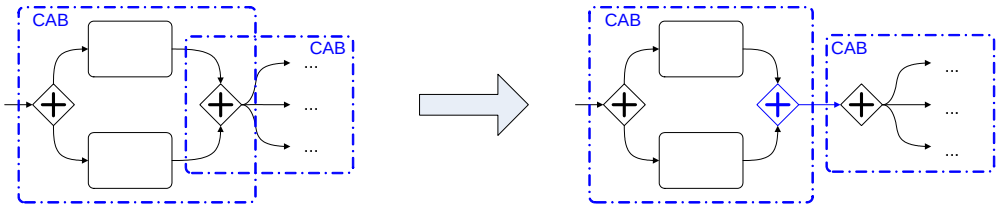


Abbildung 3.34: Aufspalten einer *Task* und eines *ParallelGateway*

Abbildung3.35(a)-(b) zeigt einen Abschnitt eines Prozesses bei dem ein *GatewayGatewayFlow* CAB sich mit einem nachfolgenden überschneidet. Das Endelement der ersten Struktur, der *ParallelGateway*, ist gleichzeitig Startelement der nachfolgenden Struktur. Damit ist eine CAB Bedingung verletzt. Nach der Aufspaltung wird die Struktur korrekt erkannt ohne daß sich die Semantik des Prozesses verändert hätte.



(a) Prozess vor der Überarbeitung (b) Prozess nach der Überarbeitung

Abbildung 3.35: Elemente aufspalten - Beispiel für das aufspalten von Elementen

Endelemente aufspalten

Diese Methode kommt zum Einsatz wenn eine Aktivität mehrere eingehende aber keine ausgehenden *SequenceFlow* Elemente besitzt. Diese Darstellung eines Endelementes ist

in BPMN durchaus üblich, kann jedoch so nicht in einen BPEL Prozess übernommen werden. In BPEL wird sie durch eine Duplizierung der Aktivität für jeden eingehenden *SequenceFlow* abgebildet. Durch Vorziehen dieses Schrittes vor die eigentliche Transformation wird diese vereinfacht, da dieser Sonderfall bei der Identifizierung von Strukturen im Modell nicht mehr berücksichtigt werden muss.

Startelemente aufspalten

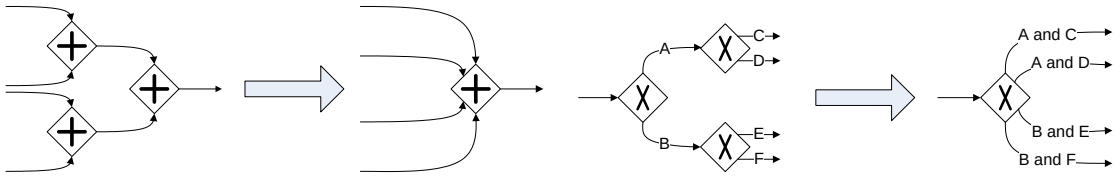
Der Begriff ist ein wenig irreführend, da nicht alle Startelemente mit mehreren ausgehenden *SequenceFlow* aufgespalten werden, sondern nur solche in Prozessen mit mehreren Startelementen. Ansonsten verläuft dieser Fall aber analog zu den übrigen beschriebenen.

Schleifen bearbeiten

Hierbei handelt es sich um einen Spezialfall der obigen Szenarien. Eine Konstruktion, wobei entweder das Startelement und/oder das Endelement einer Schleife sowohl mehrere eingehende wie auch ausgehende Verbindungen hat.

3.3.2 Gateways zusammenfassen

Eine Modifkation bei der aneinanderhängende Gateways desselben Typs zusammengefasst werden. Es kann grundsätzlich zwischen zwei Fällen unterschieden werden, je nachdem ob es sich um joinende oder splittende Gateways handelt, die sich jedoch nur geringfügig unterscheiden. Im joinenden Fall kann diese Methode angewandt werden wenn zwei oder mehr Gateways mit mehreren eingehenden Sequence Flows jeweils nur einen ausgehenden Sequence Flow zu demselben Gateway besitzen. Zusätzlich darf der nachfolgende Gateway keine anderen eingehenden Sequence Flows besitzen. In diesen Fällen wäre die Anwendung der Methode kontraproduktiv. Bei allen beteiligten Gateways muss es sich um Gateways desselben Typs handeln. In Abbildung 3.36 sind die beiden Fälle abgebildet.

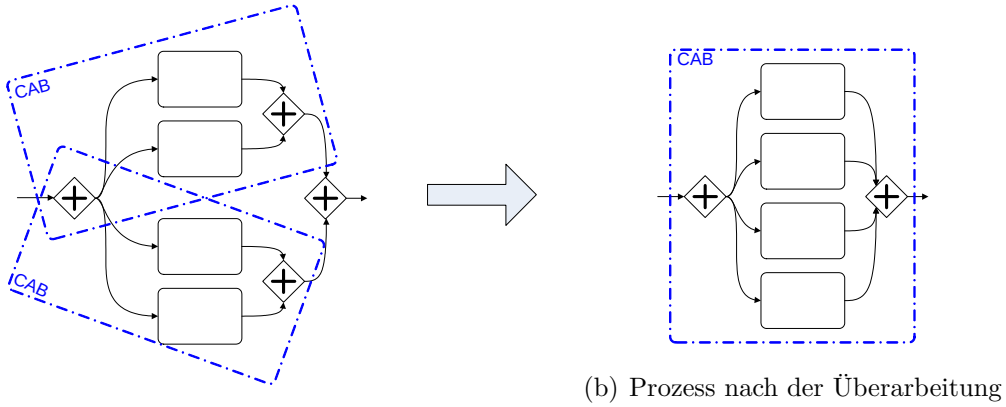


(a) joinende Gateways zusammenfassen. (b) splittende Gateways zusammenfassen.

Abbildung 3.36: Zusammenfassen von (a)Joins, (b)Splits

Abbildung 3.37 demonstriert den praktischen Einsatz anhand eines einfachen Falles mit joinenden Gateways. Der Transformationsalgorithmus erkennt im unmodifizierten Fall 3.37(a) zwei Strukturen, die jedoch bei einer näheren Untersuchung verworfen werden. So darf der Start-Gateway eines *GatewayGatewayFlow* CABs keine ausgehenden Sequence Flows besitzen, die nicht in einem der Elemente des CABs münden. Diese Bedingung

wird jedoch offensichtlich verletzt. Nach der Überarbeitung, die Abbildung 3.37(b) zeigt, kann der Prozess alleine mit Hilfe von wohlstrukturierten CABs transformiert werden.



(a) Prozess vor der Überarbeitung

(b) Prozess nach der Überarbeitung

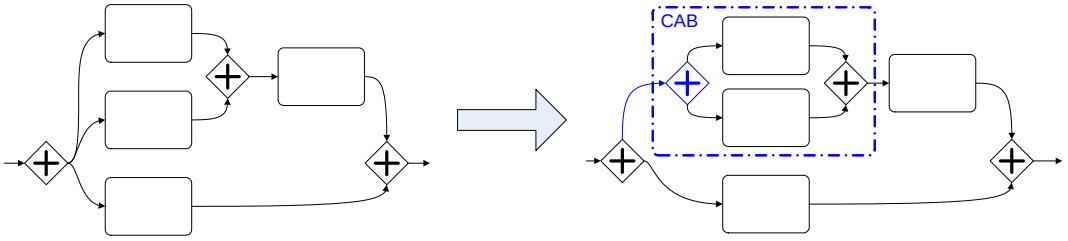
Abbildung 3.37: Beispiel für das Zusammenfassen von Gateways.

Die Überarbeitung verläuft analog, wenn es sich um splittende Gateways handelt. Dann müssen jedoch auch die Vorbedingungen der Sequence Flows überarbeitet werden. Ohne diese Modifikation könnte kein CAB diesem Teilgraphen zugeordnet werden und es käme die Control Link Transformation zum Einsatz.

3.3.3 Gateways einfügen und ausbalancieren

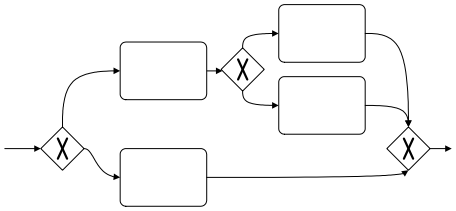
Diese Methode ist das Gegenstück zu der in 3.3.2 beschriebenen Methode Gateways zusammenzufassen. Sie wird angewandt wenn es notwendig ist, einer eigentlich wohlstrukturierte Komponente ein Element hinzuzufügen, damit sie als CAB erkannt wird. Es wird dabei, je nach vorhandener Struktur und den jeweiligen splittenden und joinenden Elementen, ein entsprechendes Element eingefügt. In einem Fall wird ein neuer split Gateway vor einem joinenden Gateway eingefügt. Im konträren Fall wird ein joinender Gateway nach einem split Gateway eingefügt. In Abbildung 3.38 sind die beiden möglichen Fälle abgebildet.

Die Methode um einen Split vor einem Join einzufügen 3.38(a)-(b) verfolgt für alle Splits des BPD alle Pfade bis zum ersten Join auf dem jeweiligen Pfad und sammelt diese. Hat ein Split mehrere joinende Gateways als Nachfolger und kommen gleichzeitig **alle** seine ausgehenden Pfade an einem dieser Joins wieder zusammen, so wird er überarbeitet. Dabei werden alle nachfolgenden Pfade überprüft und die Pfade, die vor dem finalen join in einen join münden, mit Hilfe eines neuen splitenden Gateway zusammengefasst. Analog werden alle joinenden Gateways überprüft. Dieser Fall ist in in Abbildung 3.38(c)-(d) dargestellt.

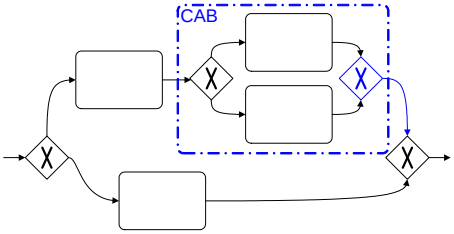


(a) Prozess vor der Überarbeitung mit mehreren join Gateways

(b) Einfügen eines split Gateway vor einem join Gateway.



(c) Prozess vor der Überarbeitung mit mehreren split Gateways

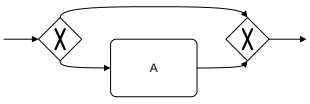


(d) Einfügen eines join Gateway nach einem split Gateway.

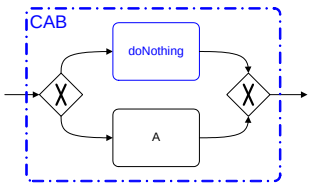
Abbildung 3.38: Gateways einzufügen: (a)-(b) vor einem join sowie (c)-(d) nach einem split

3.3.4 Leere Pfade verhindern

Damit eine Struktur als CAB erkannt wird, muss sie genau ein Startelement sowie ein Endelement besitzen. In Fällen wo ein oder mehrere Sequence Flows direkt ein Startelement mit einem Endelement eines potentiellen CABs verbinden, muss dieser CAB überarbeitet werden. So hängt diese Methode in jeden "leeren Pfad" ein *None* Element ein, wie in Abbildung 3.39 dargestellt. Damit werden auch diese Strukturen sofort als CAB erkannt ohne bei der Überprüfung der Vorbedingungen eines CABs diesen Fall berücksichtigen zu müssen.



(a) Prozess vor der Überarbeitung



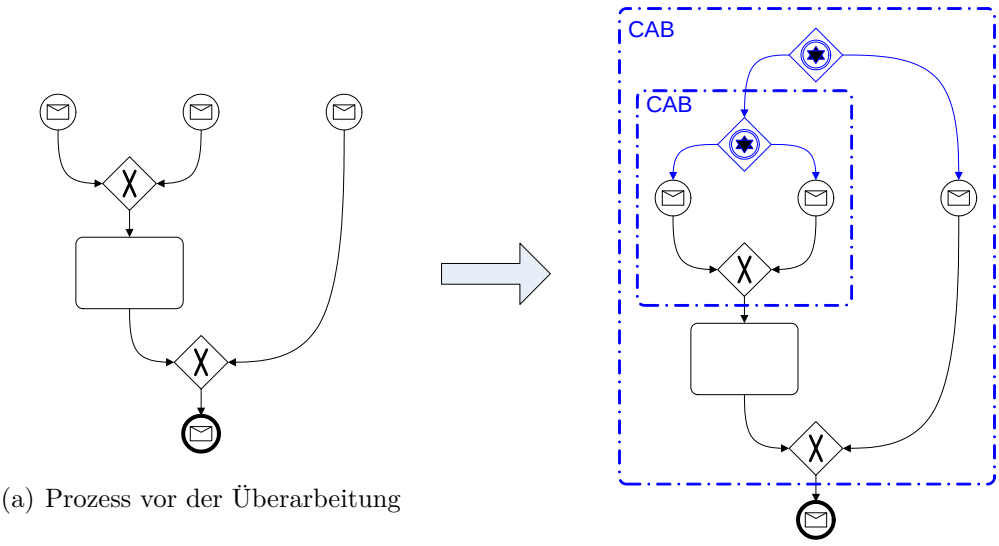
(b) Prozess nach der Überarbeitung

Abbildung 3.39: Leere Pfade im BPD verhindern.

3.3.5 Prozesse mit multiplen Startelementen überarbeiten

Bei Prozessen mit mehreren Startelementen ist es für die weiteren Transformationsschritte notwendig einen einzelnen *EventXor* vor den bisherigen Startelementen in den Prozess zu hängen.

Abbildung 3.40(a) stellt einen einfachen Prozess mit mehreren Startelementen dar, in diesem Fall handelt es sich um *Receive* Elemente. In einem ersten Schritt wird nun ein neuer *EventXor* in den Prozess gehängt. Der so erhaltene Prozess ist aber noch nicht für die Transformation bereit und muss noch ausbalanciert werden. Im nächsten Schritt kommt die in Abschnitt 3.3.3 beschriebene Methode zum Einsatz. Es wird nun zwischen die zwei *Receive* Elemente und dem neuen Startelement ein weiterer *EventXor* in den Prozess gehängt. Damit ist der Prozess in Abbildung 3.40(b) für die eigentliche Transformation fertig vorbereitet.



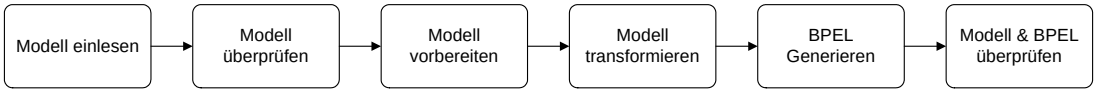
(a) Prozess vor der Überarbeitung

(b) Prozess nach der Überarbeitung

Abbildung 3.40: Start *EventXor* einfügen sowie anschliessendes Ausbalancieren

3.4 Der gesamte Ablauf

Der zu Beginn des Kapitels skizzierte Grundgedanke des Transformationsansatzes vom schrittweisen Zerlegen in Komponenten ist aber nur ein, wenn auch sehr wichtiger Schritt, in einer Kette von Schritten. Nachdem nun in den vorangegangenen Abschnitten die unterschiedlichen möglichen Komponenten samt ihrer Übersetzung sowie Überarbeitungsmethoden vorgestellt wurden, kann nun der gesamte Transformationsablauf formuliert werden.



Modell einlesen In einem ersten Schritt wird das Modell eingelesen und auf syntaktische Korrektheit überprüft.

Modell überprüfen Im nächsten Schritt wird das Modell auf seine semantische Korrektheit überprüft. So wird sichergestellt, daß sich alle Elemente an die Vorgaben der BPMN Spezifikation halten und das Modell einen läuffähigen Prozess enthält.

Modell vorbereiten In diesem Schritt wird das Modell überarbeitet und in eine für die Transformation notwendige Form gebracht.

Modell transformieren Die eigentliche Transformation findet in diesem Schritt statt. Dabei werden solange Teile des Modells durch CABs ersetzt, bis das Modell nur noch aus einem CAB besteht.

BPEL Generierung Aus dem überarbeiteten BPD wird in diesem Schritt ein BPEL Prozess erzeugt.

Test In einem letzten Schritt wird das veränderte BPD und der daraus generierte BPEL Prozess überprüft.

Die konkrete Implementierung dieses Prozesses wird im folgenden Kapitel vorgestellt.

4 Implementierung in openArchitectureWare

Nachdem in Kapitel 2 die Grundlagen und in Kapitel 3 der Transformation Ansatz beschrieben wurde, wird in diesem Kapitel die konkrete Implementierung erläutert.

- Abschnitt 4.1 beschreibt den zugrundeliegenden Ansatz der Transformation.
- Abschnitt 4.2 skizziert den Aufbau der Implementierung.
- Abschnitt 4.3 stellt die für BPMN entwickelte BPMN DSL vor.
- Abschnitt 4.4 listet die Komponenten des oAW Workflow auf.
- Abschnitt 4.5 demonstriert die Regeln zur Überprüfung der BPMN Modelle.
- Abschnitt 4.6 stellt kurz die Implementierung der Transformation vor.
- Abschnitt 4.7 zeigt, wie aus den transformierten BPMN Modellen BPEL Prozesse erzeugt werden.

4.1 Modellgetriebener Transformationsansatz

Zusätzlich zu den in Kapitel 3.1 beschriebenen Zielen des Transformationsalgorithmus, war ein Ziel die Transformation mit Hilfe von Methoden der modellgetriebenen Softwareentwicklung zu implementieren. Dazu bot sich als Framework für Modellgetriebene Softwareentwicklung openArchitectureWare an. In einem ersten Schritt vor der Implementierung war eine Entscheidung zwischen zwei unterschiedlichen Ansätzen zu treffen. Die Entscheidung für einen der beiden würde maßgeblich den Ablauf und den Aufbau des Transformationsprozesses bestimmen. In beiden Fällen würde die Basis der Transformation aus einer domänenspezifischen Sprache (DSL) zur Beschreibung von BPMN Prozessen bestehen. Diese textuelle Beschreibung wird im Zuge des Transformationsprozesses von oAW eingelesen und als Modell im weiteren Transformationsverlauf auf zwei unterschiedliche Arten verwendet.

Modell-zu-Text Transformation

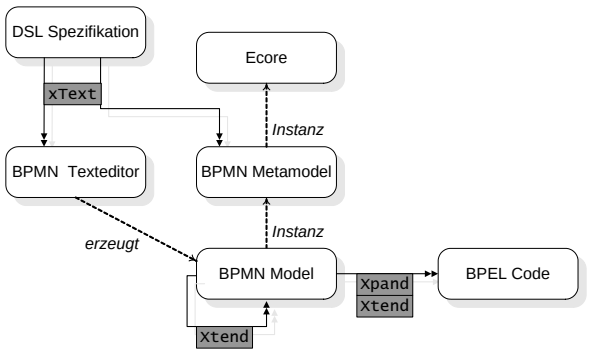


Abbildung 4.1: Modell-zu-Text Transformation.

Eine Möglichkeit wäre eine Modell-zu-Text (M2T) Transformation zu implementieren. Dabei würde aus dem BPMN Modell mit Hilfe von Templates direkt ein BPEL Prozess erzeugt werden. Abbildung 4.1 skizziert den Ablauf.

Modell-zu-Modell Transformation

Andererseits könnte die Erzeugung des BPEL Prozesses auch auf Basis einer sogenannten Modell-zu-Modell (M2M) Transformation erfolgen. In diesem Fall würde aus dem Ausgangsmodell, das auf einem BPMN Meta-Modell basiert, mit Hilfe von Transformationsfunktionen ein Modell erzeugt, das auf einem BPEL konformen Meta-Modell basiert. Anschliessend würde aus diesem Modell dann im Zuge einer M2T Transformation der BPEL Prozess generiert werden. Diesen Ansatz zeigt Abbildung 4.2.

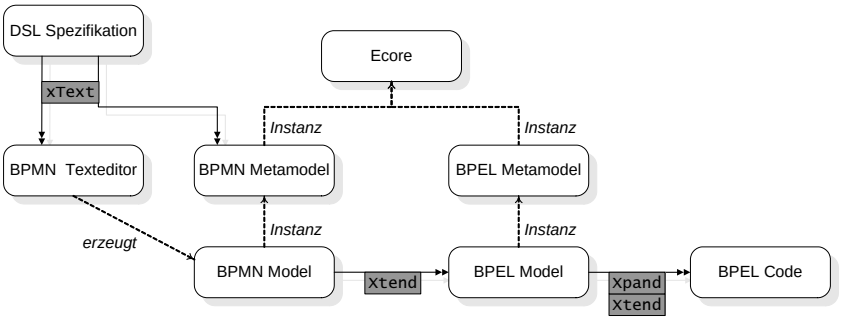


Abbildung 4.2: Modell-zu-Modell Transformation

Entscheidung

Schlußendlich fiel die Entscheidung eine Modell-zu-Text Transformation zu implementieren. Eine wesentliche Rolle spielte dabei, daß beide Spezifikationen diesselbe Domäne abdecken und dadurch viele Gemeinsamkeiten aufweisen. Stark vereinfacht könnte man die BPEL Spezifikation als eine restriktivere Form der BPMN Spezifikation sehen. Ein Metamodell für BPEL würde sich nur geringfügig von dem BPMN Metamodell unterscheiden, abgesehen von den Unterschieden, die sich durch die graph- beziehungsweise blockbasierte Darstellung der Prozesse ergeben. Ein Metamodell für BPMN kann mit überarbeiteten und stärkeren Beschränkungen auch als Metamodell für BPEL verwendet werden. Gleichzeitig darf jedoch der Aufwand für den Entwurf und Test eines Metamodells nicht unterschätzt werden. Da die Transformation nur in eine Richtung und nur von BPMN aus erfolgen sollte, würden sich auch durch die Verwendung einer M2M Transformation keine Vorteile ergeben. Mehr dazu jedoch im abschliessenden Kapitel 7

4.2 Aufbau der Projekte

Durch die Verwendung von oAW und dadurch Eclipse bei der Implementierung des Ansatzes, sind die einzelnen Komponenten über die vier Eclipse Projekte verteilt. Die Projekte enthalten unterschiedliche Komponenten des Prozesses. Die meisten der von Xtext automatisch erzeugten Artefakte befinden sich in allen vier Projekten im Verzeichnis ./src-gen.

Die "Basis" - *tuwien.bpmn2bpel.bpmn*

Dieses Projekt beinhaltet die Datei *bpmn.xtxt* mit der Xtext Definition der DSL. Mithilfe des *generate.oaw* Workflow lassen sich alle Xtext Artifakte bei einer Änderung der DSL erneut generieren.

Im Package `tuwien.bpmn2bpel.bpmn` befinden sich grundlegende Xtend Dateien sowie die Check Dateien zur Verifikation der Modelle.

Elementare Extensions um Modelle durchsuchen und bearbeiten zu können sind in einigen Xtend Dateien im Package `tuwien.bpmn2bpel.bpmn.extensions` definiert.

Das Projekt enthält auch die Xpand Templates zur Generierung von DSL Dateien aus BPMN Modellen im Package `tuwien.bpmn2bpel.bpmn.generator`. Diese werden in der eigentlichen Transformation nicht benötigt, erleichtern aber das Nachvollziehen der Transformationsschritte.

Der Editor - *tuwien.bpmn2bpel.bpmn.editor*

Enthält die Eclipse Plugins für den BPMN DSL Editor. Dieser wird zur Eingabe der Modelle verwendet. In den beiden Packages des Projektes befinden sich Xtend Dateien für die Definition von Extensions für die Unterstützung von Syntaxhervorhebung und Autovervollständigung im Editor.

Die Transformation - *tuwien.bpmn2bpel.transform*

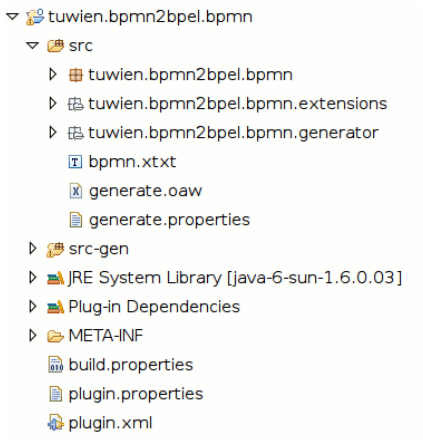
Dieses Projekt enthält alle für die Transformation und Generation von BPEL benötigten Komponenten.

Das Package `tuwien.bpmn2bpel.transform` enthält die oAW Workflow Datei sowie alle Xtend Dateien, die an der Transformation beteiligt sind. Das Package enthält auch eine Xtend Datei mit Extensions zur Verifikation der transformierten Modelle.

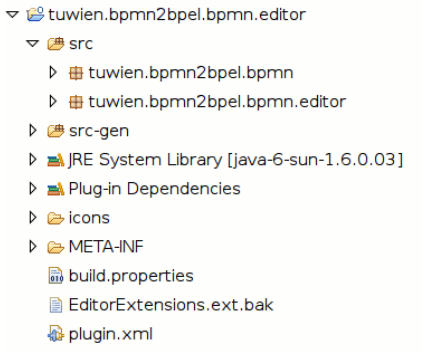
Im Package `tuwien.bpmn2bpel.generateBpel` befinden sich alle Xpand Templates die für die Erzeugung des BPEL Quelltextes benötigt werden.

Die Testfälle - *tuwien.bpmn2bpel.test*

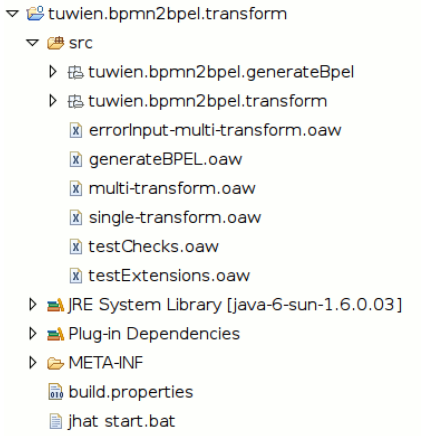
Das Projekt enthält die oAW Workflow Dateien sowie Java Klassen die für die Ausführung der Transformation zuständig sind. Diese befinden sich im Package `tuwien.bpmn2bpel.transform`. Im Verzeichnis ./input befinden sich auch alle bei der Entwicklung verwendeten Testfälle.



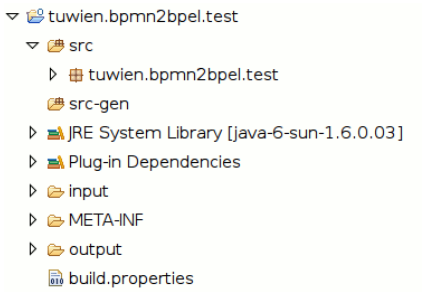
(a) Verzeichnisbaum des Projekts *tuwien.bpmn2bpel.bpmn*



(b) Verzeichnisbaum des Projekts *tuwien.bpmn2bpel.bpmn.editor*



(c) Verzeichnisbaum des Projekts *tuwien.bpmn2bpel.transform*



(d) Verzeichnisbaum des Projekts *tuwien.bpmn2bpel.test*

Abbildung 4.3: Verzeichnisbäume der vier Projekte

4.3 Domänenspezifische Sprache für BPMN

Für die Beschreibung der Prozesse wurde eine domänenspezifische Sprache mit Xtext entwickelt. Sie wurde Anhand der textuellen Beschreibung der Elemente in der BPMN Spezifikation entworfen.

4.3.1 Metamodell

Bei der Verwendung von Xtext zur Spezifikation einer DSL wird automatisch ein Metamodell aus der definierten Syntax der DSL erstellt. Da es sich bei BPMN um eine umfangreiche Sprache mit vielen Elementen handelt, ist das so entstandene Metamodell relativ groß. Abbildung 4.4 zeigt einen Ausschnitt der wichtigsten Elemente der BPMN Spezifikation. Die Hierarchien und Relationen der einzelnen Elemente entsprechen der jeweiligen Beschreibung in der BPMN Spezifikation.

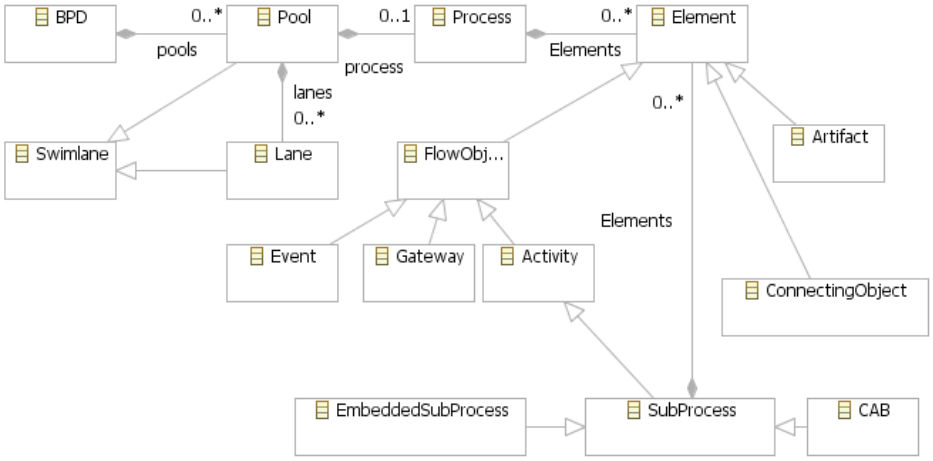


Abbildung 4.4: Ein Ausschnitt aus dem BPMN Metamodell mit den Grund-Elementen.

Abbildung 4.5 zeigt die wichtigsten Elemente für die Definition von Gateways. Dabei muss aber erwähnt werden, daß die Bedingung, welcher *SequenceFlow* eines Gateways genommen wird,im *SequenceFlow* gespeichert wird und nicht im *Gate*. Ein *Gate* dient hauptsächlich dazu *Assingment* Elemente aufzunehmen.

Die Abbildung 4.6 auf Seite 66 zeigt die Elemente die zur Definition der unterschiedlichen *Task* Typen notwendig sind, wie die *Message* und *Implementation* Elemente.

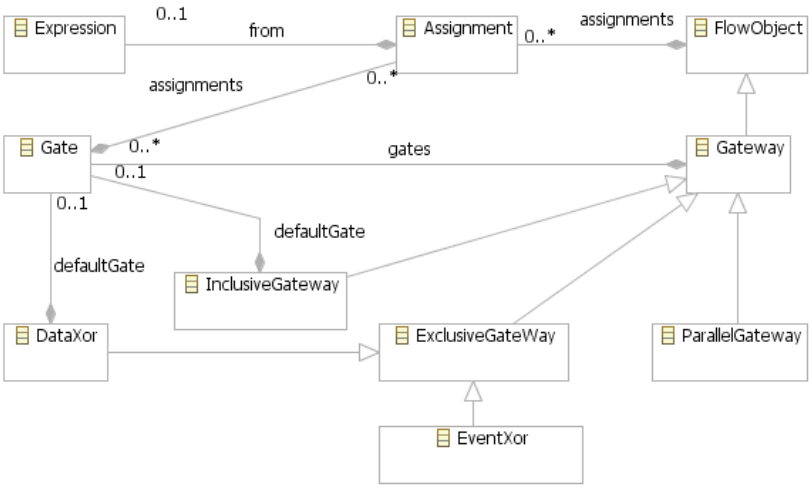


Abbildung 4.5: Ein Ausschnitt aus dem BPMN Metamodell mit unterschiedlichen Gateway Typen.

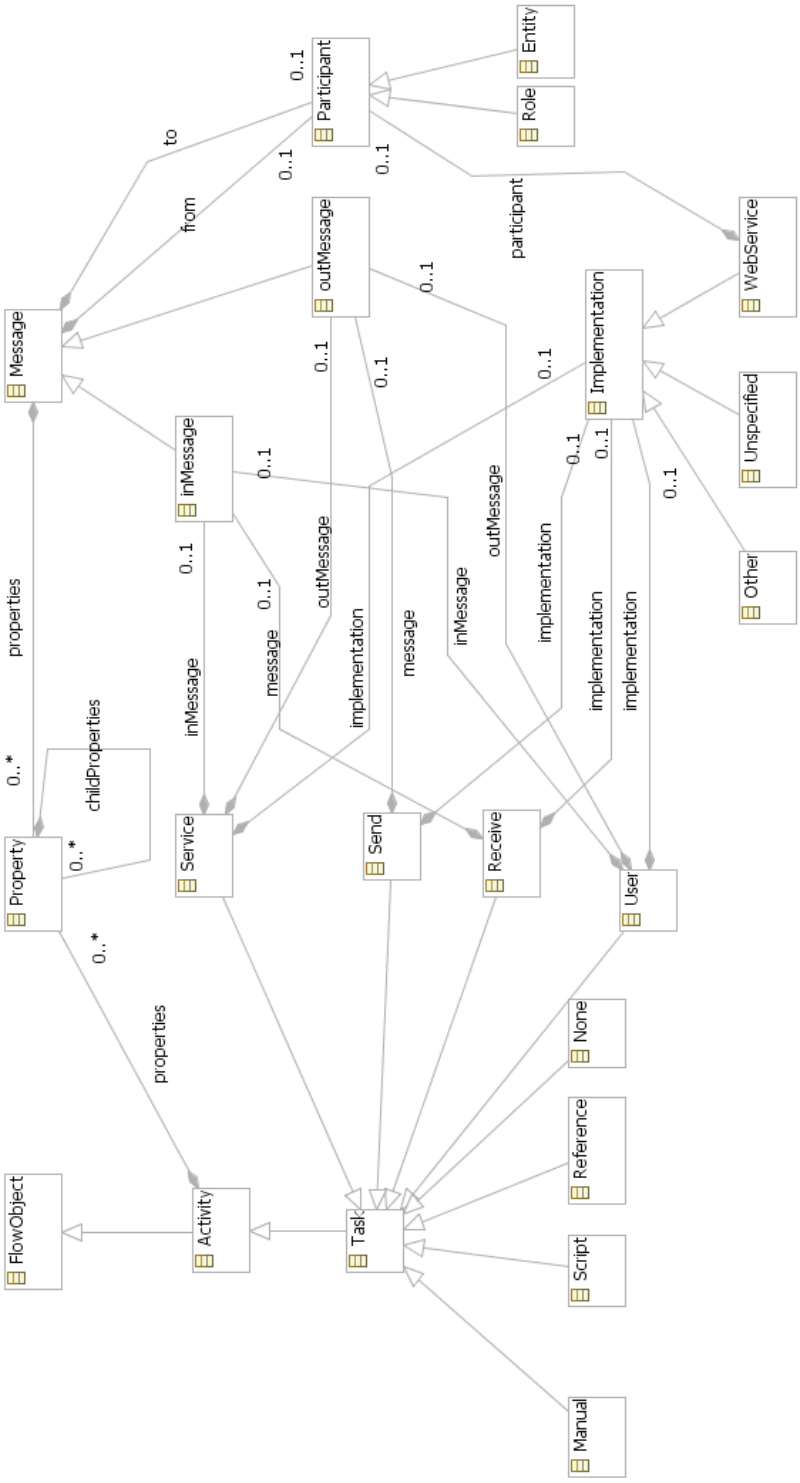


Abbildung 4.6: Ein Ausschnitt aus dem BPMN Metamodell mit unterschiedlichen Task Typen.

4.3.2 Syntax und Beispiele

Die Syntax diese Sprache orientiert sich an XML, verwendet aber weniger Sonderzeichen um die schnellere Eingabe von Modellen zu ermöglichen. So wird ein Element wie in XML üblich nicht mit `/>` sondern nur mit `>` terminiert. Ebenso werden viele nicht optionale Attribute der Elemente, sofern sie nicht angegeben werden, mit Standardwerten belegt.

Abbildung 4.7 zeigt einen einfachen Prozess sowohl in der BPMN DSL als auch graphisch.

Zeile 4 definiert einen `SequenceFlow` mit Namen ***sf1***, ***startE*** bezeichnet sein Startelement und ***t1*** sein Zielelement. Er wird passiert, sobald sein Ausgangselement aktiviert wurde, er besitzt also keine Bedingung, folglich ist er vom `ConditionType None`. Dieser muss jedoch nicht explizit angegeben werden. Die `SequenceFlow` Definitionen in Zeile 8 und 9 besitzen nicht den `ConditionType None`, daher ist ihr jeweiliger Typ explizit angegeben.

In Zeile 7 wird ein data-based Decision Gateway Namens ***split*** definiert. Er besitzt einen *conditional SequenceFlow* sowie einen *default SequenceFlow*. Folglich verweisen seine zwei Gate Definitionen auf die einzelnen SequenceFlows. Da beide Gates in diesem Fall weder Zuweisungen noch Variablen besitzen, bestehen sie nur aus dem Variablennamen der zugehörigen Sequence Flows. Dieses Beispiel zeigt, wie die BPMN DSL den Funktionsumfang der Spezifikation abdeckt, es jedoch ermöglicht auf wenigen Zeilen die Struktur eines Prozesses zu beschreiben.

```
<Diagram testProcess name "Ein einfacher Prozess"
   <Pool testPool name "Der einzige Pool"
      <None startE>
      <SequenceFlow sf1 startE t1>
      <None t1>
      <SequenceFlow sf2 t1 split>
      <DataXor split Gates sf3  DefaultGate sf4>
      <SequenceFlow sf3 split t2 ConditionType Expression "A
          ">
      <SequenceFlow sf4 split t3 ConditionType Default>
         <None t2>
         <None t3>
      <SequenceFlow sf5 t2 join>
      <SequenceFlow sf6 t3 join>
      <DataXor join DefaultGate sf7>
      <SequenceFlow sf7 join endE ConditionType Default>
      <None endE>
      >
   >
>
```

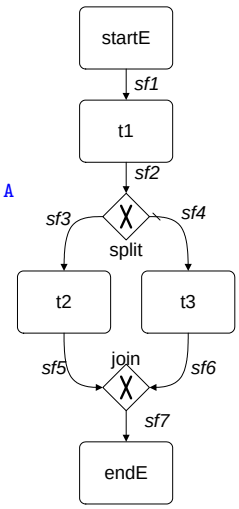


Abbildung 4.7: Ein Prozess in BPMN DSL sowie in graphischer Darstellung

Listing 4.1 zeigt ein `receive` Element, das so bereits in ein BPEL `receive` samt zugehörigen WSDL und `partnerLink` übersetzt werden kann. Die dafür notwendigen Operationen werden aus den Zeilen 15 bis 18 generiert. In Zeile 11 und 12 findet sich die Information für die notwendigen `partnerLink` Definitionen. In diesem Fall handelt es sich bei dem `receive` Element um ein Startelement wie das `Instantiate` in Zeile 14 verrät.

Listing 4.1: Ein `receive` Element.

```
...
   <Receive request
      <Assignments
         <Assignment "loginData" "expression" AssignTime End>
      >
      in
      <inMessage "login"
         <Properties
            <Property "UserName" "string">
         >
         <Role "Customer">
         <Role "this">
      >
      Instantiate
      Implementation
      <WebService
         <Role "loginService">
         "loginPort" "authorize"
      >
   >
...
```

4.4 Der Transformations-Workflow

Der Ablauf der Transformation wird von einem oAW Workflow gesteuert. Die unterschiedlichen Schritte der Transformation sind über verschiedene Komponenten des Workflow verteilt, die sequentiell abgearbeitet werden. Im Falle eines Fehlers einer Komponente werden alle nachfolgenden Komponenten übersprungen. Der konkrete oAW Workflow in Abbildung 4.8 hat im Gegensatz zum abstrakten Ablauf in Abbildung 3.4 einige Komponenten hinzugewonnen.

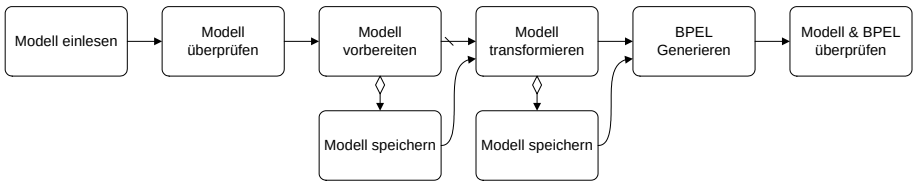


Abbildung 4.8: Der Ablauf der oAW Workflow Komponenten im Transformationsprozess

Der Aufbau der Workflow Datei *transform2.oaw* ist im folgenden beschrieben.

Einlesen des Modells

Die erste Komponente des Workflow ruft einen von Xtext automatisch generierten Workflow auf. Dieser registriert das Metamodel, liest eine BPMN Datei ein und überprüft sie auf syntaktische Korrektheit. Ist das Modell gültig wird es in der Variable `bpmnModel` gespeichert. Zu diesem Zweck verwendet der Workflow diesselbe Check Datei, die auch vom BPMN DSL Editor verwendet wird. Die Variablen *$modelPath* und *$modelFile* in Zeile 4 werden in einer Property Datei für den gesamten Workflow definiert.

Listing 4.2: Einlesen des Modells

```
<bean class="org.eclipse.mwe.emf.StandaloneSetup"/>
<bean class="tuwien.bpmn2bpel.bpmn.MetaModelRegistration"/>
<component id="bpmn-parser" class="tuwien.bpmn2bpel.bpmn.parser.ParserComponent">
      <modelFile value="${modelPath}/${modeFile}"/>
      <outputSlot value="bpmnModel"/>
</component>
<component id="bpmn-checker" class="oaw.check.CheckComponent">
   <metaModel id="mm" class="org.eclipse.m2t.type.emf.EmfRegistryMetaModel"/>
   <expression value="bpmnModel.eAllContents.union(bpmnModel)"/>
   <checkFile value="tuwien::bpmn2bpel::bpmn::GenChecks"/>
   <checkFile value="tuwien::bpmn2bpel::bpmn::Checks"/>
</component>
```

Multiple Startelemente überarbeiten

Im nächsten Schritt wird eine Xtend Erweiterung aufgerufen. Im Falle von mehreren Startelementen erzeugt diese ein *EventXor* und hängt diesen vor den bisherigen Startelementen in den Prozess. Diese kleine Korrektur ändert nichts an der Semantik des Modells.

Damit muss jedoch in späteren Schritten nicht mehr zwischen diesen beiden Fällen unterschieden werden. Das veränderte Modell wird wieder in die Variable `bpmnModel` gespeichert. Im Falle eines ungültigen Modells ist diese Änderung nicht notwendig, daher ist das Attribut `skipOnErrors` gesetzt. Die Komponente wird also übersprungen, sollten im Workflow bereits Fehler aufgetreten sein.

Listing 4.3: Multiple Startelemente überarbeiten

```
<component class="oaw.xtend.XtendComponent" skipOnErrors="true">
   <metaModel id='mm' class='org.eclipse.m2t.type.emf.EmfRegistryMetaModel'/>
   <invoke value="tuwien::bpmn2bpel::transform::preprocessEventXor::init(bpmnModel)"/>
   <outputSlot value="bpmnModel"/>
</component>
```

Modell überprüfen

Die Check Komponente überprüft das in `bpmnModel` gespeicherte Modell anhand einer Liste von Check Regeln auf semantische Korrektheit. Diese Regeln sind zum Teil relativ zeitaufwendig. Daher befinden sie sich in einer anderen Datei als der vom Editor verwendeten. Auf die Überprüfung des Modells wird in Abschnitt 4.5 noch näher eingegangen.

Listing 4.4: Modell überprüfen

```
<component class="oaw.check.CheckComponent" skipOnErrors="true">
   <metaModel  id='mm' class='org.eclipse.m2t.type.emf.EmfRegistryMetaModel'/>
   <expression value="bpmnModel.eAllContents.union({bpmnModel})" />
   <checkFile  value="tuwien::bpmn2bpel::bpmn::bpmnPatternCheck" />
</component>
```

Modell überarbeiten

Diese Komponente ruft die Xtend Erweiterung `preprocess` auf. Die Erweiterung führt die in 3.3 beschriebenen Veränderungen an dem in *`bpmnModel`* gespeicherten Modell durch. Das Ergebnis dieser Funktion wird wieder in die Variable `bpmnModel` gespeichert.

Listing 4.5: Modell überarbeiten

```
<component class="oaw.xtend.XtendComponent"  skipOnErrors="true">
   <metaModel id='mm' class='org.eclipse.m2t.type.emf.EmfRegistryMetaModel'/>
   <invoke value="tuwien::bpmn2bpel::transform::preprocess::preprocess(bpmnModel)"/>
   <outputSlot value="bpmnModel"/>
</component>
```

Modell speichern

Dieser Schritt ist ein optionaler, wie die Kapselung in einem `<if> ... </if>` Tag zeigt. Ist die entsprechende Variable gesetzt wird das aktuelle Modell gespeichert. Mit Hilfe dieser DSL Datei ist die Fehlersuche und Kontrolle der Transformation möglich.

Listing 4.6: Modell speichern

```
<if cond='${transform.generateBPMNafterTransformation}'>
   <component  id="generator" class="oaw.xpand2.Generator" skipOnErrors="true">
      <fileEncoding  value="ISO-8859-1" />
      <metaModel  id='mm' class='org.eclipse.m2t.type.emf.EmfRegistryMetaModel'/>
      <expand     value="tuwien::bpmn2bpel::bpmn::generator::bpmnExpand::Root FOR
          bpmnModel" />
      <advices value='tuwien::bpmn2bpel::bpmn::generator::bpmnExpandGeneric'/>
      <genPath value="${targetDir}/"/>
   </component>
</if>
```

Modell transformieren

An dieser Stelle findet die eigentlich Transformation des Modells statt. Einziges Argument ist das Modell in *`bpmnModel`*. Die Implementierung der Transformation ist Inhalt des Abschnittes 4.6.

Listing 4.7: Modell transformieren

```
<component class="oaw.xtend.XtendComponent" skipOnErrors="true">
   <metaModel id='mm' class='org.eclipse.m2t.type.emf.EmfRegistryMetaModel'/>
   <invoke value="tuwien::bpmn2bpel::transform::transform::transform(bpmnModel)"/>
   <outputSlot value="bpmnModel"/>
</component>
```

Modell speichern

Nach der Transformation des Modells wird dieses analog zum obigen Fall ein weiteres Mal in einer BPMN DSL Datei gespeichert.

Modell vergleichen

Diese zwei Komponenten sind für die Kontrolle bekannter Transformationen zuständig. Damit können Änderungen an den Transformationsregeln durch Vergleich der erzeugten Modelle sofort überprüft werden. Dafür besitzen alle Testmodelle eine Annotation, die eine Prüfsumme des erwarteten Modelles enthält. Findet die Xtend Erweiterung so eine Annotation im übergebenen `bpmnModel` generiert sie eine Prüfsumme und vergleicht sie mit der in der Annotation gespeicherten. Auf etwaige Unterschiede wird dann hingewiesen.

Listing 4.8: Modell vergleichen

```
<component class="org.openarchitectureware.util.stdlib.ExtIssueReporter"/>
<component class="oaw.xtend.XtendComponent">
   <metaModel  id='mm' class='org.eclipse.m2t.type.emf.EmfRegistryMetaModel'/>
   <invoke  value="...::checkModel(bpmnModel)"/>
</component>
```

BPEL erzeugen

Im abschliessenden Schritt wird, sofern bis dahin im Workflow noch kein Fehler aufgetreten ist, ein Xpand Template aufgerufen, das für die Erzeugung der BPEL und WSDL Dateien aus dem Modell zuständig ist. Detaillierter wird auf diesen Schritt noch in Abschnitt 4.7 eingegangen.

Listing 4.9: BPEL erzeugen

```
<component id="bpelGenerator" class="oaw.xpand2.Generator" skipOnErrors="true">
   <fileEncoding  value="ISO-8859-1" />
   <metaModel  id='mm' class='org.eclipse.m2t.type.emf.EmfRegistryMetaModel'/>
   <expand  value="...::generateBpel FOR bpmnModel" />
   <genPath value="${targetDir}/${inFileName}/"/>
   <advices value='...::elementsAdvices'/>
   <advices value='...::generateEventTransformation'/>
   <advices value='...::cabScopes'/>
   <beautifier
   class="oaw.xpand2.output.XmlBeautifier" fileExtensions="xml, xsl, wsdd, wsdl, bpel"
       />
</component>
```

4.5 Überprüfung des Eingabemodells

Eine Einschränkung bei der Verwendung von Check im Zuge der aktuellen Version von oAW ist die Beschränkung des von Xtext generierten Editor auf eine Check Datei. Zusätzlich unterstützt Check nicht den Import von weiteren Check Dateien. Beides zusammen führt zu einer großen und unübersichtlichen Check Datei für den Editor. So sind alle Check Regeln, die im Zuge des Transformationsprozesses zur Anwendung kommen, auf zwei Dateien aufgeteilt. Diese beiden Dateien, *Check.chk* und *bpmnPatternCheck.chk*, befinden sich im Projekt *tuwien.bpmn2bpel.bpmn* im gleichnamigen Package.

Echtzeit Überprüfungen: Check.chk

Diese Check Datei enthält hauptsächlich schnell und einfach zu prüfende Regeln da sie vom BPMN DSL Editor verwendet wird. Aufwendige Prüfungen würden den Editor verlangsamen, da dieser nach jeder Änderung des Modelles dieses mit allen Regeln auf seine Korrektheit überprüft. Dieses Verhalten lässt sich in der aktuellen oAW Version nicht modifizieren. Die Regeln der Datei können nach zwei unterschiedlichen Gesichtspunkten charakterisiert werden.

Elementare Prüfungen

Dabei handelt es sich um Prüfungen analog den zwei nachfolgenden in Listing 4.10. Diese haben in den meisten Fällen keinen direkten Bezug zu BPMN. Sie überprüfen nur Attribute, Referenzen und Typen von Elemente auf ihre Korrektheit.

Listing 4.10: Zwei Check Regeln die überprüfen, ob das Quell- und das Ziel-Attribut eines `SequenceFlow` auf ein existierendes Element verweisen.

```
    //SequenceFlow Regeln
2   context SequenceFlow ERROR "Source not a valid Flow Object" :
        eRootContainer.eAllContents.typeSelect(FlowObject).exists(f|
4           f.id == this.source
            );
6
    context SequenceFlow ERROR "Target not a valid Flow Object" :
8       eRootContainer.eAllContents.typeSelect(FlowObject).exists(f|
            f.id == this.target
10          );
```

Spezifische Prüfungen

Bei dieser Kategorie handelt es sich um Einschränkungen und Bedingungen die sich aus der BPMN Spezifikation ergeben wie Listing 4.11. Sie überprüfen ob das übergebene Modell keine der Beschränkungen der BPMN Spezifikation in Bezug auf einzelne Elemente verletzt.

Listing 4.11: Eine Check Regel überprüft ob `SequenceFlow` Elemente die von einem `InclusiveGateway` ausgehen auch vom entsprechenden Typ sind.

```
context SequenceFlow ERROR "The Sequence Flow MUST have its Condition attribute
    set to Expression":
  eContainer.eAllContents.typeSelect(InclusiveGateway).exists(x|
    x.id == this.source &&
    x.defaultGate.outgoingSequenceFlow != this.id && //nicht default SF
    x.gates.size > 1 )?
      this.conditionType == ConditionType::Expression
      :
      true;
```

Überprüfung vor der Transformation: bpmnPattern.chk

Diese Check Datei überprüft das übergebene Modell auf unterschiedliche Aspekte. Sowohl auf Konstruktionen und Strukturen von Elementen die in der BPMN Spezifikation nicht gestattet sind, als auch auf Prozesse die nicht lauffähig wären, oder zu Deadlocks führen würden, wie in Listing 4.12. Diese Regeln sind jedoch im Unterschied zu den oben beschriebenen teilweise relativ aufwendig, wodurch die Auslagerung in eine eigene Check Datei notwendig wurde.

Listing 4.12: Beispiel für eine bpmnPattern Check Regel.

```
/**
 * Methode ueberprueft ob mehrere SequenceFlows vom ExceptionFlow
 * wieder zurueck n den Prozess fliessen.
 */
context IntermediateEvent if this.doesMergeBackIntoNormalFlow() && target != null
    ERROR "Multiple SequenceFlow outgoing from the IntermediateEvent which joins
    back into the main flow.":
  let join = getNearestIntersection(this.getTarget(),this):
  getAllSuccessor(this.getElementsOfProcess(),this,join).notExists(suc|
    getDistanceBetween(this,join,this.getElementsOfProcess()) <
    getDistanceBetween(this,suc,this.getElementsOfProcess())
    );
```

4.6 Transformation des Modells

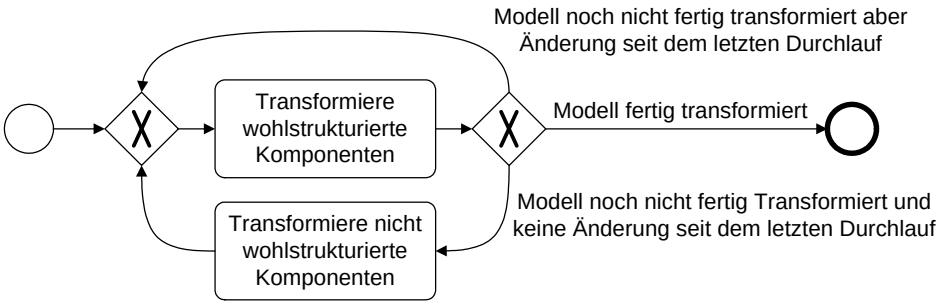


Abbildung 4.9: Die Ablauf der Xtend Transformationsfunktion.

Der grobe Ablauf der Transformation ist in Abbildung 4.9 skizziert. Die für die Transformation verantwortlichen Xtend Dateien befinden zu einem großen Teil im Projekt *tuwien.bpmn2bpel.transform* im gleichlautenden Package.

Da Xtend eine funktionale Sprache ist, kann eine Zeile im Quelltext sehr lang werden. Daher eignen sich die meisten der Extension Dateien nicht um auszugsweise wiedergegeben zu werden. Es wird in diesem Abschnitt also nur der elementare Aufbau der Erweiterungen sowie ihre ungefähre Funktionsweise beschrieben.

Extension: transform Gestartet wird die Transformation eines Modells durch den Aufruf der Extension `transform` in der Xtend Datei *transform.ext*. Das einzige Argument dieser Extension ist das zu transformierende BPMN Modell.

Listing 4.13 zeigt einen wesentlichen Ausschnitt der Datei. Die Extension `transform` ist die einzige öffentliche Extension der Datei und ruft `transformModel` auf. Dieser Extension wird als Argument das Modell übergeben sowie zwei Zähler die verwendet werden um die Tiefe der rekursiven Aufrufe zu begrenzen. Noch vor Beginn der Transformation wird in Zeile 8 der momentane Modellzustand als Prüfsumme gespeichert.

In Zeile 11 wird die Extension für die Suche und Ersetzung von CABs mit unterschiedlichen Flow Typen aufgerufen. In den nun folgenden Zeilen werden der Reihe nach die Erweiterungen für die unterschiedlichen CAB Typen aufgerufen. Bis auf den Aufruf der Erweiterung, die für die Or Fälle zuständig ist, wurden die übrigen Aufrufe aus dem Listing gekürzt. Falls das Modell bereits vor einem Aufruf fertig transformiert ist, verhindert die Extension `isTransformationComplete` weitere Aufrufe.

Falls das Modell nach diesen Aufrufen noch nicht fertig transformiert ist, es jedoch zu einer Veränderung des Modells gekommen ist, wird die Methode in Zeile 19 rekursiv aufgerufen.

Struktur der Extensions Die Extensions sind dabei in mehreren Ebenen gestaffelt. In der ersten Ebene sind jeweils alle Extensions, die für mehrere CABs eines bestimmten Typen anwendbar sind, zusammengefasst. Abbildung 4.10 zeigt einen Ausschnitt der Struktur. So enthält *flow.ext* die Extension `transformFlows` und diese wiederum ruft

Listing 4.13: Ausschnitt aus der Xtend Datei transform.ext.

```
transform(BPD model):
    let loops = 0 :
    model.transformModel(loops,0)->
    model;

private BPD transformModel(BPD model,Integer loops,Integer wholeLoop):
    let currentLoop = loops:
    let cabCount = model.getCabCount():

    // ...
    !model.isTransformationComplete()?model.transformFlows():null->
    !model.isTransformationComplete()?model.transformOrs():null->
    // Weitere Zeilen analog den obigen.

    (  cabCount < model.getCabCount()  &&  loops < 10 &&
       !model.isTransformationComplete() &&   loops == currentLoop
       )?

       model.transformModel(loops+1,wholeLoop):
       model
    ;
```

die Extensions für die unterschiedlichen Typen von Flow CABs auf. Die Extensions sind dabei für jeden Typ in einer eigenen Datei gesammelt. In den einzelnen Extensions wird überprüft, ob potentielle CABs dem jeweiligen Typ der Extension entsprechen. Ist dies der Fall, werden die Elemente gekapselt und wieder in den Prozess gehängt.

Sollte das Modell nach der Ausführung von `transform()` an der Wurzel nicht aus einem CAB bestehen wird `transform()` rekursiv erneut aufgerufen. Die Methode wird solange aufgerufen, bis sich das Modell in einem Durchlauf nicht mehr verändert oder aber es an der Wurzel nur mehr aus einem CAB besteht. Im ersten Fall kommen dann die Methoden für nichtwohlstrukturierte Modelle zum Einsatz.

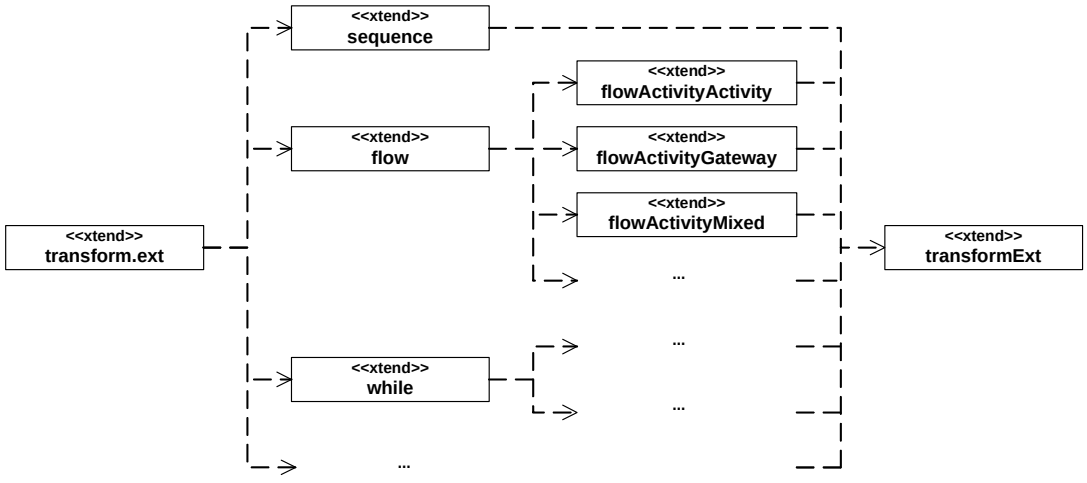


Abbildung 4.10: Struktur der Xtend Dateien.

Aufbau einer Extension In den meisten Fällen folgen die Dateien für die einzelnen CAB Typen demselben Aufbau. Sie enthalten eine öffentliche Erweiterung der das Modell übergeben wird. Diese ruft eine globale Erweiterung auf, die alle im Modell noch vorhandenen CABs zurückliefert. Diese werden nun der Reihe nach untersucht ob sie dem CAB Typen entsprechen. Abschliessend werden eventuell gefundene Strukturen gekapselt.

4.7 Generierung von BPEL

Die Generierung des BPEL Codes aus dem überarbeiteten Modell erfolgt mit Hilfe mehrerer Xpand Templates im Projekt *tuwien.bpmn2bpel.transform* im Package `tuwien.bpmn2bpel.generateBPEL`.

4.7.1 Allgemeiner Ablauf

In einem ersten Schritt erzeugt ein Template eine neue Datei und die einleitenden Tags des BPEL Prozesses. Existieren gültige *Implementation* Instanzen im BPD, oder andere BPMN Elemente mit den notwendigen Informationen, werden aus diesen WSDL und `partnerLink` Definitionen generiert. Ist dieser einleitende Schritt abgeschlossen, beginnt die Erzeugung der BPEL Aktivitäten. Dafür wird das Wurzelelement des BPD dem Template `generate` übergeben. Der BPEL Prozess wird nun von diesem Wurzelknoten aus, bei dem es sich um ein CAB handelt, rekursiv erzeugt. Für die unterschiedlichen Elemente existieren jeweils eigene `generate` Templates die den entsprechenden BPEL Code erzeugen. Es lassen sich nach Typ des Elements grob zwei Vorgehensweisen unterscheiden.

- Templates für *FlowObject* Elemente generieren den entsprechenden Code und rufen danach wieder ein Template mit den nachfolgenden Elementen als Argument auf.
- Templates für *Subprocess* Elemente erzeugen zuerst die einleitenden BPEL Elemente, rufen die Templates für alle Kinder Elemente auf, erzeugen die schliessenden Elemente um letztendlich noch die Templates für alle nachfolgenden Elemente aufzurufen.

Alogrithmus 1 auf Seite 78 skizziert den generellen Ansatz der `generate` Templates. Insbesondere bei den Templates für die CABs ist der Ablauf jedoch komplexer.

Algorithmus 1 : Der Pseudocode von `generate` zur Erzeugung des BPEL Prozesses aus dem transformierten BPD

```
Algorithmus :generate(Element this)
begin
    Erzeuge Anfangs Tags von this
    if this ist ein Subprocess then
        generate(kinder.selectInitial())
    end
    Erzeuge End Tags von this
    für jedes Element nachfolger das ein Nachfolger von this ist tue
        generate(nachfolger)
    Ende
end
```

Bei der Implementierung in Xpand wurde die Logik für den Aufruf der nachfolgenden Elemente von den Code erzeugenden Templates getrennt und in eigene Templates

ausgelagert. Einzig die Templates für die unterschiedlichen CAB Typen enthalten noch zusätzliche Kontrollstrukturen. Die folgenden drei Templates stehen stellvertretend für die Templates von *FlowObject* Elementen. Das erste Template für den „abstrakten“ Typ *Element*, ist das Default Template. Es wird aufgerufen, sollte kein eigenes `generate` Template für den Typ des übergebenen Elementes existieren. Es erzeugt einzig einen Kommentar im BPEL Prozess, wie in Zeile 2 zu sehen ist. Die beiden anderen sind für die Elemente *None* und *Receive*. Beide bedienen sich weiterer Templates. Das `generateLink` Template in Zeile 7 und 18 erzeugt die für die Control Link Transformation notwendingen ***sources*** und ***targets*** Tags innerhalb der jeweiligen Aktivität.

Listing 4.14: Drei `generate` Templates für unteschiedliche Elemente

```
«DEFINE generate FOR Element»
2    <!-- no rule defined yet for Element «this.id» -->
«ENDDEFINE»
4
«DEFINE generate FOR None»
6 <bpel:empty name="«getElementName()»">
    «EXPAND generateLink FOR this»
8 </bpel:empty>
«ENDDEFINE»
10
«DEFINE generate FOR Receive»
12    <bpel:receive
        name="«id»"
14      createInstance="«instantiate?"yes":"no"»"
        «EXPAND generateImplementation FOR implementation»
16      variable="«message.name»"
      >
18      «EXPAND generateLink FOR this»
      </bpel:receive>
20 «ENDDEFINE»
```

Alle drei Templates erzeugen nur die jeweiligen BPEL Aktivitäten. Es existieren keine Kontrollstrukturen oder Aufrufe nachfolgender Elemente. Das `generate` Template besitzt aber für unterschiedliche Typen sogenannte AROUND Templates oder Aspekte. Dabei handelt es sich um ein Feature von oAW, das es ermöglicht den Aufruf von Templates zu beeinflussen, ohne den Code des Template verändern zu müssen. Diese AROUND Templates werden nun verwendet um je nach Typ des aktuellen Elementes, beziehungsweise bestimmter Attribute, das `generate` Template mit seinen Nachfolgern aufzurufen. Dadurch ist die Logik, die die Generierung steuert, getrennt von den Templates die den eigentlichen Code erzeugen.

Das AROUND Statement in Listing 4.15 kapselt alle `generate` Aufrufe mit einem `FlowObject` als Parameter. In Zeile 2 wird ein externes Tempalte mit dem Element als Parameter aufgerufen. Dieses Template erzeugt, wenn notwendig ein `assign` Element, vor dem eigentlichen BPEL Element. In Zeile 3 wird überprüft ob für das Element überhaupt BPEL Code erzeugt werden soll. Soll eine BPEL Aktivität generiert werden, wird in Zeile 4 das Ziel-Template ausgeführt. Im Anschluss wird noch ein Template aufgerufen, das eventuell nachfolgende `assign` Elemente erzeugt und in der letzten Zeile `generate` für jeden Nachfolger.

Listing 4.15: Das AROUND `generate` Template für *FlowObject* Elemente.

```
«AROUND ... ::generate FOR FlowObject»
    «EXPAND elements::generateStartAssignment FOR this»
    «IF generateBpelElement()»
        «targetDef.proceed()»
    «ENDIF»
    «EXPAND elements::generateEndAssignment FOR this»
    «EXPAND ... ::generate FOREACH  getSuccessor()»
«ENDAROUND»
```

Der Aufbau der Templates für die unterschiedlichen CAB Typen ähnelt dem in Algorithmus 1 auf Seite 78 skizziertem allgemeinem Fall.

Das nachfolgende Template ist das einfachste aller CAB Templates und zuständig für CABs vom Typen *Sequence*. In Zeile 2 wird das einleitende BPEL `sequence` Tag ausgeschrieben und benannt. In Zeile 3 wird anschliessend ein weiteres Template aufgerufen. Dieses Template ist für die Generierung der für die Control Link Transformation notwendingen Tags zuständig, falls der CAB Teil einer solchen ist. Anschliessend wird in Zeile 5 das Startelement der Sequence in eine Variable gespeichert. Handelt es sich nur um ein Element, wird in 7 der `generate` Aufruf dieses Elements durchgeführt. Handelt es sich aber um mehrere Elemente, wird eine Fehlermeldung generiert.

Listing 4.16: Das `generate` Template für einen CAB vom Typen *Sequence*

```
«DEFINE generateSequence FOR CAB»
    <bpel:sequence name="«getElementName()»">
    «EXPAND elements::generateLink FOR this»
    <!-- Generated out of CAB «this.id»-->
    «LET Elements.getRemainingStartEvent() AS start»
        «IF start.size==1»
            «EXPAND elements::generateElement FOR start.get(0)»
        «ELSE»
            «EXPAND generateErrorBlock FOR this»
        «ENDIF»
    «ENDLET»
    </bpel:sequence>
«ENDDEFINE»
```

Alle weiteren für die unterschiedlichen CAB Typen zuständngen Templates befinden sich im Projekt *`tuwien.bpmn2bpel.transform`* im Package `generateBPEL`. Bei den Templates für die unterschiedlichen CAB Typen muss jedoch zwischen zwei Fällen unterschieden werden.

Synchronisierende CAB Typen Dabei handelt es sich um CAB Typen die im Normalfall einen synchronisierenden Gateway als Endelement besitzen. Die einzige Ausnahme bildet der CAB Typ *Sequence*. Mit Ausnahmen von *Sequence* rufen aber alle CAB Typen die Templates zur Erzeugung ihrer Nachfolger selber auf. Auf den Ablauf wird noch exemplarisch in 4.7.2 eingegangen.

Nichtsynchronisierende CAB Typen Diese CAB Typen besitzen keinen synchronisierenden Gateway als Endelement. Im Unterschied zum obigen Fall rufen die Elemente

des CAB die Templates zur Erzeugung der Nachfolger des CAB auf und nicht der CAB selber. In Abschnitt 4.7.3 wird noch die Generierung anhand eines einfachen Beispiels demonstriert.

Dieser Unterschied hat große Auswirkungen auf den Ablauf der Generierung wie die nachfolgenden beiden Abschnitte zeigen werden.

4.7.2 Ablauf bei synchronisierenden CAB Typen

Abbildung 4.11(a)–(c) illustriert den Ablauf der Templates anhand eines einfachen Prozesses, der nur aus synchronisierenden CAB Typen besteht.

Das Wurzelelement dieses Prozesses ist der äußerste CAB, in diesem Fall *CAB3* vom Typ *Sequence*. Mit diesem wird nun das `generate` Template aufgerufen (1). Das Template für diesen CAB Typen wird ausgeführt und ein `sequence` Element in die BPEL Datei geschrieben. Generell werden zuerst Kinder von CABs abgearbeitet und erst danach die nachfolgenden Elemente.

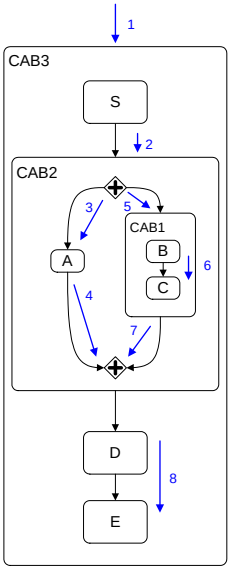
Daher ruft das Template für *Sequence* CABs im nächsten Schritt `generate` mit dem Startelement *S* des CABs auf (2). Dabei handelt es sich um ein *None* Element. Das Template für *None* Elemente erzeugt das notwendige BPEL Element und ruft danach wieder `generate` mit dem Nachfolger *CAB2* von *S* auf (3).

Dabei handelt es sich wieder um einen CAB, nur handelt es sich diesmal um einen CAB vom Typen *GatewayGatewayFlow*. Wieder wird das erste Element, es handelt sich um einen Gateway, aufgerufen (4). Dessen `generate` Statement erzeugt keinen Code im Prozess, folglich werden sofort seine Nachfolger aufgerufen (3 und 5). Dieser Ablauf wiederholt sich solange bis ein Element keine Nachfolger mehr hat. Abbildung 4.11(c) zeigt den Baum der `generate` Aufrufe.

Dieser Ansatz kommt auch auch bei CABs, die mit Hilfe der Control Link beziehungsweise eventHandler Transformation erzeugt wurden, zum Einsatz. Bei beiden CAP Typen dieses Ansatzes handelt es sich um synchronisierende.

4.7.3 Ablauf bei nicht synchronisierenden CAB Typen

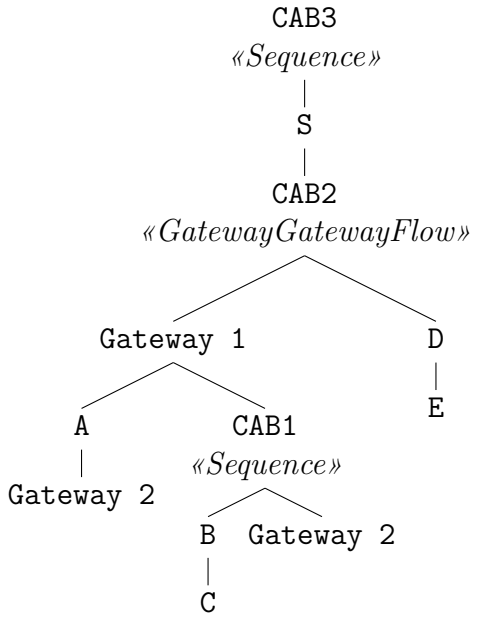
Abbildung 4.12(a)–(c) zeigt den Ablauf der Templates eines einfachen Prozesses, der jedoch einen nicht synchronisierenden CAB enthält. Der Prozess sieht dem in 4.7.3 verwendeten sehr ähnlich, jedoch handelt es sich bei *CAB2* in diesem Fall nicht um einen *GatewayGatewayFlow* CAB sondern um einen *GatewayActivityFlow*. Das Endelement ist in diesem Fall kein *ParallelGateway* sondern ein *Activity* Element. Die BPEL Generierung verläuft bis zu diesem CAB analog zum synchronen Fall. Der entscheidende Unterschied ist jedoch, daß die Elemente des CAB bei den Aufrufen ihrer Nachfolger nicht mehr auf die Elemente des CAB beschränkt sind. Wie in 4.12(a) zu sehen, enden die Aufrufe der nachfolgenden Elemente nicht mehr mit dem letzten Element des CAB. Das CAB ruft folglich auch nicht seine Nachfolger auf da dies bereits von seinen „Kinder“ Elementen vorgenommen wurde. In Abbildung 4.12(c) ist der Unterschied zum synchronisierenden Fall anhand der Aufrufe von `generate` deutlich zu erkennen.



(a) transformiertes BPD.

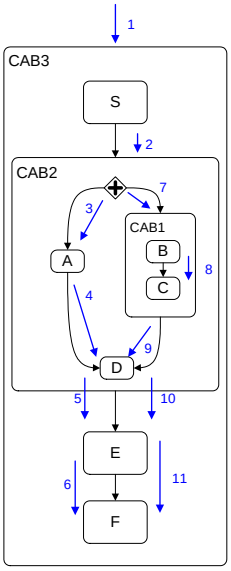
```
<sequence>
  <empty name="S"/>
  <flow>
    <empty name="A"/>
    <sequence>
      <empty name="B"/>
      <empty name="C"/>
    </sequence>
  </flow>
  <empty name="D"/>
  <empty name="E"/>
</sequence>
```

(b) BPEL Struktur.



(c) Der Baum der Aufrufe von `generate`.

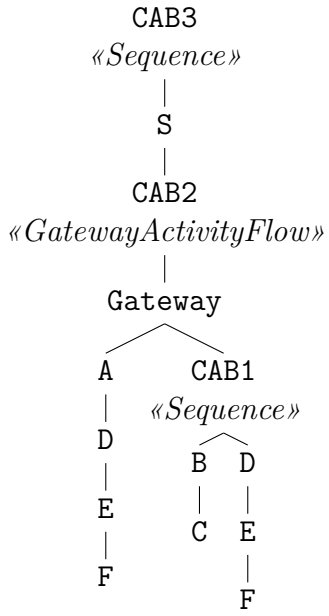
Abbildung 4.11: Generation von BPEL im Falle synchronisierender CABs



(a) transformiertes BPD.

```
<sequence>
  <empty name="S"/>
  <flow>
    <sequence>
      <sequence>
        <empty name="B"/>
        <empty name="C"/>
      </sequence>
      <empty name="D"/>
      <empty name="E"/>
      <empty name="F"/>
    </sequence>
    <sequence>
      <empty name="A">
      <empty name="D_1"/>
      <empty name="E_1"/>
      <empty name="F_1"/>
    </:sequence>
  </flow>
</sequence>
```

(b) BPEL Struktur.



(c) Der Baum der Aufrufe von `generate`.

Abbildung 4.12: Generation von BPEL im Falle nicht synchronisierender CABs

4.7.4 Generierung von sonstigen Elementen

Neben den `generate` Templates für Elemente sowie CABs existieren noch weitere, die für Attribute, wie zum Beispiel die Transformation von *Assignment* in `assign` Elemente, zuständig sind. Diese sind jedoch in den meisten Fällen relativ einfach und werden daher nicht näher erläutert.

5 Beispiele für Transformationen

In diesem Kapitel werden ausgewählte Prozesse schrittweise transformiert um die in den vorangegangenen Kapiteln vorgestellten Transformationsalgorithmen zu demonstrieren.

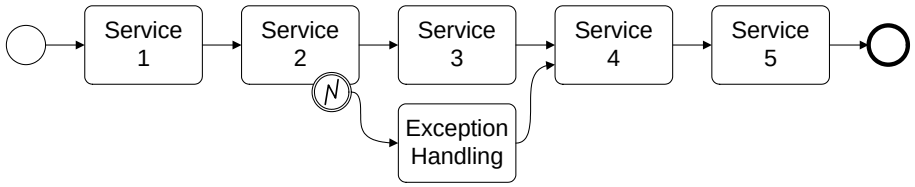
- 5.1 Wohlstrukturierte Prozesses
  - 5.1.1 Prozess mit Exception (Seite 85)
  - 5.1.2 Prozess mit multiplen Start- und Endelementen (Seite 93)
- 5.2 Nichtwohlstrukturierte Prozesse
  - 5.2.1 ohne Schleifen (Seite 100)
  - 5.2.2 mit Schleifen (Seite 106)

Diese vier ausgewählten Prozesse repräsentieren eine große Klasse von möglichen Prozessen, stellen aber nur einen Bruchteil der Testdaten dar, die verwendet wurden um den Ansatz zu überprüfen. Die Abbildung aller verwendeten Testfälle würde den Umfang dieser Arbeit sprengen. Daher werden nur einige weitere wichtige in Abschnitt 5.3 präsentiert.

5.1 Wohlstrukturierte Prozesse

5.1.1 Prozess 1 - Prozess mit Exception

Ein einfacher Prozess, der eine der Möglichkeiten der Transformation von Prozessen mit *IntermediateEvent* Elementen, die nicht im normalen Kontrollfluss liegen, demonstriert. Abbildung 5.1(a) zeigt den Prozess mit einem *IntermediateEvent* vom Typ *Exception* der an das Element ***Service2*** geheftet ist. Wie in Listing 5.1 erkennbar, wird dies in der BPMN DSL durch ein `target` Attribute des `<IntermediateEvent ... >` Elements dargestellt.



(a) BPMN Abbildung des Prozesses.

```
<Diagram exception8  name "Exception test 1"
   <Pool first name "Exception pool"
      <Process exception8 name "Exception test 1"
            <StartEvent start triggerType None>
            <SequenceFlow sf1 start service1>
            <None service1>
            <SequenceFlow sf2 service1 service2>
            <None service2>
            <SequenceFlow sf3 service2 service3>
            <None service3>
            <SequenceFlow sf4 service3 service4>
            <None service4>
            <SequenceFlow sf5 service4 service5>
            <None service5>
            <SequenceFlow sf6 service5 end>
            <EndEvent end triggerType None>
            //Attached Exception Event
            <IntermediateEvent catchError
               triggerType Error
               target service2
               errorCode "test"
            >
            <SequenceFlow sf10
               catchError ExceptionHandling>
            <None ExceptionHandling>
            <SequenceFlow sf11 ExceptionHandling service4>
      >
   >
>
```

(b) Der Prozess in BPMN DSL.

Abbildung 5.1: Prozess1 - Prozess mit Exception vor der Transformation

Dieser Prozess wird nun in einen gültigen BPEL Prozess transformiert. Da es sich bei den Elementen des Prozesses nur um *None* Elemente handelt, werden keine WSDL oder `partnerLink` Definitionen erzeugt. In einem späteren Beispiel wird auf die Erzeugung der

WSDL und `partnerLink` Definitionen aus den Element der BPMN DSL eingegangen.

Transformation des BPD

Prozess auf die Transformation vorbereiten Keine der Methoden ist für die Transformation des Prozesses notwendig.

Schritt 1 Im ersten Schritt 5.2(a) werden drei Strukturen identifiziert. Bei *CAB1* und *CAB2* handelt es sich um CABs vom Typ *Sequence*. *CAB3* ist ein CAB vom Typ *ExceptionFlow*. Bei diesem Typ handelt es sich um eine Sequence die aber von einem *IntermediateEvent* ausgeht. Dieser Unterschied ist für den nächsten Schritt relevant. Die Namen spiegeln nur die Reihenfolge ihrer Identifizierung wieder und haben ansonsten keine besondere Bedeutung.

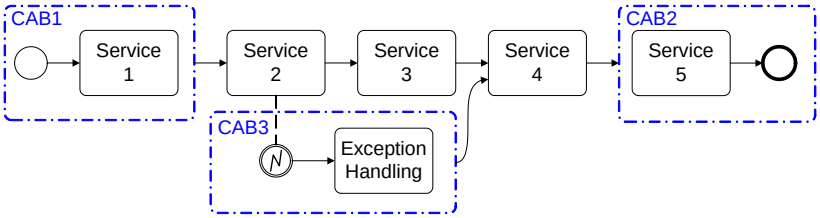
Schritt 2 In diesem Schritt wird das Element, an dem der *IntermediateEvent* des *ExceptionFlow* CABs hängt, in diesem Fall also *Service2*, in einem *ActivityScope* CAB gekapselt. Da der Prozess erst wieder bei *Service4* fortsetzt, wenn der Event ausgelöst wird, muss *Service3* noch gekapselt werden. In einem ersten Schritt werden dazu dem Prozess zwei neue Variablen hinzugefügt. Weiters wird wie in Abbildung 5.3 dem Element *Service2* und dem Event jeweils eine *Assignment* Anweisung für jede dieser Variablen hinzugefügt. Anschließend werden *Service2* und *CAB3* in *CAB4* gekapselt wie in Abbildung 5.2(b). *CAB4* ist vom Typ *ActivityScope*.

Schritt 3 Bei der nächsten Struktur, die identifiziert wird, handelt es sich um ein Gateway-Gateway XOR das *Service3* kapselt. In diesem Fall sind keine Modifkationen notwendig und die Elemente werden in einen neuen CAB vom Typen *XorSplitJoin* verschoben. Abbildung 5.2(c) stellt diesen Schritt dar.

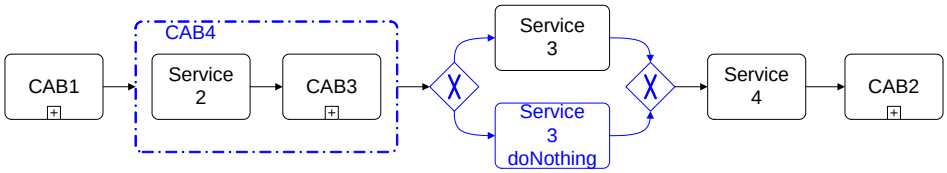
Schritt 4 Im weiteren Schleifendurchlauf wird der Prozess noch auf die übrigen wohlstrukturierten CAB Typen hin untersucht, es kann jedoch keine passende Struktur mehr im Prozess gefunden werden. Da sich der Prozess aber in diesem Durchlauf bereits gegenüber dem ursprünglichen Zustand verändert hat, wird ein weiterer Schleifendurchlauf gestartet.

Schritt 5 Die erste Struktur die im nächsten Schleifendurchlauf erkannt wird, ist wieder eine Sequence wie in Abbildung 5.2(d). Nach der Kapselung der Elemente in CAB6 besteht der Prozess nur noch aus diesem einen CAB, damit ist die Transformation abgeschlossen.

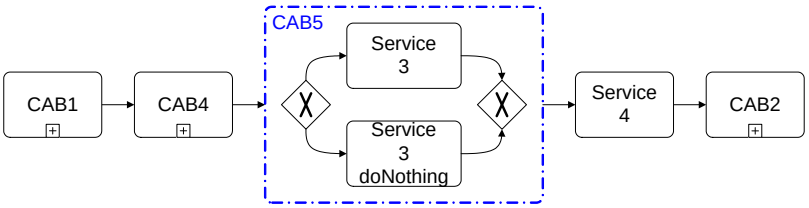
Der fertig transformierte Prozess ist nochmals als DSL in Listing 5.5 abgebildet.



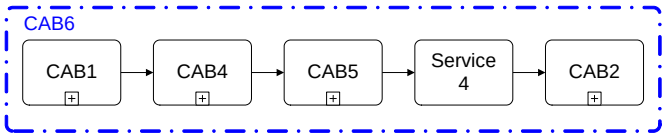
(a) Schritt 1 - Identifizierung und Ersetzung von zwei *Sequence* CABs und einem *ExceptionFlow* CAB.



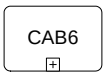
(b) Schritt 2 - Identifizierung und Ersetzung eines *ActivityScope* CABs. In diesem Fall mündet der Exception Flow nicht wieder direkt nach dem auslösenden Element in den normalen Prozessverlauf. Daher ist es notwendig ***Service3*** mit Hilfe zweier *DataXor* zu kapseln.



(c) Schritt 3 - Identifizierung und Ersetzung eines *XorSplitJoin* CABs.



(d) Schritt 5 - Identifizierung und Ersetzung eines *Sequence* CABs.



(e) Der Prozess ist fertig transformiert.

Abbildung 5.2: Prozess 1 : Die Identifizierung und Ersetzung von CABs. Abbildungen 5.2(a)-(c) stellen Schritte im ersten Schleifendurchlauf dar, 5.2(d) und 5.2(e) Schritte im zweiten Schleifendurchlauf.

```
<Process exception8 name "Exception test 1"
   <Properties
      <Property "service2_normalCompletion"  "boolean">
      <Property "catchError"  "boolean">

   >
   ...
>
```

(a) Die Definition der einzelnen Variablen als *Property* des Prozesses.

```
<None service2
   <Assignments
      <Assignment "service2_normalCompletion" "true()" AssignTime Start>
      <Assignment "catchError" "false()" AssignTime Start>
   >
 >
```

(b) *service2* mit den *Assignment* Anweisungen für die Kapselung von *service3*. Die gesetzten Werte sind die Standardwerte und bleiben unverändert kommt es nicht zu einer Exception.

```
<IntermediateEvent catchError
   <Assignments
      <Assignment "service2_normalCompletion" "false()" AssignTime End>
      <Assignment "catchError" "true()" AssignTime End>
   >
   triggerType Error
   target service2
   errorCode "test"
 >
```

(c) Der *IntermediateEvent* mit den *Assignment* Anweisungen im Falle einer Exception.

Abbildung 5.3: Setzten der für die Kapselung von *Service 3* benötigten Variablen durch *Assignment* Anweisungen in den entsprechenden Elementen.

Listing 5.5: Prozess 1 - Der transformierte Prozess in BPMN DSL.

```
<Diagram exception8 name "Exception test 1"
  <Pool first name "first pool" Participant  <Entity "exception8">
    <Process exception8 name "Exception test 1"
      <Properties
        <Property "service2_normalCompletion"  "boolean">
        <Property "catchError"  "boolean">
      >
      <CAB CAB6 Sequence
        <SequenceFlow sf3 CAB2 CAB5 ConditionType None>
        <SequenceFlow sf4 CAB5 service4 ConditionType None>
        <SequenceFlow sf5 service4 service5 ConditionType None>
        <CAB CAB2 ActivityScope
          <CAB CAB3 ExceptionFlow
            <IntermediateEvent catchError
              <Assignments
                <Assignment "catchError" " true() " AssignTime End>
                <Assignment "service2_normalCompletion" " false() "
                    AssignTime End>
              >
            triggerType Error target service2 errorCode "test">
            <SequenceFlow sf10 catchError ExceptionHandling ConditionType
                None>
            <None ExceptionHandling>
          >
          <None service2
            <Assignments
              <Assignment "service2_normalCompletion" " true() "
                  AssignTime Start>
              <Assignment "catchError" " false() " AssignTime Start>
            >
          >
        >
        <SequenceFlow sf2 service1 CAB2 ConditionType None>
        <CAB CAB5 XorSplitJoin
          <None service3>
          <SequenceFlow sf30 service3_split service3_doNothing ConditionType
              Default>
          <SequenceFlow sf31 service3 service3_join ConditionType None>
          <SequenceFlow sf32 service3_doNothing service3_join ConditionType
              None>
          <DataXor service3_split Gates sf33 DefaultGate sf30>
          <DataXor service3_join >
          <None service3_doNothing>
          <SequenceFlow sf33 service3_split service3 ConditionType
              Expression "$exception8_ProcessData.service2_normalCompletion
              or not($exception8_ProcessData.catchError)">
        >
        <None service4>
        <None service1>
        <SequenceFlow sf1 start service1 ConditionType None>
        <StartEvent start triggerType None>
        <None service5>
        <EndEvent end triggerType None>
        <SequenceFlow sf6 service5 end ConditionType None>
        >
      >
    >
  >
>
```

BPEL Generieren

Da es sich um einen wohlstrukturierten Prozess handelt erfordert die Generierung keine besonderen Aufwand und der erzeugte Code ist gut lesbar.

Aus dem *ActivityScope* CAB wird ein `scope` Element erzeugt wobei der enthaltene *ExceptionFlow* CAB als `faultHandler` Element übersetzt wird.

Abbildung 5.4 auf Seite 92 zeigt den BPEL Prozess im graphischen Editor von NetBeans[3]. Die Farben der Abbildung wurden leicht verändert um die Lesbarkeit zu erhöhen.

Listing 5.6: Prozess 1 - Vollständiger Quelltext des erzeugten BPEL Prozesses

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<bpel:process xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/
    executable" xmlns:ns1="exception8" xmlns:xsd="http://www.w3.org/2001/XMLSchema
    " exitOnStandardFault="yes" name="exception8" suppressJoinFailure="yes"
    targetNamespace="exception8">
   <bpel:import importType="http://schemas.xmlsoap.org/wsdl/" location="
       exception8PV.wsdl" namespace="exception8"/>
   <bpel:partnerLinks></bpel:partnerLinks>
   <bpel:variables>
      <bpel:variable messageType="ns1:exception8_ProcessDataMessage" name="
          exception8_ProcessData"/>
   </bpel:variables>
   <!-- Begin Elements Block -->
   <bpel:sequence>
      <bpel:empty name="service1"></bpel:empty>
      <bpel:scope>
         <bpel:variables></bpel:variables>
         <bpel:faultHandlers>
            <bpel:catch faultName="test">
               <bpel:sequence>
                  <bpel:sequence>
                     <bpel:assign>
                        <bpel:copy>
                           <bpel:from>true() </bpel:from>
                           <bpel:to part="catchError" variable="
                               exception8_ProcessData"/>
                        </bpel:copy>
                        <bpel:copy>
                           <bpel:from>false() </bpel:from>
                           <bpel:to part="service2_normalCompletion" variable="
                               exception8_ProcessData"/>
                        </bpel:copy>
                     </bpel:assign>
                  </bpel:sequence>
                  <bpel:empty name="ExceptionHandling"></bpel:empty>
               </bpel:sequence>
            </bpel:catch>
         </bpel:faultHandlers>
         <bpel:sequence>
            <bpel:assign name="service2_StartAssignment">
               <bpel:copy>
                  <bpel:from>true() </bpel:from>
                  <bpel:to part="service2_normalCompletion" variable="
                      exception8_ProcessData"/>
               </bpel:copy>
               <bpel:copy>
                  <bpel:from>false() </bpel:from>
                  <bpel:to part="catchError" variable="exception8_ProcessData"/>
               </bpel:copy>
            </bpel:assign>
            <bpel:empty name="service2"></bpel:empty>
```

```
44            </bpel:sequence>
          </bpel:scope>
46        <bpel:if>
             <bpel:condition expressionLanguage="urn:oasis:names:tc:wsbpel:2.0:
                 sublang:xpath1.0">$exception8_ProcessData.service2_normalCompletion
                 or not($exception8_ProcessData.catchError) </bpel:condition>
48           <bpel:empty name="service3"></bpel:empty>
             <bpel:else>
50              <bpel:empty name="service3_doNothing"></bpel:empty>
             </bpel:else>
52        </bpel:if>
          <bpel:empty name="service4"></bpel:empty>
54        <bpel:empty name="service5"></bpel:empty>
          <bpel:empty name="end"/>
56     </bpel:sequence>
    </bpel:process>
```

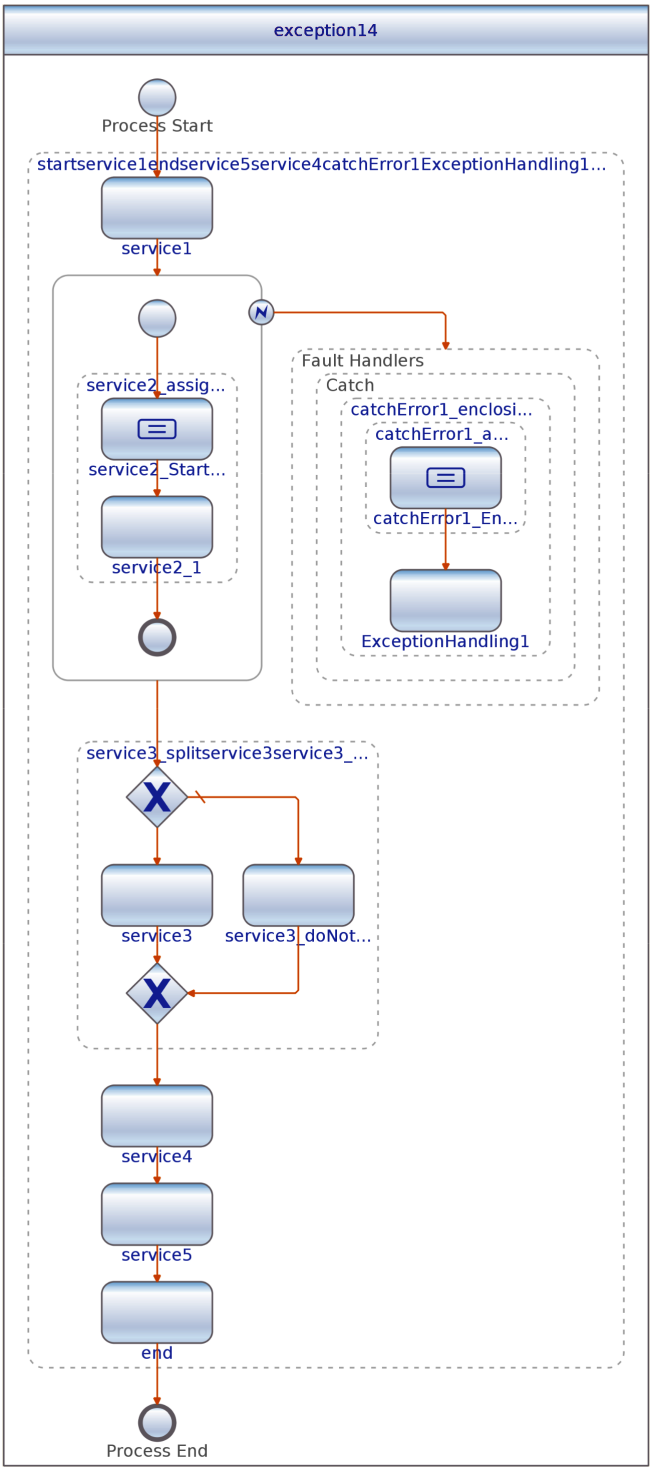


Abbildung 5.4: Prozess 1 - graphische Darstellung des BPEL Prozesses in NetBeans[3]

5.1.2 Prozess 2 - Prozess mit multiplen Start- und Endelementen

Hierbei handelt es sich um einen Prozess mit zwei Startelementen. Bei den beiden verwendeten Startelementen handelt es sich um *Receive* Elemente. Auf das *Receive* ***startB*** folgt direkt ein *EventXor* mit zwei weiteren *Receive* Elementen. Abbildung 5.5 zeigt die graphische Darstellung, Listing 5.7 auf Seite 94, die entsprechende DSL Datei.

In Zeile 7 und 14 in Listing 5.7 werden die eingehenden Nachrichten der beiden *Receive* Elemente definiert, die den Prozess starten. Die `partnerLink` und `partnerLinkType` Definitionen werden aus den *Implementation* Elementen in Zeile 9 und 16 erzeugt. In Zeile 19 wird ein weiterer *EventXor* definiert. Die zwei Nachrichten der zugehörigen *Receive* Elemente werden in Zeile 24 und 30 definiert.

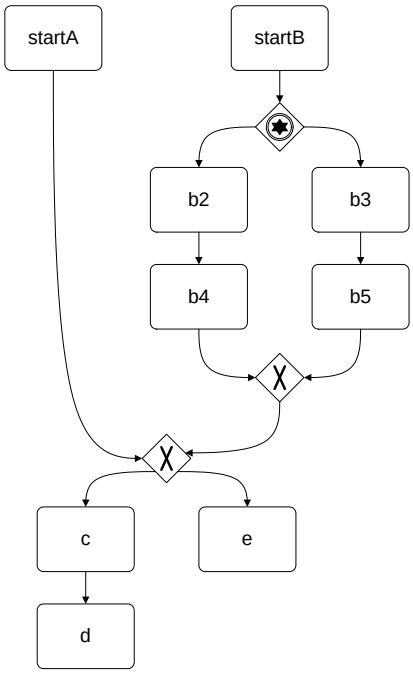


Abbildung 5.5: Prozess 2 - Ein Prozess mit multiplen Start- und Endelementen.

Transformation des BPD

Prozess auf die Transformation vorbereiten In einem ersten Schritt wird vor die zwei *Receive* ***startA*** und ***startB*** ein *EventXor* in den Prozess eingefügt und der *EventXor* ***xor*** aufgespalten wie in Abbildung 5.6 dargestellt.

Schritt 1 Anschliessend können drei Sequence Strukturen identifiziert werden, wie in Abbildung 5.7(a) dargestellt. Diese werden in den CABs ***CAB1*** bis ***CAB3*** vom Typen *Sequence* gekapselt.

Schritt 2 Im nächsten Schritt können zwei Xor Strukturen identifiziert und ersetzt werden. Diese werden in den CABs ***CAB4*** und ***CAB5*** mit dem Typen *XorSplitJoin* gekapselt. Abbildung 5.7(b).

Listing 5.7: Prozess 2 - Der Prozess in BPMN DSL

```
<Diagram beispiel2   name "Ein einfacher Prozess"
   <Pool
   beispiel2 name "first pool"
      <Process beispiel2 name "beispiel2"
         <Receive startA
            in
            <inMessage "startA" <Role "Customer"><Role "processRole">>
            Implementation
            <WebService <Role "processRole"> "startPort" "startAoperation">
         >
         <SequenceFlow sf1 startA xor>
         <Receive startB
            in
            <inMessage "startB" <Role "Customer"><Role "processRole">>
            Implementation
            <WebService <Role "processRole"> "startPort" "startBoperation">
         >
         <SequenceFlow sf2 startB event>
         <EventXor event Gates sf3 sf4>
         <SequenceFlow sf3 event b2>
         <SequenceFlow sf4 event b3>
         <Receive b2
            in
            <inMessage "b2"   <Role "bank"><Role "processRole">>
            Implementation
            <WebService <Role "processRole2"> "bankPort" "dosomething">
         >
         <Receive b3
            in
            <inMessage "b3"   <Role "bank"><Role "processRole">>
            Implementation
            <WebService    <Role "processRole2"> "bankPort" "dosometh">
         >
         <SequenceFlow sf5 b2 b4>
         <SequenceFlow sf6 b3 b5>
         <None b4>
         <None b5>
         <SequenceFlow sf7 b4 xor2>
         <SequenceFlow sf8 b5 xor2>
         <DataXor xor2 Gates sf9>
         <SequenceFlow sf9 xor2 xor>
         <DataXor xor Gates sf10 sf11>
         <SequenceFlow sf10 xor c ConditionType Expression "A">
         <SequenceFlow sf11 xor e  ConditionType Expression "B">
         <SequenceFlow sf12 c d>
         <None c>
         <None e>
         <None d>
      >
   >
>
```

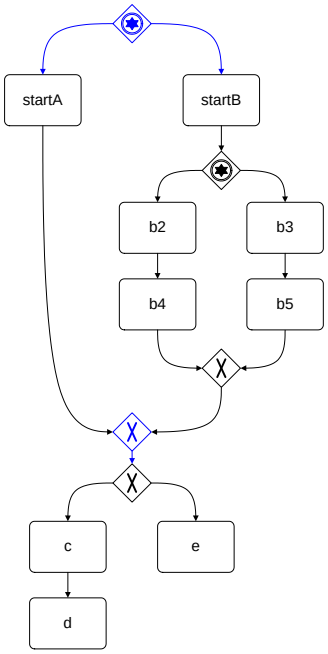
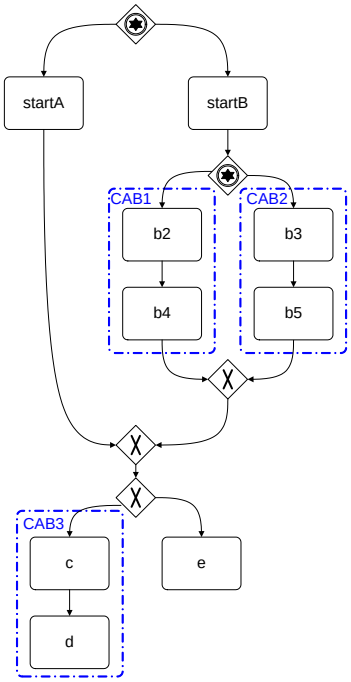


Abbildung 5.6: Überarbeitung des Prozesses vor der eigentlichen Transformation.

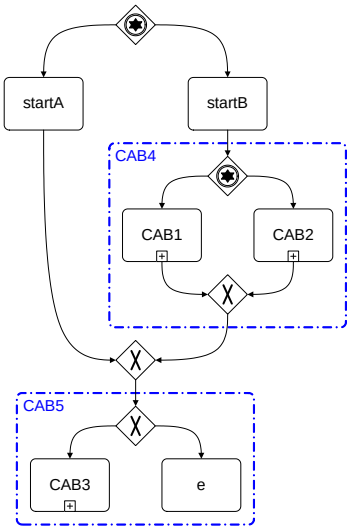
Schritt 3 Die nächste Struktur, die identifiziert werden kann, ist wieder eine Sequence. Diese wird in *CAB6* vom Typen *Sequence* gekapselt. Abbildung 5.7(c).

Schritt 4 In Schritt vier kann wieder ein Xor gefunden und ersetzt werden. Die Elemente werden in *CAB7* vom Typ *XorSplitJoin* gekapselt. Abbildung 5.7(d).

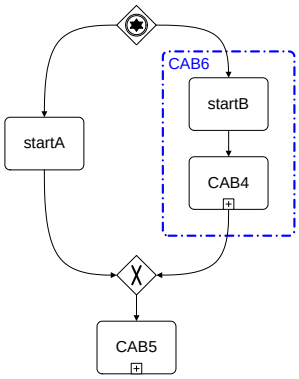
Schritt 5 Im fünften Schritt wird eine Sequence gefunden. Diese wird in *CAB8* gekapselt. Abbildung 5.7(e). Damit ist der Prozess soweit fertig transformiert und bereit für die Generation des BPEL Prozesses.



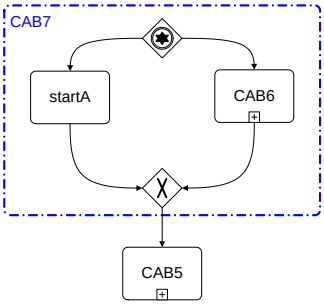
(a) Schritt 1 - Identifizierung von drei *Sequence* CABs



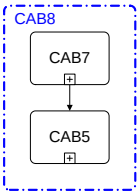
(b) Schritt 2 - Identifizierung von zwei *XorSplitJoin* CABs



(c) Schritt 3 - Identifizierung eines *Sequence* CABs



(d) Schritt 4 - Identifizierung eines *XorSplitJoin* CABs



(e) Schritt 5 - Identifizierung eines *Sequence* CABs

Abbildung 5.7: Prozess 2 : Die Identifizierung und Ersetzung von CABs. Abbildungen 5.7(a)-(b) stellen Schritte im ersten Schleifendurchlauf, 5.7(c)-(d) Schritte im zweiten Schleifendurchlauf und 5.7(e) im dritten Schleifendurchlauf dar.

BPEL Generieren

Auch dieser Prozess ist wohlstrukturiert und die Transformation daher nicht sonderlich aufwendig. Die *EventXor* und *DataXor* Elemente in den jeweiligen CABs werden vom entsprechenden Template behandelt. Es existieren also keine unterschiedlichen Templates für diese beiden Fälle. Listing 5.8 zeigt den erzeugten BPEL Prozess.

Listing 5.8: Prozess 2 - Vollständiger Quelltext des erzeugten BPEL Prozesses

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Process beispiel2 beispiel2 -->
<bpel:process xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/
    executable" xmlns:ns1="beispiel2" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     exitOnStandardFault="yes" name="beispiel2" suppressJoinFailure="yes"
    targetNamespace="beispiel2">
   <bpel:import importType="http://schemas.xmlsoap.org/wsdl/" location="
       beispiel2PV.wsdl" namespace="beispiel2"/>
   <bpel:import importType="http://schemas.xmlsoap.org/wsdl/" location="
       processRole.wsdl" namespace="beispiel2"/>
   <bpel:import importType="http://schemas.xmlsoap.org/wsdl/" location="
       processRolePL.wsdl" namespace="beispiel2"/>
   <bpel:import importType="http://schemas.xmlsoap.org/wsdl/" location="
       processRole2.wsdl" namespace="beispiel2"/>
   <bpel:import importType="http://schemas.xmlsoap.org/wsdl/" location="
       processRole2PL.wsdl" namespace="beispiel2"/>
   <bpel:partnerLinks>
      <bpel:partnerLink myRole="processRole" name="processRolePartnerLink"
          partnerLinkType="ns1:processRolePartnerLink"/>
      <bpel:partnerLink myRole="processRole2" name="processRole2PartnerLink"
          partnerLinkType="ns1:processRole2PartnerLink"/>
   </bpel:partnerLinks>
   <bpel:variables>
      <!-- Message for the Message 'b2' -->
      <bpel:variable messageType="ns1:b2" name="b2"/>
      <!-- Message for the Message 'b3' -->
      <bpel:variable messageType="ns1:b3" name="b3"/>
      <!-- Message for the Message 'startB' -->
      <bpel:variable messageType="ns1:startB" name="startB"/>
      <!-- Message for the Message 'startA' -->
      <bpel:variable messageType="ns1:startA" name="startA"/>
      <!-- Variable for Properties of the Process 'beispiel2' -->
      <bpel:variable messageType="ns1:beispiel2_ProcessDataMessage" name="
          beispiel2_ProcessData"/>
   </bpel:variables>
   <!-- Begin Elements Block -->
   <bpel:sequence name="
       startAprocessStartpre_xorJOINeventxor2b4b2b5b3startBxorcde">
      <bpel:pick createInstance="yes" name="
          startAprocessStartpre_xorJOINeventxor2b4b2b5b3startB">
         <bpel:onMessage operation="startAoperation" partnerLink="
             processRolePartnerLink" portType="ns1:startPort" variable="startA">
            <bpel:sequence>
               <bpel:empty name="emptyForStartA"/>
            </bpel:sequence>
         </bpel:onMessage>
         <bpel:onMessage operation="startBoperation" partnerLink="
             processRolePartnerLink" portType="ns1:startPort" variable="startB">
            <bpel:sequence>
               <bpel:pick name="eventxor2b4b2b5b3">
                  <bpel:onMessage operation="dosomething" partnerLink="
                      processRole2PartnerLink" portType="ns1:bankPort" variable="
                      b2">
                     <bpel:sequence>
                        <bpel:empty name="b4"/>
                     </bpel:sequence>
```

```
40                    </bpel:onMessage>
                      <bpel:onMessage operation="dosometh" partnerLink="
                          processRole2PartnerLink" portType="ns1:bankPort" variable="
                          b3">
42                       <bpel:sequence>
                            <bpel:empty name="b5"/>
44                       </bpel:sequence>
                      </bpel:onMessage>
46                 </bpel:pick>
               </bpel:sequence>
48          </bpel:onMessage>
         </bpel:pick>
50       <bpel:if name="xorcde">
            <bpel:condition>B</bpel:condition>
52          <bpel:empty name="e"/>
            <bpel:elseif>
54             <bpel:condition>A</bpel:condition>
               <bpel:sequence name="cd">
56                <!-- Generated out of CAB cd -->
                  <bpel:empty name="c"/>
58                <bpel:empty name="d"/>
               </bpel:sequence>
60          </bpel:elseif>
         </bpel:if>
62    </bpel:sequence>
   </bpel:process>
```

Listing 5.9: Prozess 2 - Eine der zwei erzeugten WSDL Dateien sowie die zugehörige `partnerLinkType` Definition

```
   <?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
2  <wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="
       beispiel2" xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="processRole"
       targetNamespace="beispiel2">
      <!-- Message Variables for processRole-->
4     <wsdl:message name="startB"></wsdl:message>
      <wsdl:message name="startA"></wsdl:message>
6     <!-- End Message Variables-->
      <!-- PortTypes for processRole-->
8     <wsdl:portType name="startPort">
         <wsdl:operation name="startBoperation">
10          <wsdl:input message="tns:startB"/>
         </wsdl:operation>
12       <wsdl:operation name="startAoperation">
            wsdl:input message="tns:startA"/>
14       </wsdl:operation>
      </wsdl:portType>
16 </wsdl:definitions>

18 <?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
   <wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:plnk2="http
       ://docs.oasis-open.org/wsbpel/2.0/plnktype" xmlns:tns="beispiel2" name="
       processRolePL" targetNamespace="beispiel2">
20    <wsdl:import location="processRole.wsdl" namespace="beispiel2"/>
      <plnk2:partnerLinkType name="processRolePartnerLink">
22       <plnk2:role name="processRole" portType="tns:startPort"/>
      </plnk2:partnerLinkType>
24 </wsdl:definitions>
```

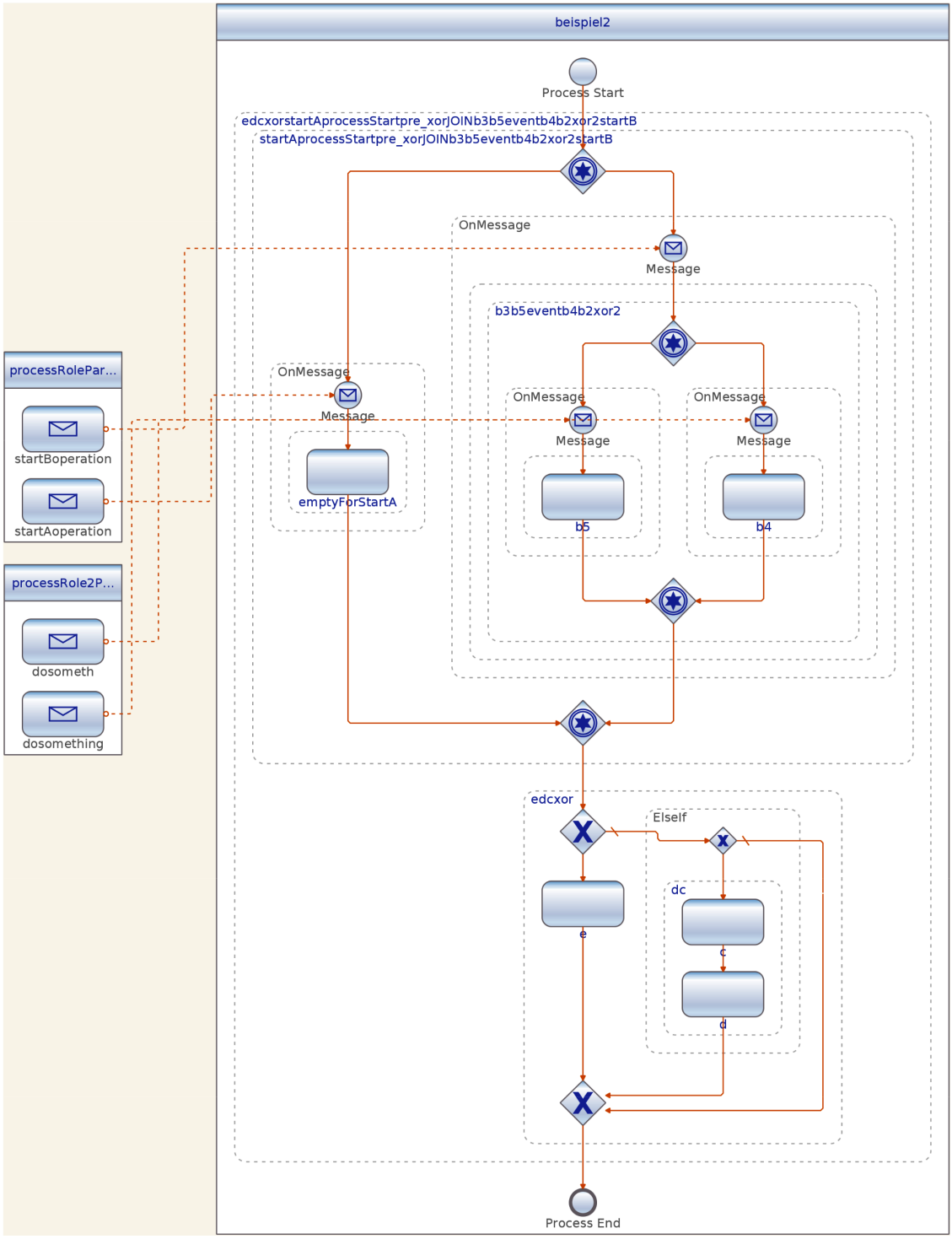


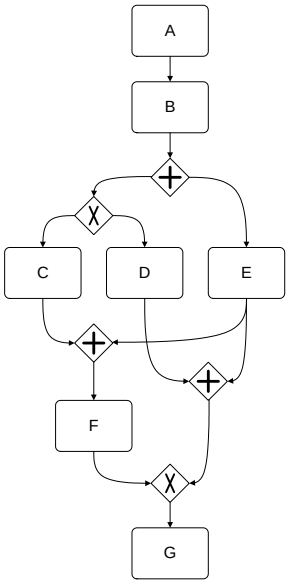
Abbildung 5.8: Prozess 2 - graphische Darstellung des BPEL Prozesses in NetBeans[3]

5.2 Nichtwohlstrukturierte Prozesse

In diesem Kapitel werden die beiden Transformationsmethoden für nichtwohlstrukturierte Strukturen anhand von zwei relativ einfachen Prozessen vorgestellt.

5.2.1 Prozess 3 - Prozess ohne Schleifen

Dieser, in Abbildung 5.9(a) dargestellte, auf den ersten Blick relativ einfache Prozess kann nicht alleine mit Hilfe der wohlstrukturierten CABs transformiert werden. Die verschachtelten *ParallelGateway* und *DataXor* Gateways verhindern dies. Der Prozess besteht auch in diesem Fall wieder nur aus *None* Elementen um das Ausgangsmodell und den erzeugten BPEL Prozess möglichst kompakt zu halten.



(a) BPMN Abbildung des Prozesses

```
<Diagram process36   name "Ein einfacher Prozess"
   <Pool testprocess36 name "first pool"
      <Process testprocess36 name "process36"
         <Properties
            <Property "a" "boolean">
            <Property "b" "boolean">
         >
         <None A>
         <None B>
         <None C>
         <None D>
         <None E>
         <None F>
         <None G>
         <ParallelGateway andSplit Gates sf3 sf4>
         <ParallelGateway andJoin1 Gates sf11>
         <ParallelGateway andJoin2 Gates sf13>
         <DataXor xorSplit Gates sf7 sf8>
         <DataXor xorJoin DefaultGate sf14>
         <SequenceFlow sf1 A B>
         <SequenceFlow sf2 B andSplit>
         <SequenceFlow sf3 andSplit xorSplit>
         <SequenceFlow sf4 andSplit E>
         <SequenceFlow sf5 E andJoin1>
         <SequenceFlow sf6 E andJoin2>
         <SequenceFlow sf7 xorSplit C ConditionType
             Expression "process36.a">
         <SequenceFlow sf8 xorSplit D ConditionType
             Expression "process36.b">
         <SequenceFlow sf9 C andJoin1>
         <SequenceFlow sf10 D andJoin2>
         <SequenceFlow sf11 andJoin1 F>
         <SequenceFlow sf12 F xorJoin>
         <SequenceFlow sf13 andJoin2 xorJoin>
         <SequenceFlow sf14 xorJoin G ConditionType
             Default>
      >
   >
>
```

(b) Der Prozess in BPMN DSL.

Abbildung 5.9: Prozess 3 - Der Prozess vor der Transformation

Transformation des BPD

Prozess auf die Transformation vorbereiten Keine der Methoden ist für den Prozess notwendig und daher nicht anzuwenden.

Schritt 1 Im ersten Schritt kann ein CAB vom Typen *Sequence* identifiziert werden. Dieses besteht aus den Elementen *A* und *B* und ist in Abbildung 5.11(a) dargestellt.

Schritt 2 Nach der Transformation aus dem vorigen Schritt kann kein wohlstrukturierte CAB mehr gefunden werden. Daher verändert sich das Modell einen Durchlauf lang nicht, was dazu führt, daß überprüft wird ob die erste nichtwohlstrukturierte Transformationsmethode anwendbar ist. Diese kann angewandt werden, da kein Element mehrmals aktiv wird und es keine unstrukturierten Schleifen gibt, folglich werden die entsprechenden Elemente in einem *Flow_ControlLink* CAB gekapselt. Die Elemente die gekapselt werden zeigt Abbildung 5.11(b). Anschliessend werden alle Vorbedingungen überarbeitet und die Elemente des CAB mit den entsprechenden Annotationen versehen, die in der Tabelle 5.10 aufgelistet sind.

Element	BPEL `transitionCondition`	BPEL `joinCondition`
pre_andSplit	-	-
C	`$testprocess36_ProcessData.a and not( $testprocess36_ProcessData.b )`	`true($pre_andSplit_LINK_C)`
D	`$testprocess36_ProcessData.b and not( $testprocess36_ProcessData.a )`	`true($pre_andSplit_LINK_D)`
E	-	`true($pre_andSplit_LINK_E)`
F	-	`( true($C_LINK_F) ) and ( true($E_LINK_F) )`
post_xorJoin	-	`( true($F_LINK_post_xorJoin) ) or ( ( true($D_LINK_post_xorJoin) ) and ( true($E_LINK_post_xorJoin) ) )`

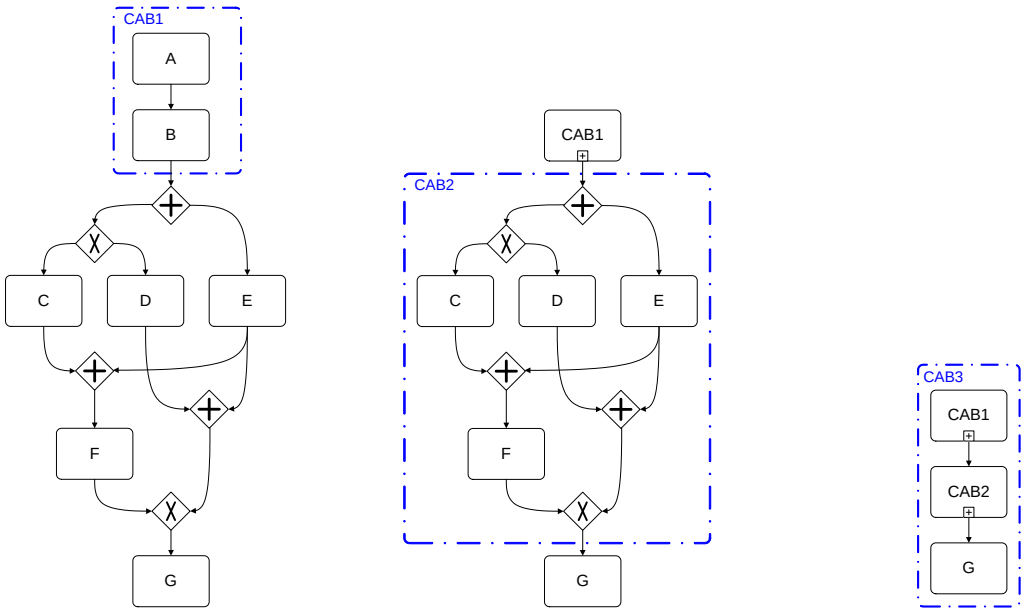
Abbildung 5.10: Die Vorbedingungen der Elemente des *Flow_ControlLink* CAB

Schritt 3 Im abschliessenden letzten Schritt wird ein CAB vom Typen *Sequence* identifiziert der aus den Elementen *CAB1*, *CAB2* und *G* besteht. Abbildung 5.11(c) stellt den Prozess unmittelbar zuvor dar.

BPEL Generieren

Listing 5.11 zeigt den aus dem transformierten Modell erzeugten BPEL Prozess. In Zeile 82 ist die abschliessende `joinCondition` zu erkennen, sie synchronisiert die zwei unterschiedlichen eingehenden Pfade wieder.

Abbildung 5.12 auf Seite 105 zeigt diesen Prozess wie er vom Editor von ActiveBpel[1] dargestellt wird. Deutlich zu erkennen ist der aus dem *Flow_ControlLink* generierte `flow`. Die Rauten auf den Verbindungslinien zwischen den einzelnen Aktivitäten repräsentieren die `transitionCondition` des jeweiligen `link`. Der BPEL Editor von NetBeans[3] unterstützt zum Zeitpunkt der Arbeit noch keine `link` Elemente.



(a) Erster Schritt (b) Zweiter Schritt (c) Dritter Schritt

Abbildung 5.11: Prozess 3 - Die Transformationsschritte

Listing 5.11: Prozess 3 - Vollständiger Quelltext des erzeugten BPEL Prozesses

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Process testprocess36 process36 -->
<bpel:process xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/
    executable" xmlns:ns1="testprocess36" xmlns:xsd="http://www.w3.org/2001/
    XMLSchema" exitOnStandardFault="yes" name="testprocess36" suppressJoinFailure=
    "yes" targetNamespace="testprocess36">
   <bpel:import importType="http://schemas.xmlsoap.org/wsdl/" location="
       testprocess36PV.wsdl" namespace="testprocess36"/>
   <bpel:variables>
      <!-- Variable for Properties of the Process 'process36' -->
      <bpel:variable messageType="ns1:testprocess36_ProcessDataMessage" name="
          testprocess36_ProcessData"/>
   </bpel:variables>
   <!-- Begin Elements Block -->
   <bpel:sequence name="
       andSplitandJoin2pre_andSplitxorJoinxorSplitCandJoin1EFpost_xorJoinDGBA">
      <!-- Generated out of CAB
          andSplitandJoin2pre_andSplitxorJoinxorSplitCandJoin1EFpost_xorJoinDGBA
          -->
      <bpel:empty name="A"/>
      <bpel:empty name="B"/>
      <bpel:flow name="
          andSplitandJoin2pre_andSplitxorJoinxorSplitCandJoin1EFpost_xorJoinD">
         <bpel:links>
            <bpel:link name="E_LINK_F"/>
            <bpel:link name="E_LINK_post_xorJoin"/>
            <bpel:link name="F_LINK_post_xorJoin"/>
            <bpel:link name="pre_andSplit_LINK_D"/>
            <bpel:link name="pre_andSplit_LINK_E"/>
            <bpel:link name="pre_andSplit_LINK_C"/>
            <bpel:link name="C_LINK_F"/>
            <bpel:link name="D_LINK_post_xorJoin"/>
         </bpel:links>
```

```
            <!-- ParallelGateway andSplit 2 Paths: [sf3, sf4] -->
            <!-- ParallelGateway andJoin1 1 Paths: [sf11]     -->
            <bpel:empty name="E">
               <bpel:targets>
                  <bpel:joinCondition>true($pre_andSplit_LINK_E)</bpel:joinCondition
                      >
                  <bpel:target linkName="pre_andSplit_LINK_E"/>
               </bpel:targets>
               <bpel:sources>
                  <bpel:source linkName="E_LINK_F">
                     <bpel:transitionCondition>true()</bpel:transitionCondition>
                  </bpel:source>
                  <bpel:source linkName="E_LINK_post_xorJoin">
                     <bpel:transitionCondition>true()</bpel:transitionCondition>
                  </bpel:source>
               </bpel:sources>
            </bpel:empty>
            <bpel:empty name="F">
               <bpel:targets>
                  <bpel:joinCondition>( true($C_LINK_F) ) and ( true($E_LINK_F) )</
                      bpel:joinCondition>
                  <bpel:target linkName="E_LINK_F"/>
                  <bpel:target linkName="C_LINK_F"/>
               </bpel:targets>
               <bpel:sources>
                  <bpel:source linkName="F_LINK_post_xorJoin">
                     <bpel:transitionCondition>true()</bpel:transitionCondition>
                  </bpel:source>
               </bpel:sources>
            </bpel:empty>
            <!-- ParallelGateway andJoin2 1 Paths: [sf13] -->
            <bpel:empty name="pre_andSplit">
               <bpel:sources>
                  <bpel:source linkName="pre_andSplit_LINK_D">
                     <bpel:transitionCondition>process36.b and not( process36.a and
                         not( process36.b )  )</bpel:transitionCondition>
                  </bpel:source>
                  <bpel:source linkName="pre_andSplit_LINK_E">
                     <bpel:transitionCondition>true()</bpel:transitionCondition>
                  </bpel:source>
                  <bpel:source linkName="pre_andSplit_LINK_C">
                     <bpel:transitionCondition>process36.a and not( process36.b )</
                         bpel:transitionCondition>
                  </bpel:source>
               </bpel:sources>
            </bpel:empty>
            <!-- DataXor xorJoin 0 Paths: []          -->
            <!-- DataXor xorSplit 2 Paths: [sf7, sf8] -->
            <bpel:empty name="C">
               <bpel:targets>
                  <bpel:joinCondition>true($pre_andSplit_LINK_C)</bpel:joinCondition
                      >
                  <bpel:target linkName="pre_andSplit_LINK_C"/>
               </bpel:targets>
               <bpel:sources>
                  <bpel:source linkName="C_LINK_F">
                     <bpel:transitionCondition>true()</bpel:transitionCondition>
                  </bpel:source>
               </bpel:sources>
            </bpel:empty>
            <bpel:empty name="post_xorJoin">
               <bpel:targets>
                  <bpel:joinCondition>( true($F_LINK_post_xorJoin) ) or ( ( true(
                      $D_LINK_post_xorJoin) ) and ( true($E_LINK_post_xorJoin) )  )</
                      bpel:joinCondition>
                  <bpel:target linkName="D_LINK_post_xorJoin"/>
```

```
84                  <bpel:target linkName="E_LINK_post_xorJoin"/>
                    <bpel:target linkName="F_LINK_post_xorJoin"/>
86              </bpel:targets>
            </bpel:empty>
88          <bpel:empty name="D">
                <bpel:targets>
90                  <bpel:joinCondition>true($pre_andSplit_LINK_D)</bpel:joinCondition
                        >
                    <bpel:target linkName="pre_andSplit_LINK_D"/>
92              </bpel:targets>
                <bpel:sources>
94                  <bpel:source linkName="D_LINK_post_xorJoin">
                        <bpel:transitionCondition>true()</bpel:transitionCondition>
96                  </bpel:source>
                </bpel:sources>
98          </bpel:empty>
        </bpel:flow>
100     <bpel:empty name="G"/>
    </bpel:sequence>
102 </bpel:process>
```

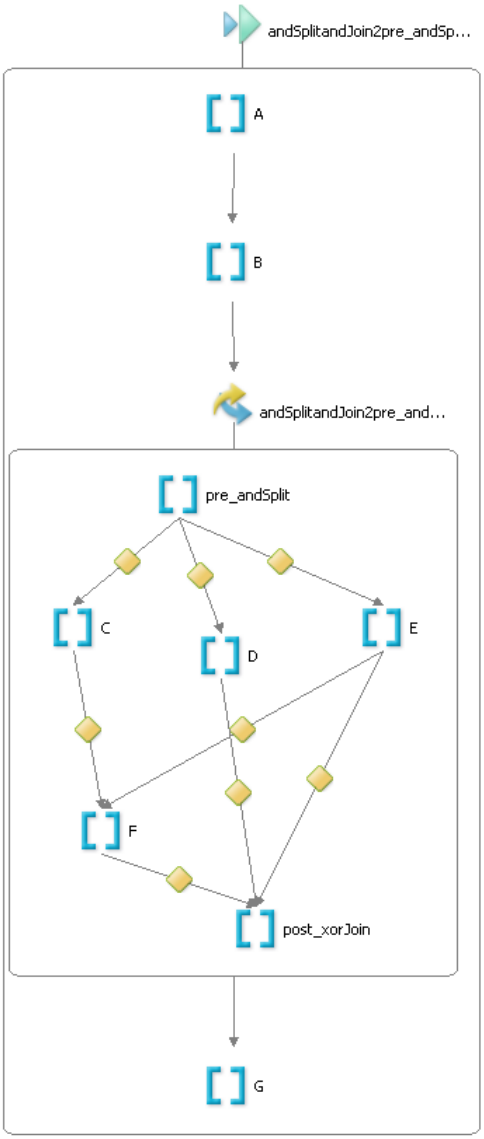
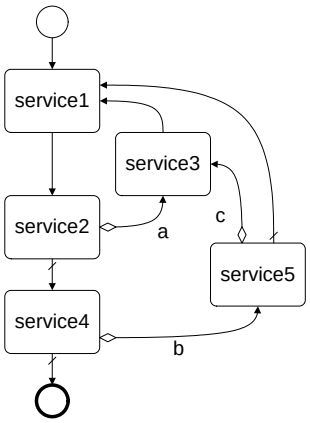


Abbildung 5.12: Prozess 3 - graphische Darstellung des BPEL Prozesses im Editor von ActiveBpel[1]

5.2.2 Prozess 4 - Prozess mit Schleifen

Der Prozess aus Abbildung 5.13(a) wird verwendet um die Transformation eines BPMN Modells mit unstrukturierten Schleifen zu demonstrieren. Er ist möglichst einfach gehalten um den erzeugten BPEL Prozess verständlich zu machen.



(a) BPMN Abbildung des Prozesses

```
<Diagram while63  name "while63"
   <Pool
   while63 name "while"
      <Process while63 name "pr"
      <Properties
            <Property "a" "boolean">
            <Property "b" "boolean">
            <Property "c" "boolean">
         >
            <StartEvent start  triggerType None>
            <SequenceFlow s1 start service1>
            <None service1>
            <SequenceFlow s2 service1 service2>
            <None service2>
            <SequenceFlow s3 service2 service3
                ConditionType Expression "
                $while63_ProcessData.a">
            <SequenceFlow s5 service2 service4
                ConditionType Default>
            <None service3>
            <SequenceFlow s4 service3 service1>
            <None service4>
            <SequenceFlow s6 service4 endevent
                ConditionType Default>
            <SequenceFlow s7 service4 service5
                ConditionType Expression "
                $while63_ProcessData.b">
            <None service5>
            <SequenceFlow s8 service5 service3
                ConditionType Expression "
                $while63_ProcessData.c">
            <SequenceFlow s10 service5 service1
                ConditionType Default>
            <EndEvent endevent  triggerType None>
      >
   >
>
```

(b) Der Prozess in BPMN DSL.

Abbildung 5.13: Prozess 4 - Der Prozess vor der Transformation

Transformation des BPD

Prozess auf die Transformation vorbereiten Zwei Methoden können auf den Prozess angewandt werden. Sie verändern jedoch nicht die grundlegende Struktur des Prozesses und haben damit, in diesem speziellen Fall, keine Auswirkung auf den Transformationsablauf. Beide werden daher nicht näher beschrieben oder abgebildet.

Schritt 1 Der Prozess enthält keine wohlstrukturierten Komponenten, folglich findet im ersten Schleifendurchlauf keine Veränderung des Modells statt. Auch die im letzten

Beispiel angewandte Methode kann hier nicht zum Einsatz kommen. Daher wird der Prozess in ein *EventTranslation* gekapselt. Anschliessend werden die Elemente des CABs annotiert. Dabei wird der Vorgänger und, sofern eine existiert, die Bedingung gespeichert die erfüllt sein muß um das jeweilige Element auszulösen. Die Bedingungen für die Elemente dieses Prozesses sind in Abbildung 5.15 aufgelistet.

Die Transformation des Modells ist mit diesem Schritt abgeschlossen wie in Abbildung 5.14 dargestellt.

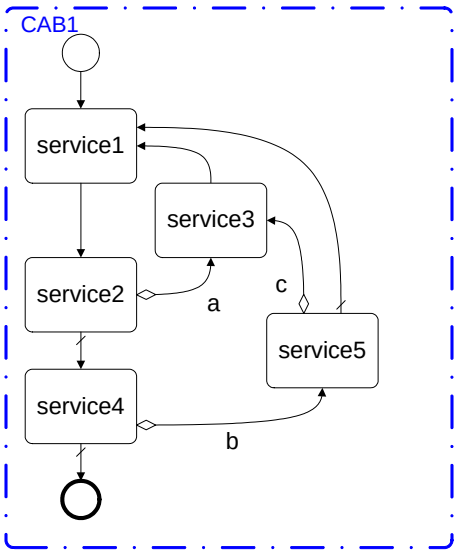


Abbildung 5.14: Prozess 4 - Der Transformationsschritt

Element	Vorgänger	Bedingung
service1	*start*	-
	service3	-
	service5	not(c)
service2	*service1*	-
service3	*service2*	a
	service5	c
service4	*service2*	not(a)
service5	*service4*	b

Abbildung 5.15: Die Vorbedingungen für die Elemente des *EventTranslation* CAB.

BPEL Generieren

Die Erzeugung des BPEL Prozesses ist auch in diesem Fall nicht sonderlich aufwendig. So werden zuerst, wie in den vorigen Beispielen, die einleitenden Tags des Prozesses generiert. Anschliessend werden alle *Activity* Elemente des *EventTranslation* CABs abgearbeitet. Dabei wird nun für jeden Vorgänger der *Activity* ein `onEvent` Element innerhalb des `eventHandler` erzeugt. In diesem Fall werden 8 `onEvent` Blöcke erzeugt, analog zu der Tabelle der Vorbedingungen 5.15. Der Inhalt aller `onEvent` Elemente einer *Activity* ist bis auf die auslösende Nachricht gleich. Abbildung 5.16 zeigt die graphische Darstellung des so erzeugten BPEL Prozesses im Editor von NetBeans[3]. Listing 5.13 zeigt die auf die wesentlichen Elemente reduzierte Struktur des Prozesses, der komplette Quelltext findet sich in Listing 5.14.

Listing 5.13: Prozess 4 - Vereinfachte Version des erzeugten BPEL Prozesses

```
<scope>
  <eventHandlers>
    <onEvent operation="Completion_service2_TO_service4" >
      <scope>
        <empty name="service4"/>
        <if><condition>pr.b</condition>
          <invoke  operation="Completion_service4_TO_service5"/>
        </if>
      </scope>
    </onEvent>
    <onEvent operation="Completion_service4_TO_service5" >
      <scope>
        <empty name="service5"/>
        <flow name="service5_completion">
          <if><condition>not( pr.d )</condition>
            <invoke operation="Completion_service5_TO_service1"/>
          </if>
          <if><condition>pr.d</condition>
            <invoke operation="Completion_service5_TO_service3"/>
          </if>
        </flow>
      </scope>
    </onEvent>
    <onEvent operation="Completion_service1_TO_service2" >
      <scope>
        <empty name="service2"/>
        <flow name="service2_completion">
          <if><condition>pr.a</condition>
            <invoke  operation="Completion_service2_TO_service3"/>
          </if>
          <if><condition>not( pr.a )</condition>
            <invoke operation="Completion_service2_TO_service4"/>
          </if>
        </flow>
      </scope>
    </onEvent>
    <onEvent operation="Completion_service2_TO_service3">
      <scope>
        <empty name="service3"/>
        <invoke  operation="Completion_service3_TO_service1"/>
      </scope>
    </onEvent>
    <onEvent operation="Completion_service5_TO_service3">
      <scope>
        <empty name="service3_2"/>
        <invoke operation="Completion_service3_TO_service1" />
```

```
            </scope>
48        </onEvent>
          <onEvent operation="Completion_start_TO_service1">
50           <scope>
                <empty name="service1"/>
52              <invoke operation="Completion_service1_TO_service2"/>
             </scope>
54        </onEvent>
          <onEvent operation="Completion_service3_TO_service1">
56           <scope>
                <empty name="service1_2"/>
58              <invoke operation="Completion_service1_TO_service2" />
             </scope>
60        </onEvent>
          <onEvent operation="Completion_service5_TO_service1">
62           <scope>
                <empty name="service1_4"/>
64              <invoke operation="Completion_service1_TO_service2"/>
             </scope>
66        </onEvent>
       </eventHandlers>
68     <sequence name="start_container">
          <invoke operation="Completion_start_TO_service1"/>
70     </sequence>
    </scope>
```

Listing 5.14: Prozess 4 - Vollständiger Quelltext des erzeugten BPEL Prozesses

```
   <?xml version="1.0" encoding="UTF-8"?>
2  <bpel:process xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/
       executable" xmlns:ns1="while63" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       exitOnStandardFault="yes" name="while63" suppressJoinFailure="yes"
       targetNamespace="while63">
     <bpel:import importType="http://schemas.xmlsoap.org/wsdl/" location="while63PV
         .wsdl" namespace="while63"/>
4    <bpel:import importType="http://schemas.xmlsoap.org/wsdl/" location="localPl/
         local_service2_TO_service4.wsdl" namespace="while63"/>
     ...
6    <bpel:partnerLinks>
        <!-- local Partner Link for the Event Translation Approach for onEvent
            service2_TO_service4 -->
8       <bpel:partnerLink myRole="receiver" name="localPL_service2_TO_service4"
            partnerLinkType="ns1:service2_TO_service4" partnerRole="sender"/>
        ...
10   </bpel:partnerLinks>
     <bpel:variables>
12      <bpel:variable messageType="ns1:completedActivity_service2_TO_service4"
            name="completed_service2_TO_service4"/>
       ...
14      <!-- Variable for Properties of the Process 'pr' -->
        <bpel:variable messageType="ns1:while63_ProcessDataMessage" name="
            while63_ProcessData"/>
16   </bpel:variables>
     <!-- Begin Elements Block -->
18   <bpel:scope>
        <bpel:eventHandlers>
20         <bpel:onEvent operation="Completion_service2_TO_service4" partnerLink="
               localPL_service2_TO_service4" portType="ns1:
               localPT_service2_TO_service4">
              <bpel:scope>
22               <bpel:sequence name="service4_container">
                    <bpel:empty name="service4"/>
24                  <bpel:if name="precondition_of_Completion_service4_TO_service5"
                        >
                       <bpel:condition>
26                     pr.b
```

```
                    </bpel:condition>
28                  <bpel:invoke inputVariable="completed_service4_TO_service5"
                        name="service4_1_completed" operation="
                        Completion_service4_TO_service5" partnerLink="
                        localPL_service4_TO_service5" portType="ns1:
                        localPT_service4_TO_service5"/>
                    </bpel:if>
30              </bpel:sequence>
                </bpel:scope>
32          </bpel:onEvent>
            <bpel:onEvent operation="Completion_service4_TO_service5" partnerLink="
                localPL_service4_TO_service5" portType="ns1:
                localPT_service4_TO_service5">
34              <bpel:scope>
                    <bpel:sequence name="service5_container">
36                      <bpel:empty name="service5"/>
                        <bpel:flow name="service5_completion">
38                          <bpel:if name="
                                precondition_of_Completion_service5_TO_service1">
                            <bpel:condition>
40                          not( pr.d )
                        </bpel:condition>
42                              <bpel:invoke inputVariable="
                                    completed_service5_TO_service1" name="
                                    service5_1_completed" operation="
                                    Completion_service5_TO_service1" partnerLink="
                                    localPL_service5_TO_service1" portType="ns1:
                                    localPT_service5_TO_service1"/>
                            </bpel:if>
44                          <bpel:if name="
                                precondition_of_Completion_service5_TO_service3">
                            <bpel:condition>
46                          pr.d
                        </bpel:condition>
48                              <bpel:invoke inputVariable="
                                    completed_service5_TO_service3" name="
                                    service5_2_completed" operation="
                                    Completion_service5_TO_service3" partnerLink="
                                    localPL_service5_TO_service3" portType="ns1:
                                    localPT_service5_TO_service3"/>
                            </bpel:if>
50                      </bpel:flow>
                    </bpel:sequence>
52              </bpel:scope>
            </bpel:onEvent>
54          <bpel:onEvent operation="Completion_service1_TO_service2" partnerLink="
                localPL_service1_TO_service2" portType="ns1:
                localPT_service1_TO_service2">
                <bpel:scope>
56                  <bpel:sequence name="service2_container">
                        <bpel:empty name="service2"/>
58                      <bpel:flow name="service2_completion">
                            <bpel:if name="
                                precondition_of_Completion_service2_TO_service3">
60                          <bpel:condition>
                            pr.a
62                      </bpel:condition>
                                <bpel:invoke inputVariable="
                                    completed_service2_TO_service3" name="
                                    service2_1_completed" operation="
                                    Completion_service2_TO_service3" partnerLink="
                                    localPL_service2_TO_service3" portType="ns1:
                                    localPT_service2_TO_service3"/>
64                          </bpel:if>
                            <bpel:if name="
                                precondition_of_Completion_service2_TO_service4">
```

```
66                          <bpel:condition>
                          not( pr.a )
68                      </bpel:condition>
                            <bpel:invoke inputVariable="
                                completed_service2_TO_service4" name="
                                service2_2_completed" operation="
                                Completion_service2_TO_service4" partnerLink="
                                localPL_service2_TO_service4" portType="ns1:
                                localPT_service2_TO_service4"/>
70                      </bpel:if>
                    </bpel:flow>
72                </bpel:sequence>
              </bpel:scope>
74          </bpel:onEvent>
            <bpel:onEvent operation="Completion_service2_TO_service3" partnerLink="
                localPL_service2_TO_service3" portType="ns1:
                localPT_service2_TO_service3">
76              <bpel:scope>
                  <bpel:sequence name="service3_container">
78                    <bpel:empty name="service3"/>
                      <bpel:invoke inputVariable="completed_service3_TO_service1"
                          name="service3_1_completed" operation="
                          Completion_service3_TO_service1" partnerLink="
                          localPL_service3_TO_service1" portType="ns1:
                          localPT_service3_TO_service1"/>
80                </bpel:sequence>
              </bpel:scope>
82          </bpel:onEvent>
            <bpel:onEvent operation="Completion_service5_TO_service3" partnerLink="
                localPL_service5_TO_service3" portType="ns1:
                localPT_service5_TO_service3">
84              <bpel:scope>
                  <bpel:sequence name="service3_container">
86                    <bpel:empty name="service3_2"/>
                      <bpel:invoke inputVariable="completed_service3_TO_service1"
                          name="service3_3_completed" operation="
                          Completion_service3_TO_service1" partnerLink="
                          localPL_service3_TO_service1" portType="ns1:
                          localPT_service3_TO_service1"/>
88                </bpel:sequence>
              </bpel:scope>
90          </bpel:onEvent>
            <bpel:onEvent operation="Completion_start_TO_service1" partnerLink="
                localPL_start_TO_service1" portType="ns1:localPT_start_TO_service1">
92              <bpel:scope>
                  <bpel:sequence name="service1_container">
94                    <bpel:empty name="service1"/>
                      <bpel:invoke inputVariable="completed_service1_TO_service2"
                          name="service1_1_completed" operation="
                          Completion_service1_TO_service2" partnerLink="
                          localPL_service1_TO_service2" portType="ns1:
                          localPT_service1_TO_service2"/>
96                </bpel:sequence>
              </bpel:scope>
98          </bpel:onEvent>
            <bpel:onEvent operation="Completion_service3_TO_service1" partnerLink="
                localPL_service3_TO_service1" portType="ns1:
                localPT_service3_TO_service1">
100             <bpel:scope>
                  <bpel:sequence name="service1_container">
102                   <bpel:empty name="service1_2"/>
                      <bpel:invoke inputVariable="completed_service1_TO_service2"
                          name="service1_3_completed" operation="
                          Completion_service1_TO_service2" partnerLink="
                          localPL_service1_TO_service2" portType="ns1:
                          localPT_service1_TO_service2"/>
```

```
104                 </bpel:sequence>
                 </bpel:scope>
106           </bpel:onEvent>
              <bpel:onEvent operation="Completion_service5_TO_service1" partnerLink="
                  localPL_service5_TO_service1" portType="ns1:
                  localPT_service5_TO_service1">
108              <bpel:scope>
                    <bpel:sequence name="service1_container">
110                    <bpel:empty name="service1_4"/>
                       <bpel:invoke inputVariable="completed_service1_TO_service2"
                           name="service1_5_completed" operation="
                           Completion_service1_TO_service2" partnerLink="
                           localPL_service1_TO_service2" portType="ns1:
                           localPT_service1_TO_service2"/>
112                 </bpel:sequence>
                 </bpel:scope>
114           </bpel:onEvent>
           </bpel:eventHandlers>
116        <bpel:sequence name="start_container">
              <bpel:invoke inputVariable="completed_start_TO_service1" name="
                  start_completed" operation="Completion_start_TO_service1" partnerLink
                  ="localPL_start_TO_service1" portType="ns1:localPT_start_TO_service1"
                  />
118        </bpel:sequence>
        </bpel:scope>
120  </bpel:process>
```

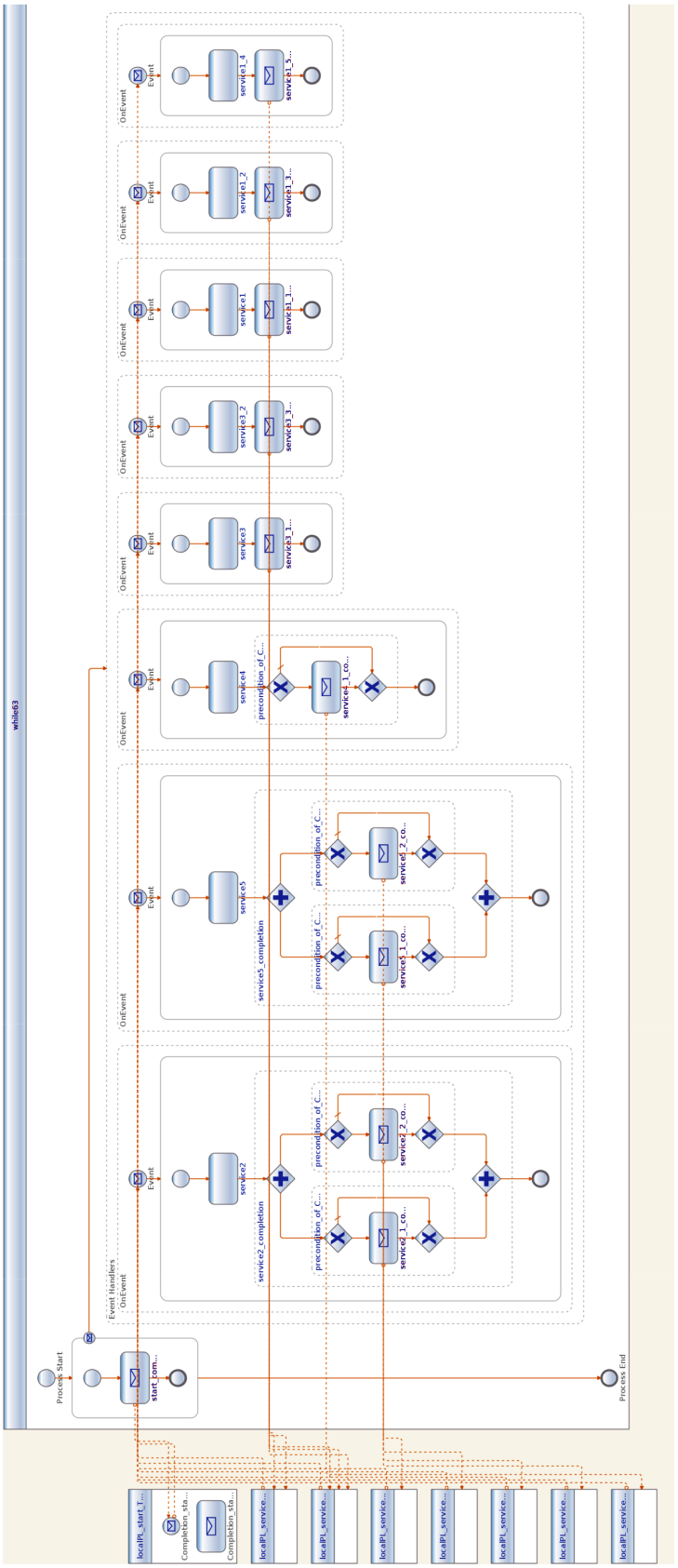


Abbildung 5.16: Prozess 4 - graphische Darstellung des BPEL Prozesses in NetBeans[3]

5.3 Testfälle

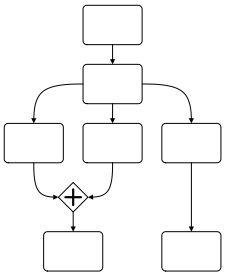
Im Zuge der Implentierung des Transformationsansatzes in oAW wurden ungefähr 400 unterschiedliche Prozesse mit Hilfe der BPMN DSL modelliert. Bei ungefähr der Hälfte dieser Prozesse handelt es sich um einfache Prozesse. Diese wurden hauptsächlich verwendet um die Check Regeln zu verfeinern und zu verhindern, daß illegale Strukturen oder nicht lauffähige Prozesse unerkannt bleiben.

Die andere Hälfte der Testprozesse entstand im Zuge der Verfeinerung der Transformationsmethoden und wurde dazu verwendet, die korrekte Transformation bestimmter Strukturen zu verifizieren.

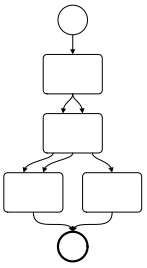
Die folgenden Abbildungen zeigen einen Ausschnitt der unterschiedlichen Klassen von verwendeten Testprozessen. Die Namen entsprechen den Dateinamen der BPMN Dateien.

5.3.1 Wohlstrukturierte Prozesse

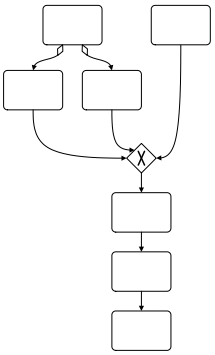
Bei einigen der hier abgebildeten Testprozesse handelt es sich um nichtwohlstrukturierte, die jedoch überarbeitet ausschließlich mit Hilfe von wohlstrukturierten CABs transformiert werden können.



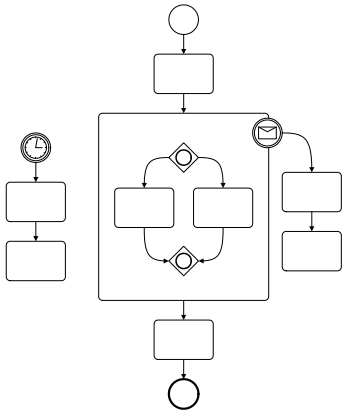
(a) andag3



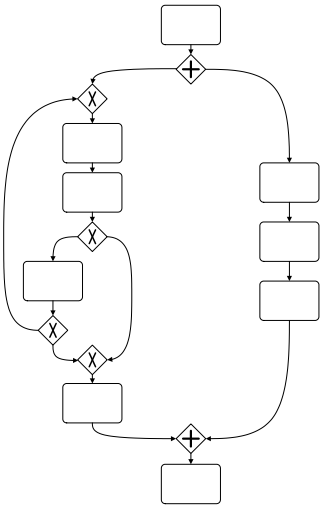
(b) andaa17



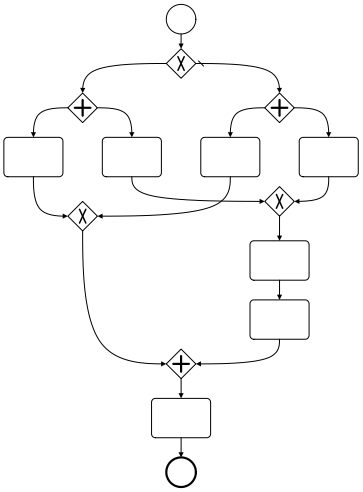
(c) start42



(d) intermediate7

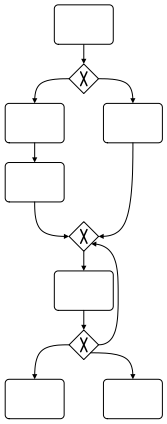


(e) process2

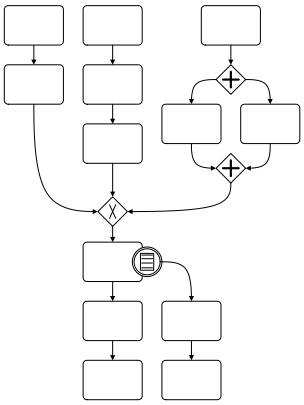


(f) process4

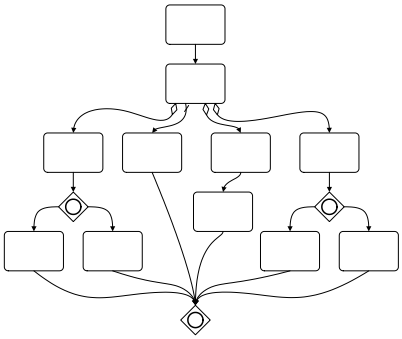
Abbildung 5.17: Wohlstrukturierte Testprozesse (a)-(f)



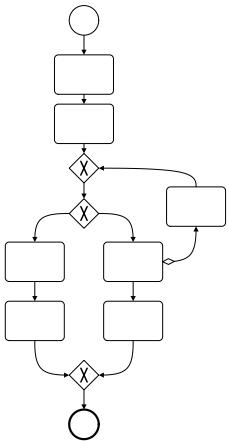
(g) while46



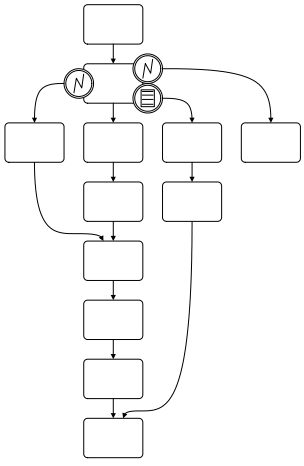
(h) start15



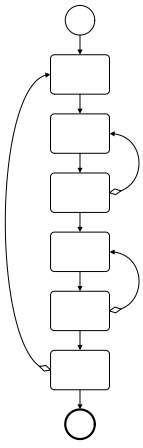
(i) orgg9



(j) while24

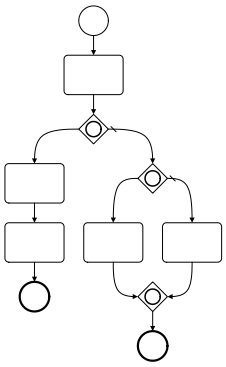


(k) exception16

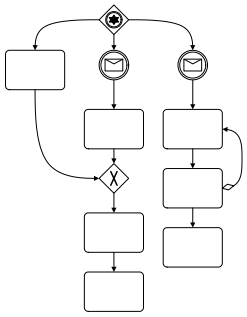


(l) while37

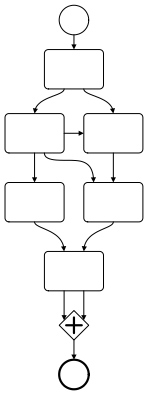
Abbildung 5.18: Wohlstrukturierte Testprozesse (g)-(l)



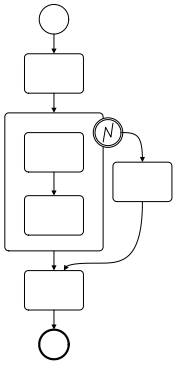
(m) orgg15



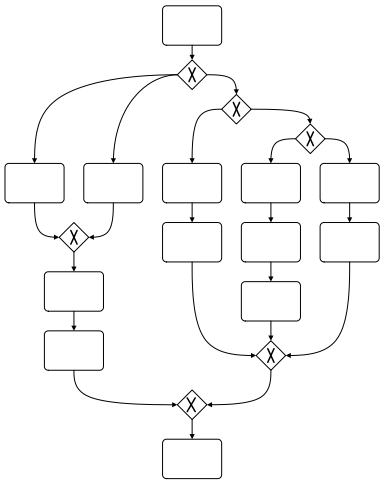
(n) pickgg11



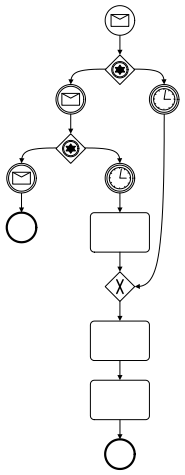
(o) process23



(p) embedded8



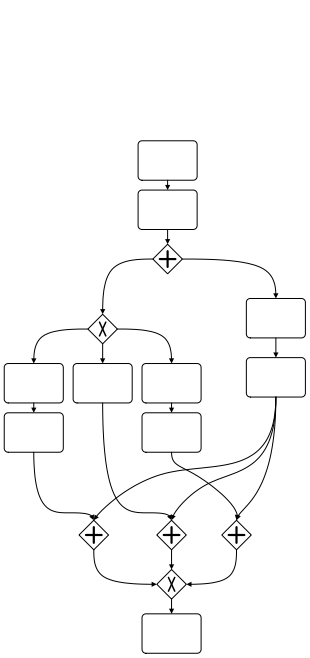
(q) xor27



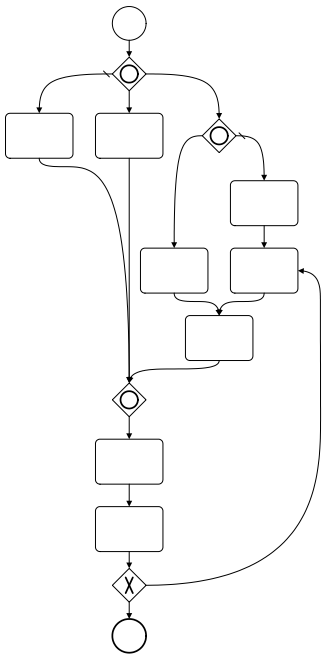
(r) pickgg21

Abbildung 5.19: Wohlstrukturierte Testprozesse (m)-(r)

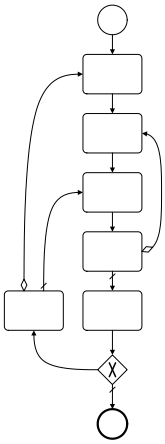
5.3.2 Nichtwohlstrukturierte Prozesse



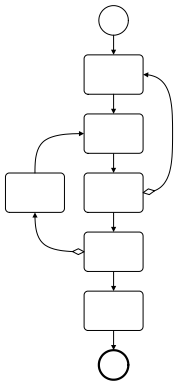
(a) process25



(b) orgg19



(c) process45



(d) while54

Abbildung 5.20: Nichtwohlstrukturierte Testprozesse (a)-(d)

6 Verwandte Arbeiten

Die Transformation eines unstrukturierten Modells in ein strukturiertes ist kein neues Problem, daher existiert eine große Anzahl von Arbeiten und Artikeln die sich der Problematik aus unterschiedlichen Blickwinkeln und mit unterschiedlichen Intentionen nähern. Unter diesen finden sich Arbeiten die sich mit der Generierung von BPEL Prozessen aus den unterschiedlichsten Modellierungssprachen, wie UML Aktivitätsdiagrammen, EPKs oder eben BPMN, beschäftigen. Ein großer Teil dieser Arbeiten bleibt jedoch abstrakt oder geht nur theoretisch auf die Transformation ein. Im folgenden Abschnitt 6.1 werden die wichtigsten Arbeiten kurz vorgestellt, um anschließend in Abschnitt 6.2 Unterschiede und Gemeinsamkeiten zum vorgestellten Transformationsansatz herauszuarbeiten.

6.1 Relevante verwandte Arbeiten

Die folgenden Artikel und Arbeiten haben einen direkten Bezug zu dem hier präsentierten Transformationsansatz oder behandeln relevante Probleme und Techniken. Es handelt sich keineswegs um eine vollständige Liste sondern nur um ausgesuchte Arbeiten.

- Kiepuszewski et al. gehen in [21] auf die möglichen Transformationen von syntaktisch nahezu unbeschränkten graphbasierten Modellen zu unterschiedlichen stark beschränkten ein. So werden unterschiedliche Möglichkeiten vorgestellt und diskutiert um Modelle mit unstrukturierte Schleifen in solche mit strukturierte Schleifen zu transformieren. Dabei werden aber keine Algorithmen an sich vorgestellt, es finden sich aber in den Abbildungen der Arbeit einige Transformationsansätze. Die Autoren zeigen, daß selbst einfache Transformationen von beliebigen Modellen in strukturierte, zumindestens die Verwendung von Hilfsvariablen erfordern. Aus ihrer Sicht wäre es ein Vorteil wenn ein WFMS beliebige Workflow Modelle unterstützen würde und nicht nur strukturierte.

- Der Transformationsansatz von Lassen und van der Aalst verwendet als Ausgangsmodell ein Petri-Netz. Der Algorithmus untersucht zuerst das Modell auf Strukturen die den `structured activities` entsprechen und transformiert diese. Sind diese nicht ausreichend für eine vollständige Transformation, greift er auf einen, dem im Abschnitt 3.2.6 gezeigten, sehr ähnlichen Ansatz zurück. Durch die Verwendung von Petri Netzen sind die Ausgangsmodelle wesentlich beschränkter, können aber auf ihre Korrektheit überprüft werden. Abbildung 6.1 zeigt einen verwendeten Beispielprozess. Die Stellen des Petri-Netzes enthalten Annotationen, die ihre BPEL Entsprechung beschreiben und während der Transformation ausgelesen werden. Der Transformationsansatz verwendet nur die `structured activities sequence`,

`switch`, `while`, `pick` und `flow`. Durch die Verwendung eines Petri-Netzes werden auch nur Modelle mit einem Start- und Endelement unterstützt. [24] [39]

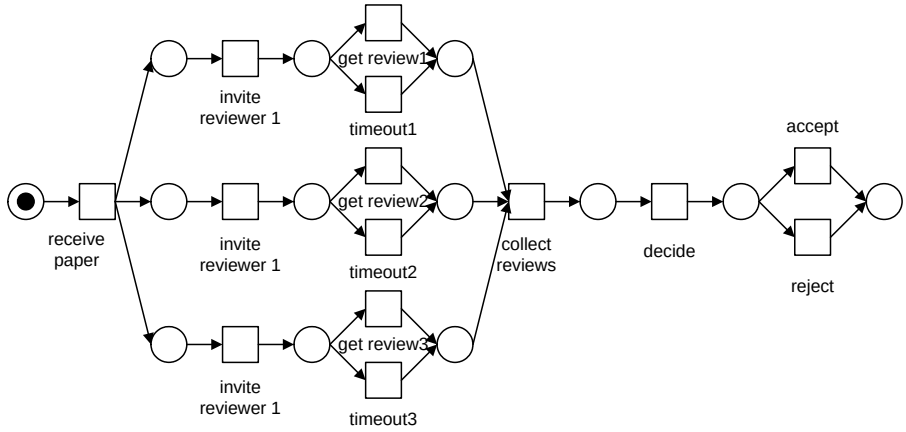


Abbildung 6.1: Die Beschreibung eines Peer-Review Prozesses in WF-Net wie es in [24] und [39] als Beispiel verwendet wird.

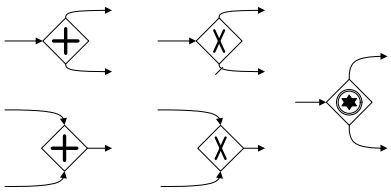
- Die Arbeit von Bordbar beschreibt sehr allgemein eine Transformation von UML Aktivitätsdiagramme in BPEL Prozesse. Die Transformationsregeln sind dabei in OCL beschrieben. Details des Transformationsprozesses fehlen mit Ausnahme einiger OCL Auszüge und es werden auch nur die Transformationen elementarer BPMN Elemente angegeben [11].

- Einer der Autoren von BPMN, S. White, geht bereits im Dokument der Spezifikation von BPMN auf die Transformation von BPMN Modellen in BPEL Prozesse ein. In einem weitere Artikel [42] beschreibt er, wie bestimmte BPMN Konstrukte mit Hilfe von `structured activities` dargestellt werden können. In beiden Fällen fehlt jedoch ein allgemeiner Ansatz, es handelt sich nur um Sammlungen von kleinen Beispielen für die Transformation bestimmter Elemente und Konstrukte. Zusätzlich sind einige der azyklischen Beispiele nicht optimal transformiert sondern verwenden nur einen großen `flow` und mehrere `link` Elemente. Die Arbeiten eignen sich aber gut um einen Einblick in die Modellierungsmöglichkeiten von BPMN zu bekommen die noch mit BPEL konform gehen.

- Der Ansatz von Ouyang [29] ist die Basis des in Kapitel 3.2.6 vorgestellten Ansatzes. Im Unterschied zu dieser Arbeit unterliegt das Ausgangsmodell jedoch wesentlich stärkeren Beschränkungen, es muss die Bedingungen eines wohlgeformtes BPD erfüllen, wie sie in Abschnitt 3.2.2 beschrieben sind. Das zugrundeliegende Metamodell unterstützt jedoch eine Teilmenge der Elemente von BPMN. Die unterstützten Elemente sind in Abbildung 6.2 dargestellt. Der Ansatz erlaubt also die Modellierung wesentlich realistischerer Prozesse und ist der mächtigste der gefundenen Transformationsansätze. Er unterstützt jedoch keine Fehlerbehandlung, keine Task Elemente mit mehreren ein- und ausgehenden Sequence Flows, keine

Gateways die gleichzeitig aufspaltend und vereinigend sind, keine OR Gateways, keine Subprozesse und keine multiplen Start- und Endelemente. Es werden auch keine Überarbeitungen vor der eigentlichen Transformation durchgeführt.

(a) Task (b) Unterstützte Event Typen



(c) Unterstützte Gateway Typen

Abbildung 6.2: Von [29] unterstützte Teilmenge von BPMN Elementen.

- Einen gänzlich anderen Ansatz, der auch unstrukturierte Schleifen in strukturierte transformieren kann, beschreiben Koehler und Hauser in ihren Arbeiten. Dieser Ansatz verwendet ein wohlgeformtes BPD als Ausgangsmodell und kompiliert aus diesem ein Programm mit "GOTOs". Auf diese Programme werden dann Methoden zu Elimination von GOTOs angewandt. Das erhaltene Programm wird anschließend in einen BPEL Prozess transformiert. Wie weit der Ansatz aber bei komplizierteren Ausgangsmodellen funktioniert, insbesonders Modellen mit Ausnahmebehandlung, wird nicht näher erläutert. Das verwendete Metamodell ist ein einfaches und in Abbildung 6.3 dargestellt. Als wohlgeformtes BPD unterliegt es ebenso den in Abschnitt 3.2.2 beschriebenen Einschränkungen. Dieser Ansatz könnte jedoch eventuell eine interessante Alternative sein, sofern er mit einem erweiterten Metamodell noch anwendbar ist[22] [19].

- Eine Erweiterung des obigen Ansatzes präsentiert Zhao gemeinsam mit Hauser in [47]. Der Ansatz wurde noch ein wenig verfeinert, das Metamodell und seine Einschränkungen blieben aber unverändert.

- Brahe beschreibt eine Architektur in der jede an der Entwicklung eines Prozesses beteiligte Rolle eine eigene DSL verwendet. So beschreibt der Analyst den Prozess in einem ersten Schritt in einer relativ einfachen DSL. Im nächsten Schritt wird diese Prozessbeschreibung vom Architekten erweitert und verfeinert. Im finalen Schritt wird aus der Prozessbeschreibung des Entwicklers der Code generiert. Jeder der Beteiligten erweitert also die Prozessbeschreibung seines Vorgängers um die für seine Rolle relevanten Details. Dabei verwendet er jeweils eine eigene DSL, die aber grundlegende Details mit den übrigen DSLs teilt. Der genaue Ablauf, die Details dieses Prozesses und ob diese Schritte auch in die umgekehrte Richtung funktionieren, werden jedoch nicht näher beschrieben.[12]

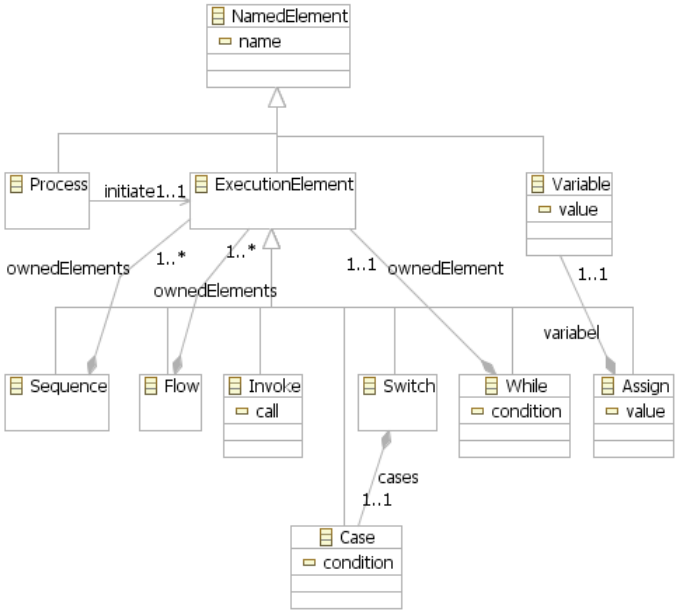


Abbildung 6.3: Das Metamodell der Arbeiten [22] und [19].

6.2 Vergleich mit verwandten Arbeiten

Bei einem Vergleich verwandter Transformationsmethoden und Ansätze mit dem in dieser Arbeit beschriebenen, treten einige Unterschiede zu Tage. So basiert keine der Arbeiten auf einem ähnlich umfangreichen **Metamodell**, die meisten der Arbeiten basieren auf einem abgewandelten Petri-Netz oder einem BPD. Die Prozesse sind damit entsprechend einfacher, dafür auch vor der Transformation formal verifizierbar. Damit einhergehend ist aber ein Verlust an **Semantik**, der stark vom verwendeten Metamodell abhängt, der jedoch auch für sich alleine betrachtet werden kann. Denn selbst unter den Petri-Netz basierten Metamodellen sind die Restriktionen nicht uniform. Generell existiert nur ein anderer Transformationsansatz der einen Teil der Elemente und Semantik von BPMN abdeckt. Selbst dieser geht jedoch nicht über die wichtigsten Elemente von BPMN hinaus und es ist auch nicht möglich Fehlerbehandlung oder Events zu modellieren. Keiner der gefundenen Ansätze erlaubt jedoch die Verwendung von nicht synchronisierenden aufspaltenden oder zusammenführenden Elementen die keine Gateways sind. Da diese aber von der BPMN Spezifikation bewusst nicht unterbunden worden sind, ist die Unterstützung derselben ein elementarer Punkt. Auch findet sich kein Transformationsansatz, der Prozesse mit mehreren Start- und Endelementen unterstützt.

Die tatsächliche **Implementierung** wird in den meisten Arbeiten nicht näher beschrieben, es findet sich nur eine Arbeit die näher auf die tatsächliche Implementierung der präsentierten Transformationsmethoden eingeht. Dabei handelt es sich um eine in Java implementierte Transformation von vorher in einem kommerziellen Modellierungstool erstellten Modellen. Durch die Verwendung von oAW bei der Implementierung des in dieser Diplomarbeit beschriebenen Transformationsansatzes und der damit einhergehen-

den modellgetriebenen Spezifikation des Metamodells sowie der Transformation mussten nur die eigentlichen Transformationsmethoden implementiert werden. Ohne Verwendung von oAW wäre wohl ein großer Teil der Zeit, die zur Entwicklung und Verfeinerung der Transformationsmethoden verwendet worden ist, in die Entwicklung und das Testen zugrundeliegender Bibliotheken und Funktionen geflossen. Durch das Fehlen einer genauen Beschreibung der Umsetzung der jeweils vorgestellten Transformationsansätzen in den Artikeln ist es schwer diese zu vergleichen, jedoch verwendet keiner der gefundener Transformationsansätze zur Beschreibung der Ausgangsprozesse eine DSL. Auch findet sich keine Information ob für die generierten BPEL Prozesse die entsprechenden WSDL Definition generiert werden.

Unterschiede im Transformationsprozess beginnen bereits bei der **Überarbeitung** des Prozesses. Ähnliche Schritte vor der Transformation finden in keiner der gefundenen Arbeiten statt. Gerade dieser Schritt sorgt jedoch dafür, daß ein nicht unbedeutender Teil der wohlstrukturierten Prozesse überhaupt als solche erkannt wird. Auch wäre eine Transformation von Prozessen mit multiplen Start- und Endelementen ansonsten nur schwer möglich. Die schrittweise Zerlegung und Kapselung von Teilen eines Prozesses wird mit mehreren Ansätzen geteilt. In vielen Fällen wird jedoch nach einer geringeren Anzahl von `structured activities` equivalenten Strukturen gesucht. Die beiden Ansätze für unstrukturierte Teile des Prozesses finden sich in mehreren Arbeiten wieder.

7 Conclusio

7.1 Resümee

Das primäre Ziel der Arbeit, einer automatischen Transformation eines gültigen BPMN Modells in einen ausführbaren BPEL Prozesss, wurde mit einer kleinen Einschränkungen erreicht. So kann Aufgrund des verwendeten Metamodells das Ausgangsmodell nicht formal auf seine Korrektheit verifiziert werden. Trotzdem kann der implementierte Generator eine große Klasse von illegalen oder fehlerhaften Prozessmodellen vorab erkennen. Aufgrund der Anzahl der Elemente des Metamodells, der relativ geringen Einschränkungen von BPMN sowie seiner nicht formal spezifizierten Restriktionen, ist eine vollständige Verifikation eines Prozessmodelles eine schwierige Aufgabe.

Durch die gezielte Auswahl und Anpassung bestehender Transformationsmethoden, wurde ein weiteres der gesteckten Ziele erreicht: die Semantik von BPMN auf das Metamodell zu übertragen. Es ist möglich, komplexe BPMN Prozesse zu formulieren und diese in ausführbare BPEL Prozesse zu transformieren. Gleichzeitig war es auch möglich die Restriktionen, denen die Prozessmodelle unterliegen, gegenüber vergleichbaren Transformationsansätzen wesentlich zu verringern. Insbesonders die Transformation von Prozessen mit multiplen Start- und Endelementen sowie nicht synchronisierenden zusammenführenden oder aufspaltenden Elementen konnte erfolgreich verwirklicht werden. Auch ist es möglich Prozesse mit Ausnahmebehandlung zu modellieren. Damit ist der Generator in der Lage eine wesentlich größere Klasse von BPMN Prozessmodellen als bisherige Transformationsmethoden abzudecken.

Aufgrund der modellgetriebenen Entwicklung der DSL, unter Verwendung der von oAW bereitgestellten Werkzeuge, war es möglich die Spezifikation des Metamodells relativ bald abzuschliessen. Damit stand mehr Zeit zur Entwicklung und Verfeinerung der eigentlichen Transformationsmethoden zur Verfügung und es konnte bei der Entwicklung ein Hauptaugenmerk auf den Entwurf und die Implementierung der eigentlichen Transformationsmethoden gelegt werden. Die Entwicklung von Teilaspekten des Generators, wie zum Beispiel Funktionen, die mit dem Einlesen sowie der Verifikation der Prozesse betraut sind, wurden durch die Verwendung des oAW Frameworks wesentlich erleichtert. Zusätzlich ist damit auch grundsätzlich eine spätere Erweiterung und Veränderung des Metamodells möglich, ohne den gesamten Transformationsprozess neu implementieren zu müssen.

Die Überarbeitung der Prozessmodelle vor der Transformation hat es bewirkt, einen wesentlich größeren Kreis von Prozessen, als ursprünglich für möglich gehalten wurde, nur mithilfe von `structured activities` alleine zu transformieren. Dies war insofern überraschend, als die meisten dieser Überarbeitungsmethoden nicht sonderlich kom-

plex sind, diese kleinen Änderungen jedoch große Auswirkungen auf das Ergebnis der Transformation haben.

7.2 Ausblick

Der Generator könnte noch in den folgenden Bereichen weiterentwickelt beziehungsweise verbessert werden.

Graphische Modellierung Eine logische Erweiterung wäre die Entwicklung eines graphischen Editors zum Entwurf der Prozesse. Dabei muss es sich nicht notwendigerweise um eine Eigententwicklung handeln, es wäre auch denkbar einen der bereits im Umfeld von Eclipse existierenden Editoren zu verwenden. Zu Beginn der Arbeit waren die Entwicklungswerkzeuge sowie die vorhandenen Editoren noch in einem Zustand, der diesen Schritt jedoch nicht ratsam erschienen ließ. Zum jetzigen Zeitpunkt gibt es jedoch sowohl einen vielversprechenden BPMN Editor[1] als auch verbesserte Entwicklungswerkzeuge samt zugehöriger Dokumentation.

Verbesserte Überarbeitung Die in dieser Arbeit vorgestellten Überarbeitungsschritte sind im Zuge von Prozessen entwickelt worden, die nicht erwartungsgemäß transformiert wurden. Es ist also durchaus möglich, daß noch weitere Methoden mit Hilfe neuer Testprozesse gefunden werden können.

Andere Transformationsmethoden Es existieren noch weitere Methoden zur Transformation unstrukturierter Prozesse in strukturierte die nicht implementiert wurden und die eventuell für bestimmte Konstruktionen bessere Resultate liefern.

Überarbeiten des Metamodells Eine weitere Option wäre die Erweiterung des Metamodells um Elemente, die bisher nur als String implementiert sind. Als Beispiel würde sich ein eigenes Metamodell zur Modellierung von Expressions anbieten. Momentan werden diese nur sehr grob überprüft. Ebenso könnte das bisherige Metamodell aber auch in mehrere kleine Metamodelle aufgespalten werden. Dieser Schritt würde das Modell sowohl übersichtlicher als auch einfacher zu warten werden lassen. Es wäre damit auch eventuell möglich die Transformation von BPMN unabhängiger ablaufen zu lassen und sie für andere graphbasierte Sprachen zu adaptieren. Zum Zeitpunkt der Entwicklung des Generators wurde aber ein kombiniertes Metamodell von oAW noch nicht unterstützt.

Verifikation der Modelle Ausbau der Regeln um die Anzahl erkannter illegaler Prozesse zu erhöhen. Gemeinsam mit einem verbesserten Metamodell, das auch Expressions abbildet, könnten die Modelle wesentlich detaillierter Überprüft werden.

BPMN Die Anpassung des Transformationsprozesses an zukünftige BPMN Versionen. Diese beseitigen optimalerweise auch die noch vorhandenen semantischen Doppeldeutigkeiten und offenen Punkte der Spezifikation.

[1] SOA Tools BPMN Modeler `http://www.eclipse.org/stp/bpmn/`

A CD

Auf der CD befinden sich alle vier zur Ausführung der Transformation benötigten Eclipse Projekte samt zugehöriger und konfigurierter Eclipse Umgebung.

A.1 Inhalt der CD

Diagramme der Testprozesse Im Verzeichnis ./visio-diagramme finden sich die Visio Diagramme aller Testprozesse.

Eclipse Im Wurzelverzeichnis der CD befinden sich zwei Archive mit den für die Ausführung der Transformation erforderlichen Eclipse Version samt oAW.

1. *eclipse-SDK-3.3.1-oAW-4.2-win32.zip*
2. *eclipse-SDK-3.3.1-oAW-4.2-linux-gtk.zip*

Projekte Ein Archiv mit den vier Eclipse Projekten liegt im Wurzelverzeichnis: *workspace.zip*

Pdf Ebenso findet sich im Wurzelverzeichnis eine Pdf Version dieses Dokuments.

A.2 Installation der oAW Umgebung

Die jeweiligen Eclipse Versionen sowie das Projekt Archiv in ein beliebiges Verzeichnis entpacken. Unter Linux ist ein Studium der Installationshilfe ratsam. Anschliessend Eclipse starten und unter File - Import `Existing Project Into Workspace` das Verzeichnis mit den Projekten wählen. Anschliessend die vier Projekte auswählen und den Import bestätigen.

A.3 Ausführen des Transformationsprozesses

Im Projekt *tuwien.bpmn2bpel.test* im Package `tuwien.bpmn2bpel.test` befinden sich alle für die Ausführung der Transformation relevanten Workflow- sowie Property-Dateien. Die Transformation kann auf zwei Arten gestartet werden. So existieren eine Workflow (*single-transform.oaw*) und eine Java Klasse (*RunTransformations.java*) für die Transformation. Die Java Klasse ist für die Batch Transformation zuständig, der oAW Workflow für einzelne Transformationen. Beide werden noch in Abschnitt A.3.1 beziehungsweise A.3.2 behandelt.

Sie können in Eclipse auf zwei Arten gestartet werden. Entweder indem die Datei im *Package Explorer* markiert und anschliessend über das Kontextmenü der rechten

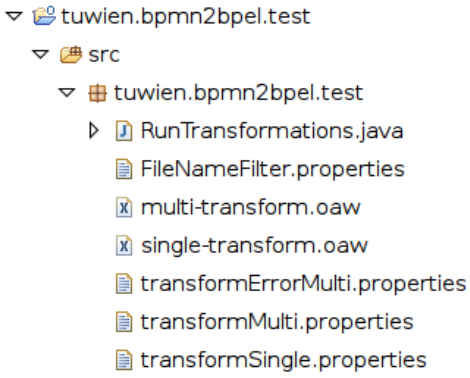


Abbildung A.1: Inhalt des Package `tuwien.bpmn2bpel.test`

Maustaste „Run As.." und entweder „1 oAW Workflow" oder „1 Java Application" auswählt und startet, wie in Abbildung A.2 gezeit, oder indem man sie über „Run" und „Open Run Dialog.." konfiguriert und anschliessend über die Dropdowns startet.

A.3.1 Batch-Transformation

Die Java Klasse *RunTransformations.java* ist für die sequentielle Transformation von mehreren Diagrammen zuständig. Die Datei verfügt über eine `main` Methode, benötigt aber sonst keine Argument. Denn die Pfade zum Quell- und Ausgabeverzeichnis werden in der Property Datei *transformMulti.properties* definiert. Die Transformation ist über die Optionen der Property Datei voreingestellt um möglichst wenig Konsolenoutput zu erzeugen. Diese Einstellung kann jedoch, wenn gewünscht, natürlich verändert werden, dies ist jedoch nicht zu empfehlen. In der Property Datei *FileNameFilter.properties* können zwei Filter auf die Dateinamen der Prozesse definiert werden um gezielt nur ausgewählte Prozesse zu transformieren. Ist die Property Datei leer oder sind die Parameter nicht gesetzt, werden alle Dateien mit gültiger Endung im Quellverzeichnis bei der Transformation berücksichtigt.

A.3.2 Einzel-Transformation

Alternativ kann für die Transformation von einzelnen Quelldateien der Workflow *single-transform.oaw* verwendet werden. Die Quelldatei sowie das Ausgabeverzeichnis werden in der Property Datei *transformSingle.properties* gesetzt. Dieser Workflow erzeugt umfangreiche Meldungen an der Konsole um den Ablauf der Transformation nachvollziehbar zu halten.

A.3.3 VM Optionen

Bei beiden Transformationen ist eine Veränderung der Java VM Parameter ratsam. Die Performanz der Transformation wird dabei wesentlich von dem verfügbaren Speicher bestimmt. Daher empfiehlt es sich in Run / Open Run Dialog unter der Kategorie

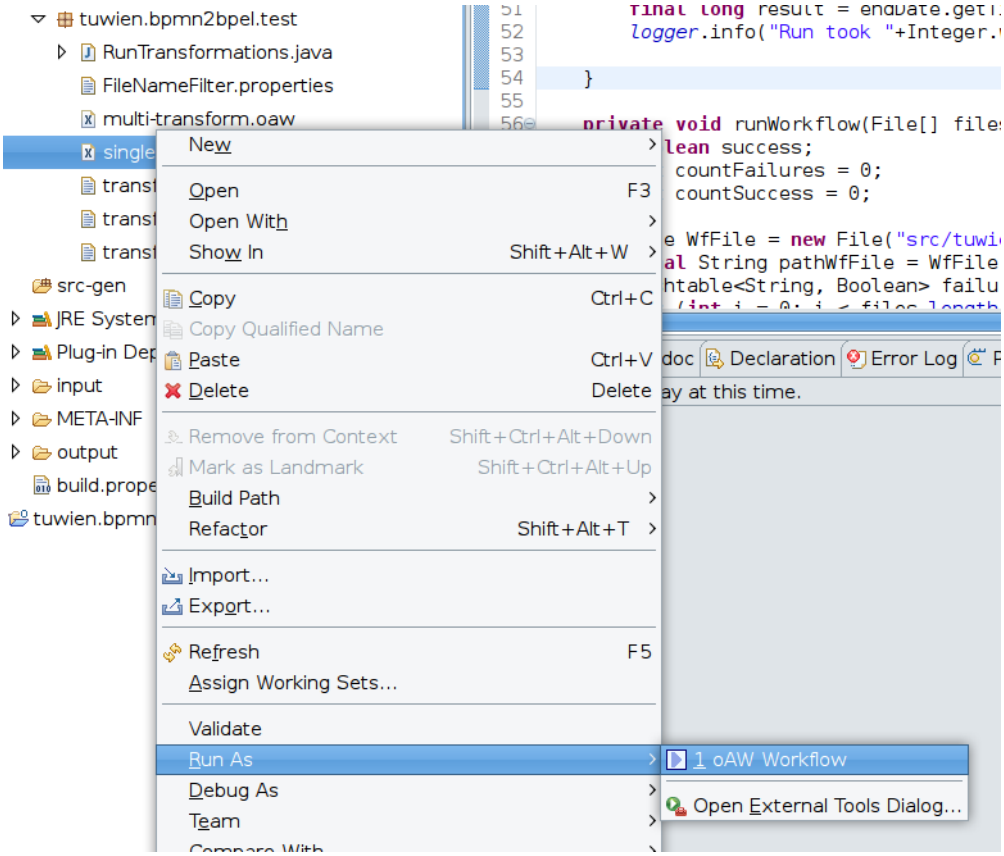


Abbildung A.2: Eine Möglichkeit die Transformation in Eclipse zu starten.

`Arguments` / `VM Arguments` den Wert `-Xmx500m` einzutragen. Dieser Wert legt den von der Java VM maximal reservierbaren Speicher auf 500 Megabyte fest. Insbesonders bei der Batch Transformation kann dieser Wert auch durchaus höher gesetzt werden, da dadurch die Anzahl an Garbage Collections, mitsamt dem verbundenen Zeiverlust, verringert werden kann. Durch setzten des Arguments `-Dcom.sun.management.jmxremote` kann die jeweilge Transformation mit Hilfe der `Jconsole` verfolgt werden.

Für die Transformationen wurden die folgenden Parameter verwendet:

```
-Xms150m
-Xmx500m
-XX:+AggressiveOpts
-XX:+UseFastAccessorMethods
```

(a) *single-transform.oaw*

```
-Xmx600m
-Xms200m
-XX:+AggressiveOpts
-XX:+UseFastAccessorMethods
-Dcom.sun.management.jmxremote
```

(b) *RunTransformations.java*

Abbildung A.3: Java VM Parameter für die beiden Transformationen

Literaturverzeichnis

Die Literaturangaben sind alphabetisch nach den Namen der Autoren sortiert. Bei mehreren Autoren wird nach dem ersten Autor sortiert. Alle Links der online verfügbaren Arbeiten waren am 4. Juni 2008 noch funktionsfähig.

[1] ActiveBpel. http://www.activevos.com. 45, 101, 105

[2] Business Process Management Initiative. http://www.bpmi.org/. 2

[3] NetBeans. http://www.netbeans.org/. 90, 92, 99, 101, 108, 113

[4] openArchitectureware. http://www.openarchitectureware.org/. 21

[5] workflowpatterns.com. http://www.workflowpatterns.com. 19

[6] Model Driven Architecture (MDA). http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01, 7 2001.

[7] A. Alamri, M. Eid, and A. El Saddik. Classification of the state-of-the-art dynamic web services composition techniques. http://inderscience.metapress.com/index/BULABFV5K7DP03TX.pdf, 2006. 8

[8] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, et al. Business Process Execution Language for Web Services, Version 1.1. http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf, 2003. 6

[9] A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, et al. DAML-S: Web Service Description for the Semantic Web. http://citeseer.ist.psu.edu/613592.html. 8

[10] A. Arkin, S. Askary, B. Bloch, IBM Francisco Curbera, BEA Yaron Goland, N. Kartha, S. Commerce, and O. Alex Yiu. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 4 2007. 6, 10

[11] B. Bordbar and A. Staikopoulos. On Behavioural Model Transformation in Web Services. http://www.springerlink.com/index/JJC02B30APUDV3CH.pdf, 2004. 120

[12] S. Brahe and B. Bordbar. A Pattern-based Approach to Business Process Modeling and Implementation in Web Services. http://www.springerlink.com/index/f564525007461160.pdf, 2006. 121

[13] A. Bucchiarone and S. Gnesi. A Survey on Services Composition Languages and Models. *International Workshop on Web Services–Modeling and Testing (WS-MaTe 2006)*, 2006. 7

[14] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, 3 2001. 6

[15] J. Eder and W. Gruber. A Meta Model for Structured Workflows Supporting Workflow Transformations. `http://www.springerlink.com/index/btpbm8ftjmv8hb4l.pdf`. 52

[16] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. SOAP Version 1.2. `http://www.w3.org/TR/soap/`, 4 2007. 6

[17] Hugo Haas and Allen Brown. Web Services Glossary. `http://www.w3.org/TR/ws-gloss/`, 2 2004. 6

[18] R. Hauser, M. Friess, J.M. Kuster, and J. Vanhatalo. Combining Analysis of Unstructured Workflows with Transformation to Structured Workflows. `http://ieeexplore.ieee.org/iel5/4031176/4031177/04031202.pdf`, 2006.

[19] R. Hauser and J. Koehler. Compiling Process Graphs into Executable Code. `http://www.zurich.ibm.com/~koe/papiere/gpce04.pdf`. 121, 122

[20] J. Hope and T. Hope. *Competing in the Third Wave: The Ten Key Management Issues of the Information Age*. Harvard Business School Press, 1997.

[21] B. Kiepuszewski, A.H.M. ter Hofstede, and C.J. Bussler. On Structured Workflow Modelling. `http://www.springerlink.com/index/R3B8597WGQGJPY6R.pdf`, 2000. 19, 21, 119

[22] J. Koehler and R. Hauser. Untangling Unstructured Cyclic Flows A Solution Based on Continuations. `http://www.springerlink.com/index/3252CE4QUFJ1UETU.pdf`, 2004. 121, 122

[23] U. Kuster, M. Stern, and B. Konig-Ries. A Classification of Issues and Approaches in Automatic Service Composition. 8

[24] K.B. Lassen and W.M.P. van der Aalst. WorkflowNet2BPEL4WS: A Tool for Translating Unstructured Workflow Processes in Readable BPEL. `http://www.springerlink.com/index/805910120q37855p.pdf`, 2006. 120

[25] F. Leymann, D. Roller, and M.T. Schmidt. Web services and business process management. `http://www.research.ibm.com/journal/sj/412/leymann.html`, 2002. 1

[26] F. Leymann, D. Roller, and S. Thatte. Goals of the BPEL4WS Specification. *xml. coverpages. org*, 2003. 8

[27] C. Ouyang, M. Dumas, S. Breutel, and A.H.M. ter Hofstede. Translating Standard Process Models to BPEL. *Proceedings of 18th International Conference on Advanced Information Systems Engineering (CAiSE 2006), June*, 2006. 30

[28] C. Ouyang, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. From Business Process Models to Process-oriented Software Systems: The BPMN to BPEL Way. 2006. 43

[29] C. Ouyang, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Translating BPMN to BPEL. `http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2006/BPM-06-02.pdf`, 2006. 47, 120, 121

[30] C. Peltz. Web services orchestration and choreography. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1236471`, 2003. 7

[31] J. Recker and J. Mendling. On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. `http://eprints.qut.edu.au/archive/00004637/`, 2006. 21

[32] W. Sadiq and M.E. Orlowska. Analyzing process models using graph reduction techniques. `http://linkinghub.elsevier.com/retrieve/pii/S0306437900000120`, 2000.

[33] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic composition of web services using semantic descriptions. `http://www.mindswap.org/papers/composition.pdf`, 2002. 8

[34] H. Smith. Business Process Management 101. *BPMI. org*, 2003.

[35] H. Smith and P. Fingar. *Business Process Management: The Third Wave*. Meghan-Kiffer Press, 2006. 1

[36] T. Stahl, M. Voelter, and K. Czarnecki. Model-Driven Software Development: Technology, Engineering, Management, 2006.

[37] WMP van der Aalst. Business Process Management: A personal view. *Business Process Management Journal*, 10(2):135–9, 2004.

[38] W.M.P. van der Aalst. Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management. `http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2004/BPM-04-03.pdf`, 2004.

[39] W.M.P. van der Aalst and K.B. Lassen. Translating Unstructured Workflow Processes to Readable BPEL: Theory and Implementation. `http://is.tm.tue.nl/staff/wvdaalst/publications/z23.pdf`, 2006. 27, 120

[40] W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske. Business Process Management: A Survey. *Business Process Management: International Conference, BPM 2003, Eindhoven, the Netherlands, June 26-27, 2003: Proceedings*, 2003.

[41] S. White. Process Modeling Notations and Workflow Patterns. *Workflow Handbook*, pages 265–294, 2004. 19

[42] S. White. Using BPMN to Model a BPEL Process. `http://www.businessprocesstrends.com/deliver_file.cfm?fileType=publication\&fileName=03-05WPMappingBPMNtoBPEL-White.pdf`, 2005. 120

[43] S.A. White et al. Business Process Modeling Notation. `http://www.bpmn.org/Documents/OMGFinalAdoptedBPMN1-0Spec06-02-01.pdf`, 2004. 3, 13, 16, 40

[44] P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. *Conceptual Modeling-Er 2003: 22nd International Conference on Conceptual Modeling, Chicago, IL, USA, October 13-16, 2003: Proceedings*, 2003. 19

[45] P. Wohed, WMP van der Aalst, M. Dumas, AHM ter Hofstede, and N. Russell. On the Suitability of BPMN for Business Process Modelling. *Business Process Management*, pages 161–176, 2006. 19

[46] M. Zairi. Business process management: a boundaryless approach to modern competitiveness. `http://www.emeraldinsight.com/Insight/viewContentItem.do?contentId=843392\&contentType=Article`, 1997. 1

[47] W. Zhao, R. Hauser, K. Bhattacharya, and B.R. Bryant. Compiling business processes: untangling unstructured loops in irreducible flow graphs. `http://inderscience.metapress.com/index/2PC7T6GC11DW0E05.pdf`, 2006. 121

[48] M. Zur Muehlen. *Workflow-Based Process Controlling: Foundation, Design, and Application of Workflow-Driven Process Information Systems*. Logos, 2004. 2

[49] M. zur Muehlen and J.C. Recker. How Much Language is Enough? Theoretical and Practical Use of the Business Process Modeling Notation. `http://eprints.qut.edu.au/archive/00012916/`, 2008. 18

[50] M. zur Muehlen and M. Rosemann. Multi-Paradigm Process Management. `http://workflow-research.de/Publications/PDF/MIZU.MIRO-BPMDS(2004).pdf`, 2004. 2

Symbolverzeichnis

BPD Business Process Diagram

BPEL Business Process Execution Language

BPMN Business Process Modeling Notation

BPM Business Process Management, siehe Kapitel 1

DSL domain specific language, domänenspezifischen Sprache

DTD Document Type Definition

EMF Eclipse Modelling Framework

M2M model to model, Modell-zu-Modell

M2T model to text, Modell-zu-Text

MDSD Model Driven Software Development

OCL Object Constraint Language

SPM Standard Process Models

WSDL Web Services Description Language, siehe Abschnitt 2.2.1

XML Extensible Markup Language

Index

BPEL Elemente
- `assign`, 10, 79, 83
- `basic activities`, 10
- `else`, 11
- `elseif`, 11, 39
- `empty`, 10
- `eventHandler`, 47, 48, 50, 108
- `exit`, 11
- `extensionActivity`, 11
- `faultHandler`, 90
- `flow`, 11, 12, 28, 33, 42, 44, 101, 120
- `forEach`, 11
- `if`, 11, 39
- `invoke`, 10
- `joinCondition`, 12, 42, 44, 101
- `link`, 12, 20, 42, 43, 45, 101, 120
- `none`, 12
- `onEvent`, 50, 108
- `onMessage`, 39
- `partnerLink`, 10, 12, 50, 67, 78, 85, 86, 93
- `partnerLinkType`, 93, 98
- `pick`, 11, 120
- `receive`, 10, 67
- `repatUntil`, 11
- `repeatUntil`, 11
- `reply`, 10
- `rethrow`, 11
- `scope`, 10, 12, 90
- `sequence`, 11, 28, 80, 81, 119
- `structured activities`, 10–12, 28, 30, 119, 120, 123, 124
- `switch`, 120
- `throw`, 10
- `transitionCondition`, 12, 44, 101
- `variable`, 12
- `wait`, 10
- `while`, 11, 20, 35, 120

BPM, 1
- lifecycle, 1

BPMN core elements, 14
- artifacts, 17
- connecting objects, 16
  - association, 17
  - message flow, 16
  - sequence flow, 16
- flow objects, 14
  - activity, 15
  - event, 15
  - gateway, 15
- swimlanes, 17

BPMN Elemente
- *IntermediateEvent*
  - Exception, 85
- *Activity*, 44, 50, 81, 108
- *Assignment*, 40, 83, 86, 88
- *Assingment*, 64
- *DataXor*, 36, 39, 40, 49, 87, 97, 100
- *Element*, 79
- *Event*, 47, 48
- *EventXor*, 38, 57, 69, 93, 97
- *FlowObject*, 25, 31, 78–80
- *Gate*, 64
- *Gateway*, 48
- *Implementation*, 64, 78, 93
- *IntermediateEvent*, 39, 40, 85, 86
- *Message*, 64
- *None*, 42, 43, 53, 56, 79, 81, 85, 100
- *ParallelGateway*, 48, 49, 53, 81, 100
- *Receive*, 57, 79, 93

SequenceFlow, 25, 30, 33, 37, 39, 40, 42, 43, 47, 48, 50, 53, 54, 64
conditional SequenceFlow, 36, 43, 67
default SequenceFlow, 34
default SequenceFlow, 36, 39, 43, 67
Subprocess, 28, 48, 78
Task, 48, 53, 64
Business Process Management, *siehe* BPM

CAB Definition, 30
CAB Typen
ActivityActivityFlow, 33
ActivityActivityOr, 34, 35
ActivityGatewayFlow, 33, 34
ActivityGatewayOr, 34
ActivityScope, 41, 86, 87, 90
EventTranslation, 107, 108
ExceptionFlow, 86, 87, 90
Flow_ControlLink, 101
GatewayActivityFlow, 33, 81
GatewayActivityOr, 34
GatewayGatewayFlow, 33, 34, 53, 54, 81
GatewayGatewayOr, 34–36
Sequence, 33, 80, 81, 86, 87, 93, 95, 96, 101
While, 36
XorSplitJoin, 39, 86, 87, 93, 95, 96

domänenspezifische Sprache, *siehe* DSL

IDE
ActiveBpel, 45, 101, 105
Eclipse, 22, 25, 62
NetBeans, 90, 92, 99, 101, 108, 113

Kompositionsmodelle, 7

Model Driven Software Development, 22
Modell-zu-Modell, 60
Modell-zu-Text, 60
Multimethoden, 23

oAW
Check, 24, 69, 70, 73, 114
Expression Language, 23
Workflow, 69
Xpand, 24, 72
Xtend, 23, 69, 76
Xtext, 22, 26

Petri Netz, 27
Prozess
BPMN
abstrakt, 13
global, 14
intern, 13
extern, 9
intern, 9
wohlstrukturiert, 97

Recipes, 25

Standard Process Model, 27

Template, 24

Web Service, 6
Choreographie, 7
Komposition, 6
Orchestrierung, 7
WSDL, 6