Transformation zwischen ausgewählten Diagrammtypen der Modellsprachen UML und BPMN für die Prozessdarstellung

Masterarbeit

vorgelegt von:	Christina Lenz
Studienbereich:	Informatik
Erstgutachter:	Professor Dr. Hans-Knud Arndt
Zweitgutachter:	Professor Dr. Klaus Turowski
Betreuer:	Dipl.-Wirt.-Inform. Torsten Urban

Kontakt:
Christina Lenz
lenz.christina@gmx.net

Kurzfassung

Business Process Management ist ein immer wichtigerer Faktor für heutige Unternehmen geworden und es gibt eine ganze Reihe von Modellierungsstandards, mit denen man Prozesse darstellen kann. Die Verwendung unterschiedlicher Modellierungsstandards führt zur Frage der Vergleichbarkeit und der Transformation von Modellen. Leider gibt es keine standardisierten, bidirektionalen Transformationswege zwischen den Modellierungsstandards, sodass Transformationen Fehler beinhalten und zu Informationsverlusten führen. In dieser Arbeit wurde ein bidirektionaler Transformationsweg zwischen ausgewählten Diagrammtypen der BPMN und der UML auf Basis des Workflow Plattern Framework erarbeitet. Dazu wurde zunächst betrachtet, welche Muster des Framework von beiden Standards unterstützt werden und dann wurde auf Grundlage dieser Ergebnisse eine XML-basierte Transformation erarbeitet. Um eine höhere Abdeckung der Notationselemente zu gewährleisten, wurden zusätzlich einige Transformationsregeln hinzugefügt. Der Transformationsweg wurde verifiziert und validiert, und die Ergebnisse sind zufriedenstellend. Der Ansatz kann auch auf Transformationen zwischen anderen Modellierungsstandards als den ausgewählten angewandt werden.

Abstract

Business Process Management has become an increasingly important factor for organizations, and there is a wide range of modeling standards which can display processes. The use of different modeling standards leads to the question of comparability and transformation of models. Unfortunately, there are no standardized bidirectional transformation paths between the modeling standards, so that transformations contain errors and lead to a loss of information. In this paper, a bidirectional transformation path between selected diagram types of the BPMN and the UML based on the Workflow Pattern Framework was developed. Initially, it was considered whether both standards supported the framework, and, corresponding with these results, an XML-based transformation was developed. To ensure greater coverage of the notation elements, some additional transformation rules have been created. The transformation path has been verified and validated, and the results are satisfying. The approach can be applied to transformations between modeling standards other than those selected.

Selbstständigkeitserklärung

Ich, Christina Lenz, versichere hiermit, dass ich meine Masterarbeit mit dem Thema

> *Transformation zwischen ausgewählten Diagrammtypen der Modellsprachen UML und BPMN für die Prozessdarstellung*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Mir ist bekannt, dass ich meine Masterarbeit zusammen mit dieser Erklärung fristgemäß nach Vergabe des Themas in zweifacher Ausfertigung und gebunden im Prüfungsamt der Otto-von-Guericke-Universität Magdeburg abzugeben oder spätestens mit dem Poststempel des Tages, an dem die Frist abläuft, zu senden habe.

Magdeburg, den 06. Januar 2014

Christina Lenz

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AD	Aktivitätsdiagramm
ARIS	Architektur integrierter Informationssysteme
ATL	ATLAS Translation Language
BPD	Business Process Diagram
BPEL	Business Process Execution Language
BPM	Business Process Management
BPMI	Business Process Management Initiative
BPML	Business Process Management Language
BPMN	Business Process Model and Notation
BPSS	Business Process Specification Schema
DIN	Deutsches Institut für Normung
EN	Europäische Norm
EPK	Ereignisgesteuerte Prozesskette
GoM	Grundsätze ordnungsgemäßer Modellierung
IBM	International Business Machines Corporation
IEEE	Institute of Electrical and Electronics Engineers
ISO	Internationale Organisation für Normung
LOI	Letter of Interest
M2M	Model-to-Model
MOLA	Model Transformation Language
OMG	Object Management Group
OMT	Object-Modeling Technique
OOSE	Object-Oriented Software Engineering
PAIS	Process Aware Information System
RFI	Request for Information
RFP	Request for Proposal
SysML	Systems Modeling Language
UML	Unified Modeling Language
WPF	Workflow Pattern Framework
WPI	Workflow Pattern Initiative
WCP	Workflow Control Pattern
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSD	XML Schema Definition
XSLT	Extensible Stylesheet Language Transformation

Listingverzeichnis

Tabellenverzeichnis

1 Einleitung

In diesem einleitenden Kapitel wird die Motivation für diese Arbeit erläutert sowie die Problem- und Zielstellung beschrieben. Außerdem erfolgt ein kurzer Überblick über den aktuellen Stand der Forschung und den weiteren Aufbau dieser Arbeit.

1.1 Motivation und Problematik

Unternehmen sehen sich zunehmend und in immer höheren Maße neuen Herausforderungen gegenüber, darunter Globalisierung und Internationalisierung der Märkte, gestiegene Ansprüche der Kunden und gesättigte Käufermärkte.[1] Um auf die neuen Anforderungen flexibel zu reagieren und die erforderlichen Anpassungen vornehmen zu können, hat sich das Geschäftsprozessmanagement als ein geeignetes und bewährtes Konzept herausgestellt.[2] Geschäftsprozessmanagement lässt sich auch als Business Process Management (BPM) bezeichnen.[3]

BPM ist in den letzten Jahren der Bereich mit der höchsten Priorität für die meisten Unternehmen geworden.[4] Dies liegt unter anderem am „steigende[n] Wettbewerbsdruck bezüglich Zeit, Kosten und Qualität“[5], die sich aus den oben genannten Herausforderungen ergeben. Dieser Meinung sind auch Ko et al.[6] und führen explizit den Bedarf nach einem schnellen Informationsaustausch, kurzen Entscheidungswegen, die Notwendigkeit, sich bei Nachfrage sofort anzupassen, sowie mehr internationale Konkurrenz auf. „Trotz der hohen Misserfolgsquoten und den kritischen Beurteilungen gibt es einen Konsens darüber, dass BPM-Projekte, wenn sie richtig und mit einem effektiven Einsatz von Informationstechnologie umgesetzt werden, bedeutende Fortschritte bei der Leistung von Organisationen tragen.“[7] BPM ist ein multidisziplinäres Gebiet, welches die Menschen, deren Arbeitsweise miteinander, die verwendete Technologie und die Zielvorgaben mit Bezug auf die Geschäftsprozesse behandelt.[8]

[1] Vgl. Schmelzer und Sesselmann [2010], S. 2
[2] Vgl. Schmelzer und Sesselmann [2010], S. 2
[3] Vgl. Schmelzer und Sesselmann [2010], S. 5
[4] Vgl. Alibabaei et al. [2009], S. 1, und Schmelzer und Sesselmann [2010], S. 3
[5] Vgl. Seidlmeier [2010], S. 1
[6] Vgl. Ko et al. [2009], S. 744f
[7] Vgl. Alibabaei et al. [2009], S. 2
[8] Vgl. Miers [2006], S. 1, und Alibabaei et al. [2009], S. 2f

Mit dem Konzept der *Architektur integrierter Informationssysteme* (ARIS) ist es möglich, ein solches BPM zu unterstützen. Es handelt sich dabei um ein in den Neunzigerjahren des vergangenen Jahrhunderts von August-Wilhelm Scheer entwickeltes „Rahmenwerk zur Beschreibung von Unternehmen und betriebswirtschaftlichen Anwendungssystemen“[9]. Dieses Rahmenwerk dient der Modellierung von Geschäftsprozessen in unterschiedlichen Sichten und auf verschiedenen Ebenen.

Grundsätzlich wird bei ARIS zwischen folgenden Sichten unterschieden: *Funktionssicht*, *Datensicht*, *Organisationssicht*, *Steuerungs-* beziehungsweise *Prozesssicht* und *Leistungssicht*. In der Funktionssicht werden fachliche Aufgaben und Tätigkeiten an einem Objekt beschrieben, die eines oder mehrere Ziele eines Unternehmens unterstützen. In der Datensicht hingegen werden reale oder abstrakte Dinge als Datenstruktur beschrieben, die für eine Aufgabe von Interesse sind (beispielsweise ein Lieferant mit Name oder eine Bestellung mit Bestellnummer). Zusätzlich werden die Beziehungen zwischen den Datenstrukturen untereinander dargestellt. Die Organisationssicht befasst sich mit der Aufbau- und Ablauforganisation, also konkret mit den Hierarchien in dem Unternehmen. Das Ziel dieser drei Sichten ist es, die Komplexität eines Prozesses zu reduzieren; allerdings gehen bei dieser Zerlegung die Beziehungen zwischen den einzelnen Komponenten verloren oder bleiben nur mit loser Kopplung bestehen.[10] Aus diesem Grund führt die vierte Sicht, die Prozesssicht, die anderen drei Sichten wieder zusammen, indem sie die Beziehungen zwischen diesen beschreibt. In der fünften Sicht, der Leistungssicht, werden materielle und immaterielle Leistungen behandelt, die von Prozessen benötigt oder erzeugt werden, zum Beispiel finanzielle Mittel und vermarktbare Produkte. Abbildung 1.1 zeigt das sogenannte ARIS-Haus bestehend aus den beschriebenen Sichten.

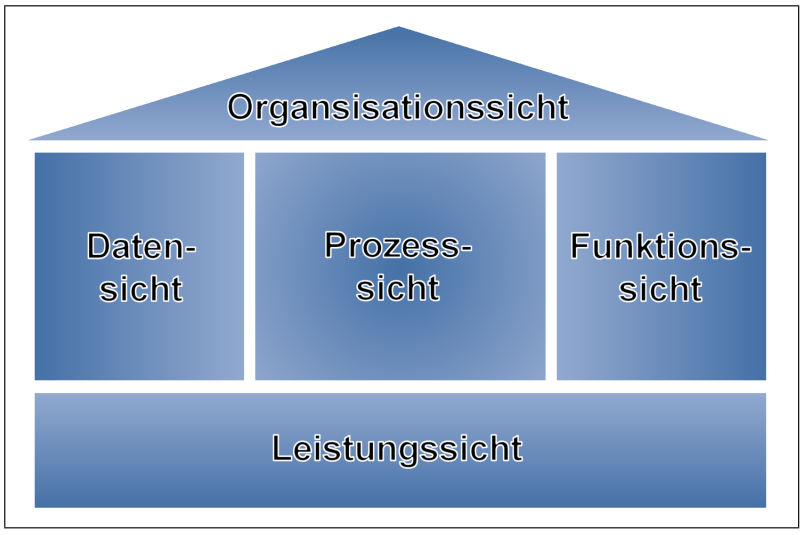


Abbildung 1.1: ARIS-Haus

[9] Vgl. Seidlmeier [2010], S. 12
[10] Vgl. Seidlmeier [2010], S. 13f

Im Rahmen dieser Arbeit liegt der Fokus auf der Prozesssicht, weil in ihr die Beziehungen zwischen den einzelnen Komponenten aus den anderen Sichten zusammengefasst werden und ein gesamtheitliches Bild des betrachteten Prozesses hergestellt wird.[11]

Um Prozesse darstellen und beschreiben zu können, haben sich seit den späten Achtzigerjahren des vergangenen Jahrhundert diverse Standards für das BPM entwickelt. Ko et al.[12] sprechen bildlich von einer Myriade und führen einige auf, wie die BPEL (Business Process Execution Language), BPSS (Business Process Specification Schema), Petri-Netze, Pi-Calculus, Rosetta-Netze, UML Aktivitätsdiagramme.[13]

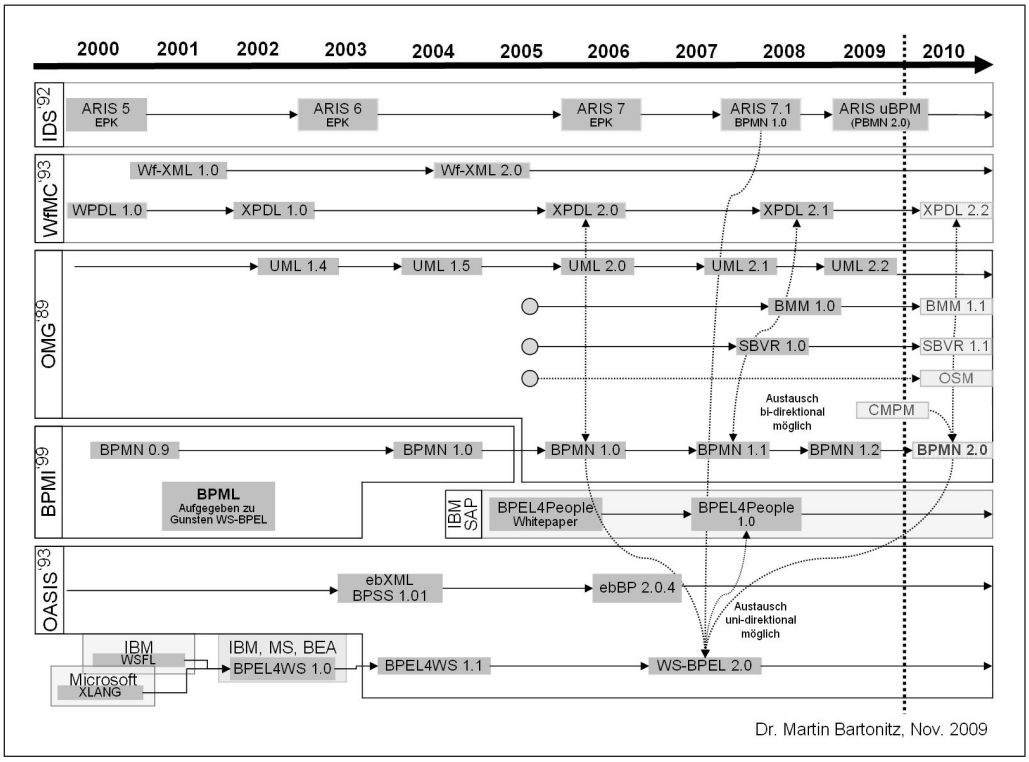


Abbildung 1.2: Historische Entwicklung wichtiger Standards im BPM (Bartonitz [2009])

Eine historische Entwicklung wichtiger Standards bis zum Jahr 2009 ist in Abbildung 1.2 dargestellt. Am linken Rand sind fünf Organisationen aufgeführt, die solche BPM-Standards entwickeln und pflegen. Darunter befinden sich beispielsweise die IDS Scheer AG, seit 2010 Software AG, mit den Ereignisgesteuerten Prozessketten (EPK) und die Object Management Group (OMG), welche sich aktuell unter anderem um die Unified Modeling Language (UML) sowie die Business Process Model and Notation (BPMN) kümmert. Rechts neben den Organisationen ist

[11] Vgl. Seidlmeier [2010], S. 21
[12] Vgl. Ko et al. [2009], S. 751
[13] Vgl. Ko et al. [2009], S. 753

jeweils die historische Entwicklung der jeweiligen Standards zu sehen.

Aus der Abbildung 1.2 geht hervor, dass Standards wie der ARIS EPK-Standard, der UML-Standard sowie der BPMN-Standard kontinuierlich weiterentwickelt werden. Eine bidirektionale Transformation zwischen diesen Standards ist jedoch nur zwischen Vor- und Nachfolger eines Standards möglich, wie anhand der Pfeile mit durchgehender Linie zu erkennen ist. Die Transformation zwischen zwei unterschiedlichen Standards ist nur sporadisch möglich. Dies ist anhand der Pfeile mit gepunkteter Linie zu erkennen. Es existiert beispielsweise ein unidirektionaler Transformationsweg zwischen BPMN 2.0 und WS-BPEL 2.0 von Omar[14].

Das BPM wird von Unternehmen verwendet, weil es hilfreich dabei ist, ein Verständnis für die Aktivitäten zu entwickeln, die zur Erreichung von Firmenzielen ausgeführt werden.[15] Eine Model-to-Model-Transformation (M2M-Transformation) ist jedoch wie bereits erwähnt nur eingeschränkt und wenn dann oft nicht bidirektional möglich.[16] Das liegt zum einen an den unterschiedlichen Organisationen, die sich um die Standards kümmern, zum anderen an den unterschiedlichen Anwendungsgebieten, für welche die Standards ursprünglich entwickelt wurden. So werden beispielsweise mit den EPK vornehmlich Geschäftsprozesse nach dem ARIS-Konzept modelliert.[17] Mit der UML hingegen sollte der Ablauf von objektorientierter Software dargestellt werden. Eine detaillierte Betrachtung erfolgt in Kapitel 3.

Aus der Abwesenheit der bidirektionalen Transformationswege zwischen den wichtigen Standards ergibt sich jedoch ein offensichtliches Problem. Die Modelle können nicht ineinander überführt werden. Es wird folgende Annahme gemacht: Zwei Unternehmen A und B streben eine Zusammenarbeit an, bei der unabhängig voneinander wichtige Prozesse modelliert werden müssen, jedoch unterschiedliche Modellierungsstandards für die Darstellung dieser Prozesse genutzt werden. Sobald die modellierten Prozesse von Unternehmen A in Unternehmen B interpretiert werden müssen, könnte es zu Problemen kommen. Der Zugang zu den modellierten Prozessen kann beispielsweise nicht gegeben sein, weil die notwendige Software dafür fehlt. Der Erwerb von Lizenzen für die kommerzielle Nutzung von Software wird schnell sehr kostenintensiv. In Tabelle 1.1 (auf der nächsten Seite) sind beispielhaft sieben Softwarewerkzeuge für die Modellierung von UML mit ihren Lizenzkosten aufgeführt.

Die Tabelle 1.1 zeigt die Lizenzkosten zum Teil pro Nutzer, zum Teil pro Floating Licence. Hier stellt sich zum einen die Frage, wie viele Lizenzen ein Unternehmen erwerben müsste, um die Prozessmodellierung durchführen zu können, zum anderen was eine Floating Licence ist. Bei einer Floating Licence handelt es sich um eine Lizenzform, bei der eine maximale Anzahl von Nutzern festgelegt ist, die gleichzeitig auf eine Ressource, in diesem Fall das Modellierungswerkzeug, zugreifen darf. Diese

[14] Vgl. Omar [2012]
[15] Vgl. Macek und Richta [2009]
[16] Vgl. Bartonitz [2009]
[17] Vgl. Ko et al. [2009], S. 758

UML-Werkzeug	Lizenzkosten
MetaEdit+ Workbench 5.0 (Introductory License)	ab 150 € / Nutzer und Jahr
Enterprise Architect 10 (Corporate Edition)	$239 / Nutzer
Altova UModel 2014 (Enterprise Edition)	299 € / Nutzer
WinA&D	$495 bis $1995 / Nutzer
Visual Paradigm for UML 10.2 (Enterprise Edition)	$1399 / Nutzer
Innovator 11	4.490 € / Floating License
RTDS 4.31	$9.000 / Floating License

Tabelle 1.1: Lizenzkosten für UML-Werkzeuge (angelehnt an Informatik [2013])

Festlegung ist abhängig vom Anbieter und den ausgehandelten Konditionen. Möchten mehr Nutzer als in der Lizenz vereinbart gleichzeitig auf die Ressource zugreifen, so müssen einige warten.[18] Ein Unternehmen wie die Volkswagen AG mit mehr als 100 Standorten und über einer halben Million Mitarbeitern[19] benötigt schätzungsweise mindestens so viele Einzelnutzerlizenzen wie Standorte und es ist unwahrscheinlich, dass je nur eine Person mit der Prozessmodellierung beauftragt ist. Es wird folgende Annahme gemacht: Jede Filiale hat mindestens fünf Abteilungen in denen je mindestens fünf Mitarbeiter mit der Prozessmodellierung betraut sind. Das macht bei einem Konzern wie der Volkswagen AG mehr als 2500 Einzelnutzerlizenzen.

Außerdem müssen die Voraussetzungen für die neue Software erst geschaffen werden, zum Beispiel durch zeitaufwendige Updates von vorhandenen Betriebssystemen, den Erwerb zusätzlicher neuer Software oder Hardware. Sollte die benötigte Software zum Lesen der Prozesse vorhanden sein, kann es dennoch teuer für die Unternehmen werden, denn es kann zu Interpretationsfehlern der modellierten Prozesse kommen und somit zu Fehlern beim weiteren Vorgehen. Derartige Interpretationsfehler kommen zustande, wenn die Mitarbeiter des Unternehmens B den von Unternehmen A verwendeten Modellierungsstandard nicht kennen. Auf diese Weise kann im Folgenden beispielsweise eine falsche Komponente oder eine Komponente falsch entwickelt werden. Wenn der Fehler erst spät erkannt wird, hat das Unternehmen gegebenenfalls bereits enorme Ressourcen aufgewandt, die jetzt verloren sind. Jedenfalls hat es zeitliche Ressourcen vergeudet und kann somit eventuell vorhandene Verträge und Deadlines nicht einhalten, was zur Folge hätte, dass es Strafzahlungen leisten muss.

Um solchen Interpretationsfehlern vorzubeugen, können Unternehmen professionelle Schulungen durchführen lassen. Aber selbst wenn nur eine kleine Gruppe von Mitarbeitern geschult wird, die ihr Wissen danach an die anderen Mitarbeiter weitergibt, entstehen an dieser Stelle wieder Kosten. Weiterhin fallen Mitarbeiter für die Zeit der Schulungen aus. Die gleichen Kosten entstehen im Übrigen auch

[18] Vgl. 3D-Coat [2013], S. 1
[19] Vgl. Volkswagen-AG [2012]

bei der Einführung einer neuen Software, denn die Mitarbeiter des Unternehmens müssen den Umgang mit dieser erst erlernen. Arbeiten mehr als zwei Unternehmen zusammen, die je unterschiedliche Modellierungsstandards verwenden, so steigen die Kosten für Schulungen exponentiell an: Während bei zwei Unternehmen nur je eine, also insgesamt zwei Schulungen durchgeführt werden müssen, müssen bei drei Unternehmen bereits je zwei Schulungen, also insgesamt sechs durchgeführt werden und so weiter.

Eine manuelle Transformation der Prozesse in den bekannten Modellierungsstandard durch eine kleine Gruppe geschulter Mitarbeiter könnte an dieser Stelle helfen. Jedoch sind solche Experten teuer und eine manuelle Transformation nimmt Zeit in Anspruch. Müssen viele Prozesse in den anderen Modellierungsstandard überführt werden, addiert sich die zeitliche Verzögerung. Außerdem kann es auch hier zu Unstimmigkeiten kommen, wenn die Prozesse ineinander überführt werden, denn einen einheitlichen Transformationsstandard gibt es nicht. So kann es passieren, dass Experte A eine Transformation auf die eine Art durchführt, Experte B auf eine gänzlich andere Art. Weiterhin ist es zum Teil nicht möglich, ein Äquivalent für ein Element aus dem Quell-Modellierungsstandard im Ziel-Modellierungsstandard zu finden. Dies führt zu Fehlern und Verlusten im transformierten Prozess und kann im Weiteren wieder zu Fehlern bei der Entwicklung von Komponenten führen, weil nicht alle benötigten Informationen (korrekt) vorhanden sind.

1.2 ARIS Business Architect

Der ARIS Business Architect[20] ist ein Software-Werkzeug, welches das bereits vorgestellte ARIS-Konzept umsetzt. Es gehört zu den führenden BPM-Werkzeugen und ist mit mehr als 3200 Installationen im deutschsprachigen Raum weit verbreitet.[21] Das Werkzeug stellt für die fünf Sichten unterschiedliche Arten von Diagrammtypen zur Verfügung wie beispielsweise Funktionsbäume für die Funktionssicht, erweiterte Entity-Relationship-Modelle für die Datensicht, Organigramme für die Organisationssicht oder EPK für die Prozesssicht. Aufgrund der Verbundenheit dieser Sichten und der Verwendung von je anderen Diagrammtypen ist eine Transformation zwischen diesen einzelnen Diagrammtypen notwendig und wird vom ARIS Business Architect angeboten.

Das Software-Werkzeug bietet allerdings nicht nur die Möglichkeit, die eigenen Diagrammtypen zu modellieren. Es können ebenso BPMN-Diagramme und UML Aktivitätsdiagramme gezeichnet werden. In Abbildung 1.3 (auf der nächsten Seite) ist beispielhaft ein Diagramm in der Notation der BPMN dargestellt. Die Notationselemente der BPMN werden in Kapitel 3 ausführlich erklärt. Zu diesem Zeitpunkt ist eine detaillierte Kenntnis jedoch noch nicht erforderlich.

[20] Version 7.1
[21] Vgl. Schmelzer und Sesselmann [2010], S. 413f

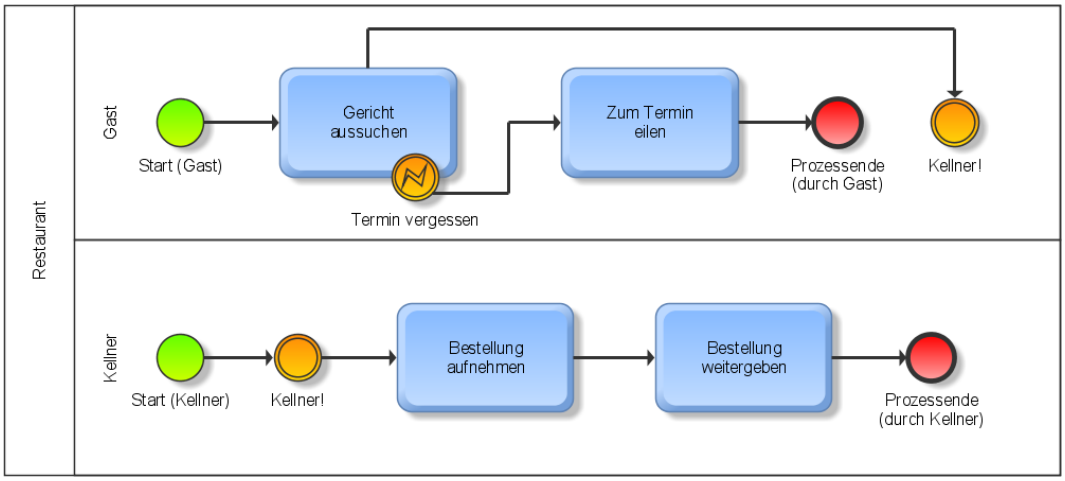


Abbildung 1.3: Beispieldiagramm in BPMN

Obwohl der ARIS Business Architect eine Modellgenerierung grundsätzlich erlaubt, funktioniert die automatische Transformation zwischen dem BPMN-Diagramm und einem UML Aktivitätsdiagramm jedoch nicht. Beim Versuch der Generierung wird ein Fehler ausgegeben und die Aktion abgebrochen. Aus diesem Grund muss eine Transformationskette angestoßen werden. In Abbildung 1.4 ist das Zwischenergebnis in der Notation einer EPK dargestellt. Die EPK wurde als Zwischenmodell ausgewählt, weil sie einen ARIS-eigenen Modelltyp darstellt.

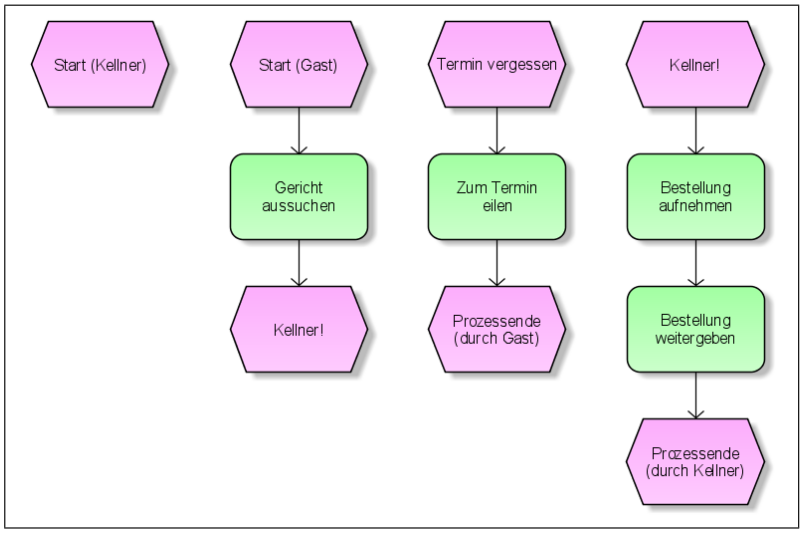


Abbildung 1.4: Generierte EPK aus BPMN-Diagramm

Bereits an dieser Stelle ist erkennbar, dass bei der Transformation zum einen Lücken entstehen, denn es fehlt das Rahmenkonstrukt „Restaurant“ sowie die Aufteilung in die Bereiche „Gast“ und „Kellner“, zum anderen wurden vier einzelne Stränge generiert, obwohl es sich um einen Prozess handelt. Akzeptabel wären noch zwei Stränge gewesen, da auch in Abbildung 1.3 optisch zwei Stränge zu sehen sind.

Es ist gegebenenfalls noch nachvollziehbar, dass der Strang rechts außen direkt nach dem zweiten Strang von links ausgeführt werden muss, weil die gleiche Bezeichnung verwendet wird. Außerdem ist anhand der Bezeichnung zu erkennen, dass es sich beim Strang ganz links offenbar um einen Start handelt. In welchem Zusammenhang jedoch der zweite Strang von rechts mit den anderen steht, kann nicht mehr erkannt werden. In Abbildung 1.5 ist das aus der EPK generierte UML-Diagramm zu sehen.

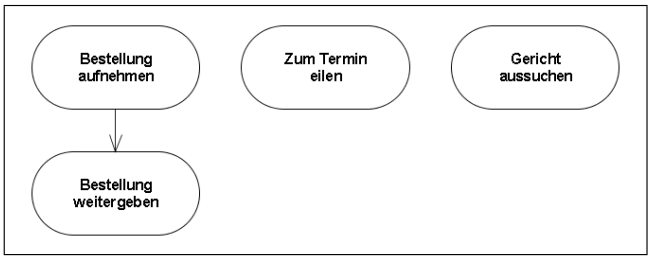


Abbildung 1.5: Generiertes UML-Diagramm aus EPK

Es sind hier lediglich die konkreten Aufgaben übrig geblieben. Ein Zusammenhang besteht nur noch zwischen „Bestellung aufnehmen“ und „Bestellung weitergeben“. Es ist deutlich zu erkennen, dass diese Transformation in hohem Maß unvollständig ist. Während in der EPK noch die einzelnen Ereignisse mit abgebildet waren, so werden im UML-Diagramm ausschließlich die Aufgaben selbst dargestellt. Start- und Endpunkt fehlen genauso wie die Verbindungen zwischen den einzelnen Elementen. Daraus ist die zeitliche Abfolge nicht mehr nachzuvollziehen.

Ein ähnliches Bild ergibt sich, wenn man von einem UML-Diagramm zu einem BPMN-Diagramm transformiert. Zwar ist hier eine direkte Transformation möglich, jedoch ist das Ergebnis nicht verwendbar, da lediglich die Arten der einzelnen Elemente dargestellt werden und diese dabei nicht alle korrekt transformiert werden. Abbildung 1.6 zeigt ein Diagramm in der Notation der UML Aktivitätsdiagramme. Die Notationselemente der UML Aktivitätsdiagramme wird in Kapitel 3 ausführlich erklärt. Zu diesem Zeitpunkt ist eine genaue Kenntnis jedoch noch nicht erforderlich.

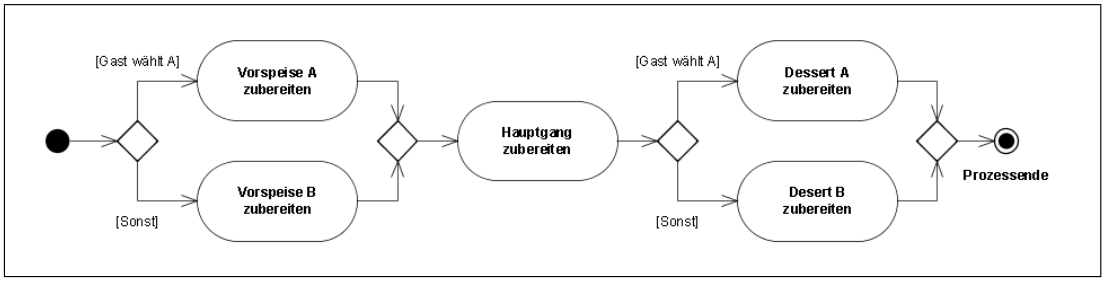


Abbildung 1.6: Beispieldiagramm in UML

In Abbildung 1.7 ist das Ergebnis der direkten Transformation des UML Aktivitäsdiagramm zu einem BPMN-Diagramm zu sehen. Die konkreten Aufgaben sind optisch zu erkennen, wenn man die Notation der BPMN kennt. Es handelt sich dabei um die abgerundeten Vierecke. Ebenso ist das Startelement zu erkennen. Es handelt sich um die Raute ganz rechts. Das Endelement fehlt, die anderen vier Elemente sind ebenfalls als Rauten dargestellt. Davon abgesehen lässt sich kaum etwas über den eigentlichen Prozess sagen.

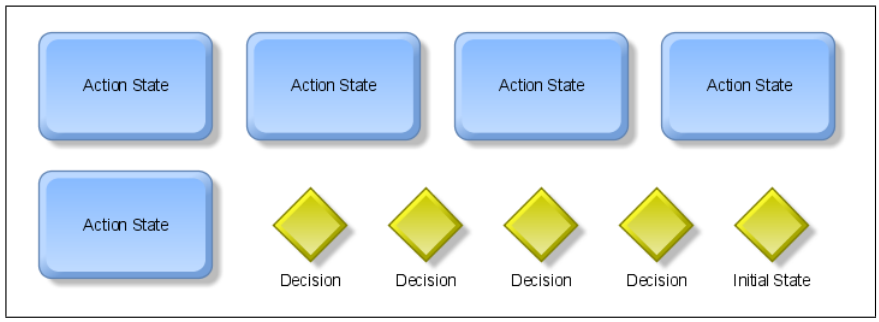


Abbildung 1.7: Generiertes BPMN-Diagramm aus UML-Diagramm

Die M2M-Transformation zwischen der BPMN und der UML funktioniert mit dem ARIS Business Architect nur sehr eingeschränkt. Auch die Generierung einer EPK funktioniert bloß für sehr einfach gehaltene Diagramme in BPMN-Notation. Wird ein UML-Diagramm aus einer EPK generiert, so ist das Ergebnis katastrophal. Eine Rücktransformation stellt nicht die originalen Diagramme wieder her, sondern fehlerhafte. Obwohl der ARIS Business Architect ein mächtiges Werkzeug für das BPM darstellt und zu den am häufigsten eingesetzten BPM-Werkzeugen im deutschsprachigen Raum gehört[22], bietet er hier für die M2M-Transformation der prominentesten Modellierungsstandards keine zufriedenstellenden Ergebnisse an. Eine manuelle Transformation nach eigenem Gutdünken würde bessere Ergebnisse erzielen.

1.3 Aufgabenstellung, Zielsetzung und Nutzen

Ein einheitlicher Standard für die automatische Transformation zwischen Modellierungsstandards wie dem ARIS EPK-Standard, dem BPMN-Standard und dem UML-Standard wäre wünschenswert, existiert jedoch zum aktuellen Zeitpunkt nur eingeschränkt. Die zu bearbeitende wissenschaftliche Lücke besteht also darin, einen Standard für die bidirektionale M2M-Transformation zwischen den bedeutenden Modellierungsstandards zu finden. Das Anliegen dieser Arbeit ist es, einen Teil der wissenschaftlichen Lücke zu schließen und aus den gewonnen Erkenntnissen, den Weg für die Erschließung eines gesamtheitlichen Standards zu finden.

[22] Vgl. Schmelzer und Sesselmann [2010], S. 413

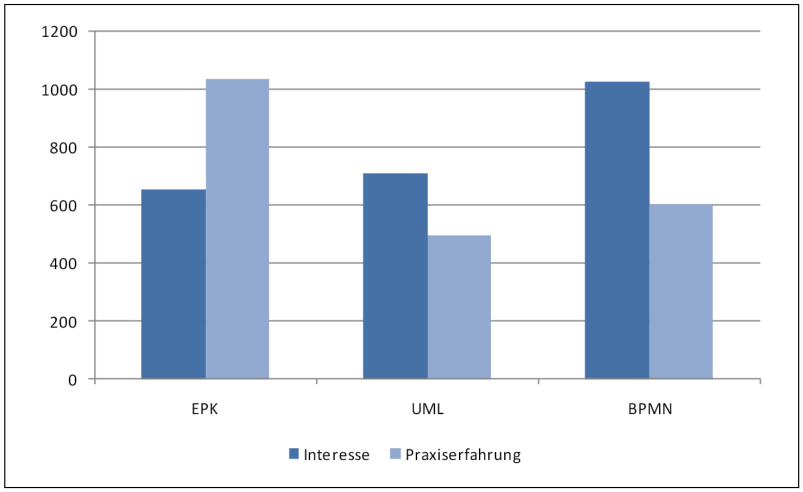


Abbildung 1.8: Popularität von Prozessnotationen auf BPM-Netzwerk.de (Stand: Juli 2010) (Freund und Rücker [2012], S. XIII)

EPK, UML und BPMN gehören zu den prominentesten graphischen Standards für die Prozessmodellierung.[23] Das wird auch in der Statistik zur Popularität von Prozessnotationen im BPM-Netzwerk deutlich (Abbildung 1.8). In diesem Netzwerk sind mehr als 7000 BPM-Professionals aus dem deutschsprachigen Raum vernetzt.[24] Auf der linken Seite der Graphik ist die Anzahl der Mitglieder festgehalten, die entweder Interesse an oder bereits Erfahrung mit dem jeweiligen Modellierungsstandard hat. Am unteren Rand sind die drei Modellierungsstandards EPK, UML und BPMN aufgeführt. In dunkelblau ist dargestellt, wie viele Mitglieder Interesse an dem jeweiligen Standard haben, in hellblau ist dargestellt, wie viele Mitglieder bereits Praxiserfahrung mit dem Standard haben. Es ist deutlich zu erkennen, dass für die EPK die meiste Praxiserfahrung vorhanden ist, aber auch, dass das Interesse an UML und BPMN hoch ist. Schmelzer und Sesselmann beobachten ebenfalls das gesteigerte Interesse an der BPMN. Während im Jahr 2007 die Verbreitung der BPMN und der UML bei der Prozessmodellierung etwa gleich war und die der EPK bei circa 60% lag, hat die BPMN die EPK im Jahr 2009 bereits überholt.[25]

Die Wahl für die Bearbeitung in dieser Arbeit fällt auf die Modellsprachen BPMN und UML, weil sie wie zuvor beschrieben zu den prominentesten Modellierungsstandards gehören. Das Interesse an der BPMN ist groß und sie wird immer öfter für die Prozessmodellierung verwendet.[26] Die UML ist zwar ein älterer Standard im Vergleich zur BPMN, gehört aber dennoch zu den am häufigsten verwendeten Modellsprachen und das Interesse für sie ist ebenfalls groß.[27] Zwar zeigt die Abbildung 1.8, dass die größte Praxiserfahrung mit der EPK besteht, allerdings verfügt diese nur über wenige Notationselemente im Vergleich zur BPMN und UML. Außerdem werden von diesen beiden Modellsprachen die meisten Kontrollflussmuster des Workflow Pattern Framework unterstützt, was

[23] Vgl. Ko et al. [2009], S. 754, und Pan et al. [2012], S. 26ff
[24] Vgl. Seidlmeier [2010], S. XIIIff
[25] Vgl. Schmelzer und Sesselmann [2010], S. 417f
[26] Vgl. Freund und Rücker [2012], S. XIII, und Schmelzer und Sesselmann [2010], S. 416ff
[27] Vgl. Freund und Rücker [2012], S. XIII, und Schmelzer und Sesselmann [2010], S. 416ff

bei der EPK nicht der Fall ist. Eine detaillierte Erläuterung dazu erfolgt in Kapitel 3.

Das Ziel dieser Masterarbeit ist es, einen Transformationsweg zwischen ausgewählten Diagrammtypen der Modellsprachen UML und BPMN für die Prozessdarstellung zu erarbeiten. Die erzielten Ergebnisse bei der Erarbeitung eines Transformationswegs sollen später auch auf die Transformation zwischen anderen Modellierungsstandards übertragen werden.

Die Forschungsfrage lautet daher: „Wie können Transformationswege in Hinblick auf die Prozessdarstellung zwischen den Modellsprachen BPMN und UML aussehen und formalisiert werden?“ Diese Frage wird am Beispiel von ausgewählten Diagrammtypen der beiden Modellsprachen untersucht. Während der Bearbeitung wird analysiert, in wie weit sich die ausgewählten Diagrammtypen wirklich für die Prozessdarstellung eignen und welche Einschränkungen in Kauf genommen werden müssen. Weiterhin wird ein Transformationsweg erarbeitet und im Anschluss verifiziert und validiert.

Durch die beschriebenen Maßnahmen soll eine standardisierte, zeitlich überschaubare und verlustfreie automatische Transformation zwischen den ausgewählten Diagrammtypen der BPMN und der UML ermöglicht werden. Der wirtschaftliche Nutzen für Unternehmen wäre enorm, da eine Überführung von Modellen aus diesen beiden Sprachen ineinander in überschaubarer Zeit verlustfrei und standardisiert möglich würde. Des Weiteren werden Einsichten in mögliche Vorgehensweisen für die Findung von Transformationswegen zwischen weiteren Modellsprachen gefunden, sodass das längerfristige Ziel, einen Standard für die bidirektionale Transformation zwischen den bedeutenden Modellierungsstandards zu finden, in der Zukunft erreicht werden kann.

1.4 Stand der Forschung

Fritzsche et al.[28] beschreiben in ihrer Arbeit Ketten von M2M-Transformationen, um ein Modell von einem Standard in einen anderen zu überführen, und Optimierungsmöglichkeiten bei derartigen Ketten. Bei der Vielzahl von Modellierungsstandards erscheint dies sinnvoll. Die Transformation in Zwischenmodelle birgt jedoch die Gefahr, wichtige Informationen aus den Modellen zu verlieren, wenn nicht für alle Notationselemente und Attribute ein passendes Gegenstück in der Notation des nächsten Modells verfügbar ist.

Kalnins und Vitolins geben in ihrer Arbeit an, dass es eine Initiative der OMG zur Entwicklung eines gemeinsamen Meta-Modells gibt, dass eine exakte Transformation zwischen diesen beiden Modellierungsstandards jedoch nicht so direkt möglich ist, wie es den Anschein hat.[29] Sie beschreiben einen Transformationsweg vom UML

[28] Vgl. Fritzsche et al. [2010], 2f
[29] Vgl. Kalnins und Vitolins [2006], S. 1

Aktivitätsdiagramm zu einem BPMN-Diagramm unter der Verwendung der Transformationssprache MOLA (Model Transformation Language). Sie verwenden jedoch nur eine Untermenge der Notationselemente der UML Aktivitätsdiagramme[30], um Arbeitsflüsse, wie sie mit der BPMN dargestellt werden, abzubilden. In Anbetracht der Tatsache, dass die BPMN jedoch die komplexeren Notationselemente besitzt, mag dies ungünstig sein. Sie verwenden ebenfalls nur eine Untermenge der BPMN-Elemente[31].

Cibrán hebt in ihrer Arbeit hervor, dass eine Zuordnung von Elementen der BPMN zu Elementen der UML Aktivitätsdiagramme nicht trivial ist aufgrund des unterschiedlichen Zwecks der Standards und ihrer verschiedenen Semantiken.[32] Dennoch beschreibt sie in ihrer Arbeit eine konzeptuelle Zuordnung, um BPMN-Modelle in UML Aktivitätsdiagramme zu überführen. Sie gibt jedoch Schwierigkeiten aufgrund der komprimierten Informationen einiger Notationselemente der BPMN an. So können in der BPMN spezielle Schleifenaufrufe mittels eines einzigen Elements dargestellt werden. In der UML-Notation sind dafür eine Reihe von Elementen von Nöten. Weiterhin lassen sich beispielsweise die Verzweigungen der BPMN nicht eindeutig auf Knotenpunkte in der UML abbilden und es gibt nicht für jedes Element der BPMN ein eindeutiges Gegenstück bei den UML Aktivitätsdiagrammen. Weiterhin beschreibt Cibrán eine automatische Transformation unter Verwendung der Transformationssprache ATL (ATLAS Transformation Language).

Macek und Richta betonen in ihrer Arbeit die Notwendigkeit zur Möglichkeit einer Transformation von BPMN-Modellen zu anderen Modellsprachen, da die BPMN sehr komplex ist: Sie enthält Notationselemente, die nicht-atomare Aktionen darstellen. Dies beschleunigt zwar den Modellierungsprozess, erschwert jedoch sowohl die Transformation zwischen den Modellen als auch die Lesbarkeit der Modelle für nicht geübte Personen.[33] Macek und Richta beschreiben einen Transformationsweg mittels XSLT (Extensible Stylesheet Language Transformation). Dafür erarbeiten sie zunächst eine XMI[34]-Repräsentation für die BPMN, da diese noch nicht standardisiert existiert. Bei der Transformation heben sie hervor, dass es einen gewissen Informationsverlust gibt, da Notationselemente in der BPMN, wie bereits angesprochen, zum Teil komprimierte Informationen beinhalten. Sie geben weiterhin an, dass eine Rücktransformation benötigt wird, um die Konsistenz zwischen zwei Modellen zu bewahren.

Raedts et al.[35] stellen einen Ansatz vor, bei welchem BPMN-Modelle zu Petri-Netzen transformiert werden. In einem weiteren Schritt werden die Petri-Netze zu mCRL2, einer Prozess-Algebra, transformiert. Ihr Ansatz basiert nicht darauf, eine M2M-Transformation zu ermöglichen, sondern die originalen Modelle einer Verifizierung und Validierung zu unterziehen. Dazu werden die erhalte-

[30] Vgl. Kalnins und Vitolins [2006], S. 4f
[31] Vgl. Kalnins und Vitolins [2006], S. 8f
[32] Vgl. Cibrán [2008], S. 2
[33] Vgl. Macek und Richta [2009], S. 2f
[34] XMI, das Kürzel steht für XML Metadata Interchange
[35] Vgl. Raedts et al. [2007]

nen Ergebnisse automatisch analysiert. Trotz der anderen Intention bildet ihre Transformation von BPMN-Modellen zu Petri-Netzen einen ersten Schritt für die M2M-Transformation. Sie verweisen darauf, dass es auch Transformationen von beispielsweise EPK und UML Aktivitätsdiagrammen zu Petri-Netzen gibt. Insofern wäre eine unidirektionale Transformation dieser Modellierungsstandards zu einem anderen Modellierungsstandard bereits möglich. Durch die automatische Verifizierung und Validierung könnten M2M-Transformationen zwischen BPMN und UML Aktivitätsdiagrammen später auf ihre Korrektheit überprüft werden, da die abgeleiteten Petri-Netze identisch sein müssen. Die Tabelle 1.2 fasst die Ansätze in der Übersicht zusammen. Es sind dabei sowohl der Verfasser des Ansatzes, die Richtung der M2M-Transformation sowie die wichtigsten Eigenschaften beschrieben.

Autor	Richtung	Eigenschaften	Kritik
Kalnins und Vitolins	UML AD zu BPMN	Transformation unter Verwendung von MOLA	Verwendung einer Untermenge der UML AD Elemente; Verwendung einer Untermenge der BPMN Elemente; nur unidirektionale Transformation
Cibrán	BPMN zu UML AD	Transformation unter Verwendung von ATL	Uneindeutigkeiten bei Transformation aufgrund der Komplexität der BPMN; nur unidirektionale Transformation
Macek und Richta	BPMN zu UML AD	Transformation unter Verwendung von XSLT	Informationsverluste aufgrund der Komplexität der BPMN; nur unidirektionale Transformation
Raedts et al.	BPMN zu Petri-Netz, UML AD zu Petri-Netz, EPK zu Petri-Netz	Verifizierung und Validierung von Modellen durch Analyse von Petri-Netzen; Vertiefung durch mCRL2	nur unidirektional; keine Transformation zwischen BPMN und UML

Tabelle 1.2: Vergleich der Ansätze

Ersichtlich ist, dass es bereits eine ganze Reihe von Ansätzen zur Transformation zwischen den Modellsprachen BPMN und UML gibt. Alle weisen einige Schwächen auf, meistens in Form von Einschränkungen beim Transformationsumfang oder der Transformationsrichtung, welche oft nur unidirektional ist. Es könnte geprüft werden, ob diese Ansätze kombiniert werden könnten, um eine bidirektionale Transformation zu erreichen. Dies soll aber nicht Teil dieser Arbeit sein.

1.5 Aufbau der Arbeit

Zur Beantwortung der Forschungsfrage werden in Kapitel 2 zunächst einige elementare Begriffe in dieser Arbeit geklärt. Zusätzlich wird betrachtet, welche Elemente für die Prozessdarstellung elementar sind und demnach von Modellierungsstandards bedient werden müssen. Anschließend erfolgt ein kurzer Einblick in die Geschichte und die Bedeutung von Prozessen für Unternehmen.

Danach folgt in Kapitel 3 eine kurze Vorstellung der Object Management Group sowie eine Einführung in die ausgewählten Modellsprachen BPMN und UML. Dabei wird zum einen deren Ursprung und Zweck erläutert, zum anderen erfolgt je eine kritische Auseinandersetzung in Bezug auf ihre Fähigkeit, Prozesse darstellen zu können. Im Anschluss daran werden die einzelnen Notationselemente kurz in der Übersicht vorgestellt.

In Kapitel 4 wird ein bidirektionaler Transformationsweg zwischen ausgewählten Diagrammtypen der beiden Modellsprachen erarbeitet. Dazu wird betrachtet, welche Konstrukte sich ineinander überführen lassen und wie dies konkret möglich ist. Im Anschluss wird ein XML-basiertes Baustein-Prinzip entwickelt, welches durch seinen Status als Meta-Ebene die M2M-Transformation ermöglicht. Danach wird betrachtet, welche Strukturen aus den Modellsprachen zu welchen Strukturen im Ziel-Modellierungsstandard überführt werden müssen. Als Abschluss wird ein Pseudo-Code für die Ausführung der Transformation entwickelt.

Es folgt in Kapitel 5 eine Verifizierung und Validierung der Ergebnisse und in Kapitel 6 eine Zusammenfassung der Arbeit sowie eine abschließende kritische Betrachtung der erzielten Ergebnisse. Zum Schluss wird ein Ausblick auf eine zukünftige Fortführung des Projekts gegeben.

2 Prozessmodellierung

In diesem Kapitel werden einige grundlegende Begriffe in dieser Arbeit erläutert, unter anderem die Begriffe Modell und Prozess, und welche Bedeutung sie für Unternehmen haben. Dazu wird betrachtet, warum Unternehmen Prozesse modellieren, wie vor dem prozessorientierten Ansatz vorgegangen wurde und welche Vorteile sich aus dem veränderten Ansatz ergeben.

2.1 Begriffserläuterungen

2.1.1 Modell

„Ein *Modell* wird verstanden als ein immaterielles Abbild der Realwelt (des Objektsystems) für Zwecke eines Subjekts. Modelle werden als Hilfsmittel zur Erklärung und Gestaltung realer Systeme eingesetzt. Erkenntnisse über Zusammenhänge und Sachverhalte bei realen Problemen können mit Hilfe von Modellen aufgrund von Ähnlichkeit gewonnen werden, die zwischen dem realen betrieblichen System und dem Modell als Abbild dieses Systems bestehen.“[36]

Ein Modell gibt demnach einen realen Sachverhalt für beispielsweise eine Person oder eine Personengruppe wieder, ist aber selbst kein greifbares Objekt. Dem widerspricht allerdings die Existenz von materiellen Modellen wie etwa dem Modell eines Sonnensystems. Dabei handelt es sich um einen vereinfachten Nachbau eines stellaren Phänomens, um Planetenbewegungen zu veranschaulichen oder Größenunterschiede in vorstellbare Dimensionen zu übertragen.

Eine andere Definition lautet: Ein Modell ist ein „materielles Objekt oder theoret[isches] Konstrukt, das einem Untersuchungsgegenstand in bestimmten Eigenschaften oder Relationen entspricht (Struktur-, Funktions- oder Verhaltensanalogie) und für sonst nicht mögl[iche] oder zu aufwendige experimentelle Untersuchungen, math[ematischen] Berechnungen, Erklärungs- oder Demonstrationszwecke oder zur Optimierung des Originals verwendet wird.“[37] Diese Definition schließt ein Modell wie das materielle Modell eines Sonnensystems ein.

Aus beiden Definitionen geht hervor, dass ein (realweltlicher) Untersuchungsgegenstand zu einem Objekt oder Konstrukt wird, von welchem anhand von Ähnlichkeiten zur real existierenden Welt Erkenntnisse gewonnen werden können. Des Weiteren geht hervor, dass die Modelle auf einer abstrahierten Ebene zu

[36] Vgl. Becker et al. [1995], S. 435
[37] Vgl. Brockhaus [2003a], S. 4871f

betrachten sind, da sie zwar Zusammenhänge und Sachverhalte wiedergeben, dies jedoch anhand von Analogien tun. Als *Modellierung* wird der Vorgang der Modell-Erstellung bezeichnet.[38] Hier geschieht die Abstraktion des realweltlichen Objekts oder Vorgangs zu einem vereinfachten Abbild der Realität. In dieser Arbeit werden Modelle von Prozessen betrachtet.

Im Rahmen dieser Arbeit haben die Begriffe *Modellsprache* (Modellierungssprache) und *Modellierungsstandard* eine große Bedeutung und werden synonym für einander verwendet. Eine Modellsprache ist eine Sprache, die es erlaubt, ein System „zu spezifizieren, visualisieren, konstruieren und dokumentieren".[39] „Modellierungssprachen werden verwendet, um ein System in Form von Abstraktionen zu beschreiben."[40] Sie werden dafür benutzt, ein reales Objekt oder einen realen Vorgang auf einer höheren Abstraktionsebene[41] mit festgelegten Elementen auf eine ebenfalls festgelegte Art und Weise zu beschreiben (Syntax und Semantik der Sprache).[42] Das Ziel ist es, etwas Komplexes leicht verständlich darzustellen.

Ein weiterer wichtiger Begriff in dieser Arbeit ist die *Modelltransformation*. Eine Transformation ist eine Art der Veränderung beziehungsweise Umformung, wie schon der lateinische Wortursprung aufzeigt.[43] Es handelt sich um die „Umformung einer [... S]truktur in eine andere unter Beibehaltung der Grundbedeutung".[44] Diese Definition ist auf diverse Gebiete anwendbar, beispielsweise in der Mathematik und der Physik, wo unter anderem Ähnlichkeitstransformationen durchgeführt werden, den Sozialwissenschaften, wo von Transformationsgesellschaften gesprochen wird[45], aber auch in der Kochkunst, wo der Zustand eines Lebensmittels geändert wird. In dieser Arbeit wird unter einer Transformation immer die Transformation von einem Modell zu einem anderen Modell verstanden, das heißt, ein Modell wird auf Grundlage festgelegter Regeln in ein Modell in einer anderen Modellsprache überführt (M2M-Transformation). Wird in dieser Arbeit Bezug auf eine andere Art der Transformation genommen, so wird dies explizit hervorgehoben.

[38] Vgl. Maria [1997], S. 7
[39] Vgl. Paige et al. [2000], S. 666
[40] Vgl. Paige et al. [2000], S. 666
[41] Vgl. Paige et al. [2000], S. 666
[42] Vgl. Harel und Rumpe [2000], S. 3, und Harel und Rumpe [2004], S.3ff
[43] Vgl. Brockhaus [2003c], S. 7626
[44] Vgl. Brockhaus [2003c], S. 7626
[45] Vgl. Brockhaus [2003c], S. 7626

2.1.2 Prozess

Gedanklich ist ein *Prozess* immer mit einem Ablauf oder Hergang von einer Sache oder etwas Abstraktem verbunden. So wird beispielsweise vom „Prozess einer Entwicklung“ gesprochen. Ein ursprünglicher Zustand wird in irgendeiner Form verändert. Dies hat jedoch nichts mit der Art von Transformation zu tun, die im vorangegangenen Abschnitt erläutert wurde, und wird daher im Weiteren als Umwandlung bezeichnet. Das deutsche Wort „Prozess“ findet seinen Ursprung im Lateinischen und bedeutet in etwa „Vorgang“, „Verlauf“ oder „Entwicklung“.[46]

Als *Prozess* wird in der Begriffsnorm des Qualitätsmanagements nach DIN EN ISO 9000:2005 ein „Satz von in Wechselbeziehung oder Wechselwirkung stehenden Tätigkeiten, der Eingaben in Ergebnisse umwandelt“ bezeichnet.[47] Es handelt sich beim Prozess um das wichtigste Element in der Prozessmodellierung beziehungsweise Prozessdarstellung.

Nach der Definition gibt es zunächst Eingaben, die während eines Prozesses in ein oder mehrere Ergebnisse umgewandelt werden. In Abbildung 2.1[48] ist der Aufbau eines grundsätzlichen Prozesses abgebildet.

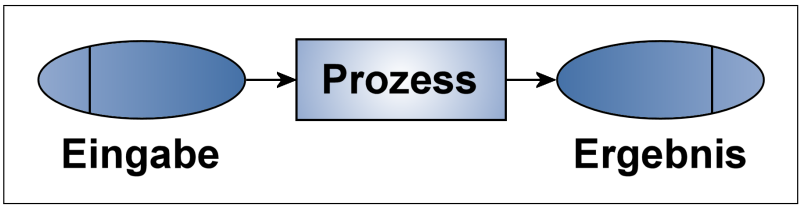


Abbildung 2.1: Grundsätzlicher Prozess

Eingaben und Ergebnisse können sowohl materiell als auch immateriell sein[49], wobei materielle Eingaben nach DIN unter anderem Ausrüstung und Bauelemente sein können und immaterielle Eingaben beispielsweise Energie und Informationen. Das gleiche gilt nach DIN für die Ergebnisse.

Ergebnisse können nach DIN sowohl beabsichtigt als auch nicht beabsichtigt sein. Als Beispiele für nicht beabsichtigte Ergebnisse werden Abfall und Umweltverschmutzung genannt. Jedoch kann nicht jedes nicht beabsichtigte Ergebnis als Abfall bezeichnet werden. An dieser Stelle wird der Begriff der Kuppelproduktion wichtig. „Die Erzeugung bestimmter Produkte aus [...] Rohstoffen bringt aufgrund natürlicher (physikalischer, chemischer oder biologischer) Gesetzmäßigkeiten [...] gewisse, nicht unbedingt beabsichtigte Nebenprodukte und Wirkungen hervor.“[50] „Kuppelproduktion liegt genau dann vor, wenn bei der Produktion eines Produk-

[46] Vgl. Brockhaus [2003b], S. 5920
[47] Vgl. DIN e.V. [2011], S. 3
[48] Bild angelehnt an DIN e.V. [2011], S. 3, Bild 1
[49] Vgl. DIN e.V. [2011], S. 3
[50] Vgl. Peterson und Faber [2004]. S. 2

tes A zwangsläufig das Produkt B entsteht.“[51]

Es kann sein, dass bei einem Prozess Kuppelprodukte entstehen, die für das Unternehmen, welches den Prozess ausführt, nicht primär relevant sind. Jedoch kann nicht jedes dieser Kuppelprodukte als Abfall bezeichnet werden. So führen Peterson und Faber das Kuppelprodukt „Schaffleisch“ an, welches bei der Wollproduktion entsteht.[52] Derartige Kuppelprodukte können von Unternehmen bewusst genutzt werden, um Vorteile zu erwirtschaften. Bei anderen Kuppelprodukten müssen die Prozesse nach Möglichkeit so verändert werden, dass sie in geringfügigen Mengen auftreten.

Weiterhin heißt es nach DIN, dass „[d]ie Ergebnisse eines Prozesses [...] häufig die Eingaben in andere Prozesse sein [können]“[53]. Dies ist in Abbildung 2.2[54] dargestellt: Die Prozesse B und C erhalten das Ergebnis beziehungsweise die Ergebnisse aus Prozess A als Eingaben. Ihre eigenen Ergebnisse erhält Prozess D als Eingaben.

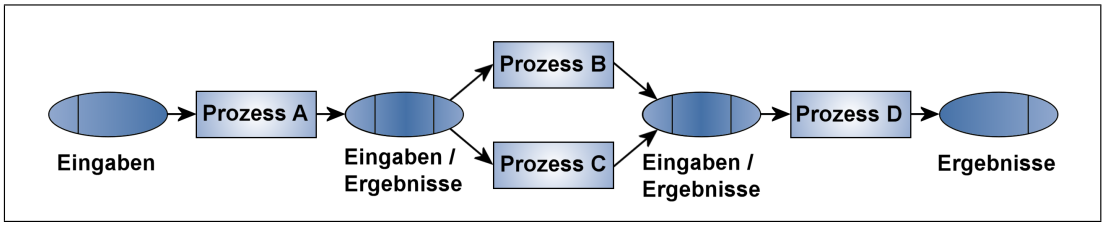


Abbildung 2.2: Allgemeingültige Prozessreihenfolge

Der Prozess selbst ergibt sich nach DIN aus den „in Wechselbeziehung oder Wechselwirkung stehenden Tätigkeiten“. Darunter versteht Seidlmeier „Aktionen, die Entscheidungen folgen“.[55] Es handelt sich dabei um eine „wiederholbare Folge von physischen oder informatorischen Tätigkeiten“. Bei Göpfert und Lindenbach handelt es sich um eine „Folge von koordinierten Aktivitäten“.[56]

Da ein Prozess in einem Unternehmen nicht nur einmalig abläuft, ist es eine Notwendigkeit, dass er wiederholbar ist und immer wieder das gleiche Ergebnis in gewünschter Variation hervorbringt. Als Beispiel kann an dieser Stelle die Produktion von Automobilen genannt werden: Ein Unternehmen wird nicht nur ein einziges Automobil produzieren, weil dies höchstwahrscheinlich wirtschaftlich unrentabel ist. Eine Ausnahme bildet an dieser Stelle die Produktion eines Prototypen. Mit gewünschter Variation ist beispielsweise die Änderung der Lackfarbe des Automobils gemeint, jedoch nicht die grundlegende Gestaltung oder Funktionsweise. Um also immer wieder dieses gleiche Ergebnis zu erhalten, müssen die Arbeitsschritte

[51] Vgl. Peterson und Faber [2004]. S. 8
[52] Vgl. Peterson und Faber [2004]. S. 8
[53] Vgl. DIN e.V. [2011], S. 6
[54] Bild angelehnt an DIN e.V. [2011], S. 6, Bild 3
[55] Vgl. Seidlmeier [2010], S. 2f
[56] Vgl. Göpfert und Lindenbach [2012], S. 1

koordiniert werden, das heißt geplant und in gleicher Reihenfolge ausgeführt werden.

Schon 1985 hat Michael Porter eine Unterscheidung der Aktivitäten in einem Unternehmen erkannt. Abbildung 2.3 zeigt die von ihm entwickelte Wertschöpfungskette[57]. Zu den unterstützenden Aktivitäten zählt Porter unter anderem das „Personalwesen“ und die „Beschaffung“, während die primäre Aktivitäten in der „Produktion“ und dem „Kundendienst“ liegen. Als Werte der Wertschöpfung können Kunden- und Mitarbeiterzufriedenheit, Termintreue, hohe Ressourcenauslastung, niedrige Kosten sowie Gewinn angeführt werden.[58]

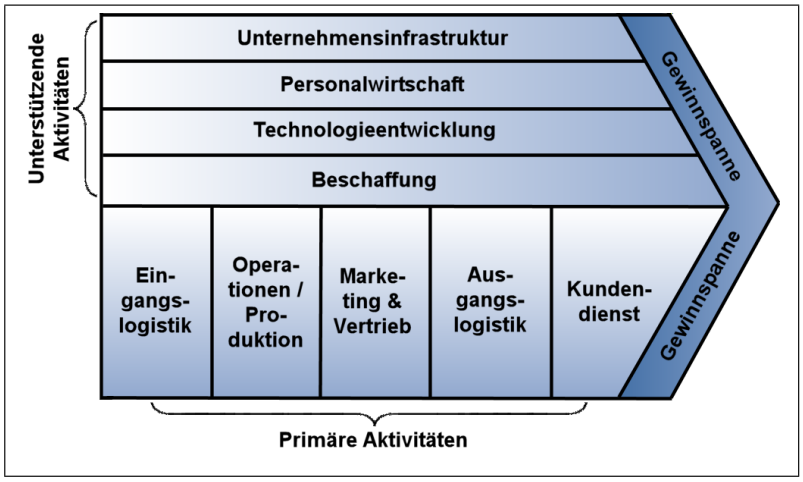


Abbildung 2.3: Wertschöpfungskette nach Porter

Eine ähnliche Teilung ist für die Prozesse in einem Unternehmen möglich. Seidlmeier unterscheidet zwischen *Geschäfts- / Kernprozessen* sowie *Unterstützungsprozessen*. Dabei haben Geschäftsprozesse eine hohe Wertschöpfung für den Kunden, wohingegen Unterstützungsprozesse ihrem Namen folgend Geschäftsprozesse lediglich unterstützen.[59]

[57] Bild angelehnt an Bergmann [2009]
[58] Vgl. Göpfert und Lindenbach [2012], S. 1
[59] Vgl. Seidlmeier [2010], S. 3

2.1.3 Grundsätze ordnungsgemäßer Modellierung

Aus den vorangegangenen Abschnitten geht hervor, dass ein Modell (eines Prozesses) einen komplexen Ablauf vereinfacht darstellt. Dies ist nicht nur durch die Abstraktion des Ablaufs und die Wahl der Modellsprache möglich, sondern auch über das Modell selbst und dessen Qualität.[60] Becker et al. haben sechs Grundsätze ordnungsgemäßer Modellierung (GoM) zusammengetragen, welche die Qualität eines Modells steigern. Die sechs GoM sind *Richtigkeit*, *Relevanz*, *Wirtschaftlichkeit*, *Klarheit*, *Vergleichbarkeit* sowie *systematischer Aufbau*. In Tabelle 2.1 sind diese GoM untergliedert in notwendige und ergänzende Grundsätze.[61]

Notwendige Grundsätze	Ergänzende Grundsätze
Grundsatz der Richtigkeit	Grundsatz der Klarheit
Grundsatz der Relevanz	Grundsatz der Vergleichbarkeit
Grundsatz der Wirtschaftlichkeit	Grundsatz des systematischen Aufbaus

Tabelle 2.1: Grundsätze ordnungsgemäßer Modellierung

Beim *Grundsatz der Richtigkeit* geht es zum einen um die formale Korrektheit eines Modells (syntaktische Richtigkeit), zum anderen um die Korrektheit des durch das Modell abgebildeten Sachverhalts (semantische Richtigkeit). Die syntaktische Richtigkeit ist gegeben, wenn alle Regeln der Modellsprache beachtet und eingehalten werden; dies kann zum Teil automatisch überprüft werden. Die semantische Richtigkeit ist gegeben, wenn sich die Projektbeteiligten einig sind, dass der dargestellte Sachverhalt zutreffend ist.[62]

Der *Grundsatz der Relevanz* bezieht sich auf die Forderung, dass die Ziele der Modellierung explizit herausgestellt werden. „Die in einem Modell enthaltenen Elemente und Beziehungen sind [...] genau dann relevant, wenn der Nutzeffekt der Modellverwendung sinken würde, falls das Modell weniger Informationen enthalten würde.“[63] Um dies zu prüfen, müssen zuvor Modellierungsziele definiert worden sein, in denen eindeutig beschrieben ist, was das Modell darstellen soll. Bei unterschiedlichen Zielgruppen des Modells, können sich die Modellierungsziele und somit die Relevanz einzelner Elemente ändern.[64]

Der dritte Grundsatz, der *Grundsatz der Wirtschaftlichkeit*, beschränkt unter anderem die Modellierungsintensität, das heißt den Detaillierungsgrad eines Modells, denn „sowohl die Modellerstellung als auch die Nutzung [sollen] möglichst kosteneffizient ablaufen“.[65] Ab einem bestimmten Detaillierungsgrad ist ein Modell für ein Unternehmen unwirtschaftlich. Wo dieser Punkt liegt, ist abhängig vom Unternehmen und dessen Zielen.

[60] Vgl. Becker et al. [1995], S. 437, und Becker et al. [2009], S. 39
[61] Vgl. Becker et al. [2009], S. 40
[62] Vgl. Becker et al. [1995], S. 437f, und Becker et al. [2009], S. 40
[63] Vgl. Becker et al. [1995], S. 438
[64] Vgl. Becker et al. [2009], S. 41
[65] Vgl. Becker et al. [2009], S. 41

Beim *Grundsatz der Klarheit* geht es unter anderem um Aspekte wie Strukturiertheit, Übersichtlichkeit und Lesbarkeit des Modells, welche sich beispielsweise in der Anordnung graphischer Elemente äußern. Dieser Grundsatz und der Grundsatz der Richtigkeit stehen in einem schwierigen Verhältnis zueinander: Eine semantisch korrekte Abbildung kann die Klarheit des Modells beeinträchtigen, während ein klar dargestelltes Modell gegebenenfalls nicht den richtigen Sachverhalt wiedergibt.[66]

Der *Grundsatz der Vergleichbarkeit* kann wie der Grundsatz der Richtigkeit in eine syntaktische und eine semantische Betrachtungsweise aufgeteilt werden. Bei der syntaktischen Vergleichbarkeit geht es darum, dass Modelle, die mit unterschiedlichen Modellsprachen erstellt wurden, konsistent zueinander sind. Bei der semantischen Vergleichbarkeit wird der Inhalt des Modells betrachtet. Beide Modelle müssen zur selben Erkenntnis führen.[67]

Der letzte GoM, der *Grundsatz des systematischen Aufbaus*, befasst sich mit dem Aufbau des Modells. Dieser Grundsatz bezieht sich unter anderem auf die konsistente Verwendung von Objekten, die in unterschiedlichen Sichten (Datensicht, Funktionssicht, Prozesssicht etc.) verwendet werden. So müssen bereits bei der Modellierung auf der Datensicht die Konsequenzen für die Prozesssicht betrachtet werden.[68]

Grundsatz ...	Kurzfassung
der Richtigkeit	Syntaktische und semantische Richtigkeit des Modells
der Relevanz	Modellierung einzig von relevanten Elementen
der Wirtschaftlichkeit	Fokus auf Kosteneffizienz durch Beschränkungen des Modells
der Klarheit	Lesbarkeit des Modells durch klare Strukturen
der Vergleichbarkeit	Modelle in unterschiedlichen Modellsprachen müssen zur selben Erkenntnis führen
des systematischen Aufbaus	Konsistente Verwendung der modellierten Elemente

Tabelle 2.2: Grundsätze ordnungsgemäßer Modellierung im Überblick

In Tabelle 2.2 sind die GoM noch einmal im Überblick aufgeführt. Die Einhaltung dieser GoM erleichtert zum einen die Modellierung, weil sie einige Richtlinien vorgeben, nach denen modelliert werden sollte, zum anderen erleichtern die GoM das Lesen von Modellen, wenn sie eingehalten worden sind. Nach einer M2M-Transformation sollte das erhaltene Modell die GoM genauso erfüllen wie das originale Modell. Das gilt insbesondere für den Grundsatz der Vergleichbarkeit.[69]

[66] Vgl. Becker et al. [1995], S. 438f, und Becker et al. [2009], S. 42
[67] Vgl. Becker et al. [1995], S. 439, und Becker et al. [2009], S. 42
[68] Vgl. Becker et al. [1995], S. 439, und Becker et al. [2009], S. 42
[69] Vgl. Becker et al. [1995], S. 437, und Becker et al. [2009], S. 39 und 43

2.2 Elementare Basiselemente für die Prozessdarstellung

Aus der Definition für den Prozess heraus lassen sich direkt verschiedene Elemente ableiten, welche für die Prozessdarstellung notwendig sind und demnach von einem Modellierungsstandard wie den EPK, der BPMN oder der UML umgesetzt werden können müssen. Göpfert und Lindenbach nennen die folgenden Elemente[70]:

- definierter Anfang und definiertes Ende
- Verzweigungs- und Zusammenführungselemente
- Steuerungselemente
- (äußere) Ereignisse
- Zeitablauf, Ressourcenverbrauch
- Verantwortliche und Beteiligte

Der Anfang eines Prozesses liegt genau dort, wo die erste Eingabe gemacht wird. Das Ende eines Prozesses liegt dort, wo das primär relevante Ergebnis den Prozess verlässt. Dazwischen können Kuppelprodukte den Prozess bereits vorzeitig verlassen haben. Die Eingabe stellt eine Ressource, materiell oder immateriell, dar, welche im Folgenden während des Prozesses verarbeitet wird.

Im Fall von Unterprozessen, also Prozessen in einem Prozess, erhalten diese Unterprozesse die Ergebnisse von vorgelagerten Unterprozessen als Eingabe. Sie geben ihre eigenen Ergebnisse wiederum an nachgelagerte Unterprozesse als Eingaben weiter. Dabei kann ein Unterprozess auch die Ergebnisse mehrerer anderer Unterprozesse benötigen oder die eigenen Ergebnisse an mehrere andere Unterprozesse weitergeben. An dieser Stelle werden die Verzweigungs- und Zusammenführungselemente wichtig. Diese stellen eine besondere Form der Steuerungselemente dar.

Die Steuerung des Prozessablaufs ist abhängig von verschiedenen Faktoren wie beispielsweise von den Ergebnissen vorangegangener Prozesse oder von externen Ereignissen. Durch die Steuerungselemente kann entschieden werden, wie der weitere Ablauf des Prozesses aussehen soll. Der Prozessablauf selbst gibt Auskunft über die zeitliche Reihenfolge der einzelnen Aktionen. Bei einer entsprechenden Darstellung kann auch festgestellt werden, welche Personen oder Akteure verantwortlich für einen bestimmten Prozess sind oder welche wenigstens daran beteiligt sind. Das Wissen darüber, wer verantwortlich für einen Prozess ist, hilft nicht nur dabei, Absprachen schneller zu treffen, sondern auch Fehler schneller zu identifizieren und zu beseitigen.

[70] Vgl. Göpfert und Lindenbach [2012], S. 1

2.3 Zweck der Prozessdarstellung

Unternehmen betreiben BPM um den wachsenden Herausforderungen, denen sie sich gegenüber sehen, zu meistern. Ein wichtiges Element dabei ist der prozessorientierte Ansatz. „Der Zweck des prozessorientierten Ansatzes ist es, die Wirksamkeit und Effizienz einer Organisation bei der Erreichung ihrer festgelegten Ziele zu verbessern“.[71] Um die Vorteile dieses Ansatzes zu erkennen, muss zunächst geklärt werden, wie Unternehmen zuvor organisiert waren.

Vor dem prozessorientierten Ansatz wurde der funktionsorientierte Ansatz verwendet. Dabei ist jede Funktion darauf spezialisiert, eine bestimmte Aufgabe zu verrichten und bearbeitet dadurch je nur Teile der Kundenleistung.[72] So bearbeitet das Marketing lediglich Marketingaufgaben und der Service nur Serviceaufgaben. Prozesse werden durch die Grenzen der unterschiedlichen Abteilungen unterbrochen und die Verantwortlichen für den Prozess wechseln.[73] Diese „Schnittstellen verursachen Koordinations- und Kontrollaufwand, erzeugen Missverständisse und Fehler, verzögern Entscheidungen, verbrauchen Zeit, erschweren die Kommunikation, führen zu Informationsverlusten und mindern insgesamt die Ergebnisqualität sowie die Produktivität.“[74] „[I]n Zeiten überschaubarer Märkte, hoher Marktstabilität, langer Produktlebenszyklen, stabiler Technologien und großer Stückzahlen [hatte der funktionsorientierte Ansatz seine] Berechtigung.“[75] Jedoch wurde bereits bei der Motivation zu dieser Arbeit hervorgehoben, dass sich die Faktoren geändert haben und Kundenorientierung wichtiger geworden ist. Abbildung 2.4 zeigt die funktionsorientierte gegenüber der prozessorientierten Arbeitsweise.

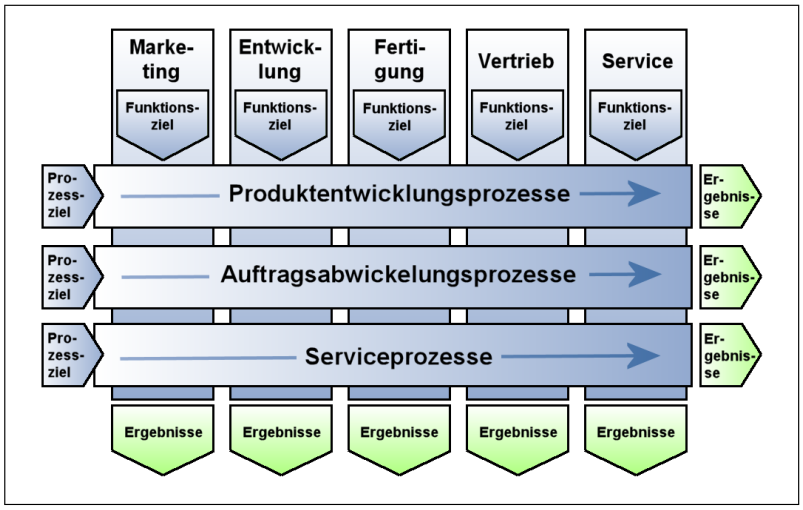


Abbildung 2.4: Funktionsorientierung vs. Prozessorientierung (Schmelzer und Sesselmann [2010], S. 74)

[71] Vgl. DIN e.V. [2011], S. 2
[72] Vgl. Schmelzer und Sesselmann [2010], S. 73
[73] Vgl. Schmelzer und Sesselmann [2010], S. 74
[74] Vgl. Schmelzer und Sesselmann [2010], S. 74
[75] Vgl. Schmelzer und Sesselmann [2010], S. 75

Beim prozessorientierten Ansatz gibt es derartige Schnittstellen wie beim funktionsorientierten Ansatz nicht. Die Prozesse erstrecken sich über die Funktionen und einzelnen Abteilungen hinweg und führen auf diese Weise zu einer hohen Wertschöpfung für den Kunden, da der einzelne Mitarbeiter das Gesamtbild erkennen kann.[76] Die Merkmale von funktionsorientierten und prozessorientierten Organisationen sind in Tabelle 2.3 zusammengefasst dargestellt.

Funktionsorganisation	Prozessorganisation
vertikale Ausrichtung	horizontale Ausrichtung
starke Arbeitsteilung	Arbeitsintegration
Verrichtungsorientierung	Objektbearbeitung
Tiefe Hierarchie	Flache Hierarchien
Statusdenken	Unternehmerrisches Erfolgsdenken
Machtorientierung	Kunden- und Terminorientierung
Abteilungsziele	Prozessziele
Ziel: Kosteneffizienz	Ziel: Kundenzufriedenheit, Produktivität
Zentrale Fremdkontrolle	Dezentrale Kontrolle
Kontrollierte Informationen	Freie und offene Informationen
Ratioprojekte	Kontinuierliche Verbesserung
Ersatzprozesse, Redundanz	Konzentration auf Wertschöpfung
Komplexität	Transparenz

Tabelle 2.3: Funktionsorganisation vs. Prozessorganisation (Schmelzer und Sesselmann [2010], S. 73)

Bei der Prozessmodellierung werden die unternehmenskritischen Prozesse identifiziert, die der Erreichung eines bestimmten Ergebnisses dienen. Dabei können suboptimale und überflüssige (Teil-)Prozessse erkannt, optimiert und korrigiert beziehungsweise beseitigt werden. Dies fördert die Fokussierung der Anstrengungen der Organisation auf ihre Prozessziele. Durch die Korrekturen werden außerdem die Ressourcen besser eingesetzt, wodurch zeitgleich Effektivität und Effizienz steigen. Ein weiterer Effekt ist, dass durch die Prozessdarstellung eine Transparenz der Arbeitsabläufe entsteht und somit die Möglichkeit besteht, gezielt weitere Verbesserungen durchzuführen. Ergänzend ist aufgrund der Transparenz klar, was zu tun ist und wer dafür verantwortlich ist. Dies gewährleistet zum einen vorhersehbare, zum anderen beständige Ergebnisse. Zusätzlich ermöglicht es der Organisation Schwachstellen zu erkennen und Gegenmaßnahmen einzuleiten. Des Weiteren kann auf unerwartete Situationen schneller reagiert werden. Insgesamt führt dies zu einer Senkung der Kosten und dem wirksameren Einsatz der Ressourcen jeder Art. Dies wiederum führt zu einer Steigerung des Vertrauens von beispielsweise Kunden in die Leistung der Organisation.[77]

[76] Vgl. Schmelzer und Sesselmann [2010], S. 75f
[77] Vgl. DIN e.V. [2011], S. 2f

Obwohl die Auseinandersetzung mit dem BPM und der Prozessdarstellung auf den ersten Blick einen Mehraufwand bedeutet, so zahlt sich die Prozessorientierung und -darstellung auf lange Sicht gesehen aus. Sollten sich Anforderungen oder Gegebenheiten verändern, können die anzupassenden Prozesse schnell identifiziert werden, wodurch das Unternehmen einen Wettbewerbsvorteil erhält.

2.4 Fazit

In den vorangegangenen Ausführungen wurden wichtige Begriffe dieser Arbeit erläutert. Es wurde dargestellt, was die Begriffe Modell, Modelltransformation und Prozess bedeuten und in welchem Zusammenhang sie zueinander stehen. Zusätzlich wurde ein Blick auf die Grundsätze ordnungsgemäßer Modellierung geworfen, welche die Modellierung vereinfachen und zu qualitativ höherwertigen Modellen führen.

Des Weiteren wurde herausgestellt, welche Elemente ein Modellierungsstandard enthalten muss, um die Prozessdarstellung zu ermöglichen. Neben einem festen Start- und Endpunkt müssen in einem Prozess der zeitliche Ablauf von einzelnen Aktionen und die Beteiligten an diesen Aktionen erkennbar sein. Mittels Steuerungselementen kann der Prozess abhängig von verschiedenen Faktoren gelenkt werden, um so das gewünschte Ergebnis zu erreichen.

Es wurde ebenfalls aufgezeigt, dass der funktionsorientierte Ansatz, den Unternehmen früher verfolgten, den Anforderungen der heutigen Zeit nicht mehr gerecht wird. Durch starres Funktionsdenken werden Prozesse unterbrochen und Verantwortungen hin- und hergeschoben. Das Gesamtergebnis für den Kunden wird nicht als Ganzes betrachtet, sondern nur als Ergebnisse der einzelnen Abteilungen. Der prozessorientierte Ansatz ermöglicht eine Orientierung auf den Kunden und ist in Zeiten der Globalisierung, der Sättigung der Märkte und den gestiegenen Ansprüchen der Kunden von großer Bedeutung.

3 Die Modellsprachen BPMN und UML

In diesem Kapitel wird zunächst die Object Management Group vorgestellt, welche die beiden Modellsprachen BPMN und UML zu ihren Standards zählt. Es wird ebenfalls betrachtet, wie die Object Management Group einen solchen Standard erhält und was sie für dessen Pflege tut. Im Anschluss werden die beiden ausgewählten Modellsprachen näher erläutert. Dazu erfolgt jeweils ein kurzer Blick auf ihren Ursprung sowie ihren Zweck. Danach schließt je eine kritische Betrachtung bezüglich ihrer Eignung, Prozesse darstellen zu können, an. Dies geschieht anhand des Workflow Pattern Framework, welcher ebenfalls kurz vorgestellt und anhand dessen im weiteren Verlauf dieser Arbeit ein Transformationsweg zwischen den Modellsprachen erarbeitet wird. Im Anschluss werden die wichtigsten Notationselemente der beiden Modellsprachen kurz dargestellt.

3.1 Object Management Group (OMG)

Die Object Management Group (OMG) ist eine internationale Non-Profit-Organisation, welche sich mit Computerindustriestandards beschäftigt. Das Konsortium wurde im Jahr 1989 von elf Unternehmen, darunter die International Business Machines Corporation (IBM), die Apple Inc. und Sun Microsystems, gegründet und hat seinen Hauptsitz derzeit in Needham, Massachusetts, USA. Das Mission Statement der OMG lautet:

> *OMGs Mission ist es, mit Hilfe unserer weltweiten Mitglieder Enterprise Integration Standards zu entwickeln, die einen Wert für die reale Welt schaffen. Die OMG hat sich ebenfalls dem Zusammenbringen von Endnutzern, Regierungsbehörden, Universitäten und Forschnungseinrichtungen in unseren Praxisgemeinschaften verschrieben, um Erfahrungen beim Übergang zu neuen Management- und Technologie-Ansätzen, wie dem Cloud Computing, zu teilen.*[78]

Zurzeit sind mehrere hundert Unternehmen aus unterschiedlichsten Bereichen wie dem Finanz- und Gesundheitswesen aber auch der Automobilindustrie und dem Versicherungswesen in der OMG vertreten. Weiterhin ist laut Angaben der OMG nahezu jede große Organisation aus der Technologieindustrie beteiligt.[79] Somit besteht eine breite und vor allem praxisnahe Wissensbasis für die Ent- und

[78] Vgl. OMG [2013c]
[79] Vgl. OMG [2013b], S. 1ff

Weiterentwicklung der Standards. Bei den viermal jährlich stattfindenden technischen Konferenzen können sich die Mitglieder der verschiedenen Arbeitsgruppen persönlich treffen und austauschen sowie OMG-Mitglieder und andere Interessierte sich über die aktuelle Arbeit des Konsortiums informieren.[80]

Die Standards beziehungsweise Spezifikationen der OMG werden in Kooperation mit Herstellern, Anwendern, akademischen Institutionen sowie Regierungsbehörden in sogenannten Task Forces (Arbeitsgruppen) entwickelt. Ein neuer Standard wird erst verabschiedet, wenn er einen mehrstufigen Prozess durchlaufen hat, in dem bewiesen wurde, dass er in der Praxis benötigt wird und sich implementieren lässt.[81] Dieser Prozess sieht wie folgt aus[82]: Der erste Schritt bei der Methodeneinführung ist optional. Dabei wird eine sogenannte Request for Information (RFI)[83] erstellt, um einer Arbeitsgruppe die Informationen zukommen zu lassen, die sie benötigt, um ein industrielles Problem zu lösen. Hier werden der Gruppe allgemeine und spezifische Anforderungen der Industrie vermittelt und / oder potenzielle Hilfe in Form von beispielsweise technischen Ressourcen offengelegt. Außerdem kann die Arbeitsgruppe auf diese Weise ihren Plan zur Lösungserstellung überprüfen und gegebenenfalls überarbeiten und anpassen.

Der nächste Schritt ist die Request for Proposal (RFP)[84], bei der die Bedürfnisse der Industrie aufgeführt sind und Software-Entwickler dazu eingeladen werden, eine Lösung für das Problem zu entwickeln. Die Identifikation der Bedürfnisse ist die „Krönung des Erfahrungsaustausches einer technischen OMG-Gruppe (sei es eine Arbeitsgruppe oder eine Gruppe mit speziellen Interessen)“. Bevor so eine RFP allerdings angefertigt wird, muss verifiziert sein, dass es jemanden gibt, der darauf reagiert und sich verpflichtet, eine Lösung zu implementieren. Um dies zu gewährleisten muss mindestens ein Mitgliedsunternehmen der OMG einen sogenannten Letter of Interest (LOI)[85] einreichen. In diesem LOI bekundet das Unternehmen seine Absicht, eine Lösung für die in der RFP benannten Bedürfnisse zu entwickeln. Wenn es so ein Unternehmen gibt, wird die RFP angefertigt.

Das Unternehmen hat anschließend Zeit, bis drei Wochen vor der nächsten technischen Konferenz eine erste Vorlage für die Umsetzung einzureichen. Diese wird während der Konferenz diskutiert und kommentiert, und das Unternehmen hat einen gewissen Zeitraum zur Verfügung, um eine überarbeitete Version zu präsentieren. Auch diese wird von den OMG-Mitgliedern gelesen und bewertet. Wenn die Mehrheit die überarbeitete Version für angemessen hält, beginnt die Abstimmung über die Annahme als OMG-Spezifikation. Diese Abstimmung findet während der Konferenz statt, die der Einreichungsfrist der überarbeiteten Version direkt folgt. Gesetz dem Fall, es erfolgt keine Ablehnung durch eine der höheren Instanzen, erhält der Vorschlag den Status *„angenommene Spezifikation“*, hat

[80] Vgl. OMG [2013c]
[81] Vgl. OMG [2013b], S. 2, und OMG [2013c]
[82] Vgl. OMG [2013d], S. 1f
[83] Request for Information, zu deutsch: Informationsanfrage
[84] Request for Proposal, zu deutsch: Angebotsanfrage
[85] Letter of Interest, zu deutsch: Interessensbekundung

jedoch noch keine Versionsnummer.

Um diese und den Status *„gültige Spezifikation“* zu erhalten, wird eine finale Arbeitsgruppe gebildet, die erste Wartungsarbeiten an der Spezifikation durchführt und eine Implementierung erstellt. Die von dieser Arbeitsgruppe überarbeitete Version wird durch die höheren Instanzen bestätigt, erhält eine offizielle Versionsnummer und gilt als gültige Spezifikation der OMG. In diesem Zeitraum erscheinen die (Software-)Produkte auf dem Markt. Auf diese Weise wird gewährleistet, dass ein Standard wirklich benutzt und nicht nur entwickelt, dann aber vergessen wird. Im Weiteren durchläuft die Spezifikation den Wartungskreis der OMG, das heißt, dass es eine Arbeitsgruppe gibt, die Belange bezüglich der Spezifikation sammelt und diese behandelt. Wenn es nötig wird, wird eine neue Version der Spezifikation veröffentlicht.

Auf diese Weise gewährleistet die OMG, dass Standards aktuell bleiben und benutzt werden, denn ein Standard würde von der OMG nicht (weiter-)entwickelt werden, wenn sich nicht mindestens ein Unternehmen um ihn kümmern würde und die Mehrheit der Mitglieder für die Spezifikation stimmen. Zwei dieser Standards der OMG sind die Modellsprachen BPMN und UML.

3.2 Business Process Model and Notation (BPMN)

3.2.1 Ursprung und Zweck

Ursprünglich stand Business Process Model and Notation (BPMN) für *Business Process Modeling Notation*. Diese wurde maßgeblich vom IBM-Mitarbeiter Stephen A. White entwickelt und 2004 von der Business Process Management Initiative (BPMI) in der Version 1.0 veröffentlicht. Im Jahr 2005 wurde die BPMI von der OMG übernommen und BPMN wurde 2006 zu einem OMG-Standard.[86] Dies ist auch in Abbildung 1.2 (auf Seite 3) zu sehen. Die OMG entwickelte die BPMN weiter und veröffentlichte im Jahr 2008 die Version 1.1 und ein Jahr darauf die Version 1.2. Seitdem steht BPMN für *Business Process Model and Notation*. Vor zwei Jahren veröffentlichte die OMG die Version 2.0[87], welche derzeit „als State of the Art bei der Modellierung von Geschäftsprozessen und deren Automatisierung“[88] gilt.

„Das primäre Ziel der BPMN Anstrengungen war es, eine Notation zur Verfügung zu stellen, die leicht verständlich für alle Business-Anwender ist; von den Business-Analysten, welche die ersten Konzepte für die Prozesse entwerfen, über die technischen Entwickler, die verantwortlich für die Umsetzung der Technologie sind, die diese Prozesse durchführt, bis hin zu den Geschäftsleuten, die diese Prozesse verwalten und überwachen.“[89]

[86] Vgl. Freund und Rücker [2012], S. 8f
[87] Vgl. OMG [2013a], Reiter „Documents“
[88] Vgl. Göpfert und Lindenbach [2012], S. V
[89] Vgl. White [2004a], S. 1

Die BPMN wurde entwickelt, um die Lücke zwischen diesen Parteien zu schließen und die Geschäftsprozesse für alle verständlich zu modellieren. Dadurch unterscheidet sie sich beispielsweise von der Unified Modeling Language (UML) (siehe Abschnitt 3.3) und der SysML[90]. Durch ihre graphische Notation sollte dies bewerkstelligt werden.[91]

Mit der BPMN werden unter anderem sogenannte Business Process Diagrams (BPDs)[92] erstellt. In ihnen wird ein theoretisches Marken-Konzept (Tokens) umgesetzt[93], welches der Petri-Netz-Logik sehr ähnlich ist.[94] Token werden produziert und am Ende konsumiert; gegebenenfalls geschieht dies implizit. Die Token wandern über den Sequenzfluss durch den Prozess. Eine Prozess-Instanz gilt als erzeugt, sobald ein Token generiert worden ist. Sie ist beendet, wenn alle erzeugten Token konsumiert worden sind.[95]

Weitere Diagrammtypen der BPMN sind die Kollaboration, die Konversation sowie die Choreographie.[96] Diese Diagrammtypen werden in dieser Arbeit im Weiteren jedoch nicht betrachtet. Wenn im Folgenden von „BPMN“ die Rede ist, bezieht sich dies ausschließlich auf die Prozessdiagramme.

3.2.2 Eignung für die Prozessdarstellung

Es stellt sich die Frage, in wie weit die BPMN überhaupt für die Prozessdarstellung geeignet ist. In Anbetracht der Tatsache, dass diese Modellsprache für die Geschäftsprozessdarstellung entwickelt wurde, sollte anzunehmen sein, dass sie sich für die Prozessdarstellung eignet, dennoch sind dieser Frage Wohed et al. ausführlich nachgegangen und haben ihre Ergebnisse in mehreren Publikationen[97] veröffentlicht. Wohed et al. haben dort die BPMN Version 1.0 gegen die von Wil van der Aalst und Arthur ter Hofstede, beides Co-Autoren der Veröffentlichungen, entwickelten Workflow Patterns getestet.

Die Workflow Pattern Initiative (WPI) begann 1999 und ist ein gemeinsames Projekt der Technischen Universitäten Eindhoven und Queensland. „Das Ziel dieser Initiative ist es, eine konzeptuelle Grundlage für Prozesstechnologien zur Verfügung zu stellen.“[98] Es werden dabei verschiedene Perspektiven betrachtet, die nach Ansicht der Entwickler von Sprachen für die Arbeitsablauf- und Geschäftsprozessdarstellung wichtig sind, so zum Beispiel der Kontrollfluss.

---

[90] Systems Modeling Language - Sie stellt eine um spezielle Elemente erweiterte Untermenge der Unified Modeling Language dar, um komplexe Systeme zu modellieren.
[91] Vgl. Ko et al. [2009], S. 756
[92] Business Process Diagram, zu deutsch: Geschäftsprozessdiagramm
[93] Vgl. OMG [2011a], S. 27
[94] Vgl. Ko et al. [2009], S. 757
[95] Vgl. Göpfert und Lindenbach [2012], S. 11
[96] Vgl. OMG [2011a], S. 31, Freund und Rücker [2012], S. 142ff, und Zimmer [2012], S. 8ff
[97] Vgl. Wohed et al. [2005a] und Wohed et al. [2006]
[98] Vgl. WPI [2011c]

Der Workflow Pattern Framework (WPF)[99] beinhaltete zum Zeitpunkt der Publikationen 20 Kontrollflussmuster (Control Flow Patterns), 40 Datenmuster (Data Patterns) sowie 43 Ressourcenmuster (Resource Patterns).

In den Tabellen 3.1 und 3.2 sind die 20 Kontrollflussmuster mit je einer Kurzbeschreibung aufgeführt. Auf eine Übersetzung der Bezeichnungen wurde an dieser Stelle verzichtet, um Übersetzungsfehlern vorzubeugen. Eine detaillierte Beschreibung der einzelnen Muster befindet sich in Anhang A.

Bezeichnung	Kurzbeschreibung
Basic Control Flow	
1. Sequence	Nachdem eine Aufgabe im Prozess beendet wurde, wird die nächste Aufgabe im Prozess angesteuert.
2. Parallel Split	Es findet eine Aufspaltung des Kontrollflusses in zwei oder mehr parallel ablaufende Kontrollflüsse statt.
3. Synchronisation	Es findet eine Zusammenführung von zwei oder mehr Kontrollflüssen in einem Kontrollfluss statt. Es muss auf alle eingehenden Kontrollflüsse gewartet werden.
4. Exclusive Choice	Aus mehreren Alternativen wird genau eine ausgewählt, um den Kontrollfluss fortzusetzen.
5. Simple Merge	Es findet eine Zusammenführung von zwei oder mehr Kontrollflüssen in einem Kontrollfluss statt.
Adv. Synchronisation	
6. Multiple Choice	Aus mehreren Alternativen werden maximal alle ausgewählt, um den Kontrollfluss fortzusetzen.
7. Synchronising Merge	Es findet eine Zusammenführung von zwei oder mehr Kontrollflüssen in einen Kontrollfluss statt. Es muss auf alle eingehenden Kontrollflüsse gewartet werden. Das Muster bezieht auf einen eindeutig identifizierbaren vorausgegangenen Punkt im Prozessablauf.
8. Multiple Merge	Es findet eine Zusammenführung von zwei oder mehr Kontrollflüssen in einen Kontrollfluss statt.
9. Discriminator	Von mehreren, konkurrierenden Möglichkeiten wird genau die erste ankommende akzeptiert. Alle weiteren werden verworfen.
Structural Patterns	
10. Arbitrary Cycles	Eine Schleife kann über mehrere Punkte betreten und verlassen werden.
11. Implicit Termination	Eine Prozess- oder Sub-Prozess-Instanz wird beendet, wenn es keine weiteren Ausführungsoptionen gibt.

Tabelle 3.1: Kurzübersicht der Kontrollflussmuster des WPF

Bei den *Kontrollflussmustern* werden sechs Gruppen unterschieden: *Basismuster* (Basic Control Flow), *erweiterte (Verzweigung und) Synchronisation* (Advanced (Branching and) Synchronisation), *strukturelle Muster* (Structural Patterns), *Muster für Multi-Instanzen* (Multiple Instances Patterns), *statusbasierte Muster* (State-Based Patterns) sowie *Abbruchmuster* (Cancellation Patterns). Basismuster definieren elementare Aspekte der Prozesskontrolle wie die Reihenfolge der einzelnen Aufgaben, während die Muster für die erweiterte Synchronisation der fortgeschrittenen Verzweigung und Synchronisation des Prozessflusses dienen,

[99] Workflow Pattern Framework, zu deutsch: Rahmen für Arbeitsablaufmuster

Bezeichnung	Kurzbeschreibung
Basic Control Flow	
Multiple Instances P.	
12. MI without Synchronization	Von einer Aufgabe können mehrere, von einander unabhängige Instanzen erzeugt werden. Es besteht keine Forderung zur abschließenden Synchronisation.
13. MI with a priori Design Time Knowledge	Von einer Aufgabe können mehrere, von einander unabhängige Instanzen erzeugt werden. Die Anzahl der Instanzen ist zum Zeitpunkt der Erstellung bekannt und es ist eine abschließende Synchronisation erforderlich, bevor weitere Aufgaben begonnen werden können.
14. MI with a priori Run Time Knowledge	Von einer Aufgabe können mehrere, von einander unabhängige Instanzen erzeugt werden. Die Anzahl der Instanzen ist abhängig von Faktoren, die erst während der Laufzeit des Prozesses bekannt werden, und es ist eine abschließende Synchronisation erforderlich, bevor weitere Aufgaben begonnen werden können.
15. MI without a priori Run Time Knowledge	Von einer Aufgabe können mehrere, von einander unabhängige Instanzen erzeugt werden. Die Anzahl der Instanzen ist abhängig von Faktoren, die erst während der Laufzeit des Prozesses bekannt werden. Es können jederzeit weitere Instanzen erzeugt werden und es ist eine abschließende Synchronisation erforderlich, bevor weitere Aufgaben begonnen werden können.
State-Based Patterns	
16. Deferred Choice	Aus mehreren Alternativen wird eine abhängig von äußeren Faktoren ausgewählt.
17. Interleaved Parallel Routing	Die Aufgaben können in beliebiger Reihenfolge ausgeführt werden. Es muss jede Aufgabe ausgeführt werden und es kann je nur eine Aufgabe ausgeführt werden.
18. Milestone	Eine Aufgabe wird erst dann ausgeführt, wenn eine Prozess-Instanz einen bestimmten Status erreicht hat.
Cancellation Patterns	
19. Cancel Activity	Eine Aufgabe wird vor ihrem natürlichen Ende abgebrochen.
20. Cancel Case	Eine Prozess-Instanz wird vor ihrem natürlichen Ende abgebrochen. Dies inkludiert alle laufenden Aufgaben sowie alle nachfolgenden.

Tabelle 3.2: Kurzübersicht der Kontrollflussmuster des WPF (Fortsetzung)

zum Beispiel durch Mehrfachauswahl. Die strukturellen Muster prüfen, ob es Einschränkungen gibt bezüglich der Art und Weise wie Prozesse gestaltet werden können, beispielsweise ob Schleifen möglich sind. Die Muster für Multi-Instanzen umfassen Situationen, in denen mehr als eine Aktivitätsinstanz zur gleichen Zeit für die gleiche Prozessinstanz aktiv ist. Die statusbasierten Muster charakterisieren Situationen, in denen der weitere Prozessverlauf vom Status der Prozessinstanz abhängig ist. Die Abbruchmuster beschreiben die Fähigkeit, den Abbruch einer einzelnen Aktivität oder eines ganzen Prozesses zu verwirklichen.[100]

[100]Vgl. Wohed et al. [2005a], S. 3ff

	BPMN
Basic Control Flow	
1. Sequence	+
2. Parallel Split	+
3. Synchronisation	+
4. Exclusive Choice	+
5. Simple Merge	+
Advanced Synchronisation	
6. Multiple Choice	+
7. Synchronising Merge	+/–
8. Multiple Merge	+
9. Discriminator	+
Structural Patterns	
10. Arbitrary Cycles	+
11. Implicit Termination	+
Multiple Instances Patterns	
12. MI without Synchronization	+
13. MI with a priori Design Time Knowledge	+
14. MI with a priori Run Time Knowledge	+
15. MI without a priori Run Time Knowledge	–
State-Based Patterns	
16. Deferred Choice	+
17. Interleaved Parallel Routing	+/–
18. Milestone	–
Cancellation Patterns	
19. Cancel Activity	+
20. Cancel Case	+

Tabelle 3.3: Unterstützung der Kontrollflussmuster des Workflow Pattern Framework [Wohed et al., 2005a, S. 13]

In Tabelle 3.3 sind die einzelnen Kontrollflussmuster sowie deren Unterstützung durch die BPMN abgebildet. Ein „+“ in der Tabelle bedeutet direkte Unterstützung des Musters, ein „+/–“ eine teilweise Unterstützung und ein „–“ ein Fehlen der Unterstützung. Es ist deutlich zu erkennen, dass die BPMN fast alle Kontrollflussmuster direkt oder wenigstens teilweise unterstützt. Einzig die Multi-Instanzen ohne a priori Wissen zur Laufzeit (MI without a priori Run Time Knowledge) und der Meilenstein (Milestone) erfahren keine volle Unterstützung durch die BPMN. Für beides bieten Wohed et al.[101] Hilfskonstruktionen an. In Abschnitt 4.2.2 wird betrachtet, wie praktikabel diese Hilfskonstruktionen sind.

Die *Datenmuster* werden in fünf Gruppen organisiert: *Muster zur Datensicht* (Data Visibility), *Muster zur Dateninteraktion (intern)* (Data Interaction Internal), *Muster zur Dateninteraktion (extern)* (Data Interaction External), *Muster zum Datentransfer* (Data Transfer) sowie *Muster zum datenbasierten Routing* (Data-based Routing). Die Muster für die Datensicht charakterisieren die Möglichkeiten, Daten innerhalb des Prozesses zu definieren und zu benutzen. Die Muster für die Dateninteraktion, sowohl intern als auch extern, beziehen sich auf die Möglichkeiten wie Daten zwischen Komponenten einer Prozessinstanz sowie der Arbeitsumgebung ausgetauscht werden können. Wie der Name der Datentransfermuster schon impliziert, geht es bei diesen Mustern um die Art und Weise, wie Daten wirklich zwischen Prozesselementen ausgetauscht werden. Es geht hier um die Mechanismen mit deren Hilfe ein Datenelement die Schnittstelle eines Prozesselementes passiert. Die letzte Gruppe sind die Muster zum datenbasierten Routing. Diese Muster

[101] Vgl. Wohed et al. [2005a], S. 10 und S. 12

erfassen die Möglichkeiten, in denen ein Datenelement mit anderen Perspektiven umgeht und somit die Gesamtausführung des Prozesses beeinflusst.[102]

Die *Ressourcenmuster* werden in sieben Gruppen organisiert: *Erstellungsmuster* (Creation Patterns), *Anstoßmuster* (Push Patterns), *Holmuster* (Pull Patterns), *Umleitungssmuster* (Detour Patterns), *Auto-Start-Muster* (Auto-Start Patterns), *Sichtmuster* (Visibility Patterns) und *Mehrfachressourcenmuster* (Multiple Resource Patterns). Die Erstellungsmuster prüfen die Beschränkungen für die Art und Weise, in der bestimmte Arbeitsaufgaben von Ressourcen beworben, ihnen zugeordnet und von ihnen ausgeführt werden können. Die Anstoßmuster beschreiben Situationen, in denen ein System pro-aktiv Arbeit einer Ressource anbietet oder zuteilt. Bei den Holmustern ist dies umgekehrt: Die Ressourcen zeigen an, welche Arbeit sie erledigen können und verpflichten sich zu deren Ausführung. Die Umleitungsmuster beschreiben Abweichungen von den normalen Zustandsübergängen im Prozess. Die Auto-Start-Muster beziehen sich auf Situationen, bei denen bestimmte Ereignisse oder Zustandsübergänge die Ausführung von Aufgaben automatisch auslösen. Die Sichtmuster beschreiben die Fähigkeit einer Ressource, den Zustand eines Systems zu erkennen, während die Mehrfachressourcenmuster Szenarien beschreiben, bei denen es eine „n zu m“ Beziehung zwischen Ressourcen und Aufgaben gibt, also mehrere Ressourcen für mehrere Aufgaben vorhanden sind.[103]

Eine detaillierte Übersicht über die Daten- und Ressourcenmuster, inklusive einer genauen Beschreibung, sowie deren Unterstützung durch die BPMN befindet sich in Anhang A. Diese Muster werden in dieser Arbeit nicht näher betrachtet, weil für die *Prozessdarstellung* einzig die Kontrollflussmuster von besonderem Interesse sind. Diese können direkt graphisch durch die Modellsprache beschrieben werden. Es sei jedoch angemerkt, dass die BPMN nur etwa die Hälfte der Datenmuster des Workflow Pattern Frameworks unterstützt und noch weniger von den Ressourcenmustern.

Wie bereits erwähnt, werden die *Kontrollflussmuster* mittels der Notationselemente der Modellsprache dargestellt. Von den Kontrollflussmustern sind die Basismuster von großem Interesse, da sie die grundlegende Prozessdarstellung gewährleisten: Darstellung einer Reihenfolge von Aktivitäten, parallele Ausführung von Aktivitäten, Synchronisation, exklusive Auswahl einer Alternative sowie einfache Zusammenführung. Die Elemente der erweiterten Verzweigung und Synchronisation sorgen dafür, dass der Kontrollfluss besser gelenkt werden kann als einzig mit den Basiselementen. Die Mehrfachauswahl ermöglicht es beispielsweise neben der Auswahl eines einzigen oder aller Stränge auch die Auswahl einer Anzahl dazwischen. Der Diskriminator ermöglicht es im Gegenzug, dass von mehreren eingehenden Alternativen nur die erste fortgeführt wird. Die strukturellen Muster stellen sicher, dass zum Beispiel Schleifen modelliert werden können und dass ein (Sub-)Prozess beendet wird, wenn keine weitere Ausführung möglich ist. Für die Prozessdarstellung scheinen die Multi-Instanzen-Muster erst einmal weniger von

[102] Vgl. Wohed et al. [2005a], S. 14ff
[103] Vgl. Wohed et al. [2005a], S. 18ff

Bedeutung zu sein. Aus diesem Grund werden sie im Rahmen dieser Arbeit nicht eingehender betrachtet. Die status-basierten Muster greifen Impulse von außerhalb auf. Ein einfaches Beispiel für die verzögerte Entscheidung ist der Anruf eines Kunden, von welchem die Art der weiteren Fortführung eines Prozesses abhängig ist. Ein Meilenstein hingegen ist ein besonderer Punkt im Verlauf eines Prozesses: Der nächste Schritt kann erst gemacht werden, wenn der Meilenstein erreicht wurde. Die letzte Gruppe stellen die Abbruchmuster dar. Diese Muster gewährleisten, dass unter bestimmten Umständen eine Aktivität oder ein Prozess augenblicklich abgebrochen werden kann.

Da somit (fast) alle Kontrollflussmuster wenigstens indirekt dargestellt werden können, ist die BPMN 1.0 geeignet für die *Prozessdarstellung*, auch wenn sie offenbar Defizite bezüglich der Daten- und Ressourcenmuster hat. Alle Elemente, die für die Darstellung der Kontrollflussmuster benötigt werden, sind auch in der BPMN 2.0 vorhanden[104], weshalb auch diese selbst für die Prozessdarstellung geeignet ist.

3.2.3 Notation der BPMN

Die Notationselemente der BPMN sind seit der Version 2.0 in fünf Basiskategorien arrangiert: *Flussobjekte*, *Verbindungsobjekte*, *Daten*, *Teilnehmer* und *Artefakte*.[105] Die geringe Anzahl an Kategorien soll es dem Leser der Diagramme ermöglichen, die Basistypen der Elemente leicht zu identifizieren und somit die Diagramme schnell zu verstehen. Zu den Flussobjekten zählen *Ereignisse*, *Aktivitäten* und *Gateways*. Als Verbindungsobjekte dienen die *Sequenzflüsse*, die *Nachrichtenflüsse*, die *Assoziationen* sowie die *Datenassoziationen*. Zu den Daten gehören die *Datenobjekte*, die *Datenein-* und *-ausgaben* sowie die *Datenspeicher*. Für die Teilnehmer gibt es zum einen den *Pool*, zum anderen die *Lane*. Zu guter Letzt gibt es noch die *Gruppe* und die *Anmerkung*, welche als Artefakte dienen.[106]

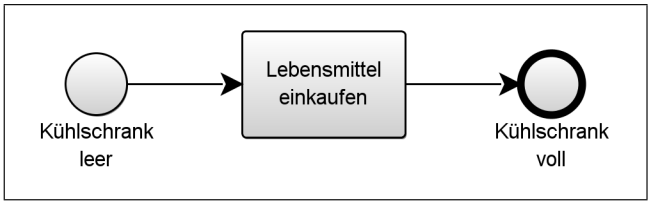


Abbildung 3.1: Einfacher Beispielprozess in BPMN

In Abbildung 3.1 ist beispielhaft ein einfacher Prozess dargestellt, der zwei der Notationselemente der BPMN verwendet: das Ereignis und die Aktivität. Das Ereignis auf der linken Seite wird in seiner einfachsten Form verwendet und zeigt an, dass es sich um den Start des Prozesses handelt, während das Ereignis auf der rechten Seite variiert wurde (dicker Rand), um zu verdeutlichen, dass es sich um

[104]Vgl. OMG [2009], S. 17ff und OMG [2011a], S. 27ff
[105]Vgl. OMG [2011a], S. 27
[106]Vgl. OMG [2011a], S. 27f

das Ende des Prozesses handelt.

In der Tabelle 3.4 sind die wichtigsten Basis-Elemente der BPMN mit einer kurzen Beschreibung aufgeführt. Eine ausführlichere Auflistung und detaillierte Beschreibungen der in dieser Arbeit verwendeten Notationselemente befindet sich in Anhang B. Weiterführende Informationen zu den einzelnen Elementen können in der Spezifikation zur BPMN 2.0[107] nachgeschlagen werden.

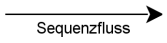
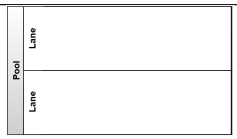
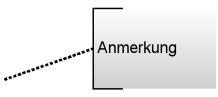
Element	Kurzbeschreibung
	Ein *Ereignis* ist etwas, das im Verlauf eines Prozesses passiert. Die Ereignisse haben Einfluss auf den Ablauf des Modells und haben in der Regel eine Ursache (Trigger) oder eine Auswirkung (Ergebnis).
	Eine *Aktivität* ist ein Oberbegriff für Arbeit, die eine Firma ausführt. Eine Aktivität kann atomar oder nicht-atomar (zusammengesetzt) sein.
	Das *Gateway* wird benutzt, um die Divergenz und die Konvergenz der Sequenzflüsse in einem Prozess zu kontrollieren. Auf diese Weise wird die Verzweigung und Zusammenführung der Pfade bestimmt.
	Der *Sequenzfluss* wird benutzt, um die Reihenfolge darzustellen, in der Aktivitäten in einem Prozess ausgeführt werden.
	Ein *Pool* ist die graphische Darstellung eines Teilnehmers. Ein Pool kann interne Details in Form eines Prozesses, der ausgeführt wird, enthalten.
	Eine *Lane* ist ein Sub-Aufteilung innerhalb eines Pools. Lanes werden verwendet, um Aktivitäten zu organisieren und zu kategorisieren.
	Ein *Datenobjekt* stellt Informationen darüber zur Verfügung, was eine Aktivität benötigt, um ausgeführt zu werden und / oder was diese erzeugt.
	Eine *Anmerkung* ist ein Mechanismus für einen Modellierer, um zusätzliche Textinformationen für den Leser eines Diagramms zur Verfügung zu stellen.

Tabelle 3.4: Kurzübersicht der Notationselemente der BPMN

[107] Vgl. OMG [2011a], S. 29ff

3.3 Unified Modeling Language (UML)

3.3.1 Ursprung und Zweck

Ein weiterer Standard der OMG ist die *Unified Modeling Language* (UML). Diese erschien 1995 erstmals in der Version 0.8, und unter dem Namen *Unified Method*, mit dem Grundgedanken, einen Standard für die graphische Modellierung von Software, vornehmlich objekt-orientierter Software, zu entwickeln. Allerdings ist diese frühe Version noch nicht von der OMG veröffentlicht worden und galt zu diesem Zeitpunkt auch noch nicht als Standard. Vielmehr stellte diese Version eine Zusammenführung der populären Methoden OMT[108] und Booch dar. Beides waren Vertreter der vielen graphischen Methoden, die für den Softwareentwicklungsprozess benutzt worden sind. Die Ideenführer der beiden genannten Methoden sind James Rumbaugh beziehungsweise Grady Booch. Beide gelten neben Ivar Jacobson als die Väter der UML. Jacobson steuerte das Object-Oriented Software Engineering (OOSE) zur UML 0.9 bei.

Im Jahr 1997 wurde die erste offizielle Version der UML (Version 1.0) veröffentlicht. Die Vorteile dieser vereinheitlichten Modellsprache mit den „besten“ Elementen aus den anderen Notationen wurden schnell von namhaften Unternehmen wie der Microsoft Corporation, der Oracle Corporation und IBM erkannt, die sich zu UML-Partnern zusammenschlossen. Damit die Modellsprache als offizieller Industriestandard anerkannt wurde, wurde eine Zertifizierung der Version 1.1 durch die OMG angestrebt. 1999 ist die UML von der OMG als Standard akzeptiert und in der Version 1.3 veröffentlicht worden. In den folgenden Jahren wurde die Modellsprache kontinuierlich weiterentwickelt (bis Version 1.5 im Jahr 2003) bis sie schließlich 2005 erstmals in der Version 2.0 erschienen ist.

Der Sprung in der Versionierung ist mit einigen deutlichen Überarbeitungen der UML zu erklären: Von der Version 1.0 bis hin zur Version 1.5 sind immer wieder kleine Korrekturen vorgenommen und neue Elemente in die Modellsprache aufgenommen worden, um sie aktuell zu halten und „mächtiger“ zu machen, jedoch wurde sie dadurch weniger überschaubar. Außerdem wurde deutlich, dass neue Aspekte in die Diagramme aufgenommen werden müssen. Aus diesem Grund entschied sich die OMG dafür, die UML vollständig zu überarbeiten, dabei aufzuräumen und auszudünnen, und neue Diagrammtypen einzuführen (zum Beispiel das Timing-Diagramm) sowie alte zu überarbeiten (zum Beispiel das Aktivitätsdiagramm). Auch nach der Veröffentlichung der Version 2.0 wurde die UML fortwährend weiterentwickelt, sodass sie derzeit in der Version 2.4.1 vorliegt und demnächst durch die Version 2.5 abgelöst werden soll.[109]

[108] Object-Modeling Technique

[109] Vgl. Fowler [2003], S. 25ff, Kecher [2006], S. 17ff und Oestereich [2012], S. 17f, 21ff

Die UML umfasst derzeit 14 Diagrammtypen, die in zwei großen Gruppen organisiert sind: *Strukturdiagramme* und *Verhaltensdiagramme*.[110] In Abbildung 3.2 ist eine Übersicht der Diagramme mit ihrer Einordnung zu sehen. In dunkelgrau sind die übergeordneten Gruppen dargestellt, in weiß die eigentlichen Diagrammtypen.

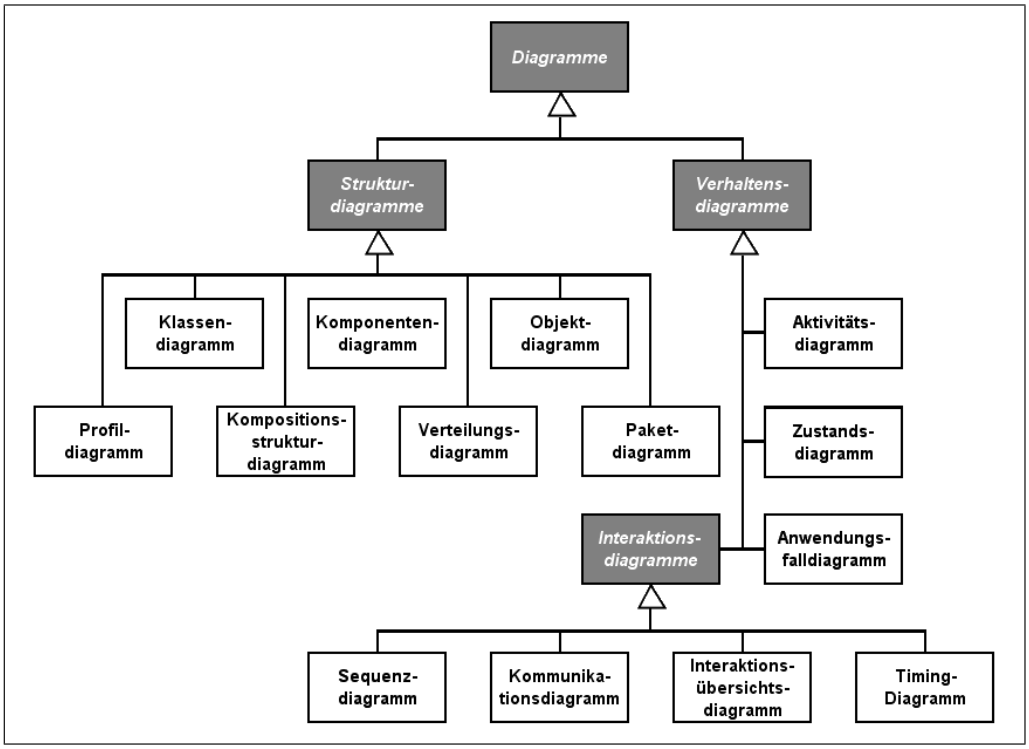


Abbildung 3.2: Diagramme der UML in der Übersicht

Von Interesse für die Prozessdarstellung sind die Diagrammtypen, die sich in der Gruppe der Verhaltensdiagramme beziehungsweise in deren Untergruppe der Interaktionsdiagramme befinden. In der Superstruktur zur UML 2.4.1 steht zu Verhaltensdiagrammen Folgendes:

> *Verhaltensdiagramme zeigen das dynamische Verhalten der Objekte in einem System, einschließlich ihrer Methoden, Kollaborationen, Aktivitäten und Zustandshistorien. Das dynamische Verhalten eines Systems kann als eine Reihe von Änderungen an dem System im Laufe der Zeit beschrieben werden.* [111]

In einem Geschäftsprozess werden ebenfalls Änderungen am vorhandenen System vorgenommen. Von besonderem Interesse ist das Aktivitätsdiagramm der UML. Das Aktivitätsdiagramm ist nach Meinung von Fachleuten der BPMN optisch sehr ähnlich.[112] Außerdem gibt es bereits Ansätze für eine Transformation zwischen der BPMN und den Aktivitätsdiagrammen der UML.[113]

[110] Vgl. OMG [2011b], S. 694f
[111] Vgl. OMG [2011b], S. 694
[112] Vgl. White [2004b], S. 2, 5f, und Ko et al. [2009], S. 759
[113] Vgl. Kalnins und Vitolins [2006], Cibrán [2008] und Macek und Richta [2009]

3.3.2 Eignung für die Prozessdarstellung

Auch für das Aktivitätsdiagramm der UML stellt sich die Frage, in wie weit es sich für die Prozessdarstellung eignet. Mit einem Aktivitätsdiagramm wird meist der Ablauf eines Anwendungsfalls beschrieben. Es bietet einen Blick darauf, wie einzelne Aktionen untereinander über Kontrollflüsse verbunden sind und zusammen spielen. Dennoch wurde es eigentlich für die die Darstellung von Abläufen in Software entwickelt. Der Frage, ob es auch für die allgemeine Prozessdarstellung geeignet ist, sind Russell et al. daher nachgegangen. Wie bei ihrer Arbeit zur BPMN haben sie die UML Aktivitätsdiagramme der Version 2.0 gegen die Muster des Workflow Pattern Frameworks getestet. Ihre Ergebnisse haben sie in mehreren Publikationen veröffentlicht.[114]

In Tabelle 3.5 ist die Unterstützung der Kontrollflussmuster durch die UML Aktivitätsdiagramme abgebildet. Das „AD“ steht dabei für „Aktivitätsdiagramm“. Ein „+“ in der Tabelle bedeutet direkte Unterstützung des Musters, ein „+/–“ eine teilweise Unterstützung und ein „–“ ein Fehlen der Unterstützung.

	UML AD
Basic Control Flow	
1. Sequence	+
2. Parallel Split	+
3. Synchronisation	+
4. Exclusive Choice	+
5. Simple Merge	+
Advanced Synchronisation	
6. Multiple Choice	+
7. Synchronising Merge	–
8. Multiple Merge	+
9. Discriminator	+
Structural Patterns	
10. Arbitrary Cycles	+
11. Implicit Termination	+
Multiple Instances Patterns	
12. MI without Synchronization	+
13. MI with a priori Design Time Knowledge	+
14. MI with a priori Run Time Knowledge	+
15. MI without a priori Run Time Knowledge	–
State-Based Patterns	
16. Deferred Choice	+
17. Interleaved Parallel Routing	–
18. Milestone	–
Cancellation Patterns	
19. Cancel Activity	+
20. Cancel Case	+

Tabelle 3.5: Unterstützung der Kontrollflussmuster des Workflow Pattern Framework [Wohed et al., 2005a, S. 13]

Die Aktivitätsdiagramme der UML unterstützen weniger Kontrollflussmuster als die BPMN. Neben den Multi-Instanzen ohne a priori Wissen zur Laufzeit und dem Meilenstein werden hier auch die synchronisierende Verschmelzung (Synchronising Merge) und das verschachtelte parallele Routing (Interleaved Parallel Routing) nicht unterstützt. Für die synchronisierende Verschmelzung können Wohed et

[114] Vgl. Wohed et al. [2005b] und Russell et al. [2006]

al.[115] keine Abhilfe anbieten. Den Vorschlag von White[116] lehnen sie jedoch als unzureichend ab, da nicht deutlich wird, wie die entsprechende Bedingung aussehen müsste, damit festgelegt ist, wie viele Token zu erwarten sind. Sie zeigen allerdings Möglichkeiten auf, die Multi-Instanzen ohne a priori Wissen zur Laufzeit sowie das verschachtelte parallele Routing und den Meilenstein zu konstruieren. Diese Hilfskonstruktionen werden in Abschnitt 4.2.2 genauer betrachtet.

Obwohl die Daten- und Ressourcenmuster nicht näher betrachtet werden, weil sie für die Prozessdarstellung von untergeordneter Bedeutung sind, sei an dieser Stelle gesagt, dass die UML Aktivitätsdiagramme ebenfalls nur knapp die Hälfte der Datenmuster unterstützen und nur 8 von 43 möglichen Ressourcenmustern. Dies entspricht weniger als einem Fünftel. Da jedoch ein Großteil der Kontrollflussmuster dargestellt werden kann, sind die UML Aktivitätsdiagramme eingeschränkt geeignet für die *Prozessdarstellung*, auch wenn sie Defizite bezüglich der Datenmuster und Ressourcenmuster haben.

3.3.3 Notation der UML Aktivitätsdiagramme

Die UML unterscheidet zwischen über einem Dutzend unterschiedlichen Diagrammtypen und beinhaltet daher sehr viele Notationselemente. Abbildung 3.3 stellt beispielhaft den gleichen Prozess, der in Abbildung 3.1 (auf Seite 34) abgebildet ist, in der Notation eines UML Aktivitätsdiagramms dar. Bis auf die fehlenden Bezeichnungen beim Start- und Endknoten sind diese Abbildungen identisch.

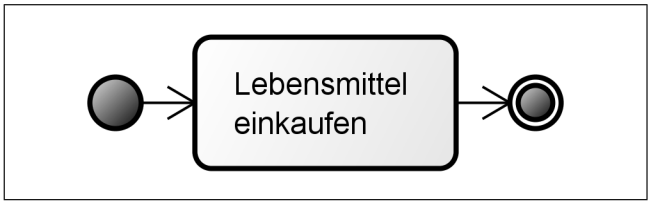


Abbildung 3.3: Einfacher Beispielprozess als UML AD

Seit der UML-Version 2.0 heißt das Aktivitätsdiagramm *Aktivität*, die früheren Aktivitäten werden nun *Aktionen* genannt.[117] Im Weiteren wird in dieser Arbeit die Aktivität als Aktivitätsdiagramm bezeichnet, um Verwirrung zu vermeiden, da es auch noch Aktivitäten gibt, die denen aus der BPMN ähneln.

Mit Aktivitätsdiagrammen können komplexe Abläufe mit ihren Variationen und Ausnahmefällen modelliert werden. Im Gegensatz zu den Versionen 1.x der UML hat sich das Aktivitätsdiagramm ab der UML 2.0 stark der Petri-Netz-Logik angenähert[118], sodass auch in diesem Fall von Token gesprochen wird, die im Laufe des Prozesses produziert und konsumiert werden. Diese Token wandern entlang des

[115] Vgl. Wohed et al. [2005b], S. 7
[116] Vgl. White [2004b], S. 11
[117] Vgl. Oestereich [2012], S. 335
[118] Vgl. Oestereich [2012], S. 335 und S. 339

Kontrollflusses durch die einzelnen Aktionen.

In Tabelle 3.6 und Tabelle 3.7 sind die wichtigsten Elemente der UML Aktivitätsdiagramme mit einer kurzen Beschreibung aufgeführt. Eine ausführlichere Auflistung und detaillierte Beschreibungen der in dieser Arbeit verwendeten Notationselemente befindet sich in Anhang C. Weiterführende Informationen zu den einzelnen Elementen können in der Superstruktur der UML 2.4.1[119] nachgeschlagen werden.

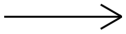
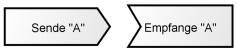
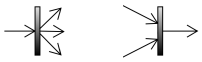
Element	Kurzbeschreibung
	Über *Start-* (links) und *Endknoten* (rechts) wird modelliert, wo ein Aktivitätsdiagramm beginnt beziehungsweise endet. Diese Knoten werden auch innerhalb von Aktivitäten verwendet.
	Eines der wichtigsten Elemente der Aktivitätsdiagramme ist die *Aktion*. Bei einer Aktion handelt es sich um eine im Modell atomare Einheit, die nicht weiter zerlegt wird.
	Aktionen können in *Aktivitäten* eingebettet sein. Auf diese Weise wird das Diagramm übersichtlicher, da Aktivitäten sowohl expandiert als auch zusammengeklappt auftreten können.
	Der *Kontrollfluss* definiert die Ausführungsreihenfolge von Aktionen im Diagramm, indem er diese gerichtet miteinander verbindet.
	Ein *Objektknoten* stellt eine Art Speicher für Objekte dar. Diese werden entweder von einer Aktion benötigt oder erzeugt.
	*Signale* werden benutzt, um einen asynchronen Nachrichtenaustausch zu model- lieren.
	Über *Entscheidungs-* und *Verbindungsknoten* wird der Kontrollfluss innerhalb des Aktivitätsdiagramms in alternative Richtungen gelenkt.
	Eine weitere Möglichkeit zur Lenkung des Kontrollflusses sind die *Gabelung* (links) sowie die *Vereinigung* (rechts).
	Eine *Anmerkung* kann zusätzliche Informationen beinhalten wie beispielsweise Bedingungen und detaillierter Erläuterungen.

Tabelle 3.6: Kurzübersicht der Notationselemente der UML AD

[119] Vgl. OMG [2011b], S.319ff

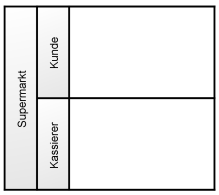
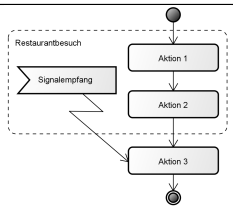
Element	Kurzbeschreibung
	Um Aktionen und Aktivitäten im Diagramm zu organisieren, können *Aktivitätsbereiche*, auch Partitionen genannt, eingesetzt werden. Mit diesen werden beispielsweise unterschiedliche Bereiche und Verantwortliche modelliert.
	Ein *Unterbrechungsbereich* umfasst eine Anzahl von Elementen, deren Ausführung beispielsweise beim Erhalt eines Signals sofort abgebrochen wird.

Tabelle 3.7: Kurzübersicht der Notationselemente der UML AD (Fortsetzung)

3.4 Fazit

Die von der OMG gepflegten Standards werden sowohl kontinuierlich und regelmäßig überprüft als auch gegebenenfalls angepasst und aktualisiert. Da ein Standard erst angenommen wird, wenn sich für ihn Interessenten finden und sich jemand bereit erklärt, ihn zu implementieren, werden keine „toten“, das heißt keine nicht nutzbaren beziehungsweise nicht benutzte Standards veröffentlicht. Es ist also offenkundig, dass sowohl BPMN als auch UML aktiv genutzte Standards sind, da sie weiterhin von der OMG weiterentwickelt werden.

BPMN und UML wurden für unterschiedliche Zwecke kreiert. Während die BPMN eine Notation zur Verfügung stellen soll, die von allen Anwendern, das heißt von den Business-Analysten über die technischen Entwickler bis hin zu den Managern, leicht zu verstehen ist, soll die UML vornehmlich die graphische Modellierung von objekt-orientierter Software ermöglichen.[120] Dennoch gehen die Fähigkeiten der UML darüber hinaus.

Die BPMN und das UML Aktivitätsdiagramm weisen viele ähnliche Notationselemente auf, wie aus den Tabellen in den Abschnitten 3.2.3 und 3.3.3 entnommen werden kann und auch von White[121] bereits aufgezeigt wurde. Was deutlich auffällt ist, dass es in der BPMN häufig mehrere Möglichkeiten gibt, ein und denselben Sachverhalt zu modellieren.[122] Dies erlaubt es zwar dem Modellierer nach weniger strikten Regelungen als beim UML Aktivitätsdiagramm zu modellieren, erschwert allerdings das Lesen des Diagramms für den ungeübten Nutzer.[123] Außerdem behindert es eine eindeutige Transformation zwischen den beiden Modellsprachen.

[120] Vgl. Ko et al. [2009], S. 758
[121] Vgl. White [2004b]
[122] Vgl. White [2004b], S. 2ff, und Cibrán [2008], S. 64ff
[123] Vgl. Macek und Richta [2009], S. 123

Zur Lesbarkeit der Diagramme haben Peixoto et al.[124] eine Studie durchgeführt. Sie haben Erstsemestern der Informatik, die zuvor weder mit der BPMN noch mit UML Aktivitätsdiagrammen zu tun gehabt haben, mehrere Diagramme in diesen Modellierungsstandards gezeigt und Ja-/Nein-Fragen zu diesen beantworten lassen. Obwohl ihre Hypothese war, dass die BPMN-Diagramme leichter verständlich sind als die UML Aktivitätsdiagramme, haben ihre Ergebnisse etwas anderes offenbart: Die Anzahl der korrekt beantworteten Fragen zu beiden Diagrammtypen war annähernd identisch.[125]

In Bezug auf die Eignung zur Prozessdarstellung stellen sich die beiden Modellsprachen als gleich geeignet beziehungsweise ungeeignet dar. Was die Kontrollflussmuster angeht, nehmen sie sich wenig (Tabelle 3.8): Die Muster, die vom UML Aktivitätsdiagramm unterstützt werden, werden ebenfalls von der BPMN unterstützt. Diese bietet jedoch noch eine teilweise Unterstützung von einigen weiteren Mustern. Kontrollflussmuster, die in der BPMN nicht direkt modelliert werden können, können über komplexe Hilfskonstruktionen modelliert werden, wie Wohed et al.[126] gezeigt haben. Beim UML Aktivitätsdiagramm lässt sich nur ein Muster nicht über Hilfskonstruktionen darstellen.[127]

	BPMN	UML AD
Basic Control Flow		
1. Sequence	+	+
2. Parallel Split	+	+
3. Synchronisation	+	+
4. Exclusive Choice	+	+
5. Simple Merge	+	+
Advanced Synchronisation		
6. Multiple Choice	+	+
7. Synchronising Merge	+/–	–
8. Multiple Merge	+	+
9. Discriminator	+	+
Structural Patterns		
10. Arbitrary Cycles	+	+
11. Implicit Termination	+	+
Multiple Instances Patterns		
12. MI without Synchronization	+	+
13. MI with a priori Design Time Knowledge	+	+
14. MI with a priori Run Time Knowledge	+	+
15. MI without a priori Run Time Knowledge	–	–
State-Based Patterns		
16. Deferred Choice	+	+
17. Interleaved Parallel Routing	+/–	–
18. Milestone	–	–
Cancellation Patterns		
19. Cancel Activity	+	+
20. Cancel Case	+	+

Tabelle 3.8: Unterstützung der Kontrollflussmuster des Workflow Pattern Framework [Wohed et al., 2005a, S. 13]

[124]Vgl. Peixoto et al. [2008], S. 4ff
[125]Vgl. Peixoto et al. [2008], S. 8ff
[126]Vgl. Wohed et al. [2005a], S. 10 und S. 12
[127]Vgl. Wohed et al. [2005b], S. 7

Zu den Daten- und Ressourcenmustern, die hier nur untergeordnet betrachtet wurden, ist anzumerken, dass nur wenige vollständig unterstützt werden. Bei den Datenmustern zeigt sich, dass die BPMN und die UML Aktivitätsdiagramme, wenn sie ein Muster unterstützen, zumeist unterschiedliche Muster unterstützen. Bei den Ressourcenmustern wird von beiden weniger als ein Füntel aller Muster unterstützt.

Die Ausführungen haben gezeigt, dass sich die beiden Modellsprachen für die BMPN und die UML Aktivitätsdiagramme ähnlich sind und dass beide die Prozessdarstellung in ausreichendem Maß gewährleisten. Auf dieser Basis wird im folgenden Kapitel ein Transformationsweg erarbeitet, der eine M2M-Transformation zwischen diesen beiden Modellsprachen ermöglicht.

4 Transformation zwischen den Modellsprachen

In diesem Kapitel wird ein Transformationsweg zwischen den Modellsprachen BPMN und UML erarbeitet. Dafür wird zunächst ein intuitiver Transformationsweg beschrieben, welcher jedoch einige Probleme aufweist. Aus diesem Grund wird anschließend auf die Kontrollflussmuster des bereits bekannten Workflow Pattern Framework zurückgegriffen und anhand dieser Muster eine M2M-Transformation angestrebt. Dafür wird zunächst gezeigt, wie die beiden Modellsprachen die jeweiligen Kontrollflussmuster darstellen. Daraus kann abgeleitet werden, welche Elemente benötigt werden, um eine Transformation zu verwirklichen. Im Anschluss daran wird ein Baustein-Prinzip mit diesen Elementen erarbeitet. Die Bausteine fungieren dabei als eine Art Meta-Ebene. Im nächsten Schritt wird gezeigt, wie die erarbeiteten Bausteine konkret in den XML-Strukturen der beiden Modellsprachen aussehen, sowie welche XML-Strukturen aus den Bausteinen erzeugt werden müssen.

4.1 Erste Überlegungen

Der logischste Transformationsweg wäre, eine 1-zu-1-Transformation zwischen der BPMN und den UML Aktivitätsdiagrammen anzustreben. Dies würde heißen, dass ein Element der BPMN genau auf ein Element der UML Aktivitätsdiagramme abgebildet wird. Diesem Ansatz folgend hat Cibrán[128] in ihrer Arbeit eine automatische Überführung von der BPMN zu den UML Aktivitätsdiagrammen angestrebt und eine Liste mit 1-zu-1-Entsprechungen begonnen.

BPMN	UML AD
Aufgabe	Aktion
Sequenzfluss	Fluss
Sub-Prozess	Aktivität
Anmerkung	Anmerkung
Pool	Aktivitätsbereich
Lane	Aktivitätsbereich

Tabelle 4.1: 1-zu-1-Transformation (Beispiel)

Tabelle 4.1 zeigt eine ähnliche, an diese Arbeit angepasste Liste. Es wird deutlich, dass zwar einige Elemente ein eindeutiges Gegenstück in der anderen Modellsprache haben, jedoch gilt dies nicht für alle Elemente. So müssen beispielsweise Pool und Lane der BPMN beide auf den Aktivitätsbereich der UML Aktivitätsdiagramme

[128] Vgl. Cibrán [2008], S. 64f

modelliert werden. Dies führt zu Uneindeutigkeiten bei der Transformation von UML Aktivitätsdiagrammen zur BPMN.

Cibrán führt weitere Probleme bei einer 1-zu-1-Transformation auf, so zum Beispiel die *reichhaltigen* sowie *überladenen* Elemente der BPMN.[129] Unter reichhaltigen Elementen versteht sie Elemente, die mehrere Aufgaben erfüllen. Cibrán führt als Beispiel das Nachrichten-Start-Ereignis der BPMN an, welches einen Prozess beim Empfang einer Nachricht startet. Die UML Aktivitätsdiagramme kennen kein entsprechendes Gegenstück, können aber mittels mehrerer Notationselemente diesen Umstand ebenfalls modellieren. Unter überladenen Elementen versteht Cibrán Elemente, die im Kontext gesehen unterschiedlich interpretiert werden müssen. So zum Beispiel die Gateways. Abhängig davon, wie diese in Beziehung zu anderen Elementen stehen, sind sie unterschiedlich zu interpretieren. Das liegt vornehmlich daran, dass ein Gateway mehrere eingehende sowie ausgehende Sequenzflüsse gleichzeitig haben kann. In Anhang B wurde festgelegt, dass ein Gateway nur eine der beiden Aufgaben, Verzweigung oder Zusammenführung, auf einmal ausführt. Dies vereinfacht Cibráns Ansatz minimal. Eine eindeutige Zuordnung ist dennoch nicht gewährleistet. Dies haben auch Macek und Richta[130] festgestellt. Eine 1-zu-1-Transformation ist also nicht unbedingt der richtige Ansatz.

4.2 Verwendung der Workflow Patterns

Obwohl Wohed et al. beziehungsweise Russell et al. beide Modellsprachen gegen den Workflow Pattern Framework getestet haben, haben sie keine Transformation auf Grundlage dieser Ergebnisse veröffentlicht. In den Abschnitten 3.2.2 und 3.3.2 wurde allerdings gezeigt, dass fast alle Kontrollflussmuster mit beiden Modellsprachen dargestellt werden können. Im Folgenden wird daher die Darstellung der Kontrollflussmuster in den beiden Modellsprachen im Detail betrachtet und daraus abgeleitet, welche Konstrukte der BPMN auf welche der UML Aktivitätsdiagramme auf Grundlage der Kontrollflussmuster abgebildet werden müssen und umgekehrt. Auf diese Weise kann eine Transformation über ein Baustein-Prinzip erreicht werden, welches eine Meta-Ebene darstellt.

4.2.1 Direkt unterstützte Muster

Den Anfang machen dabei die fünf Basismuster. Das erste Basismuster entspricht dem Kontrollflussmuster Nummer 1 (kurz #1; analog dazu alle weiteren Muster) und stellt die Sequenz dar. Sowohl bei der BPMN als auch bei den Aktivitätsdiagrammen der UML wird dies durch eine gerichtete Verbindungslinie zwischen zwei Aufgaben beziehungsweise zwei Aktionen modelliert (Abbildun-

[129] Vgl. Cibrán [2008], S. 65f
[130] Vgl. Macek und Richta [2009], S. 123ff

gen 4.1[131] und 4.2). Der optische Unterschied ist minimal, wie auch schon White[132] feststellte.

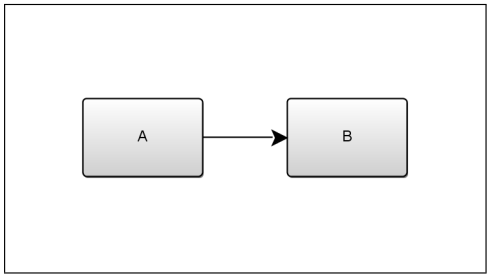


Abbildung 4.1: WCP #1 - BPMN

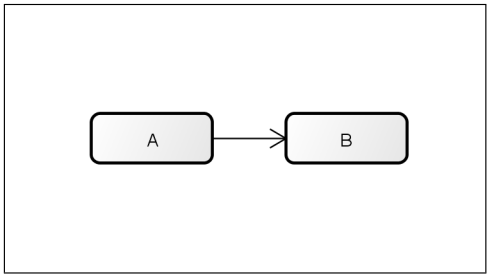


Abbildung 4.2: WCP #1 - UML AD

Das zweite Basismuster stellt die parallele Aufspaltung dar. In der BPMN wird sie mittels eines parallelen Gateways dargestellt (Abbildung 4.3), die UML Aktivitätsdiagramme stellen den Sachverhalt mittels einer Gabelung dar (Abbildung 4.4). In beiden Darstellungen ist es möglich, dass der Fluss in mehr als zwei parallele Flüsse aufgespalten wird. Die Anzahl ist theoretisch unbegrenzt.

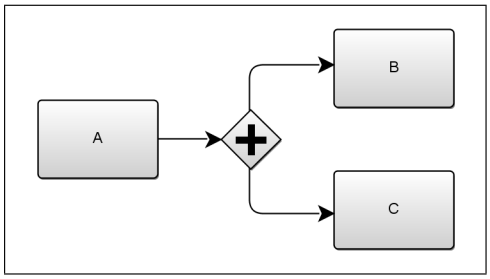


Abbildung 4.3: WCP #2 - BPMN

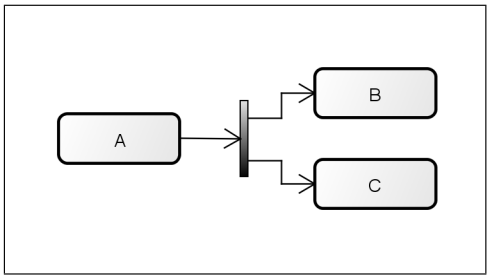


Abbildung 4.4: WCP #2 - UML AD

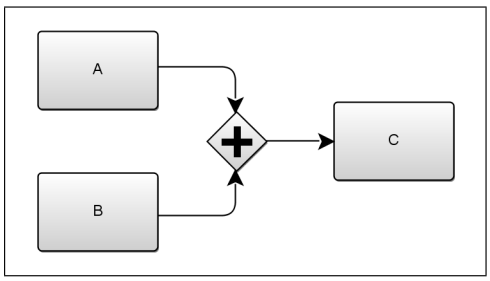


Abbildung 4.5: WCP #3 - BPMN

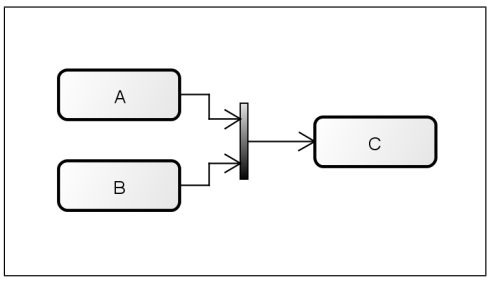


Abbildung 4.6: WCP #3 - UML AD

Auch für das dritte Muster, die Synchronisation, wird bei der BPMN ein paralleles Gateway verwendet (Abbildung 4.5). Bei den UML Aktivitätsdiagrammen wird das Gegenstück zur Gabelung, die Vereinigung, verwendet, um den Sachverhalt zu

[131] WCP steht für Workflow Control Pattern, zu deutsch: Kontrollflussmuster
[132] Vgl. White [2004b], S. 2

modellieren (Abbildung 4.6 auf der vorherigen Seite). Auch hier ist die Anzahl der zusammenzuführenden Flüsse theoretisch unbegrenzt.

Der Unterschied zwischen der Darstellung in der BPMN und den UML Aktivitätsdiagrammen ist offensichtlich, aber die Ähnlichkeiten zwischen den Kontrollflussmustern #2 und #3 in den jeweiligen Modellsprachen sind nicht abzustreiten und verdeutlichen ihre Verwandtschaft miteinander. Es ist an dieser Stelle aber auch anzumerken, dass White noch weitere Modellierungsmöglichkeiten für das zweite und dritte Kontrollflussmuster vorschlägt.[133] Diese werden im Folgenden in dieser Arbeit nicht berücksichtigt, um eine Eindeutigkeit zu erreichen.

Beim vierten und fünften Kontrollflussmuster handelt es sich um die exklusive Auswahl beziehungsweise die einfache Verschmelzung. Dem Namen folgend wird in der BPMN ein exklusives Gateway verwendet (Abbildung 4.7), um die exklusive Auswahl zu modellieren. Bei den UML Aktivitätsdiagrammen wird hingegen ein Entscheidungsknoten verwendet (Abbildung 4.8). An den ausgehenden Kanten stehen Bedingungen beziehungsweise Guards, um die Auswahl zu ermöglichen. Wie schon bei Kontrollflussmuster #2 und #3 ist auch hier die Anzahl der weiterführenden Kanten theoretisch unbegrenzt. Die BPMN kennt auch hier mehrere Möglichkeiten, dieses Kontrollflussmuster zu modellieren. Aufgrund von Einschränkungen, die in Anhang B gemacht wurden, ist dies jedoch die einzige Darstellungsmöglichkeit im Rahmen dieser Arbeit.

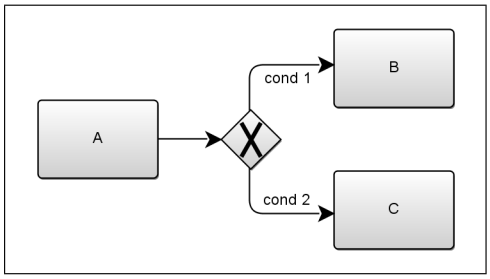


Abbildung 4.7: WCP #4 - BPMN

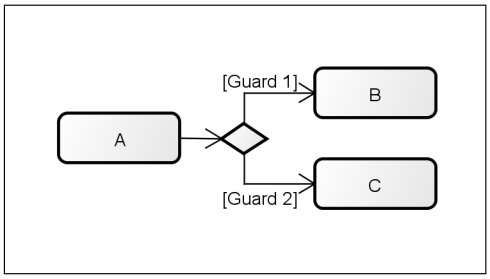


Abbildung 4.8: WCP #4 - UML AD

Um die einfache Verschmelzung zu modellieren, wird in der BPMN wiederum ein exklusives Gateway benutzt (Abbildung 4.9 auf der nachfolgenden Seite). Bei den UML Aktivitätsdiagrammen wird ein Verbindungsknoten verwendet (Abbildung 4.10 auf der folgenden Seite). Auch hier ist die Anzahl der eingehenden Kanten theoretisch unbegrenzt. Die Kontrollflussmuster #4 und #5 besitzen hier wieder eine gewisse Ähnlichkeit zueinander. Aber dieses Mal besteht ebenfalls eine Ähnlichkeit zwischen den Darstellungsformen in den beiden Modellsprachen. Bei einer Transformation ist diese intuitiver zu erfassen als beispielsweise die Transformation der Kontrollflussmuster #2 und #3.

[133] Vgl. White [2004b], S. 3f

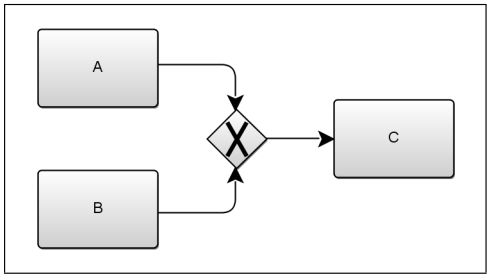


Abbildung 4.9: WCP #5 - BPMN

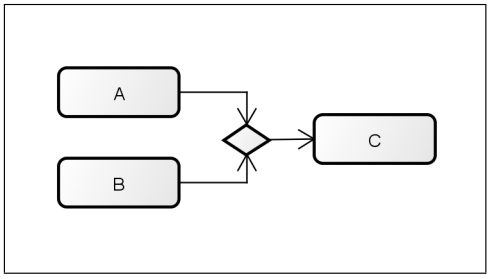


Abbildung 4.10: WCP #5 - UML AD

Bis hierher scheint es einfach zu sein, Regeln abzuleiten, die eine eindeutige Transformation zwischen der BPMN und den UML Aktivitätsdiagrammen beschreiben. Vergleicht man nun jedoch die folgenden vier Kontrollflussmuster miteinander, ergibt sich ein anderes Bild.

Kontrollflussmuster #6 beschreibt die Mehrfachauswahl. Die Darstellung in der BPMN ähnelt der Darstellung bei der exklusiven Auswahl. Allerdings wird hier ein inklusives Gateway verwendet. Alternativ könnte auch ein komplexes Gateway benutzt werden. Auch hier wird auf diese Möglichkeit verzichtet, um eine Eindeutigkeit zu erreichen. Um den gleichen Sachverhalt in der Notation der UML Aktivitätsdiagramme zu modellieren, wird eine Gabelung mit Guards benutzt. Bis auf die Guards sieht die graphische Darstellung der der parallelen Aufspaltung somit sehr ähnlich. Die Anzahl der ausgehenden Kanten ist theoretisch unbegrenzt.

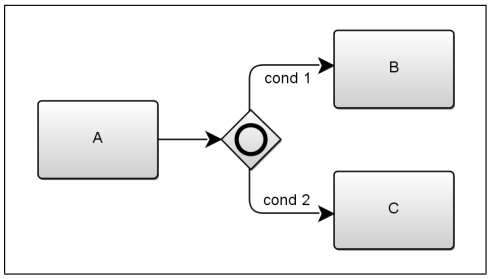


Abbildung 4.11: WCP #6 - BPMN

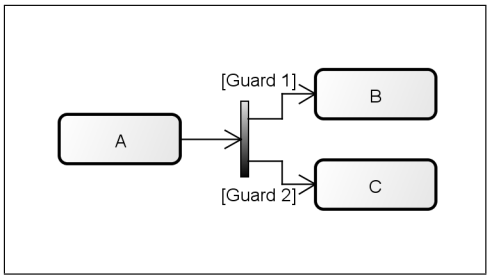


Abbildung 4.12: WCP #6 - UML AD

Das siebte Kontrollflussmuster (synchronisierende Verschmelzung) kann nicht direkt in der Notation der UML Aktivitätsdiagramme und nur teilweise in der Notation der BPMN modelliert werden. Seine Betrachtung wird daher hinten angestellt.

Die mehrfache Verschmelzung (Kontrollflussmuster #8) ist bei beiden Modellsprachen in der Darstellung identisch zu denen der einfachen Verschmelzung (Kontrollflussmuster #5) (siehe Abbildung 4.13 und 4.14). An dieser Stelle wird deutlich, dass eine Transformation anhand der Kontrollflussmuster nicht so einfach wird, wie zunächst gedacht, da die Konstrukte mehrfach auftauchen.

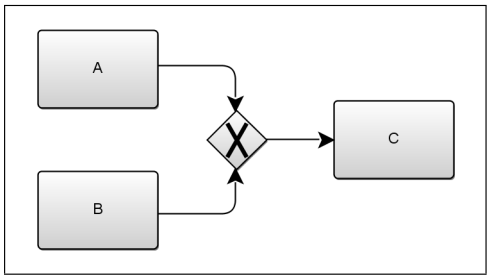


Abbildung 4.13: WCP #8 - BPMN

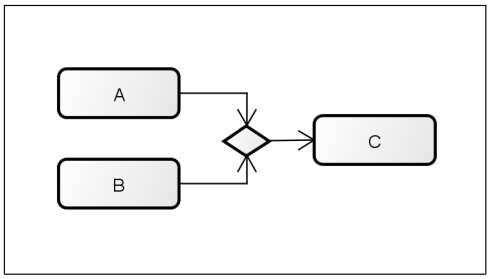


Abbildung 4.14: WCP #8 - UML AD

Kontrollflussmuster #9 (der Diskriminator) kann in der BPMN mittels eines Sub-Prozesses und eines Fehlerflusses dargestellt werden (Abbildung 4.15). Eigentlich stellt das abgebildete Konstrukt die „n von m“-Beziehung dar. Beim Diskriminator ist n=1. Die Aufgaben A bis C stehen an dieser Stelle stellvertretend für bis zu m-Aufgaben. Aufgabe E wird so konfiguriert, dass n-Marken ankommen müssen, ehe sie abgeschlossen ist. Da n=1 ist, kann diese Aufgabe wegfallen und der Fluss direkt in das Fehler-Endereignis führen. Dieses Ereignis beendet den Sub-Prozess dann über einen Fehlerfluss. Auch in der Notation der UML Aktivitätsdiagramm kann der Diskriminator modelliert werden (Abbildung 4.16). Hierzu wird ein Unterbrechungsbereich sowie ein exklusives Gateway verwendet. Erreicht eine Marke das Gateway, so wird der Bereich sofort verlassen.

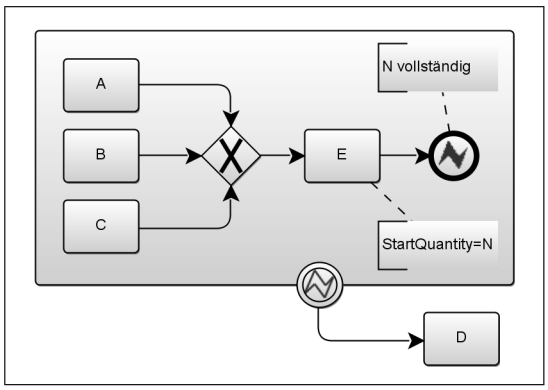


Abbildung 4.15: WCP #9 - BPMN

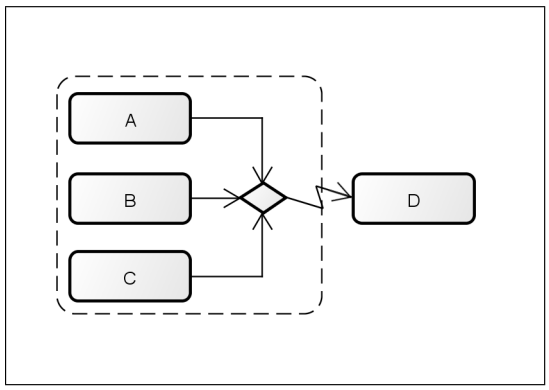


Abbildung 4.16: WCP #9 - UML AD

Die nächste Gruppe der Kontrollflussmuster stellen die strukturellen Muster dar. Diese Gruppe beinhaltet zwei Muster: arbiträre Zyklen und implizierte Terminierung. Die arbiträren Zyklen können in beiden Modellsprachen modelliert werden, da es in beiden Modellsprachen erlaubt ist, Verbindungen zu früherstehenden Elementen zu modellieren, um so eine Schleife zu schaffen. Eine eindeutige

Darstellung gibt es jedoch nicht, da diese Verbindungen sowohl über kleine als auch größere „Strecken“ modelliert werden können, das heißt, es können unter Umständen sehr einfache oder sehr komplexe Schleifenaufrufe modelliert werden.

Die implizite Terminierung hingegen lässt sich darstellen, indem hinter dem Element, nach dem kein weiteres folgen soll, ein Endereignis (Abbildung 4.17) beziehungsweise ein Flussende (Abbildung 4.18) steht.

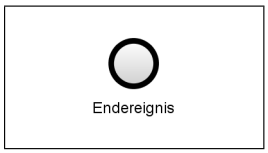


Abbildung 4.17: WCP #11 - BPMN

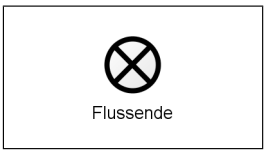


Abbildung 4.18: WCP #11 - UML AD

Die nachfolgenden vier Muster (#12 bis #15) beschäftigen sich mit den Multi-Instanzen. Ihre Betrachtung wird wie bereits erwähnt außen vor gelassen, da sie für die Prozessdarstellung von geringerer Bedeutung erscheinen. Die nächste Gruppe sind daher die statusbasierten Muster. Es handelt sich dabei um drei Muster von denen nur eines von beiden Modellsprachen direkt unterstützt wird: die verzögerte Entscheidung.

Das Kontrollflussmuster #16 (verzögerte Entscheidung) kann auf verschiedene Arten in der BPMN dargestellt werden. Aufgrund von Einschränkungen, die in Anhang B gemacht wurden, bleibt jedoch nur eine davon übrig (Abbildung 4.19). Verwendet wird das ereignis-basierte Gateway mit nachfolgenden Nachrichten-Ereignissen. In der Notation der UML Aktivitätsdiagramme wird dieser Sachverhalt mit Hilfe eines Unterbrechungsbereiches modelliert (Abbildung 4.20 auf der nachfolgenden Seite). In ihm befinden sich empfangende Signale und sobald eines davon ausgelöst wird, wird die entsprechende Aktion ausgeführt.

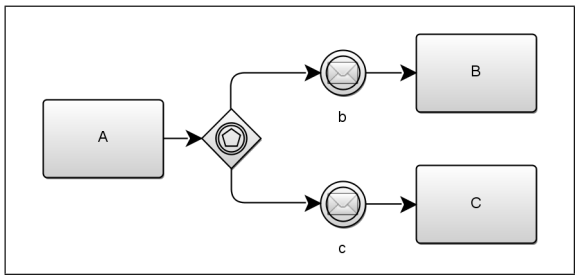


Abbildung 4.19: WCP #16 - BPMN

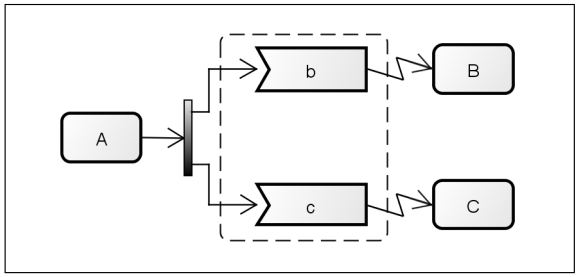


Abbildung 4.20: WCP #16 - UML AD

Die folgenden beiden Kontrollflussmuster (#17 und #18) werden von der BPMN nur teilweise und von den UML Aktivitätsdiagrammen nicht direkt unterstützt. Ihre Betrachtung wird daher hinten angestellt.

Die letzte Gruppe der Kontrollflussmuster besteht aus den Abbruchmustern. Davon gibt es zwei Stück: Abbruch der Aktivität und Abbruch des Falles. Eine laufende Aktivität wird in der BPMN unterbrochen, indem ein Fehlerfluss ausgelöst wird (Abbildung 4.21). Das angeheftete Zwischenereignis wartet auf das entsprechende Ereignis und bricht die Aufgabe ab. In der Notation der UML Aktivitätsdiagramme wird dieser Sachverhalt mittels eines Unterbrechungsbereiches modelliert (Abbildung 4.22). Es wird auf das Signal gewartet und die Aktion gegebenenfalls zum Abbruch gezwungen.

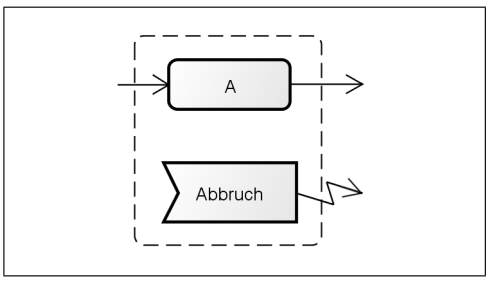


Abbildung 4.21: WCP #19 - BPMN

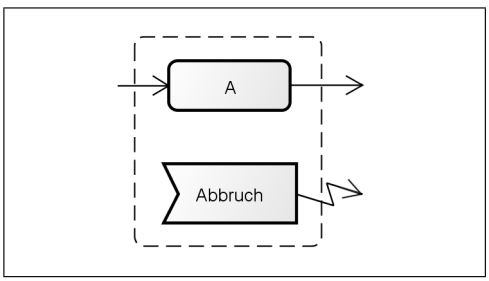


Abbildung 4.22: WCP #19 - UML AD

Das letzte Kontrollflussmuster kann in der BPMN wieder mittels verschiedener Darstellungen modelliert werden. Die eindeutigste Möglichkeit davon ist jedoch die Verwendung eines Endereignisses (Abbildung 4.23). Für die UML Aktivitätsdiagramme wird das Aktivitätsende benutzt (Abbildung 4.24). Auf diese Weise ergibt sich jedoch eine Uneindeutigkeit, denn das Endereignis wurde bereits bei Kontrollflussmuster #11 verwendet.

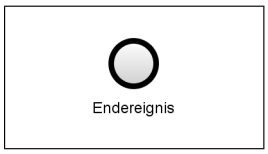


Abbildung 4.23: WCP #20 - BPMN

Abbildung 4.24: WCP #20 - UML AD

Tabelle 4.2 fasst noch einmal zusammen, welche Kontrollflussmuster direkt von beiden Modellsprachen unterstützt werden. Zusätzlich zeigt die Tabelle, welche Kontrollflussmuster nicht direkt unterstützt werden sowie bei welchen es Probleme aufgrund von Übereinstimmungen zueinander gibt.

Kontrollflussmuster	Direkte Unterstützung
#1	ja
#2	ja
#3	ja
#4	ja
#5	ja
#6	ja
#7	nein
#8	ja (aber volle Übereinstimmung mit #5)
#9	ja
#10	ja (aber ohne Abbildung)
#11	ja
#12	nicht betrachtet
#13	nicht betrachtet
#14	nicht betrachtet
#15	nicht betrachtet
#16	ja
#17	nein
#18	nein
#19	ja
#20	ja (aber Teilübereinstimmung mit #11)

Tabelle 4.2: Direkt unterstützte Kontrollflussmuster

4.2.2 Indirekt unterstützte Muster

Russell et al.[134] und Wohed et al.[135] haben Hilfskonstruktionen für die meisten der nicht direkt unterstützten Kontrollflussmuster entwickelt. Diese sind zum Teil einfach, zum Teil recht komplex.

Die synchronisierende Verschmelzung (Kontrollflussmuster #7) kann von der BPMN beispielsweise nur in strukturierter Form mittels des inklusiven Gateways dargestellt werden. Strukturiert heißt, es dürfen zwischen dem verzweigenden und dem zusammenführenden inklusiven Gateway keine Token umgeleitet werden, sodass sie nicht mehr am zusammenführenden inklusiven Gateway vorbeikommen. Die Abbildungen 4.25 und 4.26 zeigen dies unter der Annahme, dass je alle ausgehenden Kanten aktiviert werden.

In Abbildung 4.25 ist der Fluss strukturiert. Das zusammenführende inklusive Gateway wartet auf alle vom verzweigenden inklusiven Gateway aktivierten Kanten und führt somit eine Synchronisation herbei. In Abbildung 4.26 kann es jedoch sein, dass der Fluss beim exklusiven Gateway zu Aufgabe E geleitet wird. Dann kann trotz des Abschlusses von Aufgabe C die Aufgabe D nicht aktiviert werden, weil kein Token über die Kante, die von Aufgabe F kommt, geleitet wird. Das Gateway wartet allerdings auf diesen zweiten Eingang.

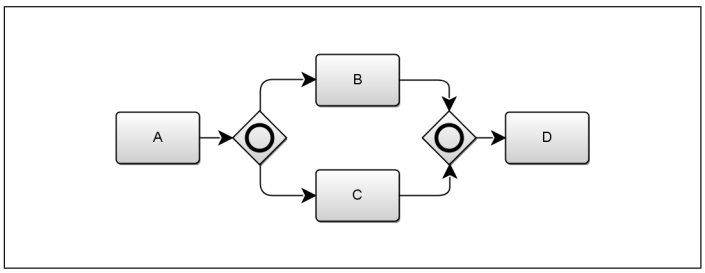


Abbildung 4.25: WCP #7 - BPMN (1)

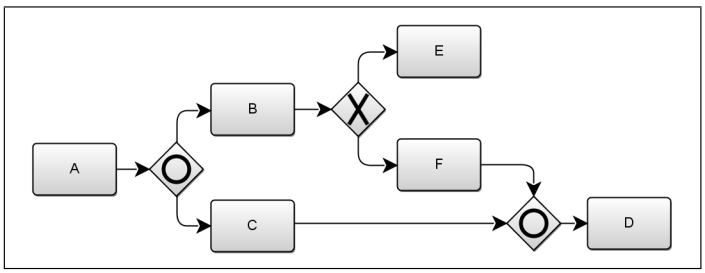


Abbildung 4.26: WCP #7 - BPMN (2)

[134]Vgl. Russell et al. [2006]
[135]Vgl. Wohed et al. [2005a], Wohed et al. [2005b] und Wohed et al. [2006]

In der Notation der UML Aktivitätsdiagramme ist die synchronisierende Verschmelzung gar nicht möglich. Dies liegt zum einen daran, dass es hier ebenfalls unstrukturierte Formen geben kann, zum anderen liegt es daran, dass eine Vereinigung nicht entscheiden kann, wie viele Kanten von der ihr vorangegangenen Gabelung aktiviert wurden. Kontrollflussmuster #7 kann daher erst einmal nicht dargestellt werden.

Anders sieht es bei Kontrollflussmuster #17 (verschachteltes paralleles Routing) aus. Für einfache Aufgaben kann bei der BPMN einfach der Ad-hoc-Sub-Prozess benutzt werden (Abbildung 4.27). Die Ausführungsreihenfolge wird als sequentiell festgelegt, wobei dies ohne graphische Darstellung bleibt. Für komplexere Strukturen wie beispielsweise Aufgaben-Sequenzen gibt es jedoch keine Darstellung.

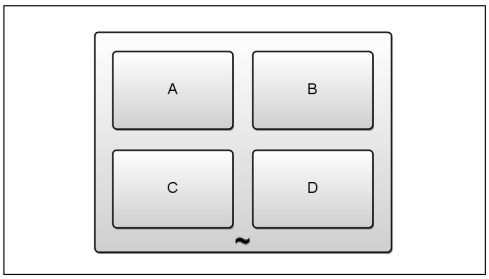


Abbildung 4.27: WCP #17 - BPMN

Die Notation der UML Aktivitätsdiagramme kennt keine direkte Darstellung für das Kontrollflussmuster #17. Abbildung 4.28 zeigt jedoch eine Lösung für vier einfache Aufgaben.

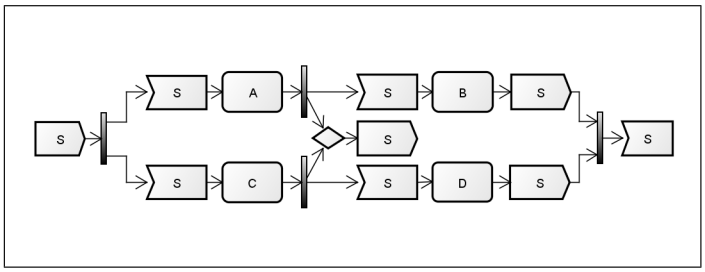


Abbildung 4.28: WCP #17 - UML AD

Über das Senden und Empfangen von immer dem gleichen Signal sollen alle Aufgaben angesprochen werden. Wohed et al.[136] betonen allerdings, dass bei dieser Hilfskonstruktion einige Annahmen getroffen werden. Unter anderem wird offenbar angenommen, dass ein Empfangssignal nur einmal auffangen kann und somit eine Aufgabe auch nur ein einziges Mal gestartet wird. Weiterhin wird angenommen, dass nur ein empfangendes Signal das ausgesendete auffängt, obwohl dies mehrere zur selben Zeit könnten. In dieser Arbeit wird jedoch davon ausgegangen, dass ein

[136]Vgl. Wohed et al. [2005b], S.12f

und dasselbe Signal zwar von verschiedenen Stellen aus ausgesendet werden darf, aber nur an einer einzigen aufgefangen werden kann. Demnach widerspricht die vorgestellte Lösung diesen Regeln. Kontrollflussmuster #17 ist somit ebenfalls nicht darstellbar.

Für den Meilenstein (Kontrollflussmuster #18) gibt es bei beiden Notationen keine direkte Unterstützung. Auch hier bieten Wohed et al.[137] Hilfskonstruktionen an. In Abbildung 4.29 ist die mögliche Lösung in der Notation der BPMN abgebildet.

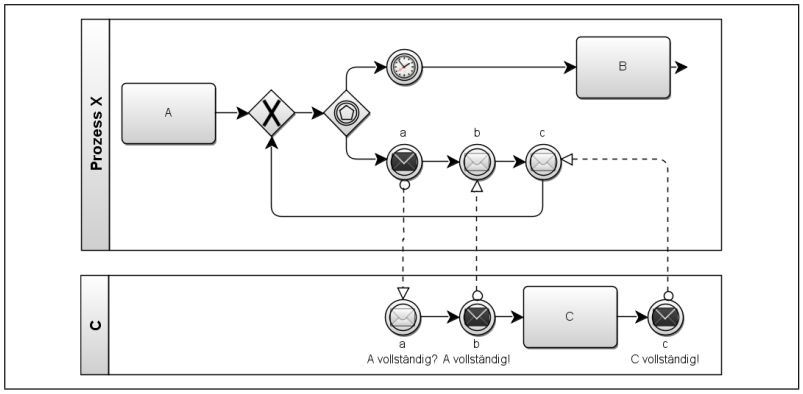


Abbildung 4.29: WCP #18 - BPMN

Die Aufgabe B wird erst ausgeführt, wenn die Aufgabe C ausgeführt worden ist. Es wird hier ein ereignis-basiertes Gateway benutzt. Dieses verzweigt auf ein Zeit-Ereignis sowie ein Nachrichten-Ereignis. Das Zeit-Ereignis stellt die Vollendung von Aufgabe C dar. Da dieses zu Beginn noch nicht eingetreten ist, wird das Nachrichten-Ereignis ausgewählt. Es wird nun abgefragt, ob Aufgabe A vollständig ist. Dies ist anzunehmen, da der Fluss ansonsten nicht an dieser Stelle angekommen sein kann. Das Ereignis b bestätigt dies also und löst Aufgabe C aus. Nach Beendigung von Aufgabe C meldet Ereignis c dies. An dieser Stelle ist anzunehmen, dass das Zeit-Ereignis ab jetzt als erfüllt gilt. Der Fluss kehrt über die Schleife zum ereignis-basierten Gateway zurück und ermöglicht nun die Abarbeitung von Aufgabe B. Es ist nicht klar, wie an dieser Stelle das Zeit-Ereignis ausgewählt wird, da auch das Nachrichten-Ereignis theoretisch auswählbar ist.

Abbildung 4.30 (auf de rnächsten Seite) zeigt eine Umsetzung in der Notation der UML Aktivitätsdiagramme. Die Darstellung ähnelt der in der Notation der BPMN. Aufgabe B wird erst dann gestartet, wenn Aufgabe C mit Sicherheit ausgeführt worden ist. Es ist anzunehmen, dass nach dem Senden des Signals 4 noch Signal 1 gesendet wird sowie der untere Kontrollfluss mittels eines Flussendes beendet wird. Beide Darstellungen ermöglichen es zu modellieren, dass Aufgabe B erst nach der Beendigung von Aufgabe C gestartet wird. Und obwohl Aufgabe C durchaus auch als Sub-Prozess, beziehungsweise Aktivität C, modelliert werden könnte, ist dies einfacher durch sequentielle Anordnung der Notationselemente darstellbar. Aus

[137] Vgl. Wohed et al. [2005a], S. 12, und Wohed et al. [2005b], S. 14

diesem Grund wird darauf verzichtet Kontrollflussmuster #18 als modellierbar anzusehen.

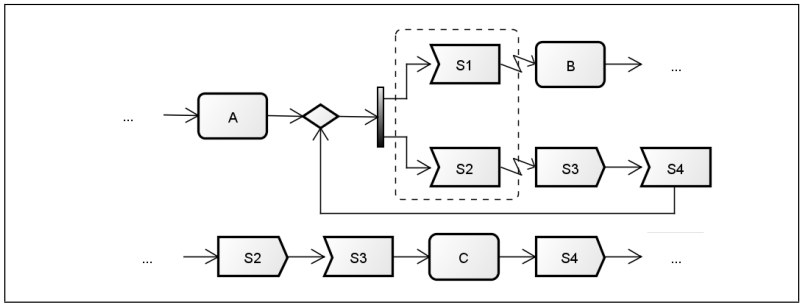


Abbildung 4.30: WCP #18 - UML AD

4.2.3 Resümee

Die vorangegangenen Ausführungen haben gezeigt, dass die Muster, die als direkt unterstützt gelten, eindeutige Darstellungen in beiden Notationen aufweisen und aus diesem Grund direkt ineinander überführbar sind. Einzige Ausnahme bilden die Kontrollflussmuster #5 und #8, welche identisch sind, sowie die Kontrollflussmuster #11 und #20, bei denen auf Seiten der BPMN in beiden Fällen ein Endereignis verwendet wird. Wie dieses Detail behandelt wird, geht aus den nachfolgenden Abschnitten hervor. Andere Kontrollflussmuster (#7, #17 und #18) lassen sich nur unter bestimmten Annahmen oder Regelungen modellieren, die in dieser Arbeit nicht übernommen werden beziehungsweise nicht gelten, oder gar nicht modellieren. Auch darauf wird in den nachfolgenden Abschnitten genauer eingegangen.

Kontrollflussmuster #10 ist bisher ohne Abbildung geblieben. Es gilt als direkt unterstützt, jedoch ohne eindeutige Darstellung. Die Abbildungen 4.31 und 4.32 (auf der nachfolgenden Seite) zeigen je ein Beispiel, wie dieses Kontrollflussmuster durch die Verwendung anderer Kontrollflussmuster modelliert werden kann. Es werden die Kontrollflussmuster #4 und #5 verwendet, um eine Schleife zu modellieren (Abbildung 4.31). Kontrollflussmuster #4 sorgt dabei für die Auswahl des Pfades „Salz fehlt“ oder des Pfades „Sonst“, und Kontrollflussmuster #5 synchronisiert den normalen Fluss mit dem Fluss von „Salz hinzufügen“. Diese Schleife entspricht somit einem „If-Then-Else“-Konstrukt.

Auch in Abbildung 4.32 werden die Kontrollflussmuster #4 und #5 verwendet. Durch die Einbettung von Entscheidungs- und Verbindungsknoten gibt es unterschiedliche Möglichkeiten, diese Schleife auszuführen. So kann beispielsweise die Reihenfolge „A“, „B“, „C“, „D“ und wieder „A“ ausgeführt werden oder aber auch die Reihenfolge „A“, „B“, „D“ und wieder „A“. Es ist genauso gut möglich, die Ausführung von „C“ diverse Male zu wiederholen, bevor „D“ ausgeführt wird.

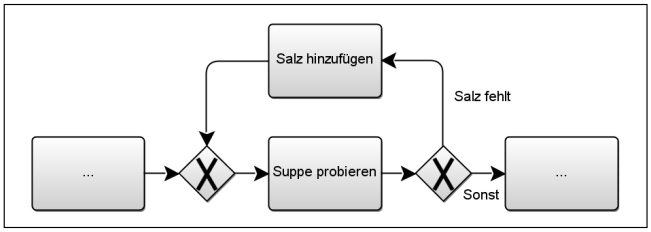


Abbildung 4.31: WCP #10 - BPMN

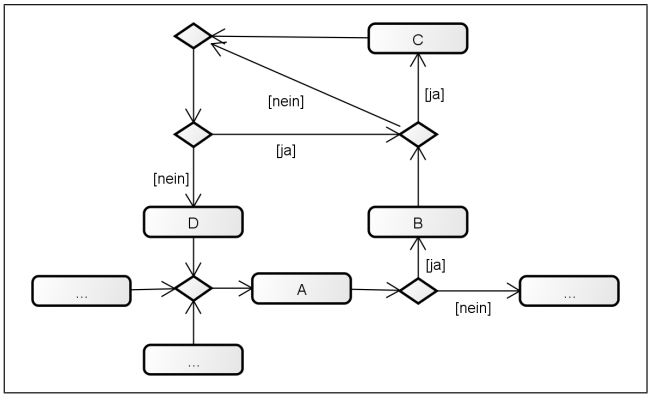


Abbildung 4.32: WCP #10 - UML AD

Weiterhin muss bei der Verwendung der Kontrollflussmuster beachtet werden, dass theoretisch in jedem der dargestellten Muster statt einer atomaren Einheit (Aufgabe beziehungsweise Aktion) auch eine nicht atomare Einheit (Sub-Prozess beziehungsweise Aktivität) verwendet werden könnte. Diese kann dann wiederum Kontrollflussmuster beinhalten. Auch diese Möglichkeit wird in den nachfolgenden Abschnitten genauer betrachtet.

Ein weiterer Punkt, der beachtet werden muss, ist, dass keines der betrachteten Kontrollflussmuster den Prozessstart behandelt. Da ein Startpunkt für einen Prozess jedoch wichtig ist und Endpunkte durch die Kontrollflussmuster #11 und #20 definiert werden, wird ein solcher Startpunkt für diese Arbeit definiert.

Weiterhin wurde deutlich, dass diverse Elemente aus der BPMN und den UML Aktivitätsdiagrammen erst einmal keine Verwendung in den Kontrollflussmustern finden. Dies würde darauf hindeuten, dass diese Elemente für die Prozessdarstellung unnötig sind. Um die Menge der verwendbaren Notationselemente allerdings nicht zu sehr zu beschränken, wird dieser Aspekt in Abschnitt 4.5 noch einmal aufgegriffen.

4.3 Modelltransformation mittels Baustein-Prinzip

Die Betrachtung der Kontrollflussmuster der Workflow Patterns hat gezeigt, dass diese weitgehend eindeutig sind und die Darstellung in beiden Modellsprachen möglich ist. Um nun eine Transformation zwischen der BPMN und den UML Aktivitätsdiagrammen zu ermöglichen, wird daher an dieser Stelle ein Baustein-Prinzip erarbeitet. Bei diesem gilt, dass die definierten Bausteine aneinander gereiht werden. Da die Bausteine in beiden Modellsprachen existieren, ist auch eine eindeutige Transformation zwischen diesen Modellsprachen möglich (Abbildung 4.33).

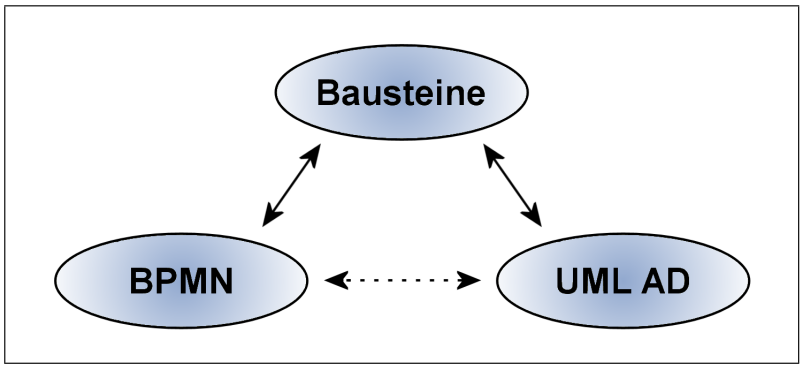


Abbildung 4.33: Transformationsrichtung

Im Folgenden werden 20 Bausteine definiert, welche alle betrachteten Kontrollflussmuster abdecken. Die Tabelle 4.3 und die Tabelle 4.4 (auf der nächsten Seite) zeigen die Beziehungen zwischen den zu definierenden Bausteinen und den betrachteten Kontrollflussmustern im Überblick. Das Kontrollflussmuster #10 ist nicht enthalten, da Schleifenkonstrukte durch andere Kontrollflussmuster modelliert werden können, wie im vorangegangenen Abschnitt gezeigt wurde. Die Kontrollflussmuster #12 bis #15 sind ebenfalls nicht enthalten, da sie sich mit den Multi-Instanzen befassen.

Baustein	Kontrollflussmuster	
#1a	—	Start
#1b	#11	Flussende
#1c	#20	Prozessende
#2a	—	atomare Einheit
#2b	—	nicht-atomare Einheit
#3	#1	Sequenz
#4	#2	Parallele Aufspaltung
#5 (a / b)	#3	Synchronisation
#6	#4	Exklusive Auswahl
#7 (a / b)	#5 / #8	Verschmelzung

Tabelle 4.3: Bausteine und Kontrollflussmuster

Zunächst wird ein Start-Baustein benötigt: Baustein Nummer 1a (kurz #1a; analog dazu alle weiteren Bausteine). Für die BPMN besteht dieser aus einem Startereignis, für die Aktivitätsdiagramme der UML aus einem Startknoten. Das heißt, obwohl es sich nicht um eines der Workflow Pattern handelt, ist es in

Baustein	Kontrollflussmuster	
#8	#6	Mehrfachauswahl
#9 (a / b)	#7	Synchronisierende Verschmelzung
#10	#9	Diskriminator
#11	#16	Verzögerte Entscheidung
#12	#19	Abbruch einer Aktivität
#13	#17	Paralleles, verschachteltes Routing
#14	#18	Meilenstein

Tabelle 4.4: Bausteine und Kontrollflussmuster (Fortsetzung)

beiden Modellsprachen darstellbar. Der erste Baustein wird demnach den Start eines Prozesses oder einer Untereinheit[138] von diesem darstellen. Es wird definiert, dass dieser Baustein keine eingehenden Kanten und genau eine ausgehende Kante besitzt. Sollte der Prozess direkt nach seinem Beginn aufgegliedert werden müssen, so wird einer der aufspaltenden Bausteine dahinter gesetzt. Obligatorisch für diesen Baustein ist eine ID-Nummer[139], optional sind eine Bezeichnung beziehungsweise ein Name sowie Anmerkungen.

Sowohl die BPMN als auch die UML können XML[140]-basiert beschrieben werden. Aus diesem Grund ist es sinnvoll, die Bausteine ebenfalls XML-basiert zu beschreiben. Dies geschieht hier anhand einer XML Schema Definition (XSD). Diese zeigt, wie die Bausteine aussehen und welche Elemente für ihre Definition benötigt werden. Das Listing 4.1 zeigt, wie Baustein #1a in der XSD aussieht.

```
1  <xs:element name="brick_01a" type="startBrickType"/> <!-- Start --->
2    <xs:complexType>
3      <xs:sequence>
4        <xs:element ref="outgoing"/>
5        <xs:element ref="labeling" minOccurs="0"/>
6        <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
7      </xs:sequence>
8      <xs:attribute name="id" type="xs:ID" use="required"/>
9    </xs:complexType>
10 </xs:element>
```

Listing 4.1: Baustein #1a in XSD

Als Gegenstück zu diesem Baustein wird an dieser Stelle direkt der End-Baustein definiert: Er besitzt beliebig viele eingehende Kanten und keine ausgehenden. Auch er verfügt über eine obligatorische ID-Nummer sowie optional über eine Bezeichnung und Anmerkungen.

Bei der Betrachtung der Workflow Patterns wurde zwischen der Beendigung eines Kontrollflusses und des gesamten Prozesses, oder einer Untereinheit desselben, gesprochen. Daher ergeben sich zwei Varianten dieses Bausteins, die sich lediglich in

[138] Als Untereinheit wird hier ein Sub-Prozess bezeichnet. Da diese Bezeichnung jedoch auch in der BPMN verwendet wird, wurde hier diese Alternative verwendet, um eine Abgrenzung zu schaffen.

[139] Eine Nummer zur eindeutigen Identifikation des Bausteins im Diagramm.

[140] Extensible Markup Language (XML)

ihrer Bezeichnung unterscheiden (Baustein #1b und #1c). Die Listings 4.2 und 4.3 zeigen, wie die entsprechenden Bausteine in der XSD aussehen können.

```
<xs:element name="brick_01b" type="brickType"/> <!-- Flussende -->
  <xs:complexType>                              <!-- WCP #11 -->
    <xs:sequence>
      <xs:element ref="incoming" maxOccurs="unbounded"/>
      <xs:element ref="labeling" minOccurs="0"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.2: Baustein #1b in XSD

```
<xs:element name="brick_01c" type="endBrickType"/> <!-- Prozessende -->
  <xs:complexType>                                 <!-- WCP #20 -->
    <xs:sequence>
      <xs:element ref="incoming" maxOccurs="unbounded"/>
      <xs:element ref="labeling" minOccurs="0"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.3: Baustein #1c in XSD

Im Weiteren werden Bausteine benötigt, die Teile eines Prozesses darstellen können, das heißt entweder eine atomare Einheit (#2a) oder eine Untereinheit (#2b). Diese stellen die Aufgaben (BPMN) beziehungsweise Aktionen (UML Aktivitätsdiagramm) sowie die Sub-Prozesse (BPMN) beziehungsweise Aktivitäten (UML Aktivitätsdiagramm) dar. Während erstere atomar sind, können letztere unter anderem wieder atomare Einheiten enthalten. Sie können sogar Untereinheiten und ganze Abläufe enthalten, weshalb es möglich sein muss, jeden definierten Baustein auch innerhalb einer Untereinheit zu verwenden. In Abbildung 4.34 ist der Baustein #2b graphisch dargestellt, um das Verständnis für diesen zu verbessern.

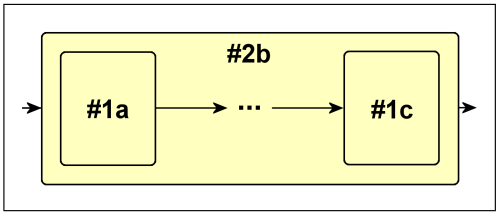


Abbildung 4.34: Baustein #2b (graphisch)

Jede Untereinheit muss mit dem Start-Baustein begonnen und mit dem Baustein für das Prozessende beendet werden. Dazwischen können alle anderen Bausteine in gültiger Reihenfolge verwendet werden. Diese beiden Bausteine (#2a und #2b) haben je eine eingehende Kante und eine ausgehende Kante. Weiterhin verfügen sie

über eine ID-Nummer sowie einen Namen. Optional besitzen auch sie Anmerkungen. Die Listings 4.4 und 4.5 zeigen, wie die Bausteine #2b und #2b in der XSD aussehen.

```
<xs:element name="brick_02a" type="brickType"/> <!-- atomare Einheit -->
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="incoming"/>
      <xs:element ref="outgoing"/>
      <xs:element ref="labeling"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute  name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.4: Baustein #2a in XSD

```
<xs:element name="brick_02b" type="brickType"/> <!-- nicht-atomare Einheit -->
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="incoming"/>
      <xs:element ref="outgoing"/>
      <xs:element ref="labeling"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="brick_01a"/>
      <xs:choice minOccurs="1" maxOccurs="unbounded">
        <xs:element ref="brick01b"/>
        <xs:element ref="brick02a"/>
        <xs:element ref="brick02b"/>
        <!-- ... -->
        <xs:element ref="brick14"/>
      </xs:choice>
      <xs:annotation>
        Zwischen Baustein #1a und #1c können beliebige Bausteine der Nummern
        2-14 (inkl. ihren Varianten) sowie 1b in gültigiger Reihenfolge gesetzt
        werden.
      </xs:annotation>
      <xs:element ref="brick_01c"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.5: Baustein #2b in XSD

Baustein #3 realisiert die Verbindung zwischen den Bausteinen (Sequenz). Das heißt, der Baustein wird immer dann verwendet, wenn zwei andere Bausteine miteinander verbunden werden müssen. Aus diesem Grund benötigt er eine Source-ID sowie eine Target-ID. Die erstgenannte ID-Nummer bezeichnet den Baustein, der als Quelle dient, die letztgenannte ID-Nummer den Ziel-Baustein. Dadurch wird auch die Verbindungsrichtung im Diagramm angegeben. Von Nutzen sind an dieser Stelle die definierten eingehenden und ausgehenden Kanten der anderen Bausteine. Jede eingehende oder ausgehende Kante eines Bausteins muss in einem Baustein #3 aufgegriffen werden. Das heißt konkret, wenn ein Baustein beispielsweise über drei ausgehende Kanten verfügt, so müssen auch drei Bausteine #3 mit dessen ID-Nummer als Source-ID existieren. Weiterhin verfügt dieser Baustein über eine eigene ID-Nummer sowie eine Richtung. Diese ist, wenn nicht anders angegeben, auf „unidirektional“ gesetzt. Optional für den Baustein sind ein Name, Anmerkungen

und Bedingungen. Listing 4.6 zeigt, wie der Baustein in der XSD aussieht.

```
<xs:element name="brick_03" type="flowType"/> <!-- Sequence -->
  <xs:complexType>                            <!-- WCP #1 -->
    <xs:sequence>
      <xs:element ref="source_id"/>
      <xs:element ref="target_id"/>
      <xs:element ref="labeling" minOccurs="0"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="direction"/>
      <xs:element ref="condition" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.6: Baustein #3 in XSD

Der vierte Baustein setzt das Kontrollflussmuster der parallelen Aufteilung um. Dieser Baustein hat genau eine eingehende Kante und beliebig viele, aber mindestens zwei, ausgehende Kanten. Wie jeder andere bisher definierte Baustein hat auch dieser eine obligatorische ID-Nummer, sowie zusätzlich eine optionale Bezeichnung und optionale Anmerkungen. Listing 4.7 zeigt den Baustein in der XSD.

```
<xs:element name="brick_04" type="brickType"> <!-- Parallel Split -->
  <xs:complexType>                            <!-- WCP #2 -->
    <xs:sequence>
      <xs:element ref="incoming"/>
      <xs:element ref="outgoing" minOccurs="2" maxOccurs="unbounded"/>
      <xs:element ref="labeling" minOccurs="0"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.7: Baustein #4 in XSD

Der nächste Baustein setzt das Gegenstück zu Baustein #4, die Synchronisation, um. Aus diesem Grund hat er beliebig viele, aber mindestens zwei, eingehende Kanten und genau eine ausgehende. Es muss auf alle eingehenden Kanten gewartet werden. Weiterhin verfügt er über eine ID-Nummer. Optional sind auch hier ein Name sowie Anmerkungen. Listing 4.8 (auf der nachfolgenden Seite) zeigt den Baustein in der XSD.

Zusätzlich zeigt Listing 4.9 (auf der nächsten Seite) den Baustein in korrespondierender Form. Dieser Baustein ist nützlich, wenn beispielsweise die von einem Baustein #4 aufgespalteten Pfade explizit wieder zusammengeführt werden sollen. In dieser zweiten Variante hat der Baustein genau eine eingehende sowie eine ausgehende Kante. Diese werden intern auf die Bausteine #4 und #5a geleitet.

```
<xs:element name="brick_05a" type="brickType"> <!-- Synchronisation I -->
  <xs:complexType>                             <!-- WCP #3 -->
    <xs:sequence>
      <xs:element ref="incoming" minOccurs="2" maxOccurs="unbounded"/>
      <xs:element ref="outgoing"/>
      <xs:element ref="labeling" minOccurs="0"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.8: Baustein #5a in XSD

Für das bessere Verständnis ist dies in Abbildung 4.35 graphisch dargestellt. Vom verzweigenden Element gehen beliebig viele, aber mindestens zwei Kanten aus. Beim zusammenführenden Element laufen beliebig viele, aber mindestens zwei Kanten zusammen. Dabei muss die Anzahl der genannten Kanten übereinstimmen. Dazwischen können andere Bausteine (mit Ausnahme der Bausteine #1a, #1b sowie #1c) in gültiger Reihenfolge und strukturiert verwendet werden.

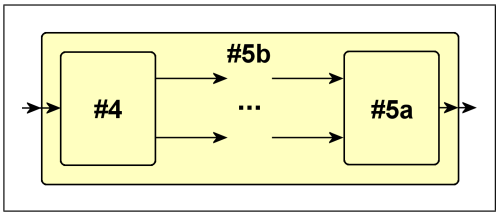


Abbildung 4.35: Baustein #5b (graphisch)

```
<xs:element name="brick_05b" type="brickType"> <!-- Synchronisation II -->
  <xs:complexType>                             <!-- WCP #3 -->
    <xs:sequence>
      <xs:element ref="incoming"/>
      <xs:element ref="outgoing"/>
      <xs:element ref="brick_04"/>
      <xs:choice minOccurs="1" maxOccurs="unbounded">
        <xs:element ref="brick02a"/>
        <xs:element ref="brick02b"/>
        <!-- ... -->
        <xs:element ref="brick14"/>
      </xs:choice>
      <xs:annotation>
        Zwischen Baustein #4 und #5a können beliebige Bausteine der Nummern
        2-14 (inkl. ihren Varianten) in gültiger Reihenfolge gesetzt werden.
      </xs:annotation>
      <xs:element ref="brick_05a"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.9: Baustein #5b in XSD

Baustein #6 gleicht formal Baustein #4 bis auf den Namen. Auch hier wird ein aufspaltendes Element modelliert, nur das es sich dieses Mal um die exklusive Auswahl handelt. Der Baustein verfügt über eine eingehende Kante, sowie beliebig

viele, aber mindestens zwei, einander ausschließende Kanten. Die Auschlusskriterien müssen später anhand der Referenzen auf die Verbindungsbausteine erkannt werden. Neben der obligatorischen ID-Nummer gibt es auch hier eine optionale Bezeichnung und Anmerkungen. Listing 4.10 zeigt die Darstellung in der XSD.

```
<xs:element name="brick_06" type="brickType"> <!-- Exclusive Choice -->
  <xs:complexType>                            <!-- WCP #4 -->
    <xs:sequence>
      <xs:element ref="incoming"/>
      <xs:element ref="outgoing" minOccurs="2" maxOccurs="unbounded"/>
      <xs:element ref="labeling" minOccurs="0"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.10: Baustein #6 in XSD

Das Gegenstück zu dem hier modellierten Kontrollflussmuster ist die Zusammenführung von mehreren Flüssen. Aus diesem Grund hat Baustein #7a beliebig viele, aber mindestens zwei eingehende Kanten und genau eine ausgehende. Weiterhin verfügt auch dieser Baustein über eine ID-Nummer. Name und Anmerkungen sind optional. Somit sieht Baustein #7a bis auf den Namen genauso aus wie Baustein #5a. Listing 4.11 zeigt die Darstellung des Bausteins in der XSD.

```
<xs:element name="brick_07a" type="brickType"> <!-- Merge I -->
  <xs:complexType>                             <!-- WCPs #5 und #8 -->
    <xs:sequence>
      <xs:element ref="incoming" minOccurs="2" maxOccurs="unbounded"/>
      <xs:element ref="outgoing"/>
      <xs:element ref="labeling" minOccurs="0"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.11: Baustein #7a in XSD

Auch für dieses zusammenführende Element wird eine korrespondierende Version definiert, die in Abbildung 4.36 und Listing 4.12 (beide auf der nachfolgenden Seite) dargestellt ist. Es gilt auch hier: Die eingehende Kante von Baustein #7b wird auf den Baustein #6 umgesetzt und die ausgehende Kante wird als ausgehende Kante von Baustein #7a gesetzt. Vom verzweigenden Element gehen beliebig viele, aber mindestens zwei, Kanten aus. Beim zusammenführenden Element laufen beliebig viele, aber mindestens zwei, Kanten zusammen. Dabei muss die Anzahl der genannten Kanten übereinstimmen. Dazwischen können andere Bausteine (mit Ausnahme der Bausteine #1a, #1b und #1c) in gültiger Reihenfolge und strukturiert verwendet werden.

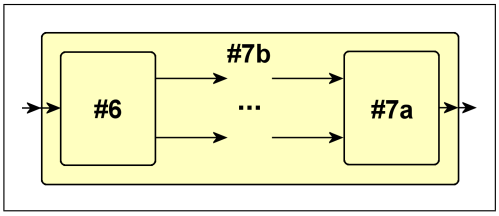


Abbildung 4.36: Baustein #7b (graphisch)

```
<xs:element name="brick_07b" type="brickType"> <!-- Merge II -->
  <xs:complexType>                             <!-- WCPs #5 und #8 -->
    <xs:sequence>
      <xs:element ref="incoming"/>
      <xs:element ref="outgoing"/>
      <xs:element ref="brick_06"/>
      <xs:choice minOccurs="1" maxOccurs="unbounded">
        <xs:element ref="brick02a"/>
        <xs:element ref="brick02b"/>
        <!-- ... -->
        <xs:element ref="brick14"/>
      </xs:choice>
      <xs:annotation>
        Zwischen Baustein #6 und #7a können beliebige Bausteine der Nummern
        2-14 (inkl. ihren Varianten) in gültiger Reihenfolge gesetzt werden.
      </xs:annotation>
      <xs:element ref="brick_07a"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.12: Baustein #7b in XSD

Der Baustein #8 modelliert das beschriebene Verhalten des sechsten Kontrollflussmusters und setzt somit die Mehrfachauswahl um. Er besitzt eine eingehende Kante sowie beliebig viele, aber mindestens zwei, ausgehende Kanten. Diese Kanten müssen sich dieses Mal jedoch nicht gegenseitig ausschließen. Die Auswahlkriterien werden später anhand der Referenzen auf die Verbindungsbausteine erkannt. Auch dieser Baustein verfügt über eine ID-Nummer. Zusätzlich können ein Name oder Anmerkungen hinzugefügt werden. Listing 4.13 zeigt die Darstellung in der XSD.

```
<xs:element name="brick_08" type="brickType"> <!-- Multiple Choice -->
  <xs:complexType>                            <!-- WCP #6 -->
    <xs:sequence>
      <xs:element ref="incoming"/>
      <xs:element ref="outgoing" minOccurs="2" maxOccurs="unbounded"/>
      <xs:element ref="labeling" minOccurs="0"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.13: Baustein #8 in XSD

Das Gegenstück zu Baustein #8 modelliert die synchronisierende Verschmelzung. Wie bei den vorangegangenen, zusammenführenden Bausteinen hat auch dieser beliebig viele, aber mindestens zwei, eingehende Kanten sowie eine ausgehende Kante. Weiterhin eine obligatorische ID-Nummer sowie optionale Elemente. Der Baustein ist in Listing 4.14 dargestellt. Bis auf den Namen gleicht der den Bausteinen #5a und #7a.

```
<xs:element name="brick_09a" type="brickType"> <!-- Synchronising Merge I -->
  <xs:complexType>                             <!-- WCPs #7 -->
    <xs:sequence>
      <xs:element ref="incoming" minOccurs="2" maxOccurs="unbounded"/>
      <xs:element ref="outgoing"/>
      <xs:element ref="labeling" minOccurs="0"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.14: Baustein #9a in XSD

Baustein #9b, also die korrespondierende Variante, hat eine eingehende Kante sowie eine ausgehende, die entsprechend auf die inkludierten Bausteine umgeleitet werden (Abbildung 4.37). Vom verzweigenden Element gehen beliebig viele, aber mindestens zwei Kanten aus. Beim zusammenführenden Element laufen beliebig viele, aber mindestens zwei Kanten zusammen. Dabei muss die Anzahl der genannten Kanten übereinstimmen. Dazwischen können andere Bausteine (mit Ausnahme der Bausteine #1a, #1b und #1c) in gültiger Reihenfolge und strukturiert verwendet werden. Wie alle bisherigen Bausteine hat auch Baustein #9b eine ID-Nummer. Ein Name oder Anmerkungen können optional gesetzt werden. Listing 4.15 (auf der nächsten Seite) zeigt den Baustein in der XSD.

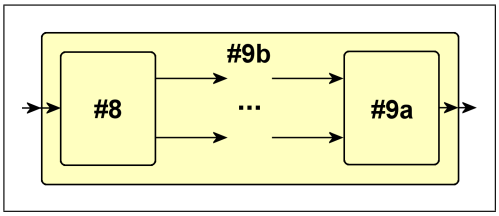


Abbildung 4.37: Baustein #9b (graphisch)

```
<xs:element name="brick_09b" type="brickType"> <!-- Synchronising Merge II -->
  <xs:complexType>                             <!-- WCP #7 -->
    <xs:sequence>
      <xs:element ref="incoming"/>
      <xs:element ref="outgoing"/>
      <xs:element ref="brick_08"/>
      <xs:choice minOccurs="1" maxOccurs="unbounded">
        <xs:element ref="brick02a"/>
        <xs:element ref="brick02b"/>
        <!-- ... -->
        <xs:element ref="brick14"/>
      </xs:choice>
      <xs:annotation>
        Zwischen Baustein #8 und #9a können beliebige Bausteine der Nummern
        2-14 (inkl. ihren Varianten) in gültiger Reihenfolge gesetzt werden.
      </xs:annotation>
      <xs:element ref="brick_09a"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.15: Baustein #9b in XSD

Baustein #10 befasst sich mit dem neunten Kontrollflussmuster (Diskriminator). Der Baustein besitzt eine eingehende Kante und eine ausgehende Kante, die nur mit einem Baustein #2a oder #2b verbunden werden darf, sowie eine ID-Nummer. In diesem Baustein müssen zusätzlich mindestens zwei atomare oder nicht-atomare Einheiten vermerkt werden. An dieser Stelle werden Bausteine #2a und #2b mit Pseudo-ID-Nummern benutzt. Das heißt, diese ID-Nummern finden keine Verwendung in Verbindungsbausteinen (Baustein #3). Auch hier können zusätzlich ein Name oder Anmerkungen vermerkt werden. Das Listing 4.16 zeigt den Baustein in der XSD.

```
<xs:element name="brick_10" type="brickType"> <!-- Discriminator -->
  <xs:complexType>                            <!-- WCP #9 -->
    <xs:sequence>
      <xs:element ref="incoming"/>
      <xs:element ref="outgoing"/>
      <xs:element ref="labeling" minOccurs="0"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
      <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element ref="brick_02a"/> <!-- konkurrierende Einheiten -->
        <xs:element ref="brick_02b"/>
      </xs:choice>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.16: Baustein #10 in XSD

Diese zehn definierten Bausteine, inklusive ihrer Varianten, decken die Kontrollflussmuster der Basismuster, der Muster der erweiterten Verzweigung und Synchronisation sowie teilweise die strukturellen Muster und die Abbruchmuster ab (implizite Terminierung sowie Abbruch eines Anwendungsfalls). Zusätzlich kann das Kontrollflussmuster der arbiträren Zyklen mittels der bereits definierten Bausteine modelliert werden. An dieser Stelle kommt es nur darauf an, wie Baustein #3

benutzt wird, um die übrigen Bausteine miteinander zu verbinden. Allerdings bieten zwei weitere der betrachteten Kontrollflussmuster direkt die Möglichkeit für weitere Bausteine.

Baustein #11 modelliert das Verhalten vom sechzehnten Kontrollflussmuster (verzögerte Auswahl). Der Baustein hat eine eingehende, sowie beliebig viele, aber mindestens zwei ausgehende Kanten, die nur in Baustein #2a und / oder #2b enden dürfen. Außerdem verfügt der Baustein über eine ID-Nummer. In diesem Baustein müssen zusätzlich beliebig viele, aber mindestens zwei, Elemente notiert werden, auf deren Grundlage die Folgebausteine #2a und / oder #2b ausgewählt werden. Dies übernimmt das Element „event“, in dem auch festgehalten wird, auf welchen Folgebaustein je Bezug genommen wird. Das heißt konkret, dass in dem „event“-Element beispielsweise steht, dass er sich auf den Baustein mit der ID-Nummer „_55“ bezieht. In Abbildung 4.38 ist dies für das bessere Verständnis beispielhaft graphisch dargestellt. Die Pfeile entsprechen jeweils einem Baustein #3. Die Anzahl der „event“-Elemente muss identisch sein mit der Anzahl der ausgehenden Kanten des Bausteins. Außerdem darf jedes dieser Elemente nur auf einen Folgebaustein verweisen. Optional können auch für diesen Baustein ein Name oder Anmerkungen hinzugefügt werden. Listing 4.17 zeigt Baustein in der XSD.

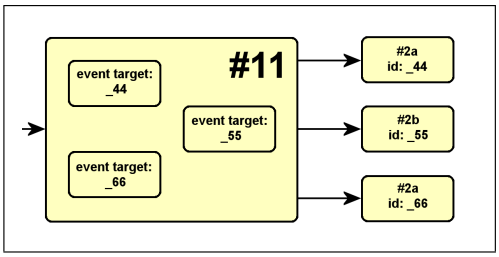


Abbildung 4.38: Baustein #11 (graphisch)

```
<xs:element name="brick_11" type="brickType"> <!-- Deferred Choice -->
  <xs:complexType>                            <!-- WCP #16 -->
    <xs:sequence>
      <xs:element ref="incoming"/>
      <xs:element ref="outgoing" minOccurs="2" maxOccurs="unbounded"/>
      <xs:element ref="labeling" minOccurs="0"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="event" minOccurs="2" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.17: Baustein #11 in XSD

Der nächste Baustein setzt das neunzehnte Kontrollflussmuster um: Abbruch einer Aktivität. Der Baustein verfügt über eine eingehende Kante sowie zwei ausgehende Kanten. Zusätzlich zur obligatorischen ID-Nummer muss eine Abbruchbedingung festgehalten werden. In dieser Abbruchbedingung ist vermerkt, zu welchem Folgebaustein im Falle des Abbruchs weitergegangen werden muss. Die zweite ausgehende Kante führt daher automatisch zu dem anderen Folgebaustein. Optional können Name und Anmerkungen gesetzt werden. Das Listing 4.18 zeigt den Baustein in der XSD.

```
<xs:element name="brick_12" type="brickType"> <!-- Cancel Activity -->
  <xs:complexType>                            <!-- WCP #19 -->
    <xs:sequence>
      <xs:element ref="incoming"/>
      <xs:element ref="outgoing" minOccurs="2" maxOccurs="2"/>
      <xs:element ref="labeling" minOccurs="0"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="exit_condition"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.18: Baustein #12 in XSD

Jetzt sind bereits 14 der 16 betrachteten Kontrollflussmuster durch Bausteine beschrieben. Obwohl sich weder das verschachtelte parallele Routing noch der Meilenstein in der Notation der BPMN beziehungsweise der UML Aktivitätsdiagramme darstellen lassen, können die entsprechenden Bausteine dennoch definiert werden, um später bereits eine Basis für andere Modellsprachen zu haben, bei denen diese Kontrollflussmuster darstellbar sind.

Baustein #13 modelliert daher das siebzehnte Kontrollflussmuster (verschachteltes paralleles Routing). Es existiert eine eingehende sowie eine ausgehende Kante. Der Baustein enthält beliebig viele, aber mindestens zwei, Bausteine #2a und / oder #2b, welche Pseudo-ID-Nummern erhalten (vergleiche Baustein #10). Diese enthaltenen Bausteine stellen die atomaren oder nicht-atomaren Einheiten dar, die in beliebiger Reihenfolge ausgeführt werden sollen. Auch hier ist es möglich, optional einen Namen sowie Anmerkungen zu setzen. Der Baustein ist in Listing 4.19 (auf der nachfolgenden Seite) abgebildet.

Der letzte Baustein modelliert den Meilenstein (Listing 4.20 auf der nächsten Seite). Eine der nachfolgenden Aufgaben wird erst dann ausgeführt, wenn ein bestimmter Status erreicht worden ist, ansonsten wird ein anderer Fluss ausgewählt. Aus diesem Grund gibt es eine eingehende Kante sowie zwei ausgehende Kanten. Eine dieser Kanten führt zu den Elementen, die erst nach Erreichung des bestimmten Status ausgeführt werden dürfen, die andere Kante führt zu einem Punkt weiter vorn im Prozess. Beim nächsten Erreichen des Bausteins ist der benötige Status dann vielleicht erreicht.

```
<xs:element name="brick_13" type="brickType"> <!-- Interleaved Parallel -->
  <xs:complexType>                            <!-- Routing - WCP #17    -->
    <xs:sequence>
      <xs:element ref="incoming"/>
      <xs:element ref="outgoing"/>
      <xs:element ref="labeling" minOccurs="0"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
      <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element ref="brick_02a"/>
        <xs:element ref="brick_02b"/>
      </xs:choice>
      <xs:annotation>
        Die Bausteine #2a und #2b können in beliebiger Reihenfolge ausgeführt
        werden. Jeder Baustein muss einmal ausgeführt werden, Baustein #13
        wird erst verlassen, wenn alle Bausteine 2a und / oder 2b ausgeführt
        worden sind.
      </xs:annotation>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.19: Baustein #13 in XSD

```
<xs:element name="brick_14" type="brickType"> <!-- Milestone -->
  <xs:complexType>                            <!-- WCP #18 -->
    <xs:sequence>
      <xs:element ref="incoming"/>
      <xs:element ref="outgoing"/>
      <xs:element ref="labeling" minOccurs="0"/>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="milestone_condition"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

Listing 4.20: Baustein #14 in XSD

In den definierten Bausteinen wurde auf andere Elemente referenziert. In Listing 4.21 (auf der folgenden Seite) werden diese Elemente vollständig aufgeführt und ihre Typen deklariert. Die meisten dieser Elemente haben als Typ eine Referenz auf eine ID-Nummer, so beispielsweise „incoming“ und „outgoing“, welche je auf die ID-Nummer eines Bausteins #3 verweisen. Dies ist nötig, um den Zusammenhang zwischen den einzelnen Bausteinen nachvollziehen zu können. Dagegen sind die Bezeichnung („labeling“) oder Anmerkungen („annotation“) reiner Text, also vom „string“-Typ. Die Elemente „event“, „exit_condition“ und „milestone_condition“ hingegen sind Elemente, die nochmals Elemente enthalten. Dies ist vor allem darum nötig, weil an dieser Stelle eine Referenz auf Folgebausteine vermerkt werden muss, die ansonsten nicht vermerkt werden könnte.

```
<xs:element name="incoming"              type="xs:IDREF"/>
<xs:element name="outgoing"              type="xs:IDREF"/>
<xs:element name="labeling"              type="xs:string"/>
<xs:element name="annotation"            type="xs:string"/>
<xs:element name="source_id"             type="xs:IDREF"/>
<xs:element name="target_id"             type="xs:IDREF"/>
<xs:element name="condition"             type="xs:string"/>

<xs:element name="event" type="eventType">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="condition"/>
      <xs:element ref="target_id"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="exit_condition" type="conditionType">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="condition"/>
      <xs:element ref="target_id"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="milestone_condition" type="conditionType">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="condition"/>
      <xs:element ref="target_id"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Listing 4.21: Referenzierte Elemente in XSD

In Tabelle 4.5 sind die verwendeten Elemente und Attribute der Bausteine noch einmal zusammengefasst aufgeführt. Neben der Bezeichnung des Elements beziehungsweise des Attributs steht dort auch die deutsche Bezeichnung sowie in welchen Bausteinen das Element / Attribut Verwendung findet, denn nicht jeder Baustein benötigt jedes Element / Attribut.

Element / Attribut	Deutsche Bezeichnung	Verwendung bei ...
id	ID-Nummer	allen Bausteinen
incoming	eingehende Kante	allen, außer #1a und #3
outgoing	ausgehende Kante	allen, außer #1b, #1c und #3
labeling	Bezeichnung / Name	allen, außer #5b, #7b und #9b
annotation	Anmerkung	allen, außer #5b, #7b und #9b
source_id	Quell-ID-Nummer	#3
target_id	Ziel-ID-Nummer	#3, #11, #12 und #14
direction	Richtung	#3
condition	Bedingung	#3, #11, #12 und #14
event	Ereignis	#11
exit_conditiion	Abbruchbedingung	#12
milestone_condition	Meilensteinbedingung	#14

Tabelle 4.5: Verwendete Elemente und Attribute

In einigen der vorangegangenen Absätze war die Rede davon, dass die Bausteine in gültiger Reihenfolge auftreten müssen (bei den Bausteinen #2b, #5b, #7b und #9b sowie allgemein). Diese gültige Reihenfolge wurde bisher jedoch noch nicht definiert. Als gültige Reihenfolge gilt:

1. Ein Prozess oder eine Untereinheit eines Prozesses (Baustein #2b) beginnt immer mit einem Baustein #1a.
2. Einem Baustein #1a folgt nicht in direktem Anschluss (Verbindung durch Baustein #3) ein Baustein #1b oder #1c.
3. Ein Prozess oder eine Untereinheit eines Prozesses (Baustein #2b) endet immer mit einem Baustein #1c.
4. Der Baustein #3 verbindet Bausteine miteinander, allerdings nicht Bausteine der eigenen Art. Das heißt, weder die Quell-ID-Nummer noch die Ziel-ID-Nummer eines Bausteins #3 gehören zu einem Baustein #3.
5. Die Bausteine #5b, #7b und #9b können nur über den inkludierten Baustein (#4, #6 oder #8) betreten beziehungsweise verlassen werden (#5a, #7a oder #9a). Das heißt, dass kein Pfad in einem Baustein #1b oder #1c endet. Es ist ebenfalls nicht möglich, dass ein Pfad plötzlich in den Baustein (#5b, #7b oder #9b) hinein oder aus diesem heraus führt.
6. Dem Baustein #10 folgt immer ein Baustein #2a oder #2b. Das gleiche gilt für Baustein #11.

Es stellt sich nun die Frage, wie die definierten Bausteine konkret in XML-basierter Form in der BPMN beziehungsweise bei den UML Aktivitätsdiagrammen aussehen. Mit dieser Frage beschäftigen sich die nächsten beiden Abschnitte.

4.3.1 Bausteine in der XML-Struktur der BPMN

Um zu sehen, nach welchen XML-Strukturen gesucht werden muss, um die Bausteine zu erkennen, wird eine XML-Struktur benötigt. Das führt dazu, dass ein Modellierungswerkzeug verwendet werden muss, welches einen XML-Export anbietet. Als Werkzeug für die Modellierung von Diagrammen in der BPMN wurde in dieser Arbeit der Yaoqiang BPMN Editor[141] verwendet. Die Wahl ist aus drei Gründen auf dieses Werkzeug gefallen. Zum einen handelt es sich dabei um Open Source Software, welche für nicht kommerzielle Zwecke kostenlos genutzt werden kann, sodass für die Verwendung in dieser Arbeit keine Kosten für die Software an sich anfallen. Zum anderen werden in diesem Editor die Spezifikationen der OMG zur BPMN 2.0 umgesetzt und der Editor regelmäßig aktualisiert, um diesen Anspruch auch weiterhin zu genügen. Das heißt, dass die Diagramme dem von der OMG definierten Standard entsprechen. Außerdem verfügt der Editor neben einer leicht zugänglichen Oberfläche über einen direkten XML-Export der erstellten Diagramme. Gerade dieses letztgenannte Detail ist im Weiteren wichtig, denn anhand dieses XML-Exports kann gezeigt werden, nach welchen XML-Strukturen gesucht werden muss, um die definierten Bausteine ableiten zu können. Deshalb wurde der Yaoqiang BPMN Editor als Modellierungswerkzeug ausgewählt.

Um die für die Ableitung wichtigen XML-Strukturen hervoruheben, wird eine vereinfachte Version der XML-Struktur verwendet. In der vollständigen XML-Struktur sind die Elemente „process“ und „bpmndi:BPMNDiagram“ die Kinder des Hauptelements „definitions“. Das für die Baustein-Identifizierung wichtige Element ist dabei „process“ mit seinen Kind-Elementen, da hier die Notationselemente der BPMN mit ihren Beziehungen zueinander aufgeführt werden. Das andere Element enthält Informationen zur Positionierung und zum Aussehen der Elemente. In Listing 4.22 ist dieses Rahmenkonstrukt kurz dargestellt.

```
1 <definitions>
2   <process id="PROCESS_1">
3     <!-- Elemente und Beziehungen -->
4   </process>
5   <bpmndi:BPMNDiagram>
6     <!-- Positionierung und Darstellung -->
7   </bpmndi:BPMNDiagram>
8 </definitions>
```

Listing 4.22: Rahmenkonstrukt (vereinfachte XML-Struktur - BPMN)

Es werden jedoch auch noch weitere Teile der XML-Struktur weggelassen, da sie ebenfalls nicht der Identifizierung der Bausteine dienlich sind. Dies betrifft bei der vom Yaoqiang BPMN Editor erzeugten XML-Struktur zum Beispiel die folgenden Attribute beim Startereignis: „isInterrupting“ und „parallelMultiple“. Auf das erstgenannte Attribut wird verzichtet, weil es im Folgenden immer den Wert „true“ hat, auf das letztgenannte wird verzichtet, weil es im Folgenden immer den Wert „false“ hat, da keine Multi-Instanzen betrachtet werden. Aus den selben

[141] Version 2.2.2; aktuell: Version 3.0.1 vom 25. November 2013; http://bpmn.sourceforge.net/

Gründen werden andere Attribute weggelassen.

Exemplarisch wird im Folgenden an drei XML-Strukturen die Ableitung des entsprechenden Bausteins gezeigt. In Anhang D sind alle Ableitungen im Detail beschrieben. Die einzelnen Bausteine sind anhand von Indikatorelementen zu identifizieren. Die Indikatorelemente sind in Tabelle 4.6 in einer Übersicht zusammengefasst.

Baustein	Indikatorelement in der BPMN
#1a	startEvent
#1b	textAnnotation (an endEvent)
#1c	endEvent
#2a	task, sendTask, receiveTask, serviceTask, userTask, manualTask, scriptTask, businessRuleTask
#2b	subProcess
#3	sequenceFlow
#4	parallelGateway (gatewayDirection=diverging)
#5a	parallelGateway (gatewayDirection=converging)
#5b	Baustein #4 (alle ausgehenden Kanten lassen sich über sequenceFlow zu Baustein #5a verfolgen)
#6	exclusiveGateway (gatewayDirection=diverging)
#7a	exclusiveGateway (gatewayDirection=converging)
#7b	Baustein #6 (alle ausgehenden Kanten lassen sich über sequenceFlow zu Baustein #7a verfolgen)
#8	inclusiveGateway (mehr als eine ausgehende Kante)
#9a	inclusiveGateway (mehr als eine eingehende Kante)
#9b	Baustein #8 (alle ausgehenden Kanten lassen sich über sequenceFlow zu Baustein #9a verfolgen)
#10	boundaryEvent (mit errorEventDefinition und an subProcess gebunden)
#11	eventBasedGateway (mit 2x intermediateCatchEvent)
#12	boundaryEvent (mit errorEventDefinition und an task gebunden)

Tabelle 4.6: Indikatorelemente der BPMN für die Transformation

Listing 4.23 zeigt die XML-Struktur des Startereignisses. Dieses ist eindeutig identifizierbar, da es als „startEvent“ deklariert wird. Es verwendet unter anderem die Attribute „id“ sowie „name“ und enthält die ID-Nummer der ausgehenden Kante als Element.

```
1 <startEvent id="_2" name="Start Event">
2   <outgoing>_4</outgoing>
3 </startEvent>
```

Listing 4.23: Baustein #1a (vereinfachte XML-Struktur - BPMN)

Die BPMN definiert verschiedene Arten von Startereignissen, darunter beispielweise Zeit- und Nachrichten-Startereignisse. Obwohl für diese Arbeit festgelegt wurde, auf diese Spezialisierungen nach Möglichkeit zu verzichten, können sie als Anmerkungen

abgespeichert werden. Um welche Spezialisierung es sich handelt wird durch eine Ereignisdefinition beschrieben. In Tabelle 4.7 sind die Ereignisdefinitionen und die dazugehörigen speziellen Anmerkungen aufgeführt. Dabei wird die Anmerkung mit den Buchstaben „SPEC“, für „special“, eingeleitet. Listing 4.24 zeigt ein Start-Ereignis mit Ereignisdefinition.

Ereignisdefinition	Anmerkung
messageEventDefinition	SPEC:MessageEventDefinition
timerEventDefinition	SPEC:TimerEventDefinition
errorEventDefinition	SPEC:ErrorEventDefinition
escalationEventDefinition	SPEC:EscalationEventDefinition
cancelEventDefinition	SPEC:CancelEventDefinition
compensateEventDefinition	SPEC:CompensateEventDefinition
conditionalEventDefinition	SPEC:ConditionalEventDefintion
linkEventDefinition	SPEC:LinkEventDefinition
signalEventDefinition	SPEC:SignalEventDefinition
terminateEventDefinition	SPEC:TerminateEventDefinition

Tabelle 4.7: Spezielle Anmerkungen bei Ereignissen

```
<startEvent id="_25" name="Start Event">
  <outgoing>_27</outgoing>
  <timerEventDefinition id="_25_ED_1"/>
</startEvent>
```

Listing 4.24: Baustein #1a (Spezialisierung) (vereinfachte XML-Struktur - BPMN)

Es stellt sich nun die Frage, wie der Baustein #1a konkret aus der XML-Struktur heraus erstellt wird. Zunächst einmal wird hier nach dem Element „startEvent“ gesucht. Die ID-Nummer des entsprechendes Elements wird als ID-Nummer des Bausteins gesetzt. Danach werden die Elemente des Bausteins befüllt: In diesem Beispiel wird „outgoing“ direkt übernommen. Die Bezeichnung („labeling“) entspricht dem Namen des Start-Elements und die Ereignisdefinition wird als Anmerkung vermerkt. So wird aus dem Element „timerEventDefinition“ eine eindeutige Anmerkung. Später kann anhand dieser speziellen Anmerkung die richtige Transformation durchgeführt werden. Listing 4.25 zeigt den aus Listing 4.24 abgeleiteten Baustein.

```
<brick_01a id="_25">
  <outgoing>_27</outgoing>
  <labeling>Start Event</labeling>
  <annotation>SPEC:TimerEventDefinition</annotation>
</brick_01a>
```

Listing 4.25: abgeleiteter Baustein #1a in XML

Komplizierter wird die Identifizierung von Baustein #5b. Wird von diesem das BPMN-Diagramm abgeleitet, so ergibt sich kein Problem. Soll dieser Baustein jedoch in einem bestehenden Diagramm identifiziert werden, so ist die Aufgabe komplexer, die korrespondierenden Gateways zu ermitteln. An dieser Stelle müssen

die ID-Nummern und dazugehörigen Sequenzflüsse untersucht werden: Es müssen alle ID-Nummern überprüft werden, die vom aufspaltendem Element (Baustein #4) fortführen. Weiterhin müssen die Wege dieser ID-Nummern bis zum zusammenführenden Element (Baustein #5a) verfolgt werden. Es handelt sich genau dann um Baustein #5b, wenn alle ID-Nummern, die vom aufspaltendem Element fortführen, im zusammenführenden Element zusammenlaufen. Das gilt insbesondere auch dann, wenn zwischen diesen beiden Elementen weitere aufspaltende oder zusammenführende Bausteine gesetzt wurden. Listing 4.26 zeigt diesen Baustein. Die Reihenfolge der Elemente ist intuitiv geordnet worden. Zwischen den beiden Gateways liegen zwei parallel abzuarbeitende Aufgaben.

```
<parallelGateway gatewayDirection="Diverging" id="_2" name="Parallel Gateway">
  <incoming>_20</incoming>
  <outgoing>_6</outgoing>
  <outgoing>_8</outgoing>
</parallelGateway>
<task id="_3" name="Task A">
  <incoming>_6</incoming>
  <outgoing>_7</outgoing>
</task>
<task id="_4" name="Task B">
  <incoming>_8</incoming>
  <outgoing>_9</outgoing>
</task>
<parallelGateway gatewayDirection="Converging" id="_5" name="Parallel Gateway">
  <incoming>_7</incoming>
  <incoming>_9</incoming>
</parallelGateway>
<sequenceFlow id="_6" sourceRef="_2" targetRef="_3"/>
<sequenceFlow id="_7" sourceRef="_3" targetRef="_5"/>
<sequenceFlow id="_8" sourceRef="_2" targetRef="_4"/>
<sequenceFlow id="_9" sourceRef="_4" targetRef="_5"/>
```

Listing 4.26: Baustein #5b (vereinfachte XML-Struktur - BPMN)

Das parallele Gateway (Baustein #4) besitzt hier genau zwei ausgehende Kanten. Diese führen zu den „sequenceFlow“-Elementen „_6“ und „_8“. Nun kann über die Source- und Target-ID-Nummern geprüft werden, ob die Sequenzflüsse in einem gemeinsamen Element enden. Dies ist in diesem Fall das Element mit der ID-Nummer „_5“. Das dazugehörige Element ist ebenfalls ein paralleles Gateway und als Baustein #5a identifizierbar. Zu beachten ist an dieser Stelle noch die Anzahl eingehender Kanten dieses Elements, denn sie muss mit der Anzahl ausgehender Kanten des ersten parallelen Gateways übereinstimmen. Das ist hier der Fall. Auf diese Weise kann hier Baustein #5b ermittelt werden. Er ist in Listing 4.27 (auf der nachfolgenden Seite) zu sehen.

Es ist zu beachten, dass der Baustein eine bisher nicht vergebene ID-Nummer erhalten muss, da er keine vorhandene übernehmen kann. Die eingehende Kante entspricht der eingehenden Kante von Baustein #4. Die ausgehende Kante entspricht der ausgehenden Kante von Baustein #5a. Die Bausteine, die zwischen #4 und #5a auftauchen, werden dann nach den bekannten Regeln abgeleitet.

```
<brick05b id="_N01">
  <incoming>_20</incoming>
  <outgoing>_21</outgoing>
  <brick04 id="_2"> <!-- ... --> </brick04>
  <brick02a id="_3"> <!-- ... --> </brick02a>
  <brick02a id="_4"> <!-- ... --> </brick02a>
  <brick03 id="_6"> <!-- ... --> </brick03>
  <brick03 id="_7"> <!-- ... --> </brick03>
  <brick03 id="_8"> <!-- ... --> </brick03>
  <brick03 id="_9"> <!-- ... --> </brick03>
  <brick05a id="_5"> <!-- ... --> </brick05a>
</brick05b>
```

Listing 4.27: abgeleiteter Baustein #5b in XML

Beim Abbruch einer Aktivität (Baustein #12) kommt ein angeheftetes Zwischenereignis zum Einsatz, also ein „boundaryEvent“-Element mit seinem Kind-Element „errorEventDefinition“. Dieses ist hier mit einer Aufgabe verbunden. Um diesen Baustein von Baustein #10 zu unterscheiden, muss überprüft werden, ob das angeheftete Ereignis mit einem Sub-Prozess oder einer Aufgabe verknüpft ist, was über das Attribut „attachedToRef“ geschieht. Listing 4.28 zeigt die XML-Struktur. In Listing 4.29 ist der abgeleitete Baustein dargestellt. Er übernimmt die ID-Nummer der Aufgabe sowie als Bezeichnung deren Name. Die eingehende Kante wird ebenfalls direkt von der Aufgabe übernommen. Die zwei ausgehenden Kanten ergeben sich aus den ausgehenden Kante der Aufgabe sowie des „boundaryEvent“-Elements.

```
<task id="_5" name="Task A">
  <incoming>_9</incoming>
  <outgoing>_10</outgoing>
</task>
<boundaryEvent attachedToRef="_5" cancelActivity="true" id="_6"
                                                  name="Boundary Event">
  <outgoing>_11</outgoing>
  <errorEventDefinition id="_5_ED_1"/>
</boundaryEvent>
```

Listing 4.28: Baustein #12 (vereinfachte XML-Struktur - BPMN)

```
<brick12 id="_5">
  <incoming>_9</incoming>
  <outgoing>_10</outgoing>
  <outgoing>_11</outgoing>
  <labeling>Task A</labeling>
  <exit_condition>
    <condition>Error Boundary Event</condition>
    <target_id>_11</target_id>
  </exit_condition>
</brick12>
```

Listing 4.29: abgeleiteter Baustein #12 in XML

Es ist nun erforderlich, eine Reihenfolge für die Identifizierung der Bausteine im Diagramm zu definieren. Dies wurde bei Baustein #12 deutlich, aber ebenso bei den Bausteinen #5b, #7b sowie #9b. Es steht fest, dass die Bausteine #5b, #7b und #9b vor den Bausteinen #4, #5a, #6, #7a, #8 sowie #9a identifiziert werden müssen, da sie diese enthalten. Die Variante b wird jeweils darüber gefunden, dass

die ID-Nummern von den Bausteinen #4, #6 beziehungsweise #8 ausgehend zu den jeweils korrespondierenden Bausteinen #5a, #7a und #9a führen, ohne dabei in ein Fluss- oder Prozessende zu laufen.

Weiterhin steht fest, dass Baustein #10 vor Baustein #2b identifiziert werden muss, denn Baustein #10 enthält einen Sub-Prozess. Zusätzlich enthält er allerdings auch ein angeheftetes Zwischenereignis, welches die Identifizierung ermöglicht. Umgekehrt ist jedoch im Sub-Prozess kein Hinweis auf das angeheftete Ereignis vermerkt. Aus dem gleichen Grund muss demnach Baustein #12 vor dem Baustein #2a identifiziert werden. Baustein #12 enthält eine Aufgabe, aber zusätzlich auch ein angeheftetes Ereignis. Zwischen Baustein #10 und #12 kann aufgrund der Referenz zu einem Sub-Prozess beziehungsweise einer Aufgabe unterschieden werden.

Alle anderen Bausteine sind eindeutig identifizierbar, weshalb ihre Reihenfolge bei der Suche nicht festgelegt werden muss. Es ist aber von Vorteil, Baustein #3 als letzten zu identifizieren und transformieren, weil er die Verbindungen zwischen den anderen Bausteinen modelliert. Daraus ergibt sich folgende mögliche Reihenfolge zur Identifizierung: #12, #11, #10, #9b, #7b, #5b, #9a, #7a, #5a, #8, #6, #4, #2b, #2a, #1c, #1b, #1a, #3. Dies entspricht in etwa der umgekehrten Definierungsreihenfolge.

Über die Bausteine wird bereits ein großer Teil, aber noch nicht alle Notationselemente der BPMN abgedeckt. Tabelle 4.8 zeigt die bisherige Abdeckung der Elemente der BPMN.

Element	Abdeckung durch ...
Startereignis (alle Typen)	#1a
Endereignis (alle Typen)	#1c
Angeheftete Ereignisse (Fehler-Ereignis)	#10 und #12
Aufgabe (alle Typen)	#2a
Sub-Prozess	#2b
Sequenzfluss	#3
Fehlerfluss	#10 und #12
Paralleles Gateway	#4, #5a und #5b
Exklusives Gateway	#6, #7a und #7b
Inklusives Gateway	#8, #9a und #9b
Ereignis-basiertes Gateway	#11
Anmerkung	alle, außer #5b, #7b und #9b

Tabelle 4.8: Abdeckung der Notationselemente durch die Bausteine

4.3.2 Bausteine in der XML-Struktur der UML Aktivitätsdiagramme

Gleichermaßen muss für die UML Aktivitätsdiagramme ein Modellierungswerkzeug ausgewählt werden, welches den Export von XML-Strukturen anbietet. Als Werkzeug für diese Modellierung wurde Astah Professional[142] verwendet. Die Wahl ist aus drei Gründen auf dieses Werkzeug gefallen. Zum einen handelt es sich bei Astah Professional um eine Software, für die Studenten eine freie akademische Lizenz erhalten können, sodass für die Verwendung in dieser Arbeit keine Kosten für die Software an sich anfallen. Mit Astah Community kann jeder eine eingeschränkte Version dieser Software benutzen. Zum anderen unterstützt Astah Professional UML 2.x und wird regelmäßig aktualisiert, um diesen Anspruch auch weiterhin zu genügen. Das heißt also, dass die Diagramme den aktuellen Richtlinien entsprechen. Außerdem bietet die Software einen XML-Export basierend auf dem XML Metadata Interchange (XMI)[143] für UML-Diagramme an. Gerade dieses letztgenannte Detail ist im Weiteren wichtig, denn anhand dieses XML-Exports kann gezeigt werden, nach welchen XML-Strukturen gesucht werden muss, um die definierten Bausteine ableiten zu können. Deshalb wurde Astah Professional als Modellierungswerkzeug ausgewählt.

Dabei wird auch hier eine vereinfachte Version der XML-Struktur verwendet, denn die exportierte Datei enthält eine ganze Reihe Informationen zu den Elementen, ihren Beziehungen zueinander sowie zur Darstellung. Nicht alle diese Informationen sind wichtig, um die Bausteine aus der XML-Struktur abzuleiten. Die Datei enthält das Hauptelement „XMI“, mit den Kindern „XMI.header“ sowie „XMI.content“. Im letztgenannten Kind-Element gibt es wiederum diverse Kind-Elemente in mehreren Ebenen. Das für die Ableitung wichtige Kind-Element ist dabei „ActivityGraph“. Es stellt hier das Rahmenkonstrukt für einen Prozess dar. Über Referenzen wird auf andere wichtige Elemente verwiesen, wenn dies nötig ist. Listing 4.30 zeigt das entsprechende Konstrukt.

```
<XMI:header> <!-- ... --> </XMI:header>
<XMI:content>
  <!-- ... -->
  <UML:Model>
    <!-- ... -->
    <UML:Namespace.ownedElement>
      <!-- ... -->
        <UML:ActivityGraph>
          <!-- ... -->
        </UML:ActivityGraph>
      <!-- ... -->
    </UML:Namespace.ownedElement>
    <!-- ... -->
  </UML:Model>
  <!-- ... -->
</XMI:content>
```

Listing 4.30: Rahmenkonstrukt (vereinfachte XML-Struktur - UML AD)

[142] Version 6.7.0 vom 14. März 2013; http://astah.net/editions/professional
[143] Vgl. OMG [2011c]

Innerhalb des „ActivityGraph“-Elements gibt es diverse Ebenen von Kind-Elementen. Um die Listings im Nachfolgenden besser lesen zu können, werden die nicht benötigten Elemente in der Darstellung weggelassen, das heißt, dass die Kind-Elemente hier quasi auf höherliegende Ebenen verlagert werden. Dies betrifft beispielsweise die Aufteilung in „UML:StateMachine.top“, „UML:StateMachine.context“ und „UML:StateMachine.transitions“. Im erstgenannten Element befinden sich zwei weitere Ebenen in denen schließlich zum Beispiel Startknoten, Endknoten und Aktion enthalten sind. Das „content“-Element verweist auf die übergeordnete UML-Modell und im „transitions“-Element befinden sich die Verbindungen zwischen den einzelnen Elementen. Im Nachfolgenden wird so getan, als würden sich diese Elemente alle direkt im „activityGraph“-Element befinden.

Es werden jedoch auch noch weitere Teile der XML-Struktur weggelassen, da sie ebenfalls nicht der Identifizierung der Bausteine dienlich sind. Dies betrifft bei der von Astah Professional erzeugten XML-Struktur zum Beispiel die folgenden Attribute beim Startknoten: „version“ und „unSolvedFlag“. Auf das erstgenannte Attribut wird verzichtet, weil es im Folgenden immer den Wert „0“ hat, auf das letztgenannte wird verzichtet, da es im Folgenden immer den Wert „false“ hat. Aus den selben Gründen werden andere Attribute weggelassen. Auch die ID-Nummern wird im Folgenden deutlich verkürzt dargestellt. Weiterhin wird beispielsweise das Element „UML:stateVertex.container“ weggelassen, welches auf ein anderes weggelassenes Element verweist und somit nur zu Verwirrung führen würde.

Exemplarisch wird im Folgenden an drei XML-Strukturen die Ableitung des entsprechenden Bausteins gezeigt. In Anhang E sind alle Ableitungen im Detail beschrieben. Die einzelnen Bausteine sind anhand von Indikatorelementen zu identifizieren. Diese Indikatorelemente sind in Tabelle 4.9 und Tabelle 4.10 in einer Übersicht zusammengefasst.

Baustein	Indikatorelement bei UML AD
#1a	PseudoState (kind=initial)
#1b	ActionState (vom Stereotyp „flow_final_node“)
#1c	FinalState
#2a	ActionState
#2b	SubactivityState
#3	Transition
#4	PseudoState (kind=fork)
#5a	PseudoState (kind=join)
#5b	Baustein #4 (alle ausgehenden Kanten lassen sich über Transitionen zu Baustein #5a verfolgen)
#6	PseudoState (kind=junction sowie mehr als einer ausgehender Kanten)
#7a	PseudoState (kind=junction sowie mehr als einer eingehendnen Kante)
#7b	Baustein #6 (alle ausgehenden Kanten lassen sich über Transitionen zu Baustein #7a verfolgen)
#8	PseudoState (kind=fork sowie dazugehörige Transition mit Bedingung)

Tabelle 4.9: Indikatorelemente der UML AD für die Transformation

Baustein	Indikatorelement bei UML AD
#9a	PseudoState (kind=join sowie dazugehörige Transition mit Bedingung)
#9b	Baustein #8 (alle ausgehenden Kanten lassen sich über Transitionen zu Baustein #9a verfolgen)
#10	InterruptibleActivityRegion (mit mehreren ActionStates)
#11	InterruptibleActivityRegion (mit mehreren ActionStates vom Stereotyp signal receipt)
#12	InterruptibleActivityRegion (mit zwei ActionStates, einer davon vom Stereotyp signal receipt)

Tabelle 4.10: Indikatorelemente der UML AD für die Transformation (Fortsetzung)

Listing 4.31 zeigt die XML-Notation des Startknotens (Baustein #1a). Er ist eindeutig identifizierbar, da er als „Pseudostate“ mit dem Attribut „kind=‘initial‘“ deklariert wird. Er verwendet unter anderem die Attribute „id“ sowie „name“ und enthält eine Referenz auf die ID-Nummer der ausgehenden Kante.

```
<UML:Pseudostate xmi.id="_02" name="Initial Node" kind="initial">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_03"/>
  </UML:StateVertex.outgoing>
</UML:Pseudostate>
```

Listing 4.31: Baustein #1a (vereinfachte XML-Struktur - UML AD)

Listing 4.32 zeigt, wie der daraus abgeleitete Baustein aussieht. Er übernimmt die ID-Nummer des Pseudo-Status als eigene ID-Nummer. Weiterhin wird der Name als Bezeichnung gesetzt und die ausgehende Kante direkt übernommen. Zu erkennen ist diese Kante auch hier durch das Wort „outgoing“.

```
<brick01a id="_2">
  <outgoing>_03</outgoing>
  <labeling>Initial Node</labeling>
</brick01a>
```

Listing 4.32: abgeleiteter Baustein #1a in XML

Den Baustein #5b in einem bestehenden Diagramm zu identifizieren wird ebenso komplex wie bei der BPMN. Alle vom aufspaltenden Element ausgehenden Verbindungen müssen im zusammenführenden Element enden. Dafür müssen die Transitionen betrachtet werden. Listing 4.33 (auf der nachfolgenden Seite) zeigt beispielhaft einen Baustein #5b.

```
<UML:Pseudostate xmi.id="_05" name="Fork Node" kind="fork">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_T06"/>
    <UML:Transition xmi.idref="_T09"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_T05"/>
  </UML:StateVertex.incoming>
</UML:Pseudostate>
<UML:Pseudostate xmi.id="_08" name="Join Node" kind="join">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_T08"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_T07"/>
    <UML:Transition xmi.idref="_T10"/>
  </UML:StateVertex.incoming>
</UML:Pseudostate>
<UML:ActionState xmi.id="_06" name="Action A">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_T07"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_T06"/>
  </UML:StateVertex.incoming>
</UML:ActionState>
<UML:ActionState xmi.id="_07" name="Action B">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_T10"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_T09"/>
  </UML:StateVertex.incoming>
</UML:ActionState>
<UML:Transition xmi.id="_T06">
  <UML:Transition.source>
    <UML:StateVertex xmi.idref="_05"/>
  </UML:Transition.source>
  <UML:Transition.target>
    <UML:StateVertex xmi.idref="_06"/>
  </UML:Transition.target>
</UML:Transition>
<UML:Transition xmi.id="_T07">
  <UML:Transition.source>
    <UML:StateVertex xmi.idref="_06"/>
  </UML:Transition.source>
  <UML:Transition.target>
    <UML:StateVertex xmi.idref="_08"/>
  </UML:Transition.target>
</UML:Transition>
<UML:Transition xmi.id="_T09">
  <UML:Transition.source>
    <UML:StateVertex xmi.idref="_05"/>
  </UML:Transition.source>
  <UML:Transition.target>
    <UML:StateVertex xmi.idref="_07"/>
  </UML:Transition.target>
</UML:Transition>
<UML:Transition xmi.id="_T10">
  <UML:Transition.source>
    <UML:StateVertex xmi.idref="_07"/>
  </UML:Transition.source>
  <UML:Transition.target>
    <UML:StateVertex xmi.idref="_08"/>
  </UML:Transition.target>
</UML:Transition>
```

Listing 4.33: Baustein #5b (vereinfachte XML-Struktur - UML AD)

Trotz des enormen Umfangs von Listing 4.33 ist der abgeleitete Baustein ziemlich klein (Listing 4.34). Zunächst einmal weist Baustein #4 auf die potenzielle Existenz von Baustein #5b hin. Die ausgehenden Kanten werden betrachtet. In diesem Fall also die Transitionen mit den ID-Nummern „_T06“ und „_T09“. Über die „source“-Elemente und die „target“-Elemente wird nun geprüft, ob die weiterführenden Transitionen in einem gemeinsamen Element enden. Dies ist das Element mit der ID-Nummer „_08“, welches sich als Baustein #5a mit passender Anzahl eingehender Kanten erweist. Somit handelt es sich hierbei um Baustein #5b. Nun werden die Bausteine #4 und #5a sowie die, die dazwischen auftauchen, nach den bekannten Regeln abgeleitet. Die eingehende Kante ergibt sich aus der eingehenden Kante von Baustein #4, die ausgehende Kante ergibt sich aus der ausgehenden Kante von Baustein #5a. Der Baustein an sich erhält eine neue ID-Nummer, weil er keine vorhandene übernehmen kann.

```
<brick05b id="_N01">
  <incoming>_T05</incoming>
  <outgoing>_T08</outgoing>
  <brick04 id="_05"> <!-- ... --> </brick04>
  <brick02a id="_06"> <!-- ... --> </brick02a>
  <brick02a id="_07"> <!-- ... --> </brick02a>
  <brick03 id="_T06"> <!-- ... --> </brick03>
  <brick03 id="_T07"> <!-- ... --> </brick03>
  <brick03 id="_T09"> <!-- ... --> </brick03>
  <brick03 id="_T10"> <!-- ... --> </brick03>
  <brick05a id="_08"> <!-- ... --> </brick05a>
</brick05b>
```

Listing 4.34: abgeleiteter Baustein #5b in XML

Bei Baustein #12 liegen genau eine Aktion und ein Signal in einem Unterbrechungsbereich. Die Aktion kann auf diese Weise jederzeit bei Signaleingang abgebrochen werden. Listing 4.35 (auf der nächsten Seite) zeigt die entsprechende XML-Struktur.

Der Baustein wird daraus wie folgt abgeleitet: Die ID-Nummer des Unterbrechungsbereiches wird übernommen und der Name der Aktion wird als Bezeichnung gesetzt. Die eingehende Kante ergibt sich aus der eingehende Kante der Aktion. Die beiden ausgehenden Kanten ergeben sich aus der ausgehenden Kante der Aktion sowie der ausgehenden Kante des Signals. Das „condition“-Element im „exit_condition“-Element wird mit dem Namen des Signals befüllt und als „target_id“ wird nochmals die ausgehende Kante des Signals angegeben. Alle weiteren Elemente können ignoriert werden. Der so abgeleitete Baustein ist in Listing 4.36 (auf der nachfolgenden Seite) dargestellt.

```
<UML:InterruptibleActivityRegion xmi.id="_I02" name="Interrup. Activity Region">
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
    <JUDE:ModelElement xmi.idref="_T51"/>
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
    <JUDE:ModelElement xmi.idref="_50"/>
    <JUDE:ModelElement xmi.idref="_51"/>
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
</UML:InterruptibleActivityRegion>
<!-- ... -->
  <UML:ActivityGraph xmi.id="_01">
  <!-- ... -->
    <UML:ActionState xmi.id="_50" name="Cancellation Signal">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_I02"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:ModelElement.stereotype>
        <UML:Stereotype xmi.idref="_S01"/>
      </UML:ModelElement.stereotype>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_T51"/>
      </UML:StateVertex.outgoing>
    </UML:ActionState>
    <UML:ActionState xmi.id="_51" name="Action A">
    <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_I02"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_T53"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_T52"/>
      </UML:StateVertex.incoming>
    </UML:ActionState>
  <!-- ... -->
  </UML:ActivityGraph>
  <!-- ... -->
  <UML:Stereotype xmi.id="_S01" name="signal receipt">
    <UML:Stereotype.baseClass>ActionState</UML:Stereotype.baseClass>
    <UML:Stereotype.extendedElement>
      <JUDE:ModelElement xmi.idref="_50"/>
    </UML:Stereotype.extendedElement>
  </UML:Stereotype>
```

Listing 4.35: Baustein #12 (vereinfachte XML-Struktur - UML AD)

```
<brick12 id="_I03">
  <incoming>_T52</incoming>
  <outgoing>_T53</outgoing>
  <outgoing>_T51</outgoing>
  <labeling>Action A</labeling>
  <exit_condition>
    <condition>Cancellation Signal</condition>
    <target_id>_T51</target_id>
  </exit_condition>
</brick12>
```

Listing 4.36: abgeleiteter Baustein #12 in XML

Nun muss noch die Reihenfolge festgelegt werden, in der die Bausteine im Diagramm identifiziert werden müssen. Die Bausteine #10, #11 und #12 müssen vor allen anderen identifiziert werden, da sie ansonsten aufgrund der enthaltenen Elemente als andere identifiziert werden könnten. Um sie zu identifizieren wird nach dem Unterbrechungsbereich gesucht, den sie alle zur Darstellung benötigen. Baustein #12 kann von Baustein #11 dadurch unterschieden werden, dass er genau ein Signal und eine Aktion im Unterbrechungsbereich enthält und keine weiteren Elemente. Baustein #11 hingegen enthält diverse, aber mindestens zwei, Signale sowie eine Gabelung. Baustein #10 enthält ebenfalls eine Gabelung, allerdings keine Signale, sondern Aktionen oder Aktivitäten. Aus diesem Grund ist die Unterscheidung einfach.

Im Weiteren gilt wie bei der BPMN, dass die Bausteine #5b, #7b und #9b vor den Bausteinen #4, #5a, #6, #7a, #8 sowie #9a identifiziert werden müssen, da sie diese enthalten. Zusätzlich muss darauf geachtet werden, dass Baustein #9b vor Baustein #5b sowie Baustein #9a vor Baustein #5a erkannt wird. Dies ist notwendig, da sich diese Bausteine nur durch die Anmerkung beim Baustein #9a unterscheiden. Ein ähnliches Problem ergibt sich bei den Bausteinen #4 und #8: Diese lassen sich nur durch die Bedingungen an den ausgehenden Kanten unterscheiden. Bei Baustein #8 sind sie vorhanden, bei Baustein #4 nicht. Aus diesem Grund muss nach Baustein #8 zuerst gesucht werden. Alle anderen Bausteine sind eindeutig identifizierbar, weshalb ihre Reihenfolge bei der Suche nicht festgelegt werden muss. Es ist aber von Vorteil, Baustein #3 als letzten zu identifizieren und transformieren, weil er die Verbindungen zwischen den anderen Bausteinen modelliert. Daraus ergibt sich folgende mögliche Reihenfolge zur Identifizierung: #12, #11, #10, #9b, #7b, #5b, #9a, #7a, #5a, #8, #6, #4, #2b, #2a, #1c, #1b, #1a, #3. Dies entspricht in etwa der umgekehrten Definierungsreihenfolge sowie der Reihenfolge der Identifizierung bei der BPMN.

Über die Bausteine wird bereits ein Großteil, aber noch nicht alle Notationselemente der UML Aktivitätsdiagramme abgedeckt. Tabelle 4.11 zeigt die bisherige Abdeckung der Elemente der Aktivitätsdiagramme der UML.

Element	Abdeckung durch ...
Startknoten	#1a
Endknoten	#1c
Flussende	#1b
Aktion	#2a
Aktivität	#2b
Kontrollfluss	#3
Gabelung	#4 und #8
Vereinigung	#5a, #5b, #9a und #9b
Entscheidungsknoten	#6
Verbindungsknoten	#7a und #7b
Anmerkung	alle, außer #5b, #7b und #9b
Unterbrechnungsbereich	#10, #11 und #12

Tabelle 4.11: Abdeckung der Notationselemente durch die Bausteine

4.4 Ableitung der XML-Struktur aus einem Baustein

Bis hier wurde gezeigt, wie aus der XML-Struktur der BPMN beziehungsweise der UML Aktivitätsdiagramme die Bausteine abgeleitet werden können. Nun ist es noch wichtig zu definieren, wie aus den Bausteinen die XML-Struktur generiert wird, die beispielsweise vom Yaoqiang BPMN Editor oder Astah Professional gelesen werden kann.

Exemplarisch wird im Folgenden an drei Bausteinen die Ableitung der entsprechenden XML-Struktur gezeigt. In Anhang F sind alle Ableitungen im Detail beschrieben. Es wird dieselbe Art vereinfachter XML-Struktur verwendet wie zuvor, da dadurch die Lesbarkeit der Listings vereinfacht wird.

Die Schritte, die zuvor zur Ableitung der Bausteine aus der XML-Struktur heraus gegangen worden sind, werden nun praktisch in umgekehrter Reihenfolge ausgeführt. Listing 4.37 zeigt den Baustein #1a.

```
<brick_01a id="_02">
  <outgoing>_05</outgoing>
  <labeling>Start Event</labeling>
  <annotation>SPEC:TimerEventDefinition</annotation>
</brick_01a>
```

Listing 4.37: Baustein #1a

Welche Elemente in der BPMN benötigt werden, um diesen Baustein darzustellen, ist aus den vorangegangenen Abschnitten klar. Für diesen Baustein wird das „startEvent“-Element (mit seinen Kind-Elementen) benötigt. So wird die ID-Nummer des Bausteins als ID-Nummer des Elements benutzt. Die Bezeichnung wird wieder zum Namen. Die ausgehende Kante wird ebenfalls direkt übernommen. Aus dem Inhalt der Anmerkung heraus ist erkennbar, dass es sich um eine besondere Anmerkung handelt. Das Schlagwort verdeutlicht, dass das Startereignis vom Typ Zeit-Startereignis ist. Dementsprechend wird eine Ereignisdefinition hinzugefügt, welche eine eigene ID-Nummer erhält. Die erhaltene XML-Struktur sieht dann wie in Listing 4.38 dargestellt aus.

```
<startEvent id="_02" name="Start Event">
  <outgoing>_05</outgoing>
  <timerEventDefinition id="_05_ED_1"/>
</startEvent>
```

Listing 4.38: abgeleitete XML-Struktur - BPMN (Baustein #1a) (1)

Handelt es sich hingegen um eine Anmerkung ohne spezielles Schlagwort, so wird diese wie in Listing 4.39 dargestellt angefügt.

```
<startEvent id="_02" name="Start Event">
  <outgoing>_05</outgoing>
</startEvent>
<!-- ... -->
<textAnnotation id="_03" textFormat="text/plain">
  <text>Annotation</text>
</textAnnotation>
<association id="_04" sourceRef="_02" targetRef="_03"/>
```

Listing 4.39: abgeleitete XML-Struktur - BPMN (Baustein #1a) (2)

Genauso funktioniert die Ableitung der XML-Struktur für die UML Aktivitätsdiagramme. Es ist bekannt, dass das entsprechende Element ein „PseudoState“-Element ist mit dem „kind“-Attribut mit dem Wert „initial“. Die ID-Nummer wird wieder als ID-Nummer des Elements übernommen, die Bezeichnung wird zum Namen-Attribut. Die ausgehende Kante wird ebenfalls direkt übernommen. Obwohl die Anmerkung auf eine Besonderheit hinweist, gibt es bei den UML Aktivitätsdiagrammen keine Entsprechung dafür, sodass die Anmerkung unverändert bestehen bleibt. Die erhaltene XML-Struktur sieht dann wie in Listing 4.40 dargestellt aus.

```
<UML:Comment xmi.id="_C01" name="" body="SPEC:TimerEventDefinition">
  <UML:Comment.annotatedElement>
    <JUDE:ModelElement xmi.idref="_02"/>
  </UML:Comment.annotatedElement>
</UML:Comment>
<!-- ... -->
  <UML:ActivityGraph xmi.id="_01">
    <!-- ... -->
    <UML:Pseudostate xmi.id="_02" name="Start Event" kind="initial">
      <UML:ModelElement.comment>
        <UML:Comment xmi.idref ="_C01"/>
      </UML:ModelElement.comment>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_05"/>
      </UML:StateVertex.outgoing>
    </UML:Pseudostate>
    <!-- ... -->
  </UML:ActivityGraph>
```

Listing 4.40: abgeleitete XML-Struktur - UML AD (Baustein #1a)

Auch die anderen Bausteine lassen sich auf diese Weise in die XML-Struktur überführen. Listing 4.41 zeigt Baustein #9a. Es ist bekannt, dass dieser Baustein in der Notation der BPMN durch das „inclusiveGateway“-Element (mit dessen Kind-Elementen) beschrieben wird. Das Attribut „gatewayDirection“ wird als unspezifiziert angegeben wie es auch schon in der Gegenrichtung der Fall gewesen ist. Die ID-Nummer des Bausteins wird als ID-Nummer „inclusiveGateway“-Elements genommen und die Bezeichnung wird zum Namen-Attribut. Die eingehenden sowie ausgehenden Kanten werden bei den „incoming“ beziehungsweise „outgoing“-Elementen eingesetzt. Daraus ergibt sich die in Listing 4.42 dargestellte Struktur.

```
<brick_09a id="_10">
  <incoming>_11</incoming>
  <incoming>_12</incoming>
  <incoming>_13</incoming>
  <outgoing>_14</outgoing>
  <labeling>Join</labeling>
</brick09a>
```

Listing 4.41: Baustein #9a

```
<inclusiveGateway gatewayDirection="Unspecified" id="_10" name="Join">
  <incoming>_11</incoming>
  <incoming>_12</incoming>
  <incoming>_13</incoming>
  <outgoing>_14</outgoing>
</inclusiveGateway>
```

Listing 4.42: abgeleitete XML-Struktur - BPMN (Baustein #9a)

Die abgeleitete Struktur für die UML Aktivitätsdiagramme sieht dagegen komplexer aus, ist aber ebenso einfach zu erzeugen. In der XML-Struktur wird der Baustein mittels eines „PseudoState“-Elements dargestellt. Dieses Mal ist die Art jedoch „join“. Die ID-Nummer des Bausteins wird zur ID-Nummer des Status-Elements, die Bezeichnung wird zum Namen-Attribut. Auch hier werden die eingehenden und ausgehenden Kanten in die entsprechenden Kind-Elemente eingefügt. Zum Schluss kommt die Besonderheit bei diesem Baustein: Da Baustein #9a eigentlich nicht in der Notation der UML Aktivitätsdiagramme erscheinen kann, wird eine Anmerkung gesetzt, die wenigstens optisch dafür sorgt, dass das gewünschte Verhalten modellierbar wird. Bei einer späteren Rücktransformation kann anhand dieser Anmerkung der richtige Baustein abgeleitet werden. Die abgeleitete XML-Struktur ist in Listing 4.43 (auf der folgenden Seite) zu sehen.

```
<UML:Comment xmi.id="_C01" name="" body="SPEC:Brick09a">
  <UML:Comment.annotatedElement>
    <JUDE:ModelElement xmi.idref="_10"/>
  </UML:Comment.annotatedElement>
</UML:Comment>
<!-- ... -->
  <UML:ActivityGraph xmi.id="_01">
  <!-- ... -->
    <UML:Pseudostate xmi.id="_10" name="Join" kind="join">
      <UML:ModelElement.comment>
        <UML:Comment xmi.idref="_C01"/>
      </UML:ModelElement.comment>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_14"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_11"/>
        <UML:Transition xmi.idref="_12"/>
        <UML:Transition xmi.idref="_13"/>
      </UML:StateVertex.incoming>
    </UML:Pseudostate>
  <!-- ... -->
  </UML:ActivityGraph>
```

Listing 4.43: abgeleitete XML-Struktur - UML AD (Baustein #9a)

Das dritte Beispiel zeigt die Generierung der XML-Struktur für Baustein #12. In Listing 4.44 ist dieser exemplarisch dargestellt.

```
<brick12 id="_5">
  <incoming>_9</incoming>
  <outgoing>_10</outgoing>
  <outgoing>_11</outgoing>
  <labeling>Action Task</labeling>
  <exit_condition>
    <condition>Exit Condition</condition>
    <target_id>_11</target_id>
  </exit_condition>
</brick12>
```

Listing 4.44: Baustein #12

In der BPMN werden für den Baustein zwei Notationselemente für die Darstellung benötigt. Zum einen eine Aufgabe, zum anderen ein spezielles angeheftetes Zwischenereignis. Letzteres wird durch das „boundaryEvent“-Element mit seinem Kind-Element „errorEventDefinition“ dargestellt. Nun wird der Baustein wie folgt auf die XML-Struktur übertragen: Die ID-Nummer des Bausteins wird zur ID-Nummer der Aufgabe. Die Bezeichnung wird zum Namen-Attribut. Die eingehende sowie die ausgehenden Kanten werden ebenfalls direkt übernommen. Der Inhalt des „condition“-Elements wird zum Namen des angehefteten Ereignisses. Dessen Attribut „attachedToRef“ erhält die ID-Nummer des Bausteins. Auf diese Weise wird die Verbindung der beiden Elemente hergestellt. Das Kind-Element „target_id“ gibt seinen Inhalt an das „outgoing“-Element des Zwischenereignisses weiter. Im gleichen Zug wird das zuvor gesetzte „outgoing“-Element wieder aus der Aufgabe entfernt. Dann entspricht es auch den festgelegten Modellierungsregeln, nach denen eine Aufgabe nur eine ausgehende Kante haben darf (Listing 4.45).

```
<task id="_5" name="Action Task">
  <incoming>_9</incoming>
  <outgoing>_10</outgoing>
</task>
<boundaryEvent attachedToRef="_5" cancelActivity="true" id="_6"
                                                        name="Exit Condition">
  <outgoing>_11</outgoing>
  <errorEventDefinition id="_5_ED_1"/>
</boundaryEvent>
```

Listing 4.45: abgeleitete XML-Struktur - BPMN (Baustein #12)

```
<UML:InterruptibleActivityRegion xmi.id="_5" name="Interruptible Activity R.">
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
    <JUDE:ModelElement xmi.idref="_11"/>
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
    <JUDE:ModelElement xmi.idref="_50"/>
    <JUDE:ModelElement xmi.idref="_51"/>
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
</UML:InterruptibleActivityRegion>
<!-- ... -->
  <UML:ActivityGraph xmi.id="_01">
  <!-- ... -->
    <UML:ActionState xmi.id="_50" name="Exit Condition">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_5"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:ModelElement.stereotype>
        <UML:Stereotype xmi.idref="_S01"/>
      </UML:ModelElement.stereotype>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_11"/>
      </UML:StateVertex.outgoing>
    </UML:ActionState>
    <UML:ActionState xmi.id="_51" name="Action Task">
    <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_5"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_10"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_9"/>
      </UML:StateVertex.incoming>
    </UML:ActionState>
  <!-- ... -->
  </UML:ActivityGraph>
  <!-- ... -->
  <UML:Stereotype xmi.id="_S01" name="signal receipt">
    <UML:Stereotype.baseClass>ActionState</UML:Stereotype.baseClass>
    <UML:Stereotype.extendedElement>
      <JUDE:ModelElement xmi.idref="_50"/>
    </UML:Stereotype.extendedElement>
  </UML:Stereotype>
```

Listing 4.46: abgeleitete XML-Struktur - UML AD (Baustein #12)

Bei den UML Aktivitätsdiagrammen werden ebenfalls mehrere Elemente benötigt, um Baustein #12 darzustellen: Es wird ein Unterbrechnungsbereich sowie eine Aktion und ein Signal benötigt. Danach werden die Werte aus dem Baustein wie folgt in die XML-Struktur eingetragen: Die ID-Nummer des Bausteins wird zur ID-Nummer des Unterbrechnungsbereichs. Die Bezeichnung wird in das Namen-Attribut der Aktion eingetragen. Die eingehende Kante des Bausteins wird zur eingehenden Kante der Aktion. Die ausgehenden Kanten werden als ausgehende Kanten der Aktion gesetzt. Der Inhalt des „target_id“-Elementes wird sowohl bei der Unterbrechnungskante des Unterbrechnungsbereiches eingesetzt als auch als ausgehende Kante beim Signal. Ebenso wird die ausgehende Kante mit der gleichen Nummer wieder aus der Aktion gelöscht. Auf diese Weise wird den festgelegten Modellierungsregeln entsprochen, bei denen eine Aktion nur eine ausgehende Kante haben darf. Listing 4.46 (auf der vorherigen Seite) zeigt die generierte XML-Struktur.

4.5 Transformation verbliebener Notationselemente

In Abschnitt 4.2.3 wurde bereits festgestellt, dass eine Reihe Notationselemente bei der Darstellung der Workflow Pattern keine Verwendung findet und dies auf Überflüssigkeit hindeuten würde. Jedoch hat jedes Notationselement eine bestimmte Aufgabe und einen Sinn. Aus diesem Grund wird an dieser Stelle eine Transformation zwischen den verbleibenden Elementen beschrieben, sofern dies möglich ist. Diese Transformation orientiert sich allerdings nicht am vorgestellten Baustein-Prinzip, sondern an einer 1-zu-1-Transformation. Weiterhin muss gesagt werden, dass diese Regeln ausschließlich auf Transformationen zwischen der BPMN und den UML Aktivitätsdiagrammen ausgelegt sind und nicht auf andere Modellsprachen angewandt werden können.

Pool/Lane und Aktivitätsbereich

In der BPMN gibt es Pools und Lanes, um unterschiedliche Teilnehmer darzustellen. Dabei sind Pools Notationselemente, die auf ein Kollaborationsdiagramm hindeuten. Solange nur ein Pool vorhanden ist, besteht jedoch keine Kollaboration. Die XML-Struktur eines Pools ist in Listing 4.47 zu sehen. Er wird außerhalb des Rahmenkonstruktes beschrieben und verweist auf den entsprechenden Prozess.

```
<collaboration id="COLLABORATION_1">
  <participant id="_2" name="Participant" processRef="PROCESS_1"/>
</collaboration>
<process id="PROCESS_1">
<!-- ... -->
</process>
```

Listing 4.47: Pool (vereinfachte XML-Struktur - BPMN)

```
<process id="PROCESS_1">
  <laneSet>
    <lane id="_9" name="Lane A">
      <childLaneSet>
        <lane id="_11" name="Lane A1">
          <flowNodeRef>_13</flowNodeRef>
          <!-- weitere -->
        </lane>
        <lane id="_12" name="Lane A2">
          <flowNodeRef>_14</flowNodeRef>
          <!-- weitere -->
        </lane>
      </childLaneSet>
    </lane>
    <lane id="_13" name="Lane B">
      <flowNodeRef>_14</flowNodeRef>
      <!-- weitere -->
    </lane>
  </laneSet>
  <!-- ... -->
</process>
```

Listing 4.48: Lane (verschachtelt) (vereinfachte XML-Struktur - BPMN)

Ein Pool kann mehrere, ineinander verschachtelte Lanes enthalten, beispielsweise um unterschiedliche Abteilungen eines Unternehmens darzustellen. Eine Lane wird, wie in Listing 4.48 (auf der vorherigen Seite) abgebildet, als Lane-Set innerhalb des entsprechenden Prozesses beschrieben und enthält Referenzen auf alle Elemente, die sie enthält. Die Elemente selbst haben jedoch keine Referenz auf die Lane oder den Pool, zu der beziehungsweise zu dem sie gehören.

Bei den UML Aktivitätsdiagrammen funktioniert das Ganze ähnlich, sieht jedoch aufgrund der vielen Referenzen zu den dazugehörigen Elementen deutlich komplizierter aus (Listing 4.50). Im Gegensatz zur BPMN wird hier nicht zwischen Pool und Lane unterschieden. Es gibt hier nur Aktivitätsbereiche, welche in der XML-Struktur als Partitionen bezeichnet werden. Außerdem muss darauf geachtet werden, dass es bei den UML Aktivitätsdiagrammen möglich ist, Partitionen senkrecht zueinander zu positionieren, was bei der BPMN nicht erlaubt ist. Um eine Transformation zu ermöglichen muss daher eine Einschränkung bei den UML Aktivitätsdiagrammen in Kauf genommen werden. Listing 4.49 und Listing 4.50 (auf der nächsten Seite) stellen die gleiche Aufteilung von Teilnehmern dar: „A“ mit den untergeordneten Einheiten „A1“ sowie „A2“, welche dann wiederum die einzelnen Elemente enthalten.

Die ID-Nummer des Pools wird zur ID-Nummer des äußeren Aktivitätsbereiches. Das „processRef“-Attribut gibt dabei an, zu welchem Prozess der Pool gehört. Der Wert wird daher in das „UML:Partition.superPartition“-Element des Aktivitätsbereiches eingetragen. Der Name des Pools wird zum Namen desselben Aktivitätsbereiches. Danach werden die Lanes des Prozesses auf die untergeordneten Partitionen transformiert: Die ID-Nummer der Lane sowie ihr Name werden zur ID-Nummer beziehungsweise zum Namen-Attribut des untergeordneten Aktivitätsbereiches. Die zugeordneten „flowNodeRef“-Elemente werden direkt übertragen (in das „UML:Partition.contents“-Element). Die ID-Nummern der untergeordneten Partitionen werden zusätzlich noch in das „UML:Partition.subGroup“-Element der übergeordneten Partition („A“) eingetragen. Gleichzeitig wird in das „UML:Partition.superPartition“-Element der untergeordneten Partitio-

```
<collaboration id="COLLABORATION_1">
  <participant id="_P01" name="Partition A" processRef="PROCESS_1"/>
</collaboration>
<process id="PROCESS_1">
  <laneSet>
    <lane id="_P02" name="Partition A1">
      <flowNodeRef>_5</flowNodeRef>
      <!-- weitere -->
    </lane>
    <lane id="_P03" name="Partition A2">
      <flowNodeRef>_10</flowNodeRef>
      <!-- weitere -->
    </lane>
  </laneSet>
  <!-- ... -->
</process>
```

Listing 4.49: Pool und Lane (verschachtelt) (vereinfachte XML-Struktur - BPMN)

```
<UML:Partition xmi.id="_P01" name="Partition A">
  <UML:Partition.activityGraph>
    <UML:ActivityGraph xmi.idref="_01"/> <!-- zugeordneter Graph -->
  </UML:Partition.activityGraph>
  <UML:Partition.superPartition>
    <UML:Partition xmi.idref="PROCESS_1"/> <!-- übergeordnete Partition -->
  </UML:Partition.superPartition>
  <UML:Partition.subGroup>
    <JUDE:ModelElement xmi.idref="_P02"/> <!-- untergeordnete Partitionen -->
    <JUDE:ModelElement xmi.idref="_P03"/>
  </UML:Partition.subGroup>
</UML:Partition>
<UML:Partition xmi.id="_P02" name="Partition A1">
  <UML:Partition.contents>
    <JUDE:ModelElement xmi.idref="_05"/> <!-- zugeordnete Elemente -->
    <!-- weitere -->
  </UML:Partition.contents>
  <UML:Partition.activityGraph>
    <UML:ActivityGraph xmi.idref="_01"/> <!-- zugeordneter Graph -->
  </UML:Partition.activityGraph>
  <UML:Partition.superPartition>
    <UML:Partition xmi.idref="_P01"/> <!-- übergeordnete Partition -->
  </UML:Partition.superPartition>
</UML:Partition>
<UML:Partition xmi.id="_P03" name="Partition A2">
  <UML:Partition.contents>
    <JUDE:ModelElement xmi.idref="_10"/> <!-- zugeordnete Elemente -->
    <!-- weitere -->
  </UML:Partition.contents>
  <UML:Partition.activityGraph>
    <UML:ActivityGraph xmi.idref="_01"/> <!-- zugeordneter Graph -->
  </UML:Partition.activityGraph>
  <UML:Partition.superPartition>
    <UML:Partition xmi.idref="_P01"/> <!-- übergeordnete Partition -->
  </UML:Partition.superPartition>
</UML:Partition>
<!-- ... -->
  <UML:ActivityGraph xmi.id="_01" name="Activity 0">
    <!-- ... -->
    <UML:ActivityGraph.partition>
      <UML:Partition xmi.id="PROCESS_1">
        <UML:Partition.activityGraph>
          <UML:ActivityGraph xmi.idref="_01"/> <!-- zugeordneter Graph -->
        </UML:Partition.activityGraph>
        <UML:Partition.subGroup>
          <JUDE:ModelElement xmi.idref="_P01"/> <!-- untergeordnete P. -->
        </UML:Partition.subGroup>
      </UML:Partition>
    </UML:ActivityGraph.partition>
  </UML:ActivityGraph>
```

Listing 4.50: Aktivitätsbereiche (verschachtelt) (vereinfachte XML-Struktur - UML AD)

nen die ID-Nummer der übergeordneten Partition eingetragen. Auch die Partition, die hier als „A“ geführt wird, hat eine solche übergeordnete Partition. Diese verweist auf das „UML:ActivityGraph.partition“-Element im Aktivitätsgraphen. Dort ist wiederum vermerkt, welches die untergeordnete Partition ist sowie in welchem Aktivitätsgraphen sich das Ganze abspielt.

Umgekehrt funktioniert die Transformation genauso: Hierbei wird auf das „UML:Partition“-Element im Aktivitätsgraphen geachtet. Darin ist vermerkt, wie die obere Ebene heißt. Diese wird verwendet, um den Pool abzuleiten. Die ID-Nummer wird also zur ID-Nummer des Pools („participant“-Element

im „collaboration“-Element). Das Namen-Attribut wird übernommen und das „processRef“-Element wird mit dem Wert aus dem „UML:Partition.superPartition“-Element befüllt. Im „UML:Partition.subGroup“-Element stehen quasi die Lanes vermerkt. Auch hier ergeben sich aus den ID-Nummern und den Namen entsprechend die ID-Nummer und der Name der jeweiligen Lane. Die enthaltenen Elemente werden aus dem „UML:Partition.contents“-Element in die „flowNodeRef“-Elemente übernommen.

Link-Ereignis und Konnektor

Ein weiteres vorgestelltes Notationselement war der Konnektor (UML Aktivitätsdiagramm). Dieses Element dient der Lesbarkeit des Diagramms und kann in das Link-Zwischenereignis (BPMN) überführt werden. Listing 4.51 und Listing 4.52 (auf der nächsten Seite) zeigen die XML-Strukturen dieser Verbindungselemente.

Bei den UML Aktivitätsdiagrammen ist ein Konnektor ein weiterer spezieller „ActionState“, der über einen Stereotypen („internal_connector“) erkannt wird. Da jedoch keine Referenz zwischen den Konnektoren besteht, können zusammengehörige Elemente nur über den Namen identifiziert werden, der demnach gleich sein muss. Diese Einschränkung fördert auch die Lesbarkeit des Diagramms.

```
<UML:ActivityGraph xmi.id="_10" name="Activity A">
  <!-- ... -->
  <UML:ActionState xmi.id="_10" name="No 1">
    <UML:ModelElement.stereotype>
      <UML:Stereotype xmi.idref="_S10"/>
    </UML:ModelElement.stereotype>
    <UML:StateVertex.incoming>
      <UML:Transition xmi.idref="_T10"/>
    </UML:StateVertex.incoming>
  </UML:ActionState>
  <UML:ActionState xmi.id="_11" name="No 1">
    <UML:ModelElement.stereotype>
      <UML:Stereotype xmi.idref="_S11"/>
    </UML:ModelElement.stereotype>
    <UML:StateVertex.outgoing>
      <UML:Transition xmi.idref="_T11"/>
    </UML:StateVertex.outgoing>
  </UML:ActionState>
  <!-- ... -->
</UML:ActivityGraph>
<UML:Stereotype xmi.id="_S10" name="internal_connector">
  <UML:Stereotype.baseClass>ActionState</UML:Stereotype.baseClass>
  <UML:Stereotype.extendedElement>
    <JUDE:ModelElement xmi.idref="_10"/>
  </UML:Stereotype.extendedElement>
</UML:Stereotype>
<UML:Stereotype xmi.id="_S11" name="internal_connector">
  <UML:Stereotype.baseClass>ActionState</UML:Stereotype.baseClass>
  <UML:Stereotype.extendedElement>
    <JUDE:ModelElement xmi.idref="_11"/>
  </UML:Stereotype.extendedElement>
</UML:Stereotype>
```

Listing 4.51: Konnektor (vereinfachte XML-Struktur - UML AD)

Bei der BPMN handelt es sich wie bereits erwähnt um Zwischenereignisse mit einer speziellen Definition. In diesem Fall handelt es sich um die „linkEventDefinition“. Im Listing 4.52 sind die Elemente „target“ und „source“ ergänzt worden, um die Zusammengehörigkeit darzustellen. Sind diese Elemente nicht vorhanden, so kann auch hier der Name der Zwischenereignisse als Referenz gewählt werden, um zu bestimmen, welche Verbindungselemente zusammengehören.

```
<intermediateThrowEvent id="_10" name="No 1">
  <incoming>_T10</incoming>
  <linkEventDefinition id="_10_ED_1" name="No 1"/>
    <target>_11</target>
  </linkEventDefinition>
</intermediateThrowEvent>
<intermediateCatchEvent id="_11" name="No 1">
  <outgoing>_T11</outgoing>
  <linkEventDefinition id="_11_ED_1" name="No 1">
    <source>_10</source>
  </linkEventDefinition>
</intermediateCatchEvent>
```

Listing 4.52: Link-Zwischenereignis (vereinfachte XML-Struktur - BPMN)

Die Transformation erfolgt nach folgenden Regeln: Die ID-Nummer des Konnektors wird zur ID-Nummer des Ereignisses. Das Namen-Attribut wird direkt übernommen sowie die eingehende oder ausgehende Kante. Das „target“-Element beziehungsweise das „source“-Element wird über den gemeinsamen Namen identifiziert. Die ID-Nummer der Ereignisdefinition ergibt sich aus der ID-Nummer plus einem Zusatz. Die Stereotypen geben den Hinweis darauf, ob zu einem werfenden oder einem fangendem Zwischenereignis transformiert werden muss. Das werfende Zwischenereignis ergibt sich aus dem sendenden Signal, das fangende Zwischenereignis aus dem empfangenden Signal.

In umgekehrter Richtung können die Ereignisdefinitionen vernachlässigt werden. Dafür müssen die Stereotypen der „ActionState“-Element entsprechend erzeugt und eingetragen werden.

Zeit-Ereignis und Zeitsignal

Bei den UML Aktivitätsdiagrammen gibt es das Zeitsignal. Das passende Äquivalent in der BPMN wäre ein Zeit-Ereignis. Da das Zeitsignal nicht zwingend eine eingehende Kante hat, kann es zum einen auf das Zeit-Startereignis, zum anderen auf das Zeit-Zwischenereignis transformiert werden. Letzteres ist genau dann der Fall, wenn es eine eingehende Kante gibt. Listing 4.53 (auf der nachfolgenden Seite) zeigt die XML-Struktur des Zeitsignals. Die Listings 4.54 und 4.55 (auf der nächsten Seite) zeigen die Strukturen für das Start- beziehungsweise Zwischenereignis der BPMN.

```
<UML:ActivityGraph xmi.id="_01" name="Activity A">
  <!-- ... -->
  <UML:ActionState xmi.id="_15" name="time signal">
    <UML:ModelElement.stereotype>
      <UML:Stereotype xmi.idref="_S15"/>
    </UML:ModelElement.stereotype>
    <UML:StateVertex.outgoing>
      <UML:Transition xmi.idref="_T15"/>
    </UML:StateVertex.outgoing>
    <UML:StateVertex.incoming>
      <!-- möglich -->
    </UML:StateVertex.incoming>
  </UML:ActionState>
  <!-- ... -->
</UML:ActivityGraph>
<UML:Stereotype xmi.id="_S15" name="accept_time_event">
  <UML:Stereotype.baseClass>ActionState</UML:Stereotype.baseClass>
  <UML:Stereotype.extendedElement>
    <JUDE:ModelElement xmi.idref="_15"/>
  </UML:Stereotype.extendedElement>
</UML:Stereotype>
```

Listing 4.53: Zeit-Signal (vereinfachte XML-Struktur - UML AD)

```
<startEvent id="_15" name="Start Event">
  <outgoing>_T15</outgoing>
  <timerEventDefinition id="_15_ED_1"/>
</startEvent>
```

Listing 4.54: Zeit-Startereignis (vereinfachte XML-Struktur - BPMN)

Mit der Transformation des Zeit-Startereignisses befasst sich allerdings bereits Baustein #1a. An dieser Stelle muss eine nachträgliche zweite Transformation erfolgen, welche das Zeit-Startereignis auf das Zeitsignal transformiert. Dafür werden alle Anmerkungen der Bausteine #1a durchgegangen und geprüft, ob eines die entsprechende Ereignisdefinition aufweist. In diesem Fall wird der Baustein zum Zeitsignal transformiert. Die ID-Nummer sowie der Name werden entsprechend übernommen, ebenso die ausgehende Kante. Der entsprechende Stereotyp wird erzeugt und eingetragen.

```
<intermediateCatchEvent id="_15" name="Intermediate Catch Event">
  <incoming>_T15</incoming>
  <outgoing>_T16</outgoing>
  <timerEventDefinition id="_15_ED_1"/>
</intermediateCatchEvent>
```

Listing 4.55: Zeit-Zwischenereignis (vereinfachte XML-Struktur - BPMN)

Im Falle eines werfenden Zwischenereignisses mit der entsprechenden Ereignisdefinition („timerEventDefinition“) wird eine Transformation zum Zeitsignal vollzogen. Die ID-Nummer des Zwischenereignisses wird zur ID-Nummer des „ActionState“-Elements. Der Name sowie die eingehenden und ausgehenden Kanten werden direkt übernommen. Der entsprechende Stereotyp wird erzeugt und eingetragen.

Verbleibende Zwischenereignisse und Signale

Bei den definierten Bausteinen wurde auf Seiten der UML Aktivitätsdiagramme mehrmals ein eingehendes Signal modelliert. Dabei wurde davon ausgegangen, dass dieses eingehende Signal von außerhalb des Diagramms stammt. Das Signal kann aber auch von innerhalb des Diagramms stammen. Die UML Aktivitätsdiagramme kennen nur eine Art sendendes Signal (Listing 4.56). Es handelt sich dabei um einen „ActionState“ mit Stereotyp („signal sending“).

```
<UML:ActivityGraph xmi.id="_01" name="Activity A">
  <!-- ... -->
  <UML:ActionState xmi.id="_20" name="Send Signal">
    <UML:ModelElement.stereotype>
      <UML:Stereotype xmi.idref="_S20"/>
    </UML:ModelElement.stereotype>
    <UML:StateVertex.incoming>
      <UML:Transition xmi.idref="_T20"/>
    </UML:StateVertex.incoming>
  </UML:ActionState>
  <!-- ... -->
</UML:ActivityGraph>
<UML:Stereotype xmi.id="_S20" name="signal sending">
  <UML:Stereotype.baseClass>ActionState</UML:Stereotype.baseClass>
  <UML:Stereotype.extendedElement>
    <JUDE:ModelElement xmi.idref="_20"/>
  </UML:Stereotype.extendedElement>
</UML:Stereotype>
```

Listing 4.56: Signal senden (vereinfachte XML-Struktur - BPMN)

Dahingegen können in der BPMN eine ganze Reihe sendender (werfender) Signale (Ereignisse) definiert werden. Die verschiedenen Arten von Start- und Endereignissen werden bereits in den Bausteinen #1a, #1b und #1c behandelt. Damit verbleiben die Zwischenereignisse. In der einfachsten Form gibt es das Blanko-Zwischenereignis zum Senden (Listing 4.57).

```
<intermediateThrowEvent id="_20" name="Send Signal">
  <incoming>_T20</incoming>
</intermediateThrowEvent>
```

Listing 4.57: Blanko-Zwischenereignis (vereinfachte XML-Struktur - BPMN)

Allgemein können die verbleibenden speziellen werfenden Zwischenereignisse („intermediateThrowEvent“) auf das Sendesignal und die speziellen fangenden Zwischenereignisse („intermediateCatchEvent“) auf das Empfangssignal transformatiert werden. Dabei gilt: Die ID-Nummer des Zwischenereignisses wird zur ID-Nummer des Signals. Der Name wird direkt übernommen sowie die eingehenden und ausgehenden Kanten. Die Ereignisdefinition wird wie schon bei den Start- und Endereignissen als eindeutige Anmerkung mitgeführt. Der Stereotyp wird entsprechend erzeugt und eingetragen. In die andere Richtung gehend wird aus der Anmerkung die entsprechende Ereignisdefinition abgeleitet.

Datenobjekt und Objektknoten

Ein weiteres wichtiges Element bei den Aktivitätsdiagrammen der UML ist der Objektknoten. Dieser kann durch das einfache Datenobjekt der BPMN abgebildet werden. Umgekehrt funktioniert diese Transformation allerdings nur eingeschränkt, denn die BPMN kennt verschiedene Elemente für Datenobjekte. So ein Datenobjekt kann nur transformiert werden, wenn es bei der einen Aufgabe als Ausgabe geführt wird und beim nachfolgenden Element (ausgenommen der Sequenzfluss) als Eingabe. In Abbildung 4.39 ist dies graphisch dargestellt.

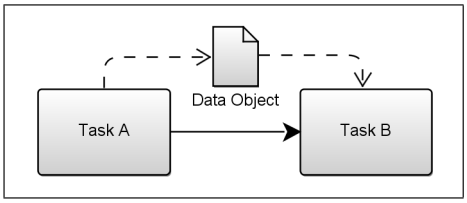


Abbildung 4.39: Datenobjekt (graphisch)

```
<dataObject id="DO_PROCESS_1_1" name="Data Object"/>
<dataObjectReference dataObjectRef="DO_PROCESS_1_1" id="_5"/>
<!-- ... -->
<task id="_2" name="Task A">
  <incoming>_8</incoming>
  <outgoing>_4</outgoing>
  <ioSpecification>
    <dataOutput id="Dout_2_5"/>
    <outputSet>
      <dataOutputRefs>Dout_2_5</dataOutputRefs>
    </outputSet>
  </ioSpecification>
  <dataOutputAssociation id="_6">
    <sourceRef>Dout_2_5</sourceRef>
    <targetRef>_5</targetRef>
  </dataOutputAssociation>
</task>
<task id="_3" name="Task B">
  <incoming>_4</incoming>
  <outgoing>_12</outgoing>
  <ioSpecification>
    <dataInput id="Din_3_5"/>
    <inputSet>
      <dataInputRefs>Din_3_5</dataInputRefs>
    </inputSet>
  </ioSpecification>
  <dataInputAssociation id="_7">
    <sourceRef>_5</sourceRef>
    <targetRef>Din_3_5</targetRef>
  </dataInputAssociation>
</task>
```

Listing 4.58: Datenobjekt (vereinfachte XML-Struktur - BPMN)

Listing 4.58 zeigt die dazugehörige XML-Struktur in der BPMN. Das Datenobjekt erhält ein eigenes Element sowie eine Datenobjektreferenz. Die Aufgaben, an die das Datenobjekt gebunden ist, werden um die Elemente „ioSpecification“ sowie „dataOutputAssociation“ beziehungsweise „dataInputAssociation“ mit den jeweiligen

Kind-Elementen erweitert.

```
<UML:ObjectFlowState xmi.id="_5" name="Data Object">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_New1"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_New2"/>
  </UML:StateVertex.incoming>
</UML:ObjectFlowState>
```

Listing 4.59: Objektknoten (vereinfachte XML-Struktur - UML AD)

Auch bei den Aktivitätsdiagrammen der UML handelt es sich beim Objektknoten um ein eigenständiges Element. Darin werden die eingehende Kante sowie die ausgehende Kante gespeichert, sodass es möglich wird, die vorgelagerte und die nachgelagerte Aktion zu erkennen. An dieser Stelle ist die XML-Struktur der UML Aktivitätsdiagramme deutlich einfacher als die der BPMN.

Die Transformation zwischen diesen Elementen erfolgt nun nach den nachstehenden Regeln: Wird von der BPMN zur den UML Aktivitätsdiagrammen transformiert, so wird die ID-Nummer des Datenobjekts zur ID-Nummer des Objektknoten. Das Namen-Attribut wird direkt übernommen. Die eingehende und ausgehende Kante zu ermitteln wird schwieriger, da in der BPMN bereits ein Sequenzfluss zwischen den beiden Aufgaben besteht. Nach der Transformation muss das Datenobjekt jedoch als Objektknoten zwischen diesen beiden Einheiten stehen. Das bedeutet, dass ein Kontrollfluss verändert und ein zusätzlicher eingefügt werden muss. Alternativ kann der betreffende Kontrollfluss entfernt und zwei neue können hinzugefügt werden. Die letztgenannte Methode ist einfacher. Die neuen Kontrollflüsse ergeben sich wie folgt: Der eine erhält als Quell-ID-Nummer die ID-Nummer der Aufgabe, welche das Datenobjekt als Output führt. Die Ziel-ID-Nummer gehört zum Datenobjekt selbst. Der zweite Kontrollfluss nimmt als Quell-ID-Nummer die ID-Nummer des Datenobjekts und als Ziel-ID-Nummer die ID-Nummer der Aufgabe, welche das Datenobjekt als Input aufführt. Der Sequenzfluss, der entfernt werden muss, hat als Quell- und Ziel-ID-Nummern die ID-Nummern der beiden Aufgaben.

Bei der Transformation von einem UML Aktivitätsdiagramm zur BPMN werden folgende Schritte gegangen: Es wird ein Datenobjekt erstellt, welches den Namen des Objektknoten hat. Zusätzlich wird eine Referenz erstellt, welche die ID-Nummer des Objektknoten verwendet. Datenobjekt und Referenz sind über Attribute verbunden. Da ein Datenobjekt sowohl eine eingehende als auch eine ausgehende Kante hat, muss jetzt noch ein Kontrollfluss entfernt und einer verändert werden. Die beiden entsprechenden Transitionen können dadurch identifiziert werden, dass sie den Objektknoten als Quell- oder Ziel-ID-Nummer haben. Der neue Kontrollfluss verwendet daher von dem einen Kontrollfluss die Quell-ID-Nummer, von dem anderen die Ziel-ID-Nummer (jeweils nicht die ID-Nummer, die vom beziehungsweise zum Objektknoten kommt / führt). Die Aufgaben erhalten eine Ergänzung um die Elemente „dataOutputAssociation“ beziehungsweise „dataInputAssociation“ sowie

„ioSpecification“ inklusive der Kind-Elemente.

Resümee

Tabelle 4.12 zeigt die soeben definierten Regeln noch einmal in der Zusammenfassung. Auf der linken Seite steht das zu transformierende Element, auf der rechten Seite steht das Element, zu dem es transformiert wird.

BPMN	UML AD
Pool	Aktivitätsbereich
Lane	Aktivitätsbereich
Link-Zwischenereignis (werfend)	Konnektor (mit Eingang)
Link-Zwischenereignis (fangend)	Konnektor (mit Ausgang)
Zeit-Startereignis	Zeitsignal (aus Baustein #1a ableiten)
Zeit-Zwischenereignis	Zeitsignal
restliche Zwischenereignisse (werfend)	Sendesignal (ggf. mit Anmerkung)
restliche Zwischenereignisse (fangend)	Empfangssignal (ggf. mit Anmerkung)
Datenobjekt	Objektknoten
UML AD	BPMN
Aktivitätsbereich (1 Ebene)	Pool
Aktivitätsbereich (2+ Ebenen)	Pool und Lane-Set
Konnektor (mit Eingang)	Link-Zwischenereignis (werfend)
Konnektor (mit Ausgang)	Link-Zwischenereignis (fangend)
Zeitsignal (mit Eingang)	Zeit-Zwischenereignis
Zeitsignal (ohne Eingang)	Zeit-Startereignis
Sendesignal (ggf. mit Anmerkung)	Zwischenereignis (werfend)
Empfangssignal (ggf. mit Anmerkung)	Zwischenereignis (fangend)
Objektknoten	Datenobjekt

Tabelle 4.12: Zusatzregeln für Transformation

Die hier eingeführten Regeln ermöglichen die Abdeckung weiterer Elemente in den Notationen der beiden Modellsprachen. Die Tabelle 4.13 fasst zusammen, welche weiteren Elemente der BPMN und der UML Aktivitätsdiagramme nun abgedeckt werden. Zwischen der linken und der rechten Spalte existiert keine Beziehung.

BPMN	UML AD
Pool	Aktivitätsbereich
Lane	Konnektor
Zwischenereignis (werfend) (alle Typen)	Zeitsignal
Zwischenereignis (fangend) (alle Typen)	Sendesignal
Datenobjekt	Empfangssignal
	Objektknoten

Tabelle 4.13: Abdeckung der Notationselemente durch weitere Regeln

Die Tabellen 4.14 und 4.15 hingegen zeigen, welche Notationselemente weiterhin nicht abgedeckt sind. Einige von diesen können allerdings unter Verwendung anderer abgedeckter Elemente imitiert werden.

Element	Abdeckung
Angeheftete Ereignisse	nein (nur Fehler-Ereignis)
Nicht-unterbrechende Ereignisse	nein (besondere Form der angehefteten Ereignisse oder Teil der Ererignisbasierten Sub-Prozesse)
Globale Aufgabe	nein
Multi-Instanz	nein (nicht betrachtet)
Schleife	nein (durch Gateways imitierbar)
Kompensation	nein
Standardfluss (Defaultfluss)	nein (durch Gateways imitierbar)
Bedingter Sequenzfluss	nein (durch Gateways imitierbar)
Komplexes Gateway	nein
Exklusives ereignis-basiertes Gateway (Instanziierung)	nein (durch mehrere Startereignisse imitierbar)
Paralleles ereignis-basiertes Gateway (Instanziierung)	nein (durch mehrere Startereignisse und paralleles Gateway imitierbar)
Ad-hoc-Sub-Prozess	nein (kann imitiert werden)
Transaktions-Sub-Prozess	nein
Ereignisbasierter Sub-Prozess	nein
Nachricht	nein (Element des Kollaborationsdiagramms)
Nachrichtenfluss	nein (Element des Kollaborationsdiagramms)
Gruppe	nein (ohne Einfluss auf Diagramm)

Tabelle 4.14: Nicht abgedeckte Elemente (BPMN)

Element	Abdeckung
Pin	nein (durch Objektknoten imitierbar)
Schleifen- und Bedingungsknoten	nein (durch Entscheidungs- und Verbindungsknoten imitierbar)

Tabelle 4.15: Nicht abgedeckte Elemente (UML AD)

Es wird deutlich, dass alle Basiselemente der BPMN abgedeckt werden, ebenso ein Großteil der erweiterten Elemente. Nur sieben Elemente werden gar nicht abgedeckt, sechs weitere können durch die Verwendung anderer Notationselemente imitiert werden. Zwei Elemente sind dem Kollaborationsdiagramm der BPMN zuzuordnen und eines bleibt ohne jeden Einfluss auf das Diagramm selbst (Gruppe). Bei genauerer Betrachtung der abgedeckten Elemente ist erkennbar, dass es sich um die handelt, mit denen vermutlich der größte Teil in einem Diagramm modelliert wird (Ereignisse, Gateways, Aufgaben und Sub-Prozesse etc.).

Bei den UML Aktivitätsdiagrammen wird eine beinah vollständige Abdeckung erreicht (18 von 20 Notationselementen). Nur zwei der vorgestellten Elemente lassen sich nicht transformieren, allerdings durch andere abgedeckte Elemente darstellen.

Die beschriebenen Einschränkungen müssen akzeptiert werden, weil sich BPMN und UML Aktivitätsdiagramme zwar ähnlich, aber nicht gleich sind. Das liegt zum einen an den unterschiedlichen Anwendungsgebieten, für die sie konzipiert worden sind, zum anderen an der Komplexität der BPMN. Dennoch ist eine weitgehende M2M-Transformation möglich.

4.6 Algorithmus zur Transformation

In den vorangegangenen Abschnitten wurden alle Bausteine und zusätzlichen Regeln für die Transformation zwischen BPMN und UML Aktivitätsdiagrammen und umgekehrt definiert. Es fehlt jedoch noch eine konkrete Ausführungsreihenfolge aller Schritte.

Zunächst einmal gibt es für jeden Baustein eine eigene Prozedur. Da sich der entwickelte Pseudo-Code an der englischen Sprache orientiert, heißt die entsprechende Prozedur zum Beispiel „brick #12“[144] und nimmt dabei Bezug auf Baustein #12 (analog dazu alle weiteren Prozeduren). Die einzelnen Schritte in den Prozeduren orientieren sich an den in den vorangegangenen Abschnitten erläuterten Regeln. So wird in der Prozedur „brick #12“ für die Richtung BPMN zu UML Aktivitätsdiagramm beispielsweise nach dem Indikatorelement „boundaryEvent“ gesucht, welches eine „errorEventDefinition“ beinhaltet und an ein „task“-Element gebunden ist. Listing 4.60 zeigt diese Schritte im Groben.

```
1  procedure (brick #12)
2    repeat until end of file
3      search indicator-element
4         > "boundaryEvent"
5      identify associated elements and attributes
6         > "errorEventDefinition"-element
7         > "attachedToRef=ID"
8         > ID belongs to "task"-element
9      deviate BPMN to brick structure
10     mark used elements as used
11 end procedure (brick #12)
```

Listing 4.60: Pseudo-Code: Ableitung Baustein #12 (BPMN zu Baustein)

Die Prozedur wird für die gesamte XML-Struktur durchführt. Auf die Erstellung eines Pseudo-Code für jede weitere der Prozeduren zur Identifizierung eines Bausteins wird hier verzichtet. In der Prozedur für Baustein #4 würde aber beispielsweise nach dem „parallelGateway“-Element mit dem „gatewayDirection“-Attributswert „diverging“ gesucht werden. Diese Informationen können den vorangegangenen Abschnitten entnommen werden.

[144] brick, zu deutsch: Baustein

Listing 4.61 zeigt den Pseudo-Code für die Transformation von der BPMN zu den UML Aktivitätsdiagrammen. In der Hauptprozedur („main“) wird zunächst die XML-Datei eingelesen, danach müssen die Bausteine in einer bestimmten Reihenfolge identifiziert werden. Dies geschieht in einer untergeordneten Prozedur („brick-identifier“). Nachdem alle Bausteine identifiziert wurden, gibt es möglicherweise noch Elemente, die mit den Zusatzregeln transformiert werden müssen. Dies geschieht in der Prozedur „extra-transformation“. Im Anschluss werden die Bausteine #3 identifiziert. Zum Schluss wird die XML-Struktur für das UML Aktivitätsdiagramm aus den Bausteinen abgeleitet. Auch hier wird sich an den Erläuterungen aus den vorangegangenen Abschnitten orientiert, weshalb keine Pseudo-Codes dafür entwickelt wurden.

```
1  procedure (main)
2    read xml-file
3    call procedure (brick-identifier)
4    call procedure (extra-transformation)
5    call procedure (brick #3)
6    call procedure (deviate-new-xml)
7  end procedure (main)

9  procedure (brick-identifier)
10   call procedure (brick #12)
11   call procedure (brick #11)
12   call procedure (brick #10)
13   call procedure (brick #9b)
14   call procedure (brick #7b)
15   call procedure (brick #5b)
16   call procedure (brick #9a)
17   call procedure (brick #7a)
18   call procedure (brick #5a)
19   call procedure (brick #8)
20   call procedure (brick #6)
21   call procedure (brick #4)
22   call procedure (brick #2b)
23   call procedure (brick #2a)
24   call procedure (brick #1c)
25   call procedure (brick #1b)
26   call procedure (brick #1a)
27 end procedure (brick-identifier)

29 procedure (extra-transformation)
30   for each rule
31     search indicator-element
32     identify associated elements and attributes
33     transform to UML AD structure
34 end procedure (extra-transformation)

36 procedure (deviate-new-xml)
37   deviate bricks to UML AD structure
38 end procedure (deviate-new-xml)
```

Listing 4.61: Pseudo-Code: BPMN zu UML AD

In die andere Richtung funktioniert diese Vorgehensweise ebenso. Die Prozeduren für die Identifizierung sehen entsprechend anders aus, da nun nach den Indikatorelementen der UML Aktivitätsdiagramme gesucht wird. Dementsprechend verändert sich der Inhalt von Prozedur „brick #12“. Dort ist jetzt das Indikatorelement ein Unterbrechnungsbereich mit genau zwei „ActionState“-Elementen,

von denen eines den Stereotyp „signal receipt“ hat. Dies ist in Listing 4.62 abgebildet.

```
procedure (brick #12)
  repeat until end of file
    search indicator-element
       > "InterruptibleActivityRegion"
    identify associated elements and attributes
       > "ActionState"-element with stereotype "signal receipt"
       > "ActionState"-element
    deviate UML AD to brick structure
    mark used elements as used
end procedure (brick #12)
```

Listing 4.62: Pseudo-Code: Ableitung Baustein #12 (UML AD zu Baustein)

Wie schon zuvor wird die Prozedur für die gesamte XML-Struktur wiederholt. Auch hier wird auf die Erstellung weiterer Pseudo-Codes für die anderen Prozeduren verzichtet. Für Baustein #4 beispielsweise würde aber an dieser Stelle nach einem „ActionState“-Element gesucht werden, bei dem das „kind“-Attribut den Wert „fork“ hat.

Der Gesamtablauf unterscheidet sich ansonsten nicht von dem zuvor. Es wird abschließend lediglich der Baustein in die XML-Struktur der BPMN überführt und nicht in die der UML Aktivitätsdiagramme.

```
procedure (main)
  read xml-file
  call procedure (brick-identifier)
  call procedure (extra-transformation)
  call procedure (brick #3)
  call procedure (deviate-new-xml)
end procedure (main)

procedure (brick-identifier)
  call procedure (brick #12)
  ...
end procedure (brick-identifier)

procedure (extra-transformation)
  for each rule
    search indicator-element
    identify associated elements and attributes
    transform to BPMN structure
end procedure (extra-transformation)

procedure (deviate-new-xml)
  deviate bricks to BPMN
end procedure (deviate-new-xml)
```

Listing 4.63: Pseudo-Code: UML AD zu BPMN

Dieser Reihenfolge folgend, kann die Transformation zwischen BPMN und UML Aktivitätsdiagrammen und umgekehrt vorgenommen werden. Baustein #3 wird dabei je nach den Zusatzregeln gesucht, weil die Zusatzregeln für die Transformation von Datenobjekten und Objektknoten den Ablauf im Diagramm beeinflussen.

4.7 Fazit

Eine 1-zu-1-Transformation zwischen der BPMN und den UML Aktivitätsdiagrammen ist schwierig und scheitert an Uneindeutigkeiten, die aufgrund der Komplexität der Modellsprachen und fehlenden Äquivalenten auftreten. Aus diesem Grund ist eine andere Methode erstrebenswert.

Da im vorangegangenen Kapitel festgestellt worden ist, dass die Kontrollflussmuster des Workflow Pattern Framework fast alle von beiden Modellsprachen unterstützt werden, wurde ihre genaue graphische Darstellung untersucht. Bei den direkt unterstützten Kontrollflussmustern hat sich ergeben, dass die meisten dieser Darstellungen eindeutig sind. Dies war lediglich bei zwei Kontrollflussmustern nicht der Fall. Bei diesen gab es zum einen eine volle Übereinstimmung zu einem anderen Kontrollflussmuster, zum anderen eine Teilübereinstimmung zu einem anderen Kontrollflussmuster. Für beide wurden Lösungen ersonnen, um die Verwendung dennoch zu ermöglichen. Bei den nicht direkt unterstützten Mustern wurde festgestellt, dass sich zwei nicht eindeutig mit den Mitteln der BPMN und den UML Aktivitätsdiagrammen modellieren lassen. Ein weiteres kann nur mit Einschränkungen verwendet werden.

Weiterhin wurden zusätzliche wichtige Elemente wie ein Muster für den Prozessstart sowie eine atomare und eine nicht-atomare Einheit definiert, welche bei den Kontrollflussmustern des Workflow Pattern Framework keine Rolle spielen.

Auf Grundlage dieser gewonnen Erkenntnisse wurden 20 Bausteine definiert, über welche eine M2M-Transformation zwischen den beiden Modellsprachen ermöglicht wird. Dabei dient das Baustein-Prinzip als eine Art Meta-Ebene. Um eine XML-basierte Transformation zu ermöglichen, sind diese Bausteine in einer XSD definiert worden. Im Anschluss daran wurde aufgezeigt, nach welchen XML-Strukturen bei der BPMN und den UML Aktivitätsdiagrammen gesucht werden muss, um die Bausteine abzuleiten, und welche XML-Strukturen abgeleitet werden müssen, wenn ein Baustein vorliegt. Dies geschah am direkten Beispiel an den XML-Exporten der Modellierungswerkzeuge Yaoqiang BPMN Editor beziehungsweise Astah Professional.

Um die Menge der abgedeckten Elemente nicht zu sehr einzuschränken, wurden zum einen spezielle Anmerkungen definiert, die wichtige Informationen speichern, zum anderen wurden Zusatzregeln definiert. Die speziellen Anmerkungen sind deutlich an ihrer Einleitung „SPEC“ zu erkennen und werden während des Transformationsprozesses beachtet. Durch die Zusatzregeln werden weitere Notationselemente abgedeckt, die bei den Kontrollflussmustern und demnach bei den definierten Bausteinen keine Beachtung finden, jedoch direkte Äquivalente aufweisen.

Auf diese Weise werden die meisten dem Prozessdiagramm zugeordneten Notationselemente transformiert. Es gibt jedoch auch eine ganze Reihe von Elementen, die nicht durch die Bausteine oder die zusätzlichen Regeln abgedeckt sind. Diese können großteils durch andere, abgedeckte Elemente imitiert werden. So verbleiben

nur sechs der betrachteten Elemente ohne irgendeine Transformation.

Im Anschluss an die Definition der Bausteine und der zusätzlichen Regeln wurde eine Ausführungsreihenfolge der einzelnen Schritte der M2M-Transformation erarbeitet. Diese Reihenfolge ist notwendig, um zum einen die Bausteine, zum anderen die übrig gebliebenen Elemente korrekt identifizieren zu können und dann entsprechend zu transformieren. Die Ausführungsreihenfolge sowie die groben Schritte wurden als Pseudo-Codes verfasst. Dabei wurde auch beispielhaft Pseudo-Code für die Identifizierung eines Bausteins geschrieben.

5 Verifizierung und Validierung

In diesem Kapitel wird gezeigt, dass der erarbeitete Transformationsweg sowie die zusätzlichen Regeln funktionieren, um ein UML Aktivitätsdiagramm in ein BPMN Prozessdiagramm zu transformieren und umgekehrt. Dass die Methodik funktioniert, soll anhand zweier graphischer Beispiele gezeigt werden. Dazu wird das Vorgehen sowohl verifiziert als auch validiert. Am Ende erfolgt eine kurze kritische Betrachtung des Ergebnisses.

5.1 Begriffserklärung Verifizierung und Validierung

Es stellt sich die Frage, was Verifizierung und Validierung sind und beinhalten, damit beide genutzt werden können, um den erarbeiteten Transformationsweg zu verifizieren und validieren. Nach Oberkampf und Trucano[145] ist Verifizierung „das Verfahren zur Beurteilung der Produkte der Software-Entwicklungsphase um Zusicherung zu leisten, dass sie die Anforderungen, die für sie in der vorherigen Phase definitiert wurden, erfüllen“. Dahingegen ist die Validierung „das Verfahren zum Testen eines Computerprogramms und der Auswertung der Ergebnisse, um die Einhaltung spezifischer Anforderungen zu gewährleisten“. Diese Definitionen gehen auf das Institute of Electrical and Electronics Engineers (IEEE) zurück.

In Kurzform nach Boehm bedeutet dies:

- „Verifizierung: Baue ich das System richtig?
- Validierung: Baue ich das richtige System?“[146]

In dieser Arbeit wurde ein Transformationsweg erarbeitet, mit dem eine M2M-Transformation zwischen der BPMN und den UML Aktivitätsdiagrammen über ein Baustein-Prinzip erreicht werden soll. Dafür wurden zum einen die Bausteine für das Baustein-Prinzip definiert, zum anderen Zusatzregeln für nicht erfasste Notationselemente der beiden Modellsprachen. Abschließend wurde eine Ausführungsreihenfolge festgelegt.

Aus diesem Grund wird im Verifizierungsprozess geprüft, ob die Bausteine und die Zusatzregeln sich eignen, die Transformation zu ermöglichen, und ob die definierte Ausführungsreihenfolge zum gewünschten Ergebnis führt. Das bedeutet zum einen, dass überprüft wird, ob die Bausteine im Diagramm identifiziert werden können,

[145] Vgl. Oberkampf und Trucano [2002], S. 214
[146] Vgl. Juristo [2005], S. 1

und zum anderen, ob die zusätzlichen Regeln alle übriggebliebenen Notationselemente abdecken. Im Validierungsprozess wird geprüft, ob das erhaltene Ergebnis den Ansprüchen der M2M-Transformation zwischen den Modellsprachen BPMN und den UML Aktivitätsdiagrammen genügt. Das heißt zum einen, dass die Transformation vollständig erfolgt, also alle verwendeten Notationselemente transformiert werden. Zum anderen muss das erhaltene Diagramm den gleichen Sachverhalt wie das ursprüngliche Diagramm modellieren. Desweiteren müssen die Grundsätze ordnungsgemäßer Modellierung (GoM) nach der Transformation noch erfüllt sein.

5.2 Beispieltransformation von BPMN zum UML Aktivitätsdiagramm

Abbildung 5.1 (auf der nächsten Seite) zeigt die Arbeit in einer Küche, in der verschiedene Gänge zubereitet werden, sobald ein Gast eingetroffen ist. Die Arbeiten an Vorspeise, Hauptgang und Nachspeise beginnen zeitgleich und werden zum Schluss wieder zusammengefasst. Danach kann sich der Koch wieder entspannen und der Prozess ist beendet.

Die Zubereitung der einzelnen Speisen erfolgt je als eigener Sub-Prozess. Als Vorspeise werden ein Salat und / oder eine andere Vorspeise zubereitet, sodass der Gast mindestens eine, aber maximal zwei der drei möglichen Vorspeisen erhält. Der Hauptgang ist abhängig von externen Faktoren. So wird beispielsweise das Überraschungsgericht zubereitet, wenn der Gast Geburtstag hat. Sollte er Veganer sein, so wird ein veganes Gericht vorbereitet, andernfalls wird das normale Hauptgericht zubereitet. Da der Koch zwei Lehrlinge hat, überlässt er diesen die Zubereitung der Nachspeise. Sie stellen zeitgleich Mousse-au-Chocolat sowie einen Bananensplit her. Dem Gast wird die Nachspeise serviert, die zuerst fertiggestellt wird.

Damit dem Gast allerdings nicht alle drei Gänge zur selben Zeit serviert werden, wird der Hauptgang erst serviert, wenn die Vorspeise beendet wurde, und das Dessert erst dann, wenn der Hauptgang beendet wurde.

Das Diagramm ist nach den Grundsätzen ordnungsgemäßer Modellierung (GoM) erstellt worden, das heißt, es wurde syntaktisch korrekt nach den Regeln der BPMN und den in dieser Arbeit definierten Beschränkungen modelliert. Der dargestellte Sachverhalt wird als korrekt betrachtet, da es sich lediglich um ein veranschaulichendes Beispiel handelt. Es wurden nur Elemente verwendet, die für die Beispieltransformation relevant sind, das heißt, sie stellen in dieser Arbeit definierte Bausteine oder Elemente mit direktem Äquivalent in der Ziel-Modellsprache dar. Dadurch ist auch der Grundsatz der Wirtschaftlichkeit erfüllt. Das Diagramm folgt einer klaren Strukturierung, ist dadurch einfach zu lesen und erfüllt somit den Grundsatz der Klarheit. Des Weiteren wurde das Diagramm systematisch aufgebaut. Die eindeutigen Bezeichnungen der Elemente

würden eine weitere Verwendung in anderen Sichten ermöglichen. Ob der Grundsatz der Vergleichbarkeit erfüllt ist, wird sich direkt nach der Transformation zeigen.

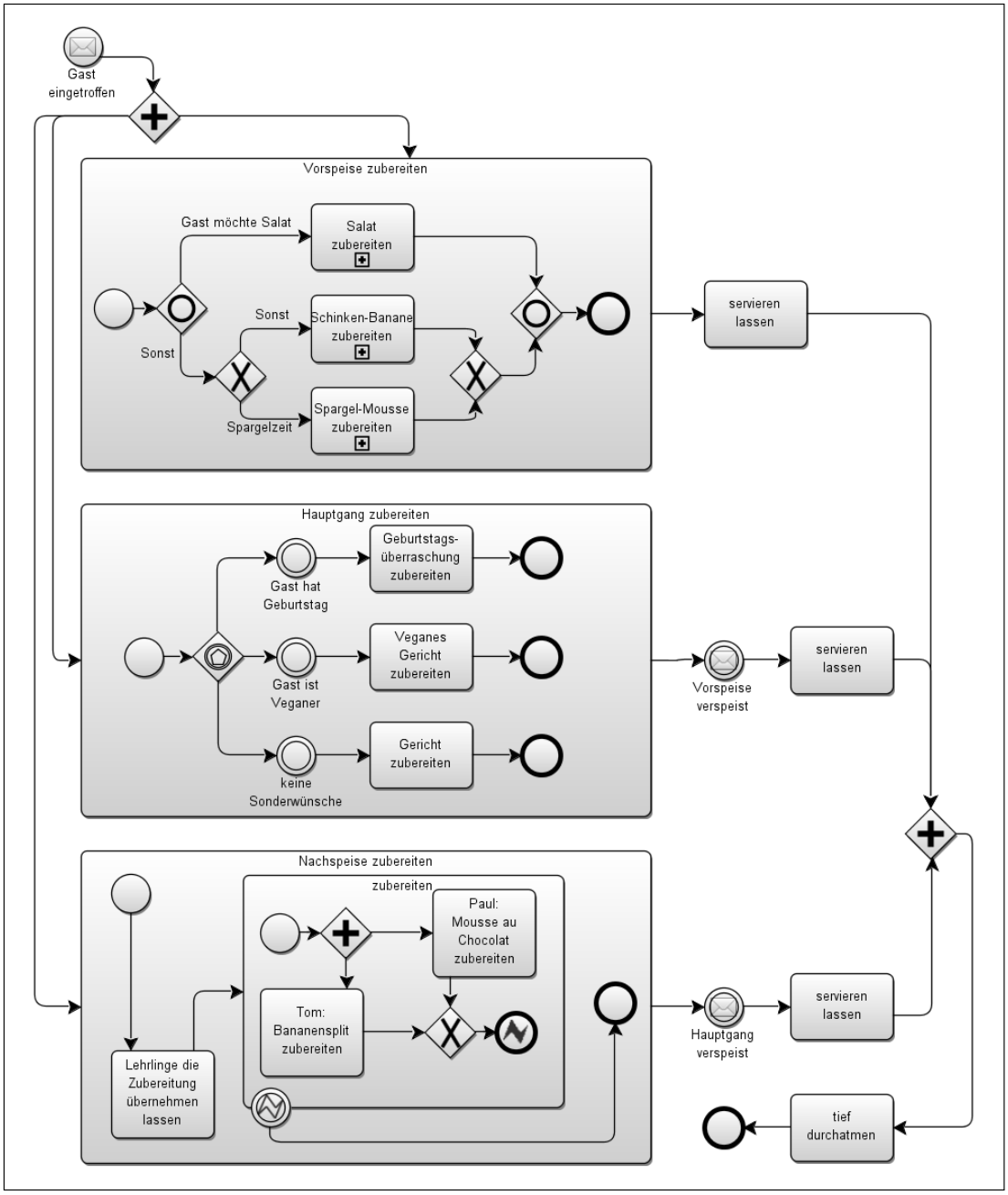


Abbildung 5.1: Diagramm in BPMN

Transformation

In Abbildung 5.2 (auf der nachfolgenden Seite) sind alle identifizierten Bausteine graphisch hervorgehoben worden. Die abgehandelten Notationselemente wurden schwächer im Hintergrund dargestellt. Auch die Verbindungslinien sind aufgrund der besseren Lesbarkeit hier abgeschwächt dargestellt. Sie wurden jedoch noch

nicht transformiert. In dem abgebildeten Diagramm sind 15 (exklusive Baustein #3) der 20 definierten Bausteine vertreten. Bis auf zwei Zwischenereignisse („Vorspeise verspeist“ und „Hauptgang verspeist“) können alle Elemente in diesem Diagramm direkt durch die definierten Bausteine transformiert werden. Die beiden Zwischenereignisse werden im nächsten Transformationsschritt durch die Zusatzregeln umgewandelt.

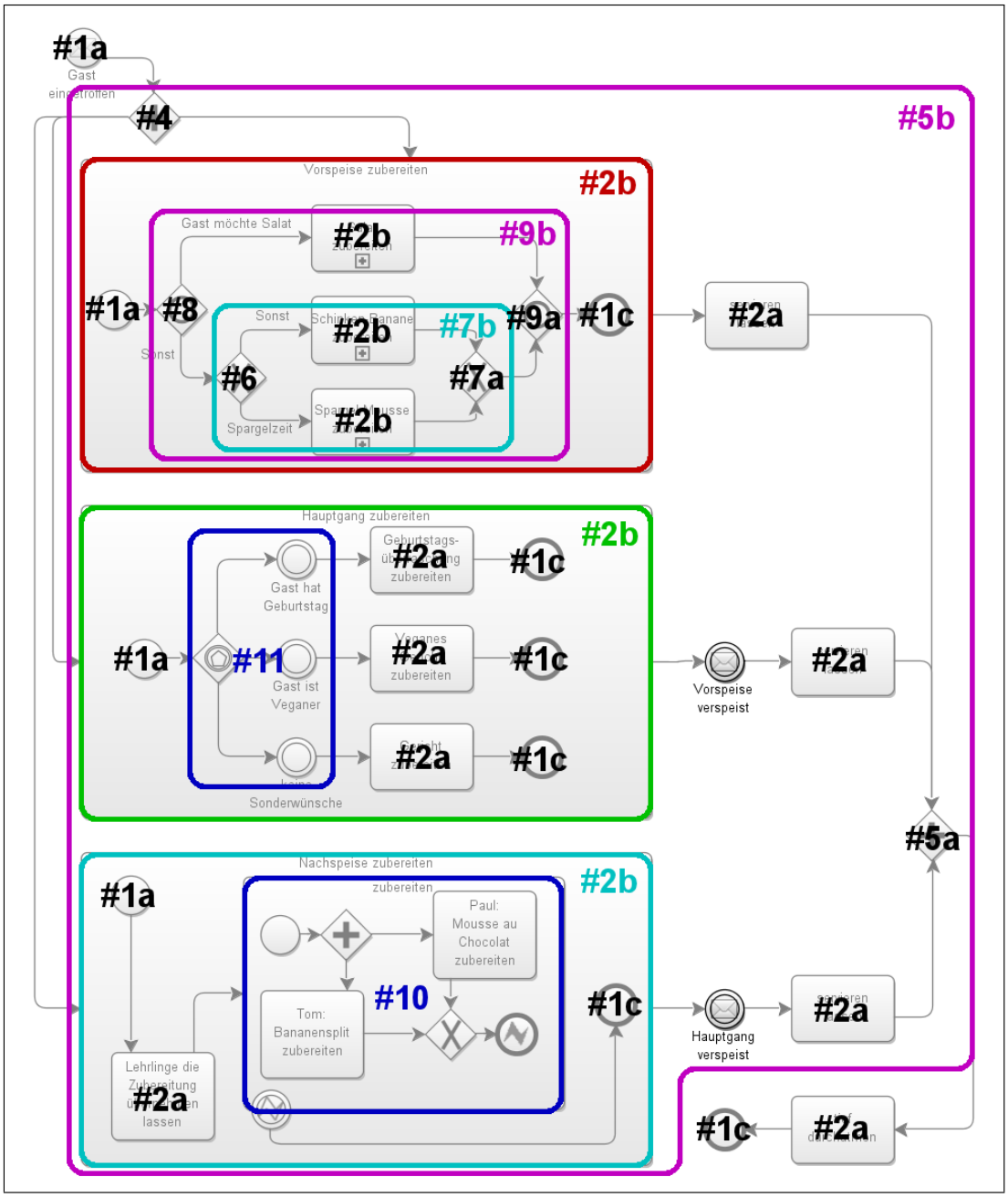


Abbildung 5.2: Diagramm in BPMN - Bausteinidentifizierung

Nach der Anwendung der zusätzlichen Regeln (hier für die beiden Zwischenereignisse) werden noch die Sequenzflüsse (Baustein #3) transformiert. Abbildung 5.3 zeigt das transformierte Diagramm. Deutlich ist auch Baustein #9a zu erkennen, der so eigentlich nicht im UML Aktivitätsdiagramm auftreten kann. Durch die spezielle Anmerkung ist das Element jedoch graphisch darstellbar. Auch die Ereignisdefinitionen der anderen Ereignisse sind als Anmerkungen deutlich zu erkennen.

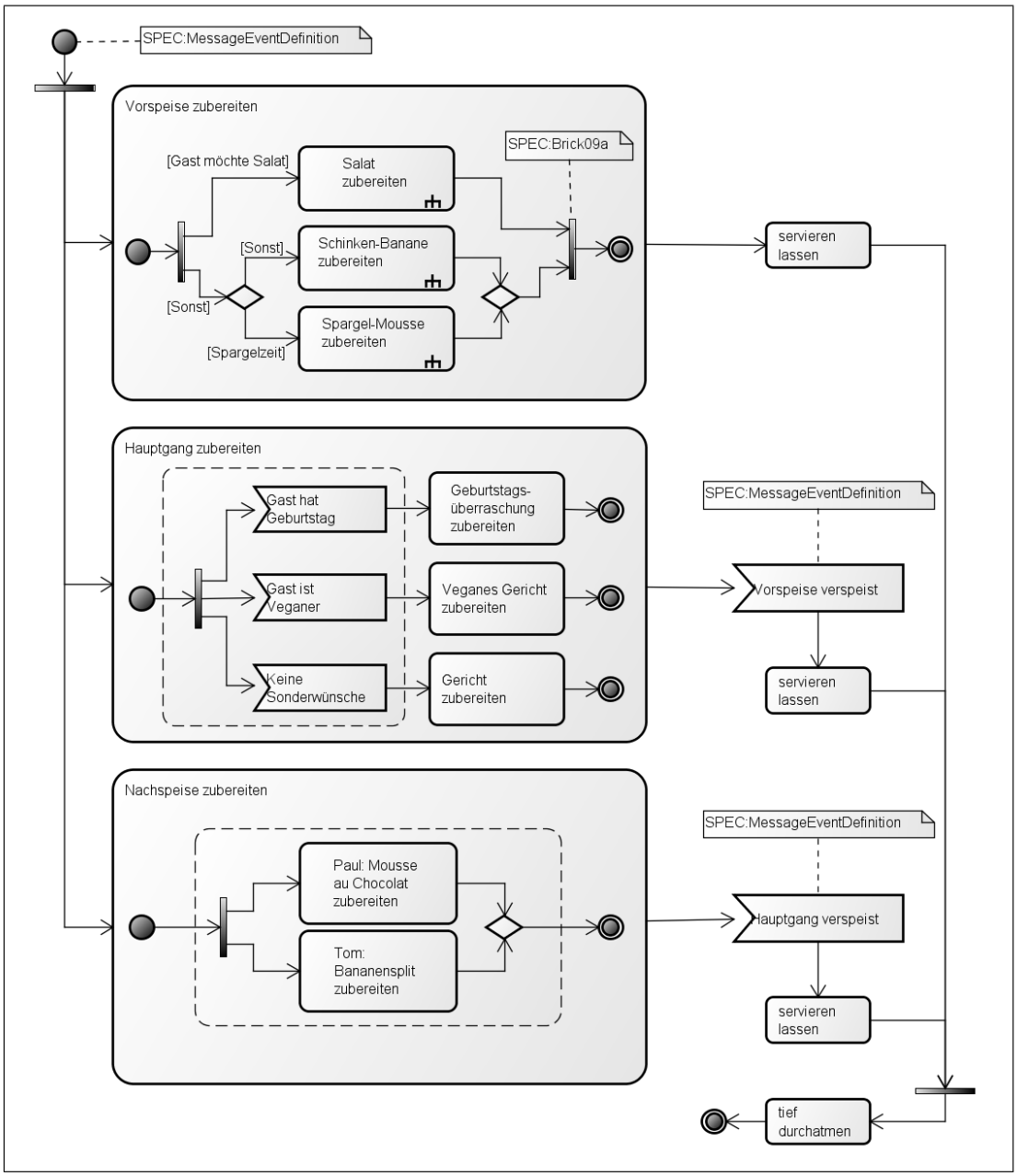


Abbildung 5.3: Transformiertes Diagramm in UML AD

Verifzierung

Durch die Bausteine und die zusätzlichen Regeln konnten alle im Diagramm verwendeten Notationselemente transformiert werden. Es wurden in diesem Diagramm nur Elemente genutzt, die den in dieser Arbeit zuvor beschriebenen Anforderungen genügen. Bei der Verwendung von nicht abgedeckten Elementen wäre es zu Fehlern in Form von Lücken im transformierten Diagramm gekommen. Die Ausführungsreihenfolge der einzelnen Schritte hat zu keinerlei Fehlern geführt und wird daher als richtig angesehen.

Validierung

Das durch die Transformation erhaltene UML Aktivitätsdiagramm modelliert den gleichen Sachverhalt wie das originale Diagramm in BPMN-Notation. Außerdem wurde das gesamte Diagramm lückenlos transformiert. Die Verwendung von nicht abgedeckten Notationselementen würde an dieser Stelle zu Fehlern führen, wie bereits angesprochen wurde. Ansonsten ist die M2M-Transformation über den erarbeiteten Transformationsweg in diese Richtung erfolgreich.

Die GoM sind auch nach der Transformation noch erfüllt. Das Diagramm ist sowohl syntaktisch als auch semantisch korrekt. Es hat den gleichen Detaillierungsgrad des Originaldiagramms, wodurch es sich weder beim Grundsatz der Wirtschaftlichkeit noch bei dem der Relevanz verschlechtert. Weiterhin besitzt das Diagramm den gleichen Grad an Klarheit wie das ursprüngliche Diagramm, da die Strukturen direkt übernommen wurden. Weil das erhaltene Diagramm den gleichen Sachverhalt wie das originale Diagramm wiedergibt, ist der Grundsatz der Vergleichbarkeit vollständig erfüllt. Zum Grundsatz des systematischen Aufbaus kann wenig gesagt werden, da hier nur die Prozesssicht betrachtet wird. Allerdings sind die Namen der einzelnen Elemente weiterhin eindeutig, sodass sie in anderen Sichten ebenfalls eindeutig identifizierbar wären.

5.3 Beispieltransformation vom UML Aktivitätsdiagramm zur BPMN

In Anlehnung an Kecher[147] zeigt Abbildung 5.4 (auf der nachfolgenden Seite) einen Restaurantbesuch. Dabei gibt es drei beteiligte Parteien: den Gast, den Kellner sowie den Koch. Diese sind dem Pool „Restaurant“ zugeordnet. Der Gast hat die Möglichkeit, sich ein Gericht auszusuchen. Fällt ihm während dieser Aktion auf, dass er einen wichtigen Termin vergessen hat, so kann er den Restaurantbesuch abbrechen und der ganze Prozess wird beendet. Anderenfalls ruft er nach dem Kellner, um eine Bestellung aufzugeben.

Der Kellner wartet bereits auf das entsprechende Signal, nimmt die Bestellung auf und gibt sie an den Koch weiter, welcher die Mahlzeit zubereitet. Der Kellner serviert das Gericht anschließend dem Gast. Dieser hat nach dem Verzehr der Mahlzeit die Möglichkeit, sich entweder über diese zu beschweren oder nach der Rechnung zu verlangen. Im Falle einer Beschwerde entschuldigt sich der Kellner und dem Gast wird die Möglichkeit eingeräumt, ein neues Gericht zu bestellen. Schlägt er diese Möglichkeit aus, kann er das Restaurant verlassen und der Prozess gilt als beendet.

Verlangt der Gast nach der Rechnung, so wird die Zahlung abgewickelt. Der genaue Vorgang ist hier nicht näher erläutert, aber beim Zahlungsvorgang kann es passieren, dass der Geldbetrag nicht in Ordnung ist. In diesem Falle wird die Polizei vom Koch gerufen und der Kellner hält gleichzeitig den Gast im Restaurant fest, bis die Polizei eintrifft. Danach gilt der Prozess als beendet. Stimmt der Geldbetrag hingegen, wird der Gast vom Kellner verabschiedet und verlässt das Restaurant. Auch in diesem Fall gilt der Prozess als beendet. Die farbliche Hervorhebung der Signale dient ausschließlich der besseren Lesbarkeit des Diagramms und hat ansonsten keinen Einfluss auf dieses.

Das Diagramm ist nach den GoM erstellt worden, das heißt, es wurde syntaktisch korrekt nach den Regeln der UML Aktivitätsdiagramme und den in dieser Arbeit definierten Beschränkungen modelliert. Der dargestellte Sachverhalt wird als korrekt betrachtet, da es sich lediglich um ein veranschaulichendes Beispiel handelt. Es wurden nur Elemente verwendet, die für die Beispieltransformation relevant sind, das heißt, sie stellen in dieser Arbeit definierte Bausteine oder Elemente mit direktem Äquivalent in der Ziel-Modellsprache dar. Dadurch ist auch der Grundsatz der Wirtschaftlichkeit erfüllt. Das Diagramm folgt einer klaren Strukturierung, ist dadurch einfach zu lesen und erfüllt somit den Grundsatz der Klarheit. Des Weiteren wurde das Diagramm systematisch aufgebaut. Die eindeutigen Bezeichnungen der Elemente würden eine weitere Verwendung in anderen Sichten ermöglichen. Ob der Grundsatz der Vergleichbarkeit erfüllt ist, wird sich direkt nach der Transformation zeigen.

[147] Vgl. Kecher [2006], Abbildung 9.65, S. 283

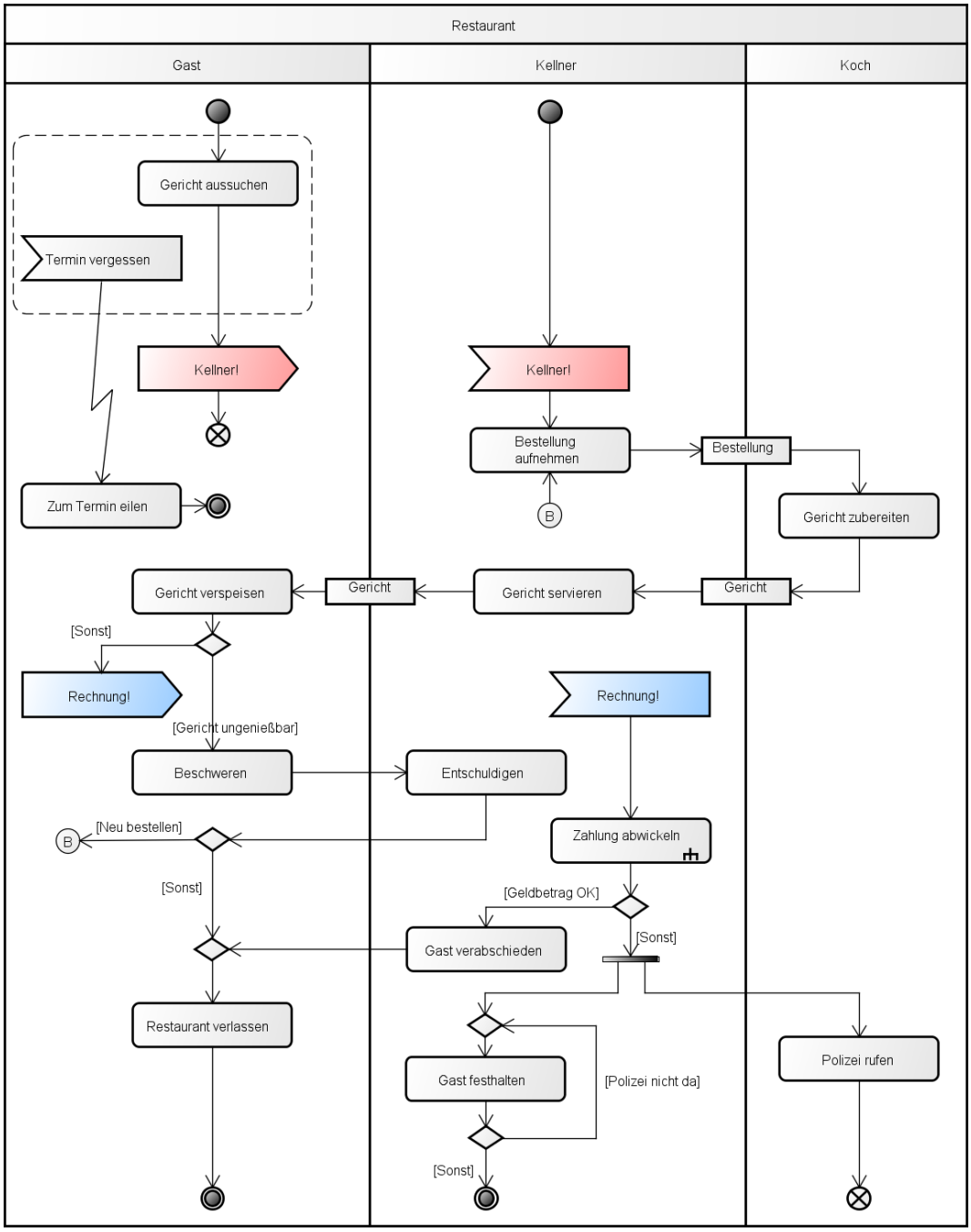


Abbildung 5.4: Diagramm in UML AD

Transformation

In Abbildung 5.5 (auf nächsten Seite) sind alle identifizierten Bausteine graphisch hervorgehoben worden. Die entsprechenden Notationselemente wurden schwächer im Hintergrund dargestellt. Auch die Verbindungslinien sind aufgrund der besseren Lesbarkeit hier abgeschwächt dargestellt. In diesem Diagramm sind 9 (exklusive Baustein #3) der 20 definierten Bausteine vertreten. Elemente, die noch nicht identifiziert werden konnten, sind noch deutlich dargestellt, so zum Beispiel die

Signale, die Konnektoren sowie die Aktivitätsbereiche. Diese werden im nächsten Transformationsschritt durch die zusätzlichen Regeln umgewandelt.

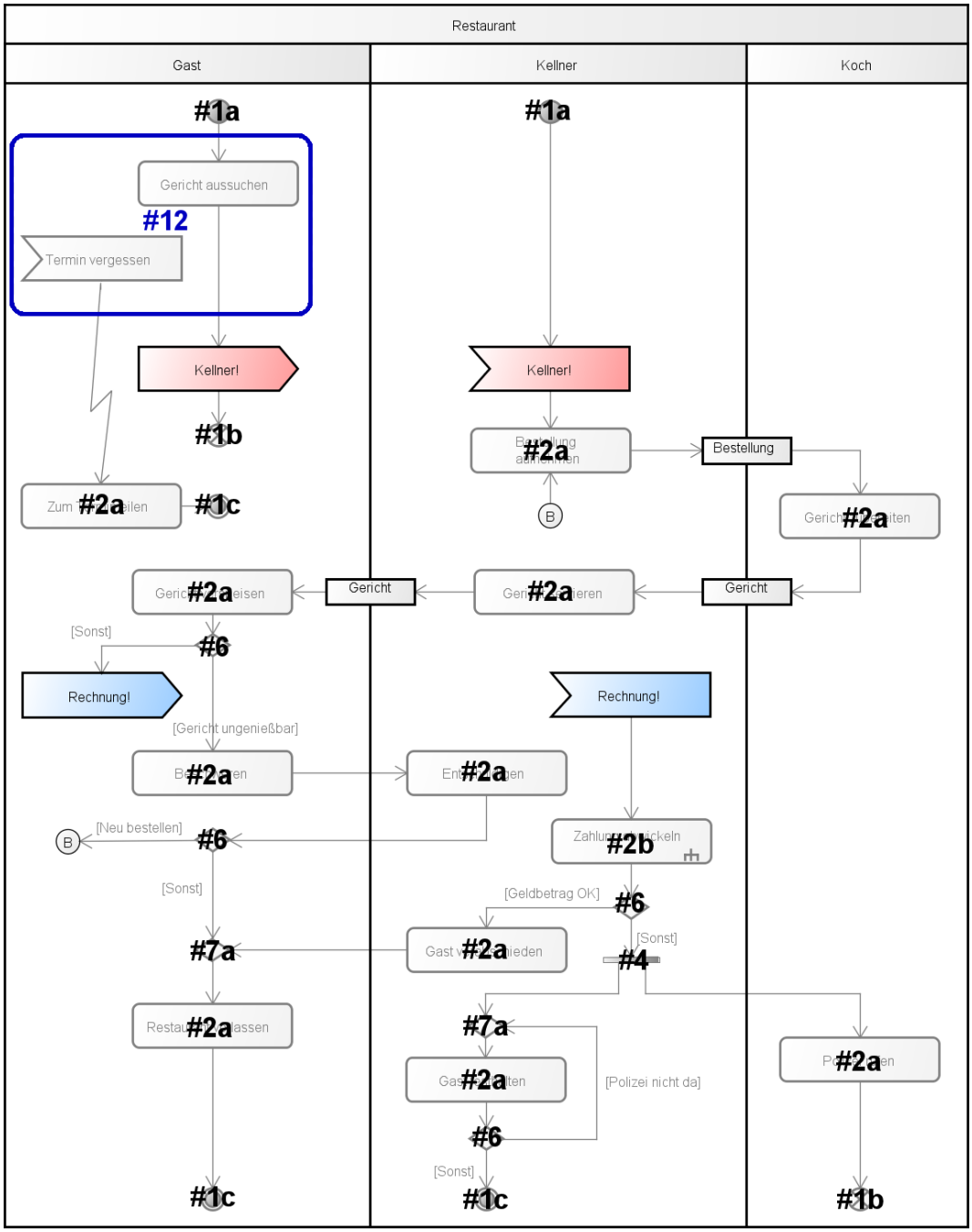


Abbildung 5.5: Diagramm in UML AD - Bausteinidentifizierung

Nach der Anwendung der zusätzlichen Regeln werden noch die Kontrollflüsse (Baustein #3) transformiert. Abbildung 5.6 zeigt das transformierte Diagramm. Auch hier gilt, dass die farbliche Hervorhebung ausschließlich der besseren Lesbarkeit des Diagramms dient. Deutlich sind auch die transformierten Flussenden im Diagramm zu sehen, da sie als spezielle Anmerkungen auftauchen.

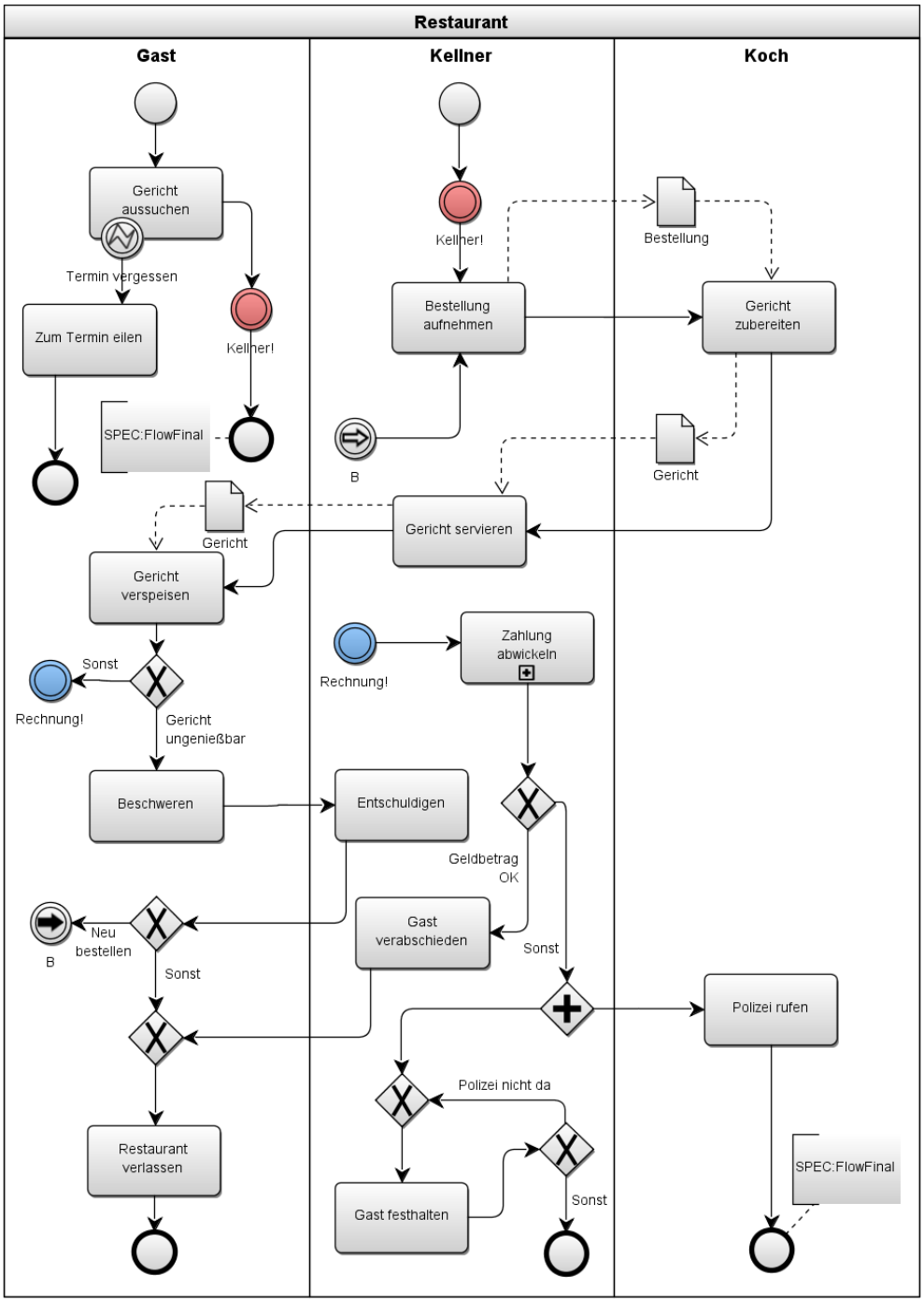


Abbildung 5.6: Transformiertes Diagramm in BPMN

Verifizierung

Durch die Bausteine und die zusätzlichen Regeln konnten alle im Diagramm verwendeten Notationselemente transformiert werden. Auch in diesem wurden nur Elemente genutzt, die den in dieser Arbeit zuvor beschriebenen Anforderungen genügen. Bei der Verwendung von nicht abgedeckten Elementen wäre es zu Fehlern in Form von Lücken im transformierten Diagramm gekommen. Die Ausführungsreihenfolge der einzelnen Schritte hat zu keinerlei Fehlern geführt und wird daher als richtig angesehen.

Validierung

Das durch die Transformation erhaltene Prozessdiagramm der BPMN modelliert den gleichen Sachverhalt wie das originale Diagramm in der Notation des UML Aktivitätsdiagramms. Außerdem wurde das gesamte Diagramm lückenlos transformiert. Die Verwendung von nicht abgedeckten Notationselementen würde an dieser Stelle zu Fehlern führen, wie bereits angesprochen wurde. Ansonsten ist die M2M-Transformation über den erarbeiteten Transformationsweg in diese Richtung erfolgreich.

Die GoM sind auch hier nach der Transformation erfüllt. Das Diagramm ist sowohl syntaktisch als auch semantisch korrekt. Es hat den gleichen Detaillierungsgrad wie das originale Diagramm, wodurch es sich weder beim Grundsatz der Wirtschaftlichkeit noch bei dem der Relevanz verschlechtert. Weiterhin besitzt das Diagramm den gleichen Grad an Klarheit wie das ursprüngliche Diagramm, da die Strukturen direkt übernommen wurden. Weil das erhaltene Diagramm den gleichen Sachverhalt wie das originale Diagramm wiedergibt, ist der Grundsatz der Vergleichbarkeit vollständig erfüllt. Es besteht eine direkte Vergleichbarkeit mit dem ursprünglichen Diagramm. Zum Grundsatz des systematischen Aufbaus kann kaum etwas gesagt werden, da nur die Prozesssicht betrachtet wird. Allerdings sind die Namen der einzelnen Elemente eindeutig gewählt worden, sodass sie in anderen Sichten ebenfalls eindeutig identifizierbar sind.

5.4 Fazit

In diesem Kapitel wurde überprüft, ob der entwickelte Transformationsweg das gewünschte Ergebnis einer M2M-Transformation zwischen der BPMN und den UML Aktivitätsdiagrammen und umgekehrt ermöglicht.

Während des Verifizierungsprozesses lag dabei das Augenmerk darauf, ob die 20 definierten Bausteine und die entwickelten Zusatzregeln sowie die Ausführungsreihenfolge an sich für die Transformation geeignet sind. Sowohl bei der beispielhaften Transformation vom BPMN Prozessdiagramm zum UML Aktivitätsdiagramm als auch in die Gegenrichtung war diese Transformation vollständig und führte zu keinen Fehlern. Alle Notationselemente wurden in das Zieldiagramm

überführt.

Im Validierungsprozess wurde überprüft, ob das erhaltene Ergebnis den Ansprüchen der M2M-Transformation genügt. Da die Transformation vollständig verlief und die beiden erhaltenen Diagramme je das wiedergaben, was im originalen Diagramm modelliert worden ist, ist die Transformation über den entwickelten Transformationsweg möglich und führt zu einem zufriedenstellenden Ergebnis.

Das Ergebnis ist solange zufriedenstellend, wie die in dieser Arbeit definierten Modellierungseinschränkungen eingehalten werden. Dazu zählt zum einen die ausschließliche Verwendung der abgedeckten Elemente, zum anderen die Einschränkung aus der detaillierten Beschreibung der Notationselemente beispielsweise bei den Gateways der BPMN: Ein Gateway erfüllt nur eine Aufgabe auf einmal, das heißt entweder es verzweigt den Sequenzfluss oder es führt diesen zusammen. Wird diese Regel nicht eingehalten, so ist eine Identifizierung der korrekten Bausteine nicht mehr möglich, da diese so definiert sind, dass Gateways nur eine dieser Aufgaben ausführen. Außerdem muss eine den Bausteinen nahe Modellierungsweise eingehalten werden. Dies betrifft vor allem die Bausteine #10, #11 und #12, welche aus mehreren Notationselementen bestehen.

Werden diese Modellierungseinschränkungen eingehalten, so funktioniert die Transformation in beide Richtungen ohne Fehler und vollständig. Daraus kann geschlossen werden, dass die M2M-Transformation über den erarbeiteten Transformationsweg mit einem Baustein-Prinzip und zusätzlichen Regeln bidirektional ist. Durch das Setzen der speziellen Anmerkungen kann auch das originale Diagramm nach der Transformation wiederhergestellt werden, indem der Ablauf für die Gegenrichtung ausgeführt wird. Auch die GoM werden bei der Transformation übernommen.

Somit ist auch das Ziel dieser Arbeit erreicht, bei welchem ein bidirektionaler Transformationsweg zwischen den BPMN Prozessdiagrammen und den UML Aktivitätsdiagrammen erarbeitet werden sollte. Auf den erhaltenen Ergebnissen kann auch für weitere M2M-Transformationen aufgebaut werden, wie im abschließenden Kapitel erläutert wird.

6 Zusammenfassung und Ausblick

In den vorangegangenen Kapiteln wurde eine neue Methode zur Transformation von Diagrammen in BPMN zu UML Aktivitätsdiagrammen und umgekehrt für die Prozessdarstellung erarbeitet. Dazu wurden zunächst in Kapitel 2 wichtige Begriffe für dieser Arbeit erläutert und ein kurzer Einblick in die Geschichte und die Bedeutung von Prozessen für Unternehmen gegeben. Im anschließenden Kapitel 3 wurden die Object Management Group (OMG), welche die Verantwortung für die beiden betrachteten Modellierungsstandards Business Process Model and Notation (BPMN) und Unified Modeling Language (UML) trägt, sowie die Standards selbst vorgestellt.

Es wurde weiterhin geprüft, in wie weit sich die beiden Modellsprachen für die Darstellung von Prozessen eignen. Dies wurde anhand des Workflow Pattern Framework getan. Dieser enthält drei Arten von Mustern: Kontrollflussmuster, Datenmuster und Ressourcenmuster. Es wurde festgestellt, dass beide Modellsprachen erhebliche Defizite bei den Datenmustern und Ressourcenmustern aufweisen. Zugleich wurde aber auch festgestellt, dass für die Prozessdarstellung die Kontrollflussmuster von besonderer Bedeutung sind und diese sowohl bei der BPMN als auch bei den Aktivitätdiagrammen der UML zu großen Teilen abgedeckt werden. Aus diesem Grund wurde geschlussfolgert, dass sich beide Modellsprachen für die Prozessdarstellung eignen. Im Anschluss wurden ihre jeweiligen Notationselemente vorgestellt und Einschränkungen bezüglich ihrer Verwendung definiert.

In Kapitel 4 wurde dann der Transformationsweg erarbeitet. Zunächst erfolgte eine kurze Überlegung zu einer vollständigen 1-zu-1-Transformation zwischen den beiden Modellsprachen. Dieser Ansatz hat sich jedoch schon bei anderen Autoren als schwierig erwiesen, da die BPMN eine Modellsprache ist, die viele Elemente aufweist, und die unterschiedliche Möglichkeiten bietet, einen bestimmten Sachverhalt darzustellen. Aus diesem Grund wurde im nächsten Schritt ein Baustein-Prinzip entwickelt, welches sich an den bereits verwendeten Workflow Patterns, genauer gesagt den Kontrollflussmustern, orientiert. Da die Darstellung eines Kontrollflussmusters in einer Modellsprache in den meisten Fällen eindeutig ist, kann eine Transformation anhand dieser Kontrollflussmuster geschehen.

Dafür wurden zunächst die Bausteine definiert und eine XML Schema Definition aufgezeigt. Danach wurde gezeigt, nach welchen Strukturen in den XML-Export-Dateien gesucht werden muss, um einen bestimmten Baustein zu identifizieren. Es wurde ebenso gezeigt, welche XML-Struktur erzeugt werden muss, um aus den Bausteinen wieder ein Diagramm in der BPMN oder den UML Aktivitätsdiagrammen zu schaffen.

Da über das vorgestellte Baustein-Prinzip nicht alle, aber bereits ein großer Teil der Notationselemente beider Modellsprachen abgedeckt werden konnte, wurden zusätzlich einige weitere Regeln definiert, die eine Transformation ermöglichen. Im Anschluss wurde ein Algorithmus erstellt, welcher die Reihenfolge vorgibt, in welcher die einzelnen Schritte für die Transformation gegangen werden müssen.

In Kapitel 5 wurde graphisch gezeigt, wie aus einem Diagramm in der Notation der BPMN über das Baustein-Prinzip und die zusätzlichen Regeln ein Aktivitätsdiagramm der UML und wie aus einem UML Aktivitätsdiagramm ein BPMN-Diagramm wird. Diese M2M-Transformation geschah vollständig, das heißt, dass alle verwendeten Elemente transformiert wurden, und erzielte ein zufriedenstellendes Ergebnis, das heißt, dass die erhaltenen Diagramme den ursprünglichen Sachverhalt wiedergaben.

Die Forschungsfrage „Wie können Transformationswege in Hinblick auf die Prozessdarstellung zwischen den Modellsprachen BPMN und UML aussehen und formalisiert werden?“ wurde bearbeitet und es ist ein möglicher Transformationsweg zwischen den Prozessdiagrammen der BPMN und den Aktivitätdiagrammen der UML entwickelt worden. Diese Transformation erfolgt über ein Baustein-Prinzip auf Grundlage des Workflow Pattern Framework und einiger zusätzlicher Regeln für eine vollständigere Abdeckung bei der Transformation. Die Tabellen 6.1 und 6.2 (auf der nachfolgenden Seite) zeigen noch einmal die Ansätze anderer Autoren und stellen ihre Eigenschaften denen des hier erarbeiteten Ansatzes gegenüber.

Autor	Richtung	Eigenschaften	Kritik
Kalnins und Vitolins	UML AD zu BPMN	Transformation unter Verwendung von MOLA	Verwendung einer Untermenge der UML AD Elemente; Verwendung einer Untermenge der BPMN Elemente; nur unidirektionale Transformation
Cibrán	BPMN zu UML AD	Transformation unter Verwendung von ATL	Uneindeutigkeiten bei Transformation aufgrund der Komplexität der BPMN; nur unidirektionale Transformation
Macek und Richta	BPMN zu UML AD	Transformation unter Verwendung von XSLT	Informationsverluste aufgrund der Komplexität der BPMN; nur unidirektionale Transformation

Tabelle 6.1: Vergleich der Ansätze

Autor	Richtung	Eigenschaften	Kritik
Raedts et al.	BPMN zu Petri-Netz, UML AD zu Petri-Netz, EPK zu Petri-Netz	Verifizierung und Validierung von Modellen durch Analyse von Petri-Netzen; Vertiefung durch mCRL2	nur unidirektional; keine Transformation zwischen BPMN und UML
Lenz	BPMN zu UML AD, UML AD zu BPMN	Transformation auf Grundlage der Kontrollflussmuster des Workflow Pattern Framework; XML-basierte Transformation; birektionale Transformation	Verwendung einer Untermenge der UML AD- und BPMN-Elemente; Verwendung zusätzlicher Regeln für vollständigere Abdeckung; strikte Einhaltung vorgeschriebener Modellierungsregeln notwendig

Tabelle 6.2: Vergleich der Ansätze (Fortsetzung)

Wie schon bei Kalnins und Vitolins erfolgt die Transformation auf einer eingeschränkten Menge der Notationselemente aufgrund von deren Komplexität und dem Fehlen von eindeutigen Äquivalenten. Allerdings wird die Menge der Notationselemente bei den Aktivitätsdiagrammen der UML nur um zwei Elemente beschränkt, welche durch andere Notationselemente imitierbar sind. Bei der BPMN dürfen über ein Dutzend Elemente nicht benutzt werden. Von diesen können allerdings fast die Hälfte durch andere Elemente imitiert werden. Außerdem werden die wichtigsten Elemente wie Flüsse, Ereignisse, Aufgaben etc. abgedeckt, sodass die Modellierung von durchaus komplexen Prozessen möglich ist.

Es gibt in dem Ansatz auch einige Einschränkungen zur Verwendung bestimmter Elemente wie beispielsweise bei den Gateways: Ein Gateway kann zu einem Zeitpunkt nur verzweigen oder nur zusammenführen, aber nicht beide Aufgaben parallel ausführen. Diese Einschränkungen vereinfacht allerdings unter Umständen die Lesbarkeit des Diagramms. Außerdem können derartige Modellierungen erkannt und vor der Transformation geändert werden, sodass der Ansatz dann wiederum angewandt werden kann. Dies erfordert zwar einen zusätzlichen Schritt, ermöglicht aber die Anwendung des erarbeiteten Ansatzes.

Die Verwendung zusätzlicher Regeln bei der Transformation führt zu einer größeren Abdeckung der Notationselemente als die Verwendung der Kontrollflussmuster des Workflow Pattern Framework allein. Dies zeigt, dass der Ansatz nur für bestimmte Modellierungskonstrukte funktioniert. Allerdings zeigt die erfolgreiche Verwendung der zusätzlichen Regeln auch Möglichkeiten auf, einzelne Elemente nachträglich umzuwandeln, sodass eine vollständigere M2M-Transformation erreicht wird.

Da eine XSD für die Bausteine angegeben wurde und sowohl für die BPMN als auch für die UML Aktivitätsdiagramme eine Beschreibung in XML vorliegt, sollte es möglich sein, ein Programm, was die automatische M2M-Transformation umsetzt, zeitnah zu implementieren. In dieser Arbeit wurde dies beispielhaft getan und damit gezeigt, dass die Konstruktionen alle eindeutig identifizierbar sind. Durch die mögliche automatische Transformation können die Modelle ohne großen Zeitverlust ineinander überführt werden.

Ein entscheidender Vorteil des hier entwickelten Ansatzes ist, dass die Transformation über das Baustein-Prinzip bidirektional abläuft: Es kann nicht nur von den BPMN Prozessdiagrammen zu den UML Aktivitätsdiagrammen transformiert werden sondern auch von den UML Aktivitätsdiagrammen zu den BPMN Prozessdiagrammen. Da die Transformation in beide Richtungen möglich ist, kann auch das originale Diagramm durch eine weitere Transformation wieder hergestellt werden. Um diese bidirektionale Transformation zu ermöglichen wird im Ansatz mit speziellen Anmerkungen gearbeitet. Dies schränkt die Lesbarkeit des Modells eventuell ein, wenn viele dieser speziellen Anmerkungen verwendet werden müssen.

Der Ansatz mit dem Baustein-Prinzip zur Transformation sollte dennoch weiter verfolgt werden, denn mit dem Workflow Pattern Framework wurden nicht nur die BPMN und die UML Aktivitätsdiagramme getestet, sondern auch noch eine Reihe anderer Modellsprachen. So wurden unter anderen auch die Business Process Execution Language (BPEL), die EPK von ARIS und der SAP Workflow getestet.[148] Daraus ergeben sich potentiell unter anderem die in Abbildung 6.1 dargestellten Transformationen. Die Bausteine dienen dabei als eine Art Meta-Ebene. Auf diese Weise können auch Diagramme in BPMN-Notation in EPK transformiert werden oder EPK zu UML Aktivitätsdiagrammen. Für die automatische Modellgenerierung im ARIS Business Architect wäre dies ein entscheidender Zugewinn.

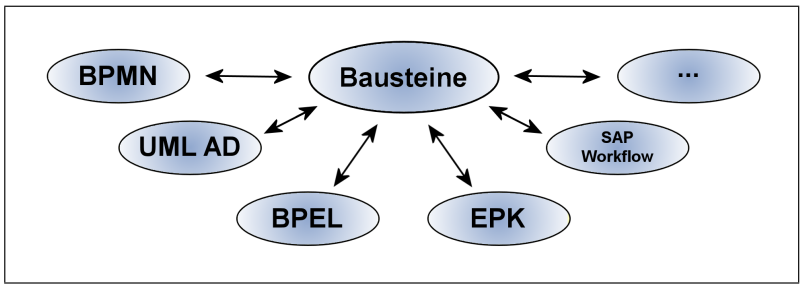


Abbildung 6.1: Mögliche weitere Transformationen

Allerdings werden bei diesen Modellierungsstandards (BPEL, EPK etc.) nicht so viele Kontrollflussmuster unterstützt, wie das bei der BPMN oder den UML Aktivitätsdiagrammen der Fall ist. Demnach wäre die Abdeckung durch die Bausteine, die auf diesen Kontrollflussmustern basieren, nicht so vollständig. Dennoch kann eine (Teil-)Transformation vorgenommen werden. Das ist vor allem dann der Fall, wenn

[148] Vgl. WPI [2011a]

sich bei der Modellierung auf die Basismuster der Kontrollflussmuster beschränkt wird. Es ist außerdem auch hier möglich, einige Zusatzregeln zu definieren, um eine vollständigere Abdeckung zu erhalten. Diese Zusatzregeln müssten allerdings für jede Modellsprache definiert werden.

Das in dieser Arbeit entwickelte Prinzip kann nicht nur dabei helfen, einen standardisierten Transformationsweg zwischen BPMN-Diagrammen und UML Aktivitätsdiagrammen zu definieren; es kann auch auf die bidirektionale Transformation zwischen weiteren bedeutenden Modellierungsstandards übertragen werden und ermöglicht somit die Erschließung eines gesamtheitlichen Standards. Die M2M-Transformation bietet noch viele Forschungsmöglichkeiten und wird auch in Zukunft von großer Bedeutung bleiben.

Literaturverzeichnis

[3D-Coat 2013] 3D-Coat: *Upgrading Floating Licences V3 -> V4*. 2013. – URL `http://pilgway.com/files/floating.pdf` (Zuletzt besucht am 26. Dezember 2013)

[Alibabaei et al. 2009] Alibabaei, Ahmad ; Bandara, Wasana ; Aghdasi, Mohammad: Means of Achieving Business Process Management Success Factors. In: *Proceedings of the 4th Mediterranean Conference on Information Systems*, 2009

[Bartonitz 2009] Bartonitz, Martin: *Historische Entwicklung wichtiger Standards im Business Process Management*. November 2009. – URL `http://upload.wikimedia.org/wikipedia/commons/d/df/Standards_BPM_2009_11.jpg` (Zuletzt besucht am 26. Dezember 2013)

[Becker et al. 2009] Becker, Jörg ; Mathas, Christoph ; Winkelmann, Axel: *Geschäftsprozessmanagement (Informatik im Fokus)*. Springer, 2009

[Becker et al. 1995] Becker, Jörg ; Rosemann, Michael ; Schütte, Reinhard: Grundsätze ordnungsgemäßer Modellierung. In: *Wirtschaftsinformatik 37*, 1995, S. 435–445

[Bergmann 2009] Bergmann, Tobias: *Klassische Wertkette nach M. Porter in Deutsch*. Dezember 2009. – URL `http://upload.wikimedia.org/wikipedia/commons/f/f1/Wertkette.JPG` (Zuletzt besucht am 26. Dezember 2013)

[Brockhaus 2003a] Brockhaus (Hrsg.): *Brockhaus Universallexikon, Band 15*. F. A. Brockhaus GmbH, 2003

[Brockhaus 2003b] Brockhaus (Hrsg.): *Brockhaus Universallexikon, Band 18*. F. A. Brockhaus GmbH, 2003

[Brockhaus 2003c] Brockhaus (Hrsg.): *Brockhaus Universallexikon, Band 23*. F. A. Brockhaus GmbH, 2003

[Cibrán 2008] Cibrán, María A.: Translating BPMN Models into UML Activities. In: *1st International Workshop on Model-Driven Engineering For Business Process Management*, 2008, S. 61–72

[DIN e.V. 2011] DIN e.V.: *ISO 9000 Einführungs- und Unterstützungspaket: Leitfaden zum Konzept und zur Anwendung des prozessorientierten Ansatzes für Managementsysteme*. 2011

[Fowler 2003] Fowler, Martin: *UML konzentriert*. 3. Auflage. Addison-Wesley Verlag, 2003

[Freund und Rücker 2012] Freund, Jakob ; Rücker, Bernd: *Praxishandbuch BPMN 2.0.* 3. Auflage. Carl Hanser Verlag, 2012

[Fritzsche et al. 2010] Fritzsche, Mathias ; Gilani, Wasif ; Lämmel, Ralf ; Jouault, Frédéric: Model Transformation Chains in Model-Driven Performance Engineering: Experiences and Future Research Needs. In: *Lecture Notes in Informatics*, 2010, S. 213–220

[Göpfert und Lindenbach 2012] Göpfert, Jochen ; Lindenbach, Heidi: *Geschäftsprozessmodellierung mit BPMN 2.0: Business Process Model and Notation.* Oldenbourg Wissenschaftsverlag, 2012

[Harel und Rumpe 2000] Harel, David ; Rumpe, Bernhard: *Modeling Languages: Syntax, Semantics and All That Stuff - Part I: The Basic Stuff*, August 2000

[Harel und Rumpe 2004] Harel, David ; Rumpe, Bernhard: *Modeling Languages: Syntax, Semantics and all that Stuff (or What's the Semantics of „Semantics"?)*, Juli 2004

[Informatik 2013] OOSE Innovative Informatik: *oose-UML-Toolliste.* 2013. – URL `http://www.oose.de/wp-content/uploads/2011/11/oose-UML-Toolliste.pdf` (Zuletzt besucht am 26. Dezember 2013)

[Juristo 2005] Juristo, Natalia: Chapter 3: Verification and Validation: Current and Best Practice. In: *Validation, Verification and Certification of Embedded Systems*, 2005

[Kalnins und Vitolins 2006] Kalnins, Audris ; Vitolins, Valdis: Use of UML and Model Transformations for Workflow Process Definitions. In: *Databases and Information Systems (BalticDBIS'2006)*, 2006, S. 3–15

[Kecher 2006] Kecher, Christoph: *UML 2.0 - Das umfassende Handbuch.* 2. Auflage. Galileo Press, 2006

[Ko et al. 2009] Ko, Ryan K. ; Lee, Stephen S. ; Lee, Eng W.: Business Process Management (BPM) Standards: A Survey. In: *Business Process Management Journal* 15 (2009), S. 744–791

[Macek und Richta 2009] Macek, Ondrej ; Richta, Karel: The BPM to UML Activity Diagram Transformation using XSLT. In: *Database, Texts, Specifications, and Objects (DATESO 2009)*, 2009, S. 119–129

[Maria 1997] Maria, Anu: Introduction to Modeling and Simulation. In: *Proceedings of the 1997 Winter Simulation Conference*, 1997, S. 7–13

[Miers 2006] Miers, Derek: *The Keys to BPM Project Success*, Januar 2006

[Oberkampf und Trucano 2002] Oberkampf, William L. ; Trucano, Timothy G.: Verification and validation in computational fluid dynamics. In: *Progress in Aerospace Sciences 38*, 2002, S. 209–272

[Oestereich 2012] Oestereich, Bernd: *Analyse und Design mit der UML 2.5 - Objektorientierte Softwareentwicklung*. 10. Auflage. Oldenbourg Wissenschaftsverlag, 2012

[Omar 2012] Omar, Omana: *Eine BPMN-nach-BPEL-Transformation*, Universität Stuttgart, Diplomarbeit, 2012

[OMG 2009] OMG: *Business Process Model and Notation (BPMN)*. Version 1.2, Januar 2009

[OMG 2011a] OMG: *Business Process Model and Notation (BPMN)*. Version 2.0, Januar 2011

[OMG 2011b] OMG: *OMG Unified Modeling Language (OMG UML) Superstructure*. Version 2.4.1, August 2011

[OMG 2011c] OMG: *UML - XMI of the merged L3 UML 2.4.1 as an instance of UML 2.4.1 using XMI 2.4.1*. Juli 2011. – URL `http://www.omg.org/spec/UML/20110701/UML.xmi` (Zuletzt besucht am 26. Dezember 2013)

[OMG 2013a] OMG: *Object Management Group - Business Process Model and Notation*. 2013. – URL `http://www.bpmn.org/` (Zuletzt besucht am 26. Dezember 2013)

[OMG 2013b] OMG: *OMG - We Set the Standard*. 2013. – URL `http://www.omg.org/memberservices/OMG_Backgrounder.pdf` (Zuletzt besucht am 26. Dezember 2013)

[OMG 2013c] OMG: *OMG - We Set the Standard - About OMG*. April 2013. – URL `http://www.omg.org/gettingstarted/gettingstartedindex.htm` (Zuletzt besucht am 26. Dezember 2013)

[OMG 2013d] OMG: *Snapshot: The OMG Technology Adopting Process*. 2013. – URL `http://www.omg.org/memberservices/TechAdoptProcess.pdf` (Zuletzt besucht am 26. Dezember 2013)

[Paige et al. 2000] Paige, R. F. ; Ostroff, J. S. ; Brooke, P. J.: Principles for Modeling Language Design. In: *Information and Software Technology 42*, 2000, S. 665–675

[Pan et al. 2012] Pan, Jeff Z. ; Staab, Steffen ; Assmann, Uwe ; Ebert, Jürgen ; Zhao, Yuting: *Ontology-Driven Software Development*. Springer Verlag, 2012

[Peixoto et al. 2008] Peixoto, Daniela C. C. ; Batista, Vitor A. ; Atayde, Ana P. ; Borges, Eduardo P. ; Resende, Rodolfo F. ; Pádua, Clarindo Isaías P. S.: A Comparison of BPMN and UML 2.0 Activity Diagrams. In: *Proceedings of the 7th Brazilian Symposium on Software Quality (SBQS2008)*, 2008, S. 1–12

[Peterson und Faber 2004] Peterson, Thomas ; Faber, Malte: Verantwortung und das Problem der Kuppelproduktion. Reflexionen über die Grundlagen der Umweltpolitik. In: *Discussion Paper Series, No. 411*, 2004

[Raedts et al. 2007] Raedts, Ivo ; Petkoviÿ, Marija ; Usenko, Yaroslav S. ; Werf, Jan M. van der ; Groote, Jan F. ; Somers, Lou: Transformation of BPMN Models for Behaviour Analysis. In: *Proceedings of the 5th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS-2007)*, 2007, S. 126–137

[Russell et al. 2006] Russell, Nick ; Aalst, Wil M. van der ; Hofstede, Arthur H. ter ; Wohed, Petia: On the Suitability of UML 2.0 Activity Diagrams for Business Process Modeling. In: *Proceedings of the 3rd Asia-Pacific Conference on Conceptual Modelling (APCCM'06)*, 2006, S. 95–104

[Schmelzer und Sesselmann 2010] Schmelzer, Hermann J. ; Sesselmann, Wolfgang: *Geschäftsprozessmanagement in der Praxis: Kunden zufrieden stellen - Produktivität steigern - Wert erhöhen.* 7. Auflage. Hanser Wirtschaft, 2010

[Seidlmeier 2010] Seidlmeier, Heinrich: *Prozessmodellierung mit ARIS - Eine beispielorientierte Einführung für Studium und Praxis.* 3. Auflage. Vieweg+Teubner Verlag, 2010

[Volkswagen-AG 2012] Volkswagen-AG: *Produktionsstandorte.* Dezember 2012. – URL `http://www.volkswagenag.com/content/vwcorp/content/de/the_group/production_plants.html` (Zuletzt besucht am 26. Dezember 2013)

[White 2004a] White, Stephen A.: *Introduction to BPMN.* IBM Corporation, 2004

[White 2004b] White, Stephen A.: *Process Modeling Notations and Workflow Patterns.* IBM Corporation, 2004

[Wohed et al. 2006] Wohed, Petia ; Aalst, Wil M. van der ; Dumas, Marlon ; Hofstede, Arthur H. ter ; Russell, Nick: On the Suitability of BPMN for Business Process Modelling. In: *Lecture Notes in Computer Science*, 2006, S. 161–176

[Wohed et al. 2005a] Wohed, Petia ; Aalst, Wil M. van der ; Dumas, Marlon ; Hofstede, Arthur H. ter ; Russell, Nick: *Pattern-based Analysis of BPMN - An Extensive Evaluation of the Control-flow, the Data and the Resource Perspectives.* 2005

[Wohed et al. 2005b] Wohed, Petia ; Aalst, Wil M. van der ; Dumas, Marlon ; Hofstede, Arthur H. ter ; Russell, Nick: Pattern-based Analysis of the Control-flow Perspective of UML Activity Diagrams. In: *Proceedings of the 24th International Conference on Conceptual Modeling (ER05)*, 2005, S. 63–78

[WPI 2011a] WPI: *Workflow Process Initiative - Workflow Patterns - Evaluations.* 2011. – URL `http://www.workflowpatterns.com/evaluations/` (Zuletzt besucht am 26. Dezember 2013)

[WPI 2011b] WPI: *Workflow Process Initiative - Workflow Patterns - Patterns.* 2011. – URL `http://www.workflowpatterns.com/patterns` (Zuletzt besucht am 26. Dezember 2013)

[WPI 2011c] WPI: *Workflow Process Initiative - Workflow Patterns - Welcome to the Workflow Patterns Homepage*. 2011. – URL `http://www.workflowpatterns.com/` (Zuletzt besucht am 26. Dezember 2013)

[Zimmer 2012] Zimmer, Christoph: *BPMN 2.0 Metamodel*, Oktober 2012

A Workflow Patterns

Auf den folgenden Seiten befindet sich eine Auflistung aller von Wohed et al.[149] verwendeten Muster bei der Analyse der BPMN und UML zu ihrer Tauglichkeit zur Prozessdarstellung. Die Muster sind zunächst zu Vergleichszwecken in tabellarischer Form aufgeführt, im Anschluss werden sie beschrieben[150]. Die Muster sind in Originalsprache (englisch) aufgelistet, um Übersetzungsfehlern vorzubeugen.

[149] Vgl. Wohed et al. [2005a] und Wohed et al. [2006]
[150] Vgl. WPI [2011b]

A.1 Kontrollflussmuster (Control Flow Patterns)

	BPMN	AD
Basic Control Flow		
1. Sequence	+	+
2. Parallel Split	+	+
3. Synchronisation	+	+
4. Exclusive Choice	+	+
5. Simple Merge	+	+
Advanced Synchronisation		
6. Multiple Choice	+	+
7. Synchronising Merge	+/–	–
8. Multiple Merge	+	+
9. Discriminator	+	+
Structural Patterns		
10. Arbitrary Cycles	+	+
11. Implicit Termination	+	+
Multiple Instances Patterns		
12. MI without Synchronization	+	+
13. MI with a priori Design Time Knowledge	+	+
14. MI with a priori Run Time Knowledge	+	+
15. MI without a priori Run Time Knowledge	–	–
State-Based Patterns		
16. Deferred Choice	+	+
17. Interleaved Parallel Routing	+/–	–
18. Milestone	–	–
Cancellation Patterns		
19. Cancel Activity	+	+
20. Cancel Case	+	+

Tabelle A.1: Unterstützung der Kontrollflussmuster des Workflow Pattern Frameworks [Wohed et al., 2005a, S. 13]

Ein „+“ in den Tabellen bedeutet direkte Unterstützung des Musters, ein „+/–“ eine teilweise Unterstützung und ein „–“ ein Fehlen der Unterstützung.

1. **Sequence:** A task in a process is enabled after the completion of a preceding task in the same process.

2. **Parallel Split:** The divergence of a branch into two or more parallel branches each of which execute concurrently.

3. **Synchronization:** The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled.

4. **Exclusive Choice:** The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on a mechanism that can select one of the outgoing branches.

5. **Simple Merge:** The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.

6. **Multi-Choice:** The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to one or more of the outgoing branches based on a mechanism that selects one or more outgoing branches.

7. **Synchronizing Merge:** The convergence of two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled.

8. **Multi-Merge:** The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.

9. **Discriminator** The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch when the first incoming branch has been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on.

10. **Arbitrary Cycles:** The ability to represent cycles in a process model that have more than one entry or exit point. It must be possible for individual entry and exit points to be associated with distinct branches.

11. **Implicit Termination:** A given process (or sub-process) instance should terminate when there are no remaining work items that are able to be done either now or at any time in the future and the process instance is not in deadlock. There is an objective means of determining that the process instance has successully completed.

12. **Multiple Instances without Synchronization:** Within a given process instance, multiple instances of a task can be created. These instances are independent of each other and run concurrently. There is no requirement to synchronize them upon completion. Each of the instances of the multiple instance task that are created must execute within the context of the process instance from which they were started (i.e. they must share the same case identifier and have access to the same data elements) and each of them must execute independently from and without reference to the task that started them.

13. **Multiple Instances with a priori Design-Time Knowledge:** Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronize the task instances at completion before any subsequent tasks can be triggered.

14. **Multiple Instances with a priori Run-Time Knowledge:** Within a given process instance, multiple instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications, but is known before the task instances must be created. Once initiated, these instances are independent of each other and run concurrently. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered.

15. **Multiple Instances without a priori Run-Time Knowledge:** Within a given process instance, multiple instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. Once initiated, these instances are independent of each other and run concurrently. At any time, whilst instances are running, it is possible for additional instances to be initiated. It is neccesary to synchronize the instances at completion before any subsequent tasks can be triggered.

16. **Deferred Choice:** A point in a process where one of several branches is chosen based on interaction with the operating environment. Prior to the decision, all branches represent possible future courses of execution. The decision is made by initiating the first task in one of the branches i.e. there is no explicit choice but rather a race between different branches. After the decision is made, execution alternatives in branches other than the one selected are withdrawn.

17. **Interleaved Parallel Routing:** A set of tasks has a partial ordering defining the requirements with respect to the order in which they must be executed. Each task in the set must be executed once and they can be completed in any order that accords with the partial order. However, as an additional requirement, no two tasks can be executed at the same time (i.e. no two tasks can be active for the same process instance at the same time).

18. **Milestone:** A task is only enabled when the process instance (of which it is part) is in a specific state (typically a parallel branch). The state is assumed to be a specific execution point (also known as a milestone) in the process model. When this execution point is reached the nominated task can be enabled. If the process instance has progressed beyond this state, then the task cannot be enabled now or at any future time (i.e. the deadline has expired). Note that the execution does not influence the state itself, i.e. unlike normal control-flow dependencies it is a test rather than a trigger.

19. **Cancel Task:** An enabled task is withdrawn prior to it commencing execution. If the task has started, it is disabled and, where possible, the currently running instance is halted and removed.

20. **Cancel Case:** A complete process instance is removed. This includes currently executing tasks, those which may execute at some future time and all subprocesses. The process instance is recorded as having completed unsuccessfully.

A.2 Datenmuster (Data Patterns)

	BPMN	AD
Data Visibility		
1. Task Data	+	+/–
2. Block Data	+	+
3. Scope Data	–	–
4. Multiple Instance Data	+/–	+
5. Case Data	+	–
6. Folder Data	–	–
7. Workflow Data	–	+
8. Environment Data	–	–
Data Interaction (Internal)		
9. Task to Task	+	+
10. Block Task to Sub-Workflow Decomposition	+	+
11. Sub-Workflow Decomposition to Block Task	+	+
12. to Multiple Instance Task	–	+
13. from Multiple Instance Task	–	+
14. Case to Case	–	–
Data Interaction (External)		
15. Task to Environment - Push-Orientated	+	–
16. Environment to Task - Pull-Orientated	+	–
17. Environment to Task - Push-Orientated	+	–
18. Task to Environment - Pull-Orientated	+	–
19. Case to Environment - Push-Orientated	–	–
20. Environment to Case - Pull-Orientated	–	–
21. Environment to Case - Push-Orientated	–	–
22. Case to Environment - Pull-Orientated	–	–
23. Workflow to Environment - Push-Orientated	–	–
24. Environment to Workflow - Pull-Orientated	–	–
25. Environment to Workflow - Push-Orientated	–	–
26. Workflow to Environment - Pull-Orientated	–	–
Data Transfer		
27. by Value - Incoming	+	–
28. by Value - Outgoing	+	–
29. Copy In / Copy Out	+/–	–
30. by Reference - Unlocked	–	–
31. by Reference - Locked	+	+
32. Data Transformation - Input	+/–	+
33. Data Transformation - Output	+/–	+
Data-based Routing		
34. Task Precondition - Data Existence	+	+
35. Task Precondition - Data Value	–	+
36. Task Postcondition - Data Existence	+	+
37. Task Postcondition - Data Value	–	+
38. Event-based Task Trigger	+	+
39. Data-based Task Trigger	+	–
40. Data-based Routing	+	+

Tabelle A.2: Unterstützung der Datenmuster des Workflow Pattern Frameworks [Wohed et al., 2006, S. 11]

Ein „+“ in den Tabellen bedeutet direkte Unterstützung des Musters, ein „+/–“ eine teilweise Unterstützung und ein „–“ ein Fehlen der Unterstützung.

1. **Task Data:** Data elements can be defined by tasks which are accessible only within the context of individual execution instances of that task.

2. **Block Data:** Block tasks (i.e. tasks which can be described in terms of a corresponding subprocess) are able to define data elements which are accessible by each of the components of the corresponding subprocess.

3. **Scope Data:** Data elements can be defined which are accessible by a subset of the tasks in a case.
4. **Multiple Instance Data:** Tasks which are able to execute multiple times within a single case can define data elements which are specific to an individual execution instance.
5. **Case Data:** Data elements are supported which are specific to a process instance or case. They can be accessed by all components of the process during the execution of the case.
6. **Folder Data:** Data elements can be defined which are accessible by multiple cases on a selective basis. They are accessible to all components of the cases to which they are bound.
7. **Workflow Data:** Data elements are supported which are accessible to all components in each and every case of the process and are within the context of the process itself.
8. **Environment Data:** Data elements which exist in the external operating environment are able to be accessed by components of processes during execution.
9. **Task to Task:** The ability to communicate data elements between one task instance and another within the same case. The communication of data elements between two tasks is specified in a form that is independent of the task definitions themselves.
10. **Block Task to Sub-Workflow Decomposition:** The ability to pass data elements from a block task instance to the corresponding subprocess that defines its implementation. Any data elements that are available to a block task are able to be passed to (or be accessed) in the associated subprocess although only a specifically nominated subset of those data elements are actually passed to the subprocess.
11. **Sub-Workflow Decomposition to Block Task:** The ability to pass data elements from the underlying subprocess back to the corresponding block task. Only nominated data elements defined as part of the subprocess are made available to the (parent) block task.
12. **To Multiple Instance Task:** The ability to pass data elements from a preceding task instance to a subsequent task which is able to support multiple execution instances. This may involve passing the data elements to all instances of the multiple instance task or distributing them on a selective basis. The data passing occurs when the multiple instance task is enabled.
13. **From Multiple Instance Task:** The ability to pass data elements from a task which supports multiple execution instances to a subsequent task. The data passing occurs at the conclusion of the multiple instance task. It involves aggregating data elements from all instances of the task and passing them to a subsequent task.

14. **Case to Case:** The passing of data elements from one case of a process during its execution to another case that is executing concurrently.

15. **Task to Environment - Push:** The ability of a task to initiate the passing of data elements to a resource or service in the operating environment.

16. **Environment to Task - Pull:** The ability of a task to request data elements from resources or services in the operational environment.

17. **Environment to Task - Push:** The ability for a task to receive and utilise data elements passed to it from services and resources in the operating environment on an unscheduled basis.

18. **Task to Environment - Pull:** The ability of a task to receive and respond to requests for data elements from services and resources in the operational environment.

19. **Case to Environment - Push:** The ability of a case to initiate the passing of data elements to a resource or service in the operational environment.

20. **Environment to Case - Pull:** The ability of a case to request data from services or resources in the operational environment.

21. **Environment to Case - Push:** The ability of a case to accept data elements passed to it from services or resources in the operating environment.

22. **Case to Environment - Pull:** The ability of a case to respond to requests for data elements from a service or resource in the operating environment.

23. **Workflow to Environment - Push:** The ability of a process environment to pass data elements to resources or services in the operational environment.

24. **Environment to Workflow - Pull:** The ability of a process environment to request global data elements from external applications.

25. **Environment to Workflow - Push:** The ability of services or resources in the operating environment to pass global data to a process.

26. **Workflow to Environment - Pull:** The ability of the process environment to handle requests for global data from external applications.

27. **Data Transfer by Value - Incoming:** The ability of a process component to receive incoming data elements by value avoiding the need to have shared names or common address space with the component(s) from which it receives them.

28. **Data Transfer by Value - Outgoing:** The ability of a process component to pass data elements to subsequent components as values avoiding the need to have shared names or common address space with the component(s) to which it is passing them.

29. **Data Transfer - Copy In/Copy Out:** The ability of a process component to copy the values of a set of data elements from an external source (either within or outside the process environment) into its address space at the commencement of execution and to copy their final values back at completion.

30. **Data Transfer by Reference - Unlocked:** The ability to communicate data elements between process components by utilizing a reference to the location of the data element in some mutually accessible location. No concurrency restrictions apply to the shared data element.

31. **Data Transfer by Reference - With Lock:** The ability to communicate data elements between process components by passing a reference to the location of the data element in some mutually accessible location. Concurrency restrictions are implied with the receiving component receiving the privilege of read-only or dedicated access to the data element. The required lock is declaratively specified as part of the data passing request.

32. **Data Transformation - Input:** The ability to apply a transformation function to a data element prior to it being passed to a process component. The transformation function has access to the same data elements as the receiving process component.

33. **Data Transformation - Output:** The ability to apply a transformation function to a data element immediately prior to it being passed out of a process component. The transformation function has access to the same data elements as the process component that initiates it.

34. **Task Precondition - Data Existence:** Data-based preconditions can be specified for tasks based on the presence of data elements at the time of execution. The preconditions can utilize any data elements available to the task with which they are associated. A task can only proceed if the associated precondition evaluates positively.

35. **Task Precondition - Data Value:** Data-based preconditions can be specified for tasks based on the value of specific parameters at the time of execution. The preconditions can utilize any data elements available to the task with which they are associated. A task can only proceed if the associated precondition evaluates positively.

36. **Task Postcondition - Data Existence:** Data-based postconditions can be specified for tasks based on the existence of specific parameters at the time of task completion. The postconditions can utilize any data elements available to the task with which they are associated. A task can only proceed if the associated postcondition evaluates positively.

37. **Task Postcondition - Data Value:** Data-based postconditions can be specified for tasks based on the value of specific parameters at the time of execution. The postconditions can utilize any data elements available to the task with which they are associated. A task can only proceed if the associated postcondition evaluates positively.

38. **Event-Based Task Trigger:** The ability for an external event to initiate a task and to pass data elements to it.

39. **Data-Based Task Trigger:** Data-based task triggers provide the ability to trigger a specific task when an expression based on data elements in the process instance evaluates to true. Any data element accessible within a process instance can be used as part of a data-based trigger expression.

40. **Data-Based Routing:** Data-based routing provides the ability to alter the control-flow within a case based on the evaluation of data-based expressions. A data-based routing expression is associated with each outgoing arc of an OR-split or XOR-split. It can be composed of any data-values, expressions and functions available in the process environment providing it can be evaluated at the time the split construct with which it is associated completes. Depending on whether the construct is an XOR-split or OR-split, a mechanism is available to select one or several outgoing arcs to which the thread of control should be passed based on the evaluation of the expressions associated with the arcs.

A.3 Ressourcenmuster (Resource Patterns)

	BPMN	AD
Creation Patterns		
1. Direct Allocation	+	+
2. Role-based Allocation	+	+
3. Deferred Allocation	–	–
4. Authorization	–	–
5. Separation on Duties	–	–
6. Case Handling	–	–
7. Retain Familiar	–	–
8. Capability-based Allocation	–	–
9. History-based Allocation	–	–
10. Organizational Allocation	–	–
11. Automatic Execution	+	+
Push Patterns		
12. Distribution by Offer-Single Resource	–	–
13. Distribution by Offer-Multiple Resources	–	–
14. Distribution by Allocation-Single Resource	+	+
15. Random Allocation	–	–
16. Round Robin Allocation	–	–
17. Shortest Queue	–	–
18. Early Distribution	–	–
19. Distribution on Enablement	+	+
20. Late Distribution	–	–
Pull Patterns		
21. Resource-Initiated Allocation	–	–
22. Resource-Initiated Execution - Allocated Work Item	–	–
23. Resource-Initiated Execution - Offered Work Item	–	–
24. System-Determined Work Queue Content	–	–
25. Resource-Determined Work Queue Content	–	–
26. Selection Autonomy	–	–
Detour Patterns		
27. Delegation	–	–
28. Escalation	–	–
29. Deallocation	–	–
30. Stateful Reallocation	–	–
31. Stateless Reallocation	–	–
32. Suspension / Resumption	–	–
33. Skip	–	–
34. Redo	–	–
35. Pre-Do	–	–
Auto-Start Patterns		
36. Commencement on Creation	+	+
37. Commencement on Allocation	–	–
38. Piled Execution	–	–
39. Chained Execution	+	+
Visibility Patterns		
40. Configurable Unallocated Work Item Visibility	–	–
41. Configurable Allocated Work Item Visibility	–	–
Multiple Resource Patterns		
42. Simultaneous Execution	+	+
43. Additional Resource	–	–

Tabelle A.3: Unterstützung der Ressourcenmuster des Workflow Pattern Frameworks [Wohed et al., 2006, S. 13]

Ein „+“ in den Tabellen bedeutet direkte Unterstützung des Musters, ein „+/–“ eine teilweise Unterstützung und ein „–“ ein Fehlen der Unterstützung.

1. **Direct Distribution:** The ability to specify at design time the identity of the resource(s) to which instances of this task will be distributed at runtime.

2. **Role-Based Distribution:** The ability to specify at design-time one or more roles to which instances of this task will be distributed at runtime. Roles serve as a means of grouping resources with similar characteristics. Where an instance of a task is distributed in this way, it is distributed to all resources that are members of the role(s) associated with the task.

3. **Deferred Distribution:** The ability to specify at design-time that the identification of the resource(s) to which instances of this task will be distributed will be deferred until runtime.

4. **Authorization:** The ability to specify the range of privileges that a resource possesses in regard to the execution of a process. In the main, these privileges define the range of actions that a resource can initiate when undertaking work items associated with tasks in a process.

5. **Separation of Duties:** The ability to specify that two tasks must be executed by different resources in a given case.

6. **Case Handling:** The ability to allocate the work items within a given case to the same resource at the time that the case is commenced.

7. **Retain Familiar:** Where several resources are available to undertake a work item, the ability to allocate a work item within a given case to the same resource that undertook a preceding work item.

8. **Capability-Based Distribution:** The ability to distribute work items to resources based on specific capabilities that they possess. Capabilities (and their associated values) are recorded for individual resources as part of the organizational model.

9. **History-Based Distribution:** The ability to distribute work items to resources on the basis of their previous execution history.

10. **Organisational Distribution:** The ability to distribute work items to resources based their position within the organisation and their relationship with other resources.

11. **Automatic Execution:** The ability for an instance of a task to execute without needing to utilise the services of a resource.

12. **Distribution by Offer - Single Resource:** The ability to distribute a work item to a selected individual resource on a non-binding basis.

13. **Distribution by Offer - Multiple Resources:** The ability to distribute a work item to a group of selected resources on a non-binding basis.

14. **Distribution by Allocation - Single Resource:** The ability to distribute a work item to a specific resource for execution on a binding basis.

15. **Random Allocation:** The ability to allocate work items to a selected resource chosen from a group of eligible resources on a random basis.

16. **Round Robin Allocation:** The ability to allocate a work item to a selected resource chosen from a group of eligible resources on a cyclic basis.

17. **Shortest Queue:** The ability to allocate a work item to a selected resource chosen from a group of eligible resources on the basis of having the shortest work queue.

18. **Early Distribution:** The ability to advertise and potentially distribute a work items to resources ahead of the moment at which it is actually enabled.

19. **Distribution on Enablement:** The ability to advertise and distribute a work items to resources at the moment that the task to which it corresponds is enabled for execution.

20. **Late Distribution:** The ability to advertise and distribute work items to resources after the task to which the work item corresponds has been enabled for execution.

21. **Resource-Initiated Allocation:** The ability for a resource to commit to undertake a work item without needing to commence working on it immediately.

22. **Resource-Initiated Execution - Allocated Work Item:** The ability for a resource to commence work on a work item that is allocated to it.

23. **Resource-Initiated Execution - Offered Work Item:** The ability for a resource to select a work item offered to it and commence work on it immediately.

24. **System-Determined Work Queue Content:** The ability of the system to order the content and sequence in which work items are presented to a resource for execution.

25. **Resource-Determined Work Queue Content:** The ability for resources to specify the format and content of work items listed in the work queue for execution.

26. **Selection Autonomy:** The ability for resources to select a work item for execution based on its characteristics and their own preferences.

27. **Delegation:** The ability for a resource to allocate an unstarted work item previously allocated to it (but not yet commenced) to another resource.

28. **Escalation:** The ability of a system to distribute a work item to a resource or group of resources other than those it has previously been distributed to in an attempt to expedite the completion of the work item.

29. **Deallocation:** The ability of a resource (or group of resources) to relinquish a work item which is allocated to it (but not yet commenced) and make it available for distribution to another resource or group of resources.

30. **Stateful Reallocation:** The ability of a resource to allocate a work item that they are currently executing to another resource without loss of state data.

31. **Stateless Reallocation:** The ability for a resource to reallocate a work item that it is currently executing to another resource without retention of state.

32. **Suspension/Resumption:** The ability for a resource to suspend and resume execution of a work item.

33. **Skip:** The ability for a resource to skip a work item allocated to it and mark the work item as complete.

34. **Redo:** The ability for a resource to redo a work item that has previously been completed in a case. Any subsequent work items (i.e. work items that correspond to subsequent tasks in the process) must also be repeated.

35. **Pre-Do:** The ability for a resource to execute a work item ahead of the time that it has been offered or allocated to resources working on a given case. Only work items that do not depend on data elements from preceding work items can be "pre-done".

36. **Commencement on Creation:** The ability for a resource to commence execution on a work item as soon as it is created.

37. **Commencement on Allocation:** The ability to commence execution on a work item as soon as it is allocated to a resource.

38. **Piled Execution:** The ability to initiated the next instance of a task (perhaps in a different case) once the previous one has completed with all associated work items being allocated to the same resource. The transition to Piled Execution mode is at the instigation of an individual resource. Only one resource can be in Piled Execution mode for a given task at any time.

39. **Chained Execution:** The ability to automatically start the next work item in a case once the previous one has completed. The transition to Chained Execution mode is at the instigation of the resource.

40. **Configurable Unallocated Work Item Visibility:** The ability to configure the visibility of unallocated work items by process participants.

41. **Configurable Allocated Work Item Visibility:** The ability to configure the visibility of allocated work items by process participants.

42. **Simultaneous Execution:** The ability for a resource to execute more than one work item simultaneously.

43. **Additional Resources:** The ability for a given resource to request additional resources to assist in the execution of a work item that it is currently undertaking.

B Notationselemente der BPMN

Basiselemente der BPMN

In der BPMN gibt es zwölf Basiselemente, die variiert werden, um mehr Details eines Prozesses zu beschreiben. Im Folgenden werden diese zwölf Elemente vorgestellt. Als Referenz dienen die Spezifikation der BPMN 2.0[151] und die Publikationen von Freund und Rücker[152] sowie Göpfert und Lindenbach[153].

Ereignis

Ein Ereignis (Abbildung B.1) ist etwas, das im Verlauf eines Prozesses passiert. Die Ereignisse haben Einfluss auf den Ablauf des Modells und in der Regel eine Ursache (Trigger) oder eine Auswirkung (Ergebnis). Aus diesem Grund wird zwischen „werfenden“ (throwing) und „fangenden“ (catching) Ereignissen unterschieden. Ereignisse werden als Kreise dargestellt. In diese Kreise können Marker gesetzt werden, um eine Unterscheidung zwischen Triggern und Ergebnissen zu ermöglichen. Es gibt drei Arten von Ereignissen: Start-, Zwischen- und Endereignisse. Start- und Endereignis müssen theoretisch nicht modelliert werden, aber wenn es ein Startereignis gibt, so muss mindestens ein Endereignis modelliert werden. Im Rahmen dieser Arbeit werden die Start- und Endereignisse modelliert, um das Lesen der Diagramme zu vereinfachen.

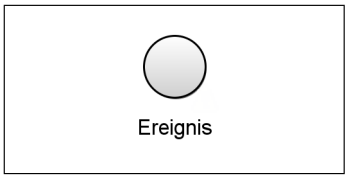


Abbildung B.1: Ereignis

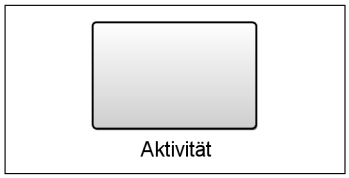


Abbildung B.2: Aktivität

[151] Vgl. OMG [2011a], S. 29f
[152] Vgl. Freund und Rücker [2012], S. 19ff
[153] Vgl. Göpfert und Lindenbach [2012], S. 4ff

Aktivität

Eine Aktivität (Abbildung B.2 auf der vorherigen Seite) ist ein Oberbegriff für Arbeit, die eine Firma ausführt. Eine Aktivität kann atomar oder nicht-atomar (zusammengesetzt) sein. Die Arten von Aktivitäten, die Teil eines Prozessmodells sind, sind: Prozess, Sub-Prozess und Aufgabe. Alle werden als abgerundete Rechtecke dargestellt.

Gateway

Das Gateway (Abbildung B.3) wird benutzt, um die Divergenz und die Konvergenz der Sequenzflüsse in einem Prozess zu kontrollieren. Auf diese Weise wird die Verzweigung und Zusammenführung der Pfade bestimmt. Interne Marker zeigen an, auf welche Art die Verzweigung oder Zusammenführung geschieht.

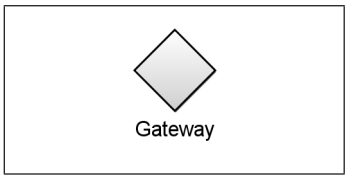


Abbildung B.3: Gateway

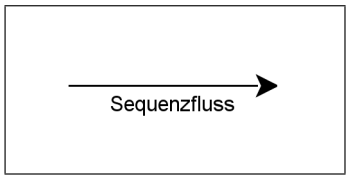


Abbildung B.4: Sequenzfluss

Sequenzfluss

Der Sequenzfluss (Abbildung B.4) wird benutzt, um die Reihenfolge darzustellen, in der Aktivitäten in einem Prozess ausgeführt werden. Er wird als durchgezogene Linie mit einer Pfeilspitze dargestellt. Das Diagramm kann sowohl horizontal (von links nach rechts) als auch vertikal (von oben nach unten) gezeichnet werden. Eine andere Richtung ist aus Gründen der Lesbarkeit nicht empfehlenswert.

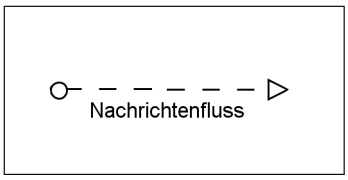


Abbildung B.5: Nachrichtenfluss

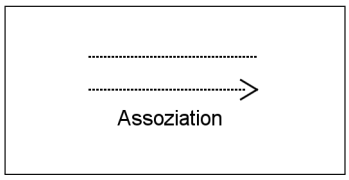


Abbildung B.6: Assoziation

Nachrichtenfluss

Ein Nachrichtenfluss (Abbildung B.5) wird verwendet, um den Fluss von Nachrichten zwischen zwei Teilnehmern zu zeigen, die bereit sind zu senden und zu empfangen. In der BPMN werden diese Teilnehmer durch zwei separate Pools im Diagramm dargestellt. Aus diesem Grund ist der Nachrichtenfluss ein Element der Kollaborationsdiagramme der BPMN. Für die Darstellung des Nachrichtenflusses

wird eine gestrichelte Linie mit einem nicht ausgefüllten Kreis auf der einen und einer nicht ausgefüllten Pfeilspitze auf der anderen Seite verwendet.

Assoziation

Eine Assoziation (Abbildung B.6 auf der vorherigen Seite) wird verwendet, um Informationen und Artefakte mit graphischen Elementen zu verknüpfen. Text-Anmerkungen und andere Artefakte können den graphischen Elementen zugeordnet werden. Eine Pfeilspitze an der Assoziation zeigt eine Richtung des Flusses an, wenn dies angemessen ist (beispielsweise bei Daten). Ansonsten wird nur eine gepunktete Linie verwendet.

Pool

Ein Pool (Abbildung B.7) ist die graphische Darstellung eines Teilnehmers. Ein Pool kann interne Details in Form eines Prozesses, der ausgeführt wird, enthalten. Oder ein Pool hat keine internen Details, das heißt, dass er beispielsweise als „Blackbox“[154] fungiert. Ein Pool besitzt eine Bezeichnung und wird als Rechteck dargestellt. Dabei kann er sowohl vertikal als auch horizontal verwendet werden. Bei der horizontalen Verwendung ist die Bezeichnung auf der linken Seite, bei der vertikalen oben.

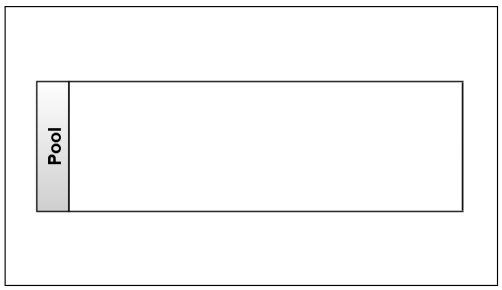


Abbildung B.7: Pool (horizontal)

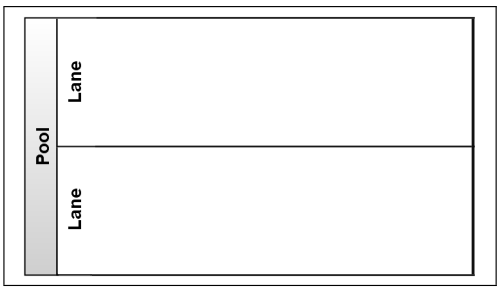


Abbildung B.8: Lane (horizontal)

Lane

Eine Lane (Abbildung B.8) ist ein Sub-Aufteilung innerhalb eines Pools. Sie erstreckt sich über die gesamte Länge des Prozesses, entweder vertikal oder horizontal. Lanes werden verwendet, um Aktivitäten zu organisieren und zu kategorisieren. Es ist auch möglich, mehrere Lanes in einer Lane zu modellieren, also eine tiefergehende Verschachtelung darzustellen.

[154] Ein Objekt, dessen innerer Aufbau beziehungsweise innere Funktionsweise unbekannt ist oder als nicht von Bedeutung erachtet wird. Es wird lediglich das äußere Verhalten betrachtet.

Datenobjekt

Ein Datenobjekt (Abbildung B.9) stellt Informationen darüber zur Verfügung, was eine Aktivität benötigt, um ausgeführt zu werden und / oder was diese erzeugt. Ein Datenobjekt kann ein einzelnes Objekt oder eine Sammlung von Objekten repräsentieren. Datenein- und -ausgabe stellen die gleichen Informationen für Prozesse zur Verfügung.

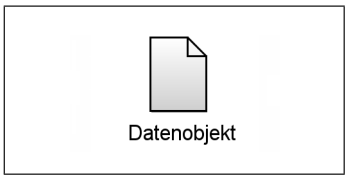


Abbildung B.9: Datenobjekt

Abbildung B.10: Nachricht

Nachricht

Eine Nachricht (Abbildung B.10) wird benutzt, um den Inhalt einer Kommunikation zwischen zwei Teilnehmern darzustellen.

Gruppe

Eine Gruppe (Abbildung B.11) ist eine Bündelung von grafischen Elementen, die innerhalb der gleichen Kategorie sind. Diese Art der Gruppierung hat keinen Einfluss auf den Sequenzfluss innerhalb der Gruppe. Die Kategorienname erscheint im Diagramm als Gruppen-Label. Kategorien können zur Dokumentation oder Analyse verwendet werden. Gruppen sind eine Möglichkeit, mit der Kategorien von Objekten visuell im Diagramm dargestellt werden können. Sie werden als abgerundetes Rechteck mit Punkt-Strich-Linie dargestellt. Auf dieses Element wird im Weiteren verzichtet, da es den Ablauf im Diagramm nicht beeinflusst.

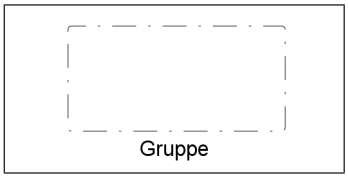


Abbildung B.11: Gruppe

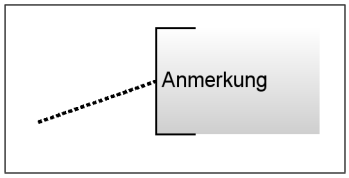


Abbildung B.12: Anmerkung

Anmerkung

Eine Anmerkung (Abbildung B.12) ist ein Mechanismus für einen Modellierer, um zusätzliche Textinformationen für den Leser eines Diagramms zur Verfügung zu stellen.

Erweiterte Elemente der BPMN

In der Spezifikation der BPMN 2.0 sind in der Liste der erweiterten Elemente[155] diverse Einträge, die die Basiselemente verfeinern. Von diesen werden hier nur die vorgestellt, die in dieser Arbeit im Weiteren verwendet werden. Elemente, die keine Variation haben, werden im Folgenden ebenfalls nicht aufgeführt.

Ereignis

Es gibt drei mögliche Ereignistypen: Start-, Zwischen- und Endereignisse. Bei den Start- und Zwischenereignissen wird zusätzlich zwischen unterbrechenden und nicht-unterbrechenden unterschieden. Die letztgenannten sind neu in der BPMN 2.0. Es wird ebenfalls zwischen angehefteten und nicht-angehefteten Zwischenereignissen unterschieden. Tabelle B.1 (auf der nachfolgenden Seite) zeigt alle in der BPMN definierten Ereignisse.

Die detaillierte Beschreibung der einzelnen Ereignisse kann in der Spezifikation der BPMN 2.0[156] nachgelesen werden. Alle Ereignisse werden als Kreis dargestellt. Während Startereignisse mit einer einfachen dünnen Linie umrandet werden, werden Zwischenereignisse mit einer doppelten Linie dargestellt. Endereignisse werden mit einer dicken Linie umrandet. Die nicht-unterbrechenden Ereignisse werden mit einer gestrichelten Linie dargestellt, um sie von den unterbrechenden zu unterscheiden. Zwischen fangenden und werfenden Ereignissen wird unterschieden, indem etwaige Marker bei den werfenden Ereignissen massiv und bei fangenden nur umrandet dargestellt werden.

Grundsätzlich könnte jedes Ereignis durch ein Blanko-Ereignis mit Anmerkung dargestellt werden. Dies würde jedoch die Lesbarkeit des Modells stark beeinträchtigen, wenn viele Ereignisse vorhanden sind. Einige der Ereignisse werden auch von anderen Notationselementen benötigt. Sofern möglich, wird auf die Verwendung der spezifizierten Ereignisse in dieser Arbeit verzichtet.

[155] Vgl. OMG [2011a], S. 31f
[156] Vgl. OMG [2011a], S. 238ff

	fangend		werfend		nicht-unterbrechend	
Blanko						
Nachricht						
Zeit						
Fehler						
Eskalation						
Abbruch						
Kompensation						
Bedingung						
Link						
Signal						
Terminierung						
Mehrfach						
Mehrfach/Parallel						

Tabelle B.1: Ereignistypen der BPMN

Aktivität

Als Aktivitäten gelten Prozesse, Sub-Prozesse und Aufgaben. Aufgaben (Abbildung B.13) sind die kleinsten Einheiten in einem Prozess und stellen eine atomare Aktivität dar, eine weitere Verfeinerung ist demnach nicht möglich. Um eine Aufgabe zu kennzeichnen, wird das Element mit einem Namen versehen.

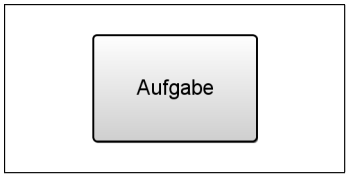


Abbildung B.13: Aufgabe

Zusätzlich kann die Aufgabe mit einem Symbol versehen werden, um die Art der Aufgabe zu definieren. Die BPMN definiert sieben Arten von Aufgaben: Service-Aufgabe, Skript-Aufgabe, Benutzer-Aufgabe, Manuelle Aufgabe, Empfangs-Aufgabe, Sende-Aufgabe sowie Geschäftsregel-Aufgabe. Im Weiteren werden in dieser Arbeit alle Aufgaben als unspezifiziert behandelt, das heißt, sie erhalten kein Symbol zur Definition der Aufgabe.

Ein Sub-Prozess (Abbildung B.14 bzw. Abbildung B.15) ist eine nicht-atomare, das heißt zusammengesetzte, Aktivität. Er kann Sub-Aktivitäten beinhalten, also wiederum Aufgaben oder Sub-Prozesse. Ein kollabierter Sub-Prozess hat ein kleines Kreuz in einem Quadrat an der unteren Seite, während in einem expandiertem Sub-Prozess die Elemente sichtbar sind, die er enthält. Wichtig ist, dass ein Sequenzfluss die Grenze eines Sub-Prozesses nicht überschreiten kann. Innerhalb eines Sub-Prozesses befindet sich also immer ein abgeschlossener (Teil-)Prozess.

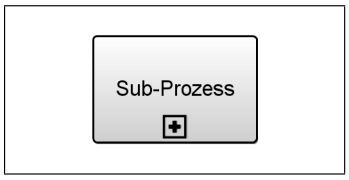


Abbildung B.14: Sub-Prozess (kollabiert)

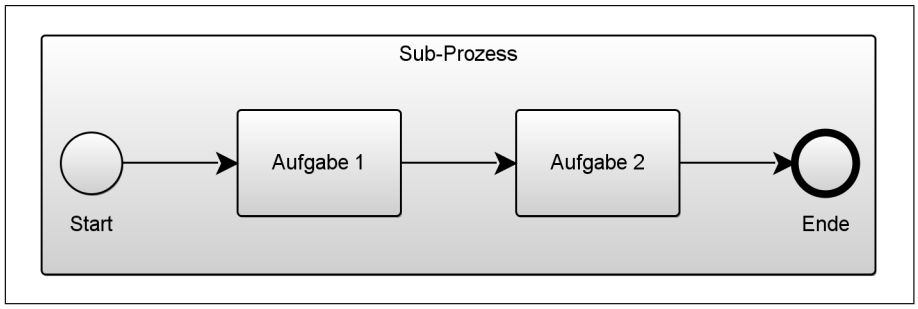


Abbildung B.15: Sub-Prozess (expandiert)

Globale Aktivität

Es ist durchaus möglich, dass einzelne Aktivitäten von mehreren anderen Aktivitäten benötigt werden. Um sie nicht für jede Aktivität einzeln zu modellieren, ist es möglich die Aktivität für andere aufrufbar zu machen. Dann handelt es sich um eine globale Aktivität. Sie wird durch eine dickere Umrandung gekennzeichnet und befindet sich nicht im eigentlichen Sequenzfluss. Die Aktivität wird vollständig ausgeführt, danach setzt sich der Sequenzfluss nach ihrem Aufruf wieder fort.

Multi-Instanzen

Es ist für Aktivitäten möglich, als Multi-Instanz aufzutreten. Dabei wird zwischen *paralleler* und *sequentieller* Ausführung unterschieden. Das Symbol für eine parallel Multi-Instanz stellen drei parallele Striche, die senkrecht verlaufen, dar (Abbildung B.16). Das Symbol für die sequentielle Multi-Instanz stellen drei parallele Striche, die waagerecht verlaufen, dar (Abbildung B.17). Die Abbildungen B.16 und B.17 dienen lediglich als Beispiel, es gibt auch sequentielle Aufgaben sowie parallele Sub-Prozesse.

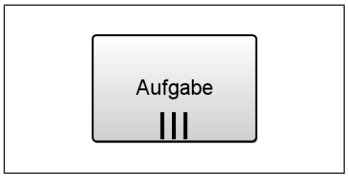


Abbildung B.16: parallel

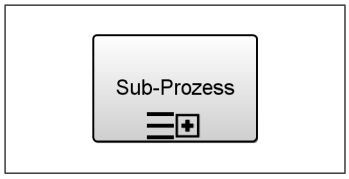


Abbildung B.17: sequentiell

Multi-Instanzen sollen der Effizienz dienen, indem mehrere Instanzen der Aufgabe gebildet werden. So kann eine parallele Aktivität beispielsweise von mehreren Mitarbeitern bearbeitet werden, bei einer sequentiellen wird die Aufgabe für jedes Objekt wiederholt.

Schleife

Es ist sowohl für Aktivitäten als auch Sub-Prozesse möglich, als Schleife aufzutreten. Das Symbol dafür ist ein kreisförmiger Pfeil (Abbildung B.18 und B.19). Auf diese spezielle Kennzeichnung einer Aktivität wird jedoch im Weiteren verzichtet, da das Verhalten der Schleife durch andere Notationselemente modelliert werden kann.

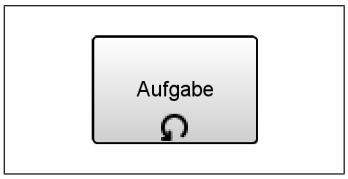


Abbildung B.18: Aufgabe (Schleife)

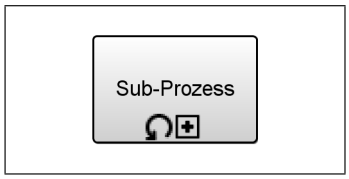


Abbildung B.19: Sub-Prozess (Schleife)

Kompensation

Es gibt noch eine vierte spezielle Markierung: die Kompensation (Abbildung B.20 auf der nächsten Seite). Sie hat mit einem angehefteten Zwischenereignis zu tun: dem fangenden Kompensations-Zwischenereignis. Zwischen diesem Ereignis und der Kompensationsaktivität besteht eine gerichtete Assoziation, kein Sequenzfluss. Dadurch wird der Zusammenhang verdeutlicht. Wird nun irgendwo im Prozess ein werfendes Kompensations-Ereignis erreicht, so werden die entsprechenden

Kompensationsaktivitäten ausgeführt. Eine Kompensation wird benutzt, wenn ein Fehler aufgetreten ist und zuvor ausgeführte Aufgaben oder deren Ergebnisse rückgängig gemacht werden müssen.

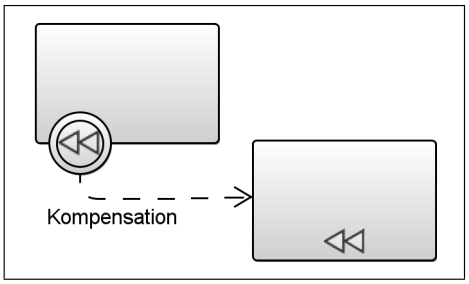


Abbildung B.20: Kompensation

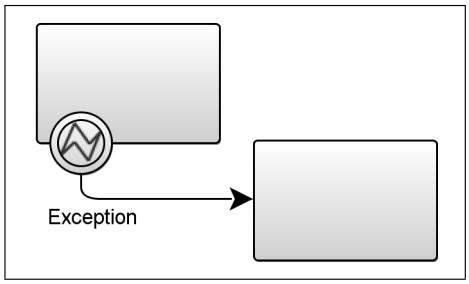


Abbildung B.21: Fehlerfluss

Sequenzfluss

Auch der Sequenzfluss kann in der BPMN verfeinert werden, indem er als „Default"-Fluss (Standardfluss) oder als bedingter Sequenzfluss gekennzeichnet wird. Die Abbildungen B.22 und B.23 zeigen die beiden Darstellungen. Auf beide wird im Folgenden verzichtet, da die Verhaltensmuster über Gateways modelliert werden können.

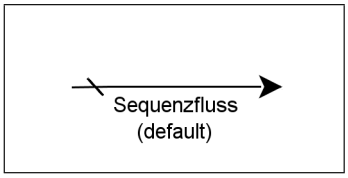


Abbildung B.22: Standardfluss

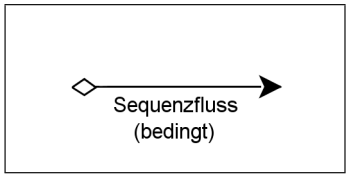


Abbildung B.23: bedingter Fluss

Es gibt noch eine dritte Art besonderen Sequenzfluss, nämlich den „Fehlerfluss" (Exception). Ein solcher Fluss hängt mit dem Fehler-Zwischenereignis zusammen, das an eine Aktivität geheftet wurde. Der Fehlerfluss wird ausgelöst, wenn es zu einem Fehler im normalen Sequenzfluss gekommen ist. In Abbildung B.21 ist der Fehlerfluss dargestellt, er sieht der Kompensation sehr ähnlich.

Gateway

Um den Sequenzfluss zu lenken, werden in der BPMN mehrere Gateways unterschieden: exklusives Gateway, inklusives Gateway, paralleles Gateway, komplexes Gateway sowie ereignis-basiertes Gateway. Die meisten dieser Gateways werden sowohl zur Verzweigung als auch zur Zusammenführung des Flusses genutzt, sodass sie zwei Verhaltensmuster aufweisen. Sie können diese Verhaltensmuster auch gleichzeitg aufweisen, also zusammenführen und verzweigen in einem Schritt. Dieses Verhalten kann allerdings auch durch zwei hintereinander gesetzte Gateways modelliert werden und wird daher auf diese Art und Weise modelliert. Ebenfalls

wird in dieser Arbeit darauf geachtet, dass nach Möglichkeit ein verzweigendes Gateway immer seinen korrespondierenden Partner hat, das heißt, dass es zu jedem verzweigenden auch ein zusammenführendes Gateway der gleichen Art gibt. Manchmal soll jedoch ein Verhalten modelliert werden, das so nicht umsetzbar ist. In diesem Fall wird davon abgewichen.

Beim exklusiven Gateway (Abbildung B.24) wird der Sequenzfluss abhängig von den Bedingungen an genau eine ausgehende Kante weitergeleitet. Die Bedingungen werden nacheinander geprüft. Somit wird eine „XOR"-Beziehung modelliert. Wenn dieses Gateway benutzt wird, um den Fluss zusammenführen, dann wird jedes Mal, wenn eine Kante eine Marke mitbringt, der Sequenzfluss fortgesetzt. Es gibt zwei Möglichkeiten, um das exklusive Gateway darzustellen: entweder ohne internen Marker, also als „Blanko-Gateway", oder durch das „X" in der Raute. Um Verwirrung vorzubeugen, wird in dieser Arbeit im Weiteren nur die Rautendarstellung mit dem „X" verwendet.

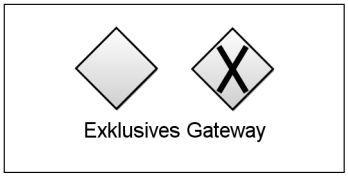


Abbildung B.24: Exklusives Gateway

Abbildung B.25: Inklusives Gateway

Das inklusive Gateway (Abbildung B.25) wird durch einen Kreis in der Raute gekennzeichnet. Eingesetzt bei der Verzweigung wird je nach Bedingung mindestens eine, gegebenenfalls aber auch alle ausgehenden Kanten ausgewählt. Hier wird demnach eine „OR"-Beziehung modelliert. Bei der Zusammenführung wird auf so viele Kanten gewartet, wie bei der Verzweigung aktiviert wurden, also mindestens auf eine, aber gegebenenfalls auch alle eingehenden Kanten.

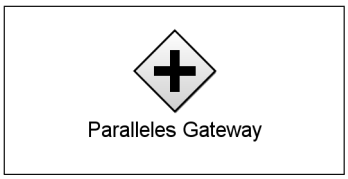


Abbildung B.26: Paralleles Gateway

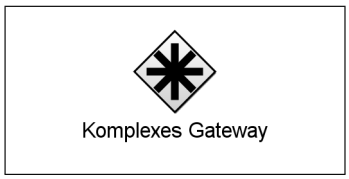


Abbildung B.27: Komplexes Gateway

Das parallele Gateway (Abbildung B.26) modelliert die „AND"-Beziehung. Bei der Verzweigung werden alle ausgehenden Kanten aktiviert, was einer Parallelisierung des Ablaufes entspricht. Das gleiche Verhalten kann erzeugt werden, indem aus einer Aktivität einfach zwei Sequenzflüsse herauskommen. Um Verwirrung vorzubeugen, wird in dieser Arbeit im Weiteren für dieses Verhalten immer das Gateway benutzt. Bei der Zusammenführung mit diesem Gateway wird auf alle eingehenden Kanten gewartet, ehe der Sequenzfluss fortgesetzt wird. Dieses Gateway hat als internen

Marker ein Plus.

Das komplexe Gateway (Abbildung B.27 auf der vorangegangenen Seite) wird benutzt, um ein Verzweigungs- und Zusammenführungsverhalten zu modellieren, das nicht von den anderen Gateways erfasst wird. Das Gateway wird durch einen Asterisk gekennzeichnet.

Es gibt noch drei weitere Gateways in der BPMN: das ereignis-basierte Gateway, das exklusive ereignis-basierte Gateway sowie das parallele ereignis-basierte Gateway. Das erstgenannte Gateway (Abbildung B.28) erscheint im normalen Sequenzfluss, wenn eine Entscheidung von eintretenden Ereignissen bestimmt wird. Ein solches Ereignis kann beispielsweise der Erhalt einer Nachricht eines anderen Teilnehmers sein. Es müssen mindestens zwei ausgehende Kanten an dieser Verzweigung anliegen, aber es wird nur eine ausgewählt.

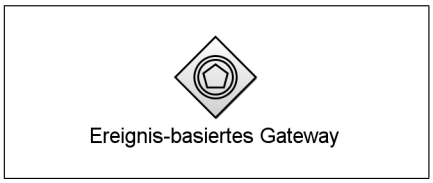


Abbildung B.28: Ereignis-basiertes Gateway

Das ereignis-basierte Gateway gibt es auch als „Startereignis“ (Abbildung B.29). Dann soll es anzeigen, dass mehrere Ereignisse einen Prozess auslösen können. Sobald eines der Ereignisse eintritt, wird der normale Sequenzfluss aufgenommen. Da dieser Umstand allerdings auch mit mehreren Startereignissen modelliert werden kann, wird auf seine Verwendung im Folgenden verzichtet.

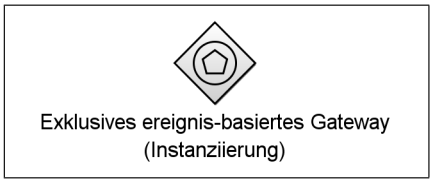


Abbildung B.29: Exklusives ereignis-basiertes Gateway (Instanziierung)

Das in der BPMN 2.0 neu hinzugekommende parallele ereignis-basierte Gateway (Abbildung B.30 auf der nächsten Seite) funktioniert ähnlich wie das exklusive ereignis-basierte Gateway. Beim Eintreten eines Ereignisses wird der Prozess inizi-iert, das heißt ins Leben gerufen. Im Gegensatz zur exklusiven ereignis-basierten Version wartet das Gateway jetzt jedoch darauf, dass auch alle Ereignisse eintreten, ehe der normale Sequenzfluss aufgenommen wird.

Grundsätzlich kann jedes Gateway durch ein Blanko-Gateway mit Anmerkung dargestellt werden. Dies würde jedoch die Lesbarkeit des Modells stark beeinträchtigen, wenn viele Verzweigungen und Zusammenführungen vorhanden sind.

Abbildung B.30: Paralleles ereignis-basiertes Gateway (Instanziierung)

Ad-hoc-Sub-Prozess

Es gibt für Sub-Prozesse noch ein paar mehr spezielle Markierungen als für Aufgaben. Darunter befindet sich der Ad-hoc-Sub-Prozess. Er wird durch eine Tilde (Abbildung B.31) gekennzeichnet. Dieser Sub-Prozess-Typ ist nicht vollständig definiert, denn es werden in ihm keine Angaben zur Reihenfolge oder zur Anzahl der Wiederholung einzelner Aktivitäten gemacht. Im Weiteren wird auf seine Verwendung daher wenn möglich verzichtet.

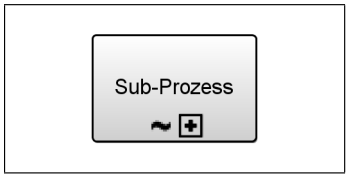


Abbildung B.31: Ad-hoc-Sub-Prozess

Transaktions-Sub-Prozess

Ein weiterer Spezialfall für Sub-Prozesse ist der Transaktions-Sub-Prozess (Abbildung B.32). Er wird durch einen doppelten Rand gekennzeichnet, außerdem wird ein fangendes Abbruch-Zwischenereignis angeheftet. Innerhalb des Sub-Prozesses gibt es ein Abbruch-Enderegnis. Wird dieses erreicht, werden alle Aktivitäten abgebrochen und alle beinhalteten kompensatorischen Aktivitäten ausgeführt. Damit versucht dieser Sub-Prozess das „ganz-oder-gar-nicht“-Prinzip zu realisieren. Anschließend geht der Sequenzfluss beim fangenden Abbruch-Zwischenereignis weiter. Die Abbruch-Ereignisse treten nur im Zusammenhang mit dem Transaktions-Sub-Prozess auf.

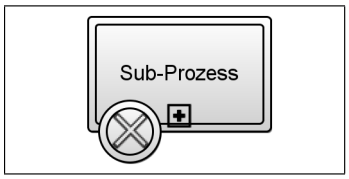


Abbildung B.32: Transaktions-Sub-Prozess

Ereignis-basierter Sub-Prozess

Diese Sub-Prozesse werden durch eine gestrichtelten Umrandung und durch ein fangendes unterbrechendes oder ein nicht-unterbrechendes Startereignis in der oberen linken Ecke gekennzeichnet. Abbildung B.33 zeigt beispielhaft einen Ereignis-Sub-Prozess mit einem nicht-unterbrechendem Nachrichten-Ereignis. Das bedeutet, dass der übergeordnete Prozess nicht abgebrochen wird, wenn dieser Sub-Prozess beendet ist. Ist das Ereignis unterbrechend, so wird der übergeordnete Prozess abgebrochen und der Sequenzfluss wird nach ihm fortgesetzt, sobald der Sub-Prozess beendet wurde.

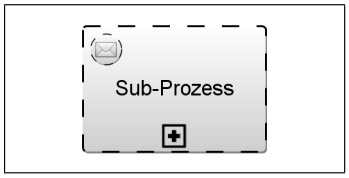


Abbildung B.33: Ereignis-Sub-Prozess

Datenobjekt

In Abschnitt zu den Basiselementen der BPMN wurde bereits alles zu den Datenobjekten gesagt, was zu ihnen wichtig ist. Lediglich ihre Darstellung kann verfeinert werden. In Abbildung B.36 wird eine Sammlung von Daten verbildlicht, während in Abbildung B.35 die Eingabe beziehungsweise das Ergebnis von Daten dargestellt wird. Als Eingabe werden Daten markiert, die benötigt werden. Als Ergebnis werden Daten markiert, die erzeugt werden. Die Darstellungen Sammlung und Eingabe beziehungsweise Ergebnis können auch kombiniert werden, um Sammlungen als Eingabe beziehungsweise Ergebnis zu kennzeichnen. Daten, die auch noch nach Beendigung einer Prozess-Instanz verfügbar sein sollen, werden in Datenlagern (Abbildung B.36) abgelegt.

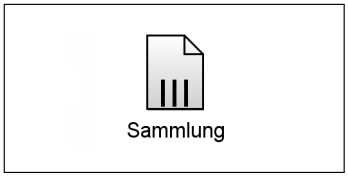


Abbildung B.34: Sammlung

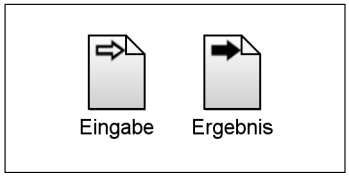


Abbildung B.35: Eingabe und Ergebnis

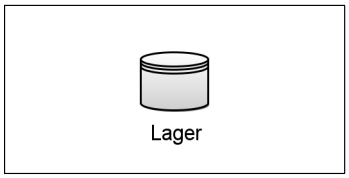


Abbildung B.36: Lager

C Notationselemente der UML AD

Im Folgenden werden die Notationselemente der UML Aktivitätsdiagramme vorgestellt. Weiterführende Informationen zu den einzelnen Elementen können in der Superstruktur der UML 2.4.1[157] sowie in den Publikationen von Fowler[158], Kecher[159] und Oestereich[160] nachgeschlagen werden.

Aktion und Aktivität

Eines der wichtigsten Elemente der Aktivitätsdiagramme ist die Aktion (Abbildung C.1). Bei einer Aktion handelt es sich um eine im Modell atomare Einheit, die nicht weiter zerlegt wird. Sie wird als Rechteck mit abgerundeten Ecken dargestellt und besitzt einen Aktionsnamen. Durch Anmerkungen (Abbildung C.17 auf Seite 161) kann die Aktion mit Bedingungen versehen werden, die vor oder nach ihr eingetreten sein müssen beziehungsweise eintreten werden.

Aktionen können in Aktivitäten eingebettet sein (Abbildung C.2 und Abbildung C.3 auf der nächsten Seite). Aktivitäten werden ebenfalls als Rechteck mit abgerundeten Ecken dargestellt. Wenn eine Aktivität zusammengeklappt ist, ist dies an dem kleinen Symbol in der unteren rechten Ecke zu erkennen. Ist sie nicht zusammengeklappt, so sind die enthaltenden Aktionen und gegebenenfalls Objektknoten (Abbildung C.4 auf der nächsten Seite) zu erkennen.

Abbildung C.1: Aktion

Abbildung C.2: Aktivität (kollabiert)

[157] Vgl. OMG [2011b], S.319ff
[158] Vgl. Fowler [2003], S. 29ff
[159] Vgl. Kecher [2006], S. 19ff
[160] Vgl. Oestereich [2012], S. 237ff

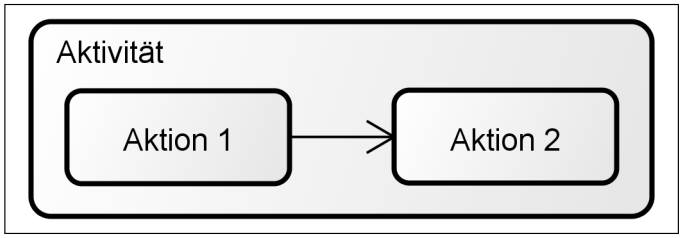


Abbildung C.3: Aktivität (expandiert)

Objektknoten

Ein Objektknoten (Abbildung C.4) stellt eine Art Speicher für Objekte dar. Diese werden entweder von einer Aktion benötigt oder erzeugt. Aus diesem Grund können sie als Parameter auf eine Aktivitätsgrenze gesetzt werden. In diesem Fall werden die Knoten entsprechend mit Stereotypen gekennzeichnet. Objektknoten werden als einfaches Rechteck dargestellt. Es gibt jedoch noch eine weitere Darstellungsform: Pins (Abbildung C.5). Dies sind kleine Quadrate auf der linken oder rechten Seite einer Aktion. Um Verwirrung vorzubeugen, wird in dieser Arbeit im Weiteren nur die Darstellung mit dem Objektknoten verwendet.

Abbildung C.4: Objektknoten

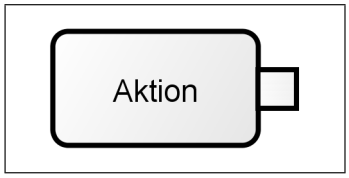


Abbildung C.5: Aktion mit Pin (rechts)

Objekte können über einen Zustand verfügen. Dieser wird in eckigen Klammern unter den Objektnamen gesetzt. Weiterhin können sie eine Angabe haben, wie viele Objekte sich maximal in dem Knoten befinden können. Diese Angaben werden in geschweiften Klammern unter den Objektknoten geschrieben. Wenn ein Objektknoten von mehreren Aktionen benutzt wird, so gilt er entweder als zentraler Puffer oder als Datenspeicher. In letzterem Fall werden lediglich Kopien der Objekte weitergegeben statt der Objekte selbst. Und noch eine weitere Funktion wird vom Objektknoten erfüllt: Er kann als Exception-Objektknoten (Fehler) dienen. Kommt es zu einem Fehler wird eine entsprechende Aktion benachrichtigt, die den Fehler behandelt.

Kontroll- und Objektfluss

Der Kontrollfluss (Abbildung C.6) definiert die Ausführungsreihenfolge von Aktionen im Diagramm, indem er diese gerichtet miteinander verbindet. Alternativ verbindet der Kontrollfluss beispielsweise auch Aktionen mit Konnektoren (Abbildung C.7) oder Entscheidungs- und Verbindungsknoten (Abbildung C.14). Konnektoren sind ein Hilfsmittel, um die Übersichtlichkeit im Diagramm beizubehalten. Sie werden beispielsweise benutzt, um zu verhindern, dass sich Kontrollflüsse kreuzen und es zu Verwirrung kommt.

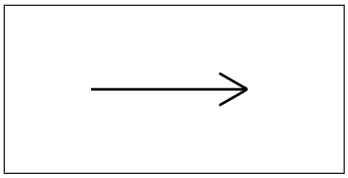


Abbildung C.6: Fluss

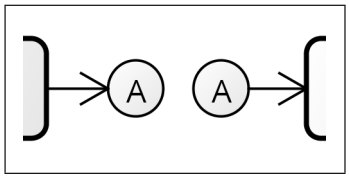


Abbildung C.7: Konnektor

Über das vom Aussehen her gleiche Notationselement werden aber auch Aktionen mit Objektknoten und umgekehrt verbunden. In diesem Fall spricht man vom Objektfluss. An diesen kann in geschweiften Klammern ein Gewicht geschrieben werden, das anzeigt, wie viele Objekte nötig sind, um den Prozess fortzusetzen.

Signal-Sendung und Signal-Empfang

Einige Aktionen werden erst ausgelöst, wenn sie ein entsprechendes Signal erhalten. Die Abbildungen C.8 und C.9 zeigen das Senden des Signals „A“ und das Empfangen des Signals „A“. Diese Signale werden verwendet, um einen asynchronen Nachrichtenaustausch zu modellieren. Das Signal zur Sendung kann von mehreren Aktionen aufgerufen werden, währenddessen das Empfangssignal nur von einer einzigen Aktion aufgefangen werden darf, da ansonsten der weitere Fluss im Diagramm nicht vorhersehbar ist.

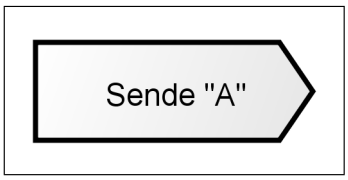


Abbildung C.8: Signal senden

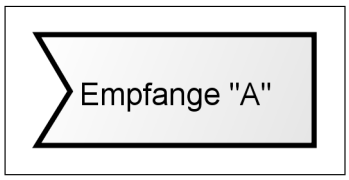


Abbildung C.9: Signal empfangen

Taucht ein Signal immer zum gleichen Zeitpunkt auf, so kann das zeitliche Ereignis (Abbildung C.10 auf der nächsten Seite) benutzt werden. In der Abbildung aktiviert dieses Signal zum Beispiel immer um 22 Uhr eine Aktion. Es ist jedoch auch möglich, das Signal immer nach einem bestimmten Zeitraum (beispielsweise nach 20 Minuten) zu aktivieren.

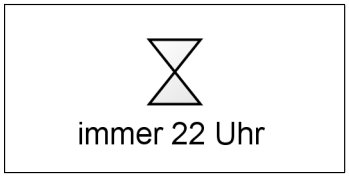


Abbildung C.10: Zeitliches Signal

Start- und Endknoten

Um zu signalisieren, wo ein Aktivitätsdiagramm beziehungsweise eine Aktivität gestartet und beendet wird, gibt es drei Symbole: einen Startknoten (Abbildung C.11) und zwei Endknoten (Abbildungen C.12 und C.13). Der Startknoten wird als ausgefüllter Kreis dargestellt. Der eine Endknoten beendet alle Kontrollflüsse in der Aktivität beziehungsweise dem Aktivitätsdiagramm. Er wird als Kreis mit doppeltem Rand dargestellt. Der innere Kreis ist dabei ausgefüllt. Der andere Endknoten beendet lediglich den Kontrollfluss, an den er angeschlossen ist. Er wird als einfacher Kreis mit einem „X“ dargestellt.

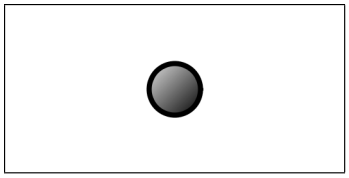


Abbildung C.11: Startknoten

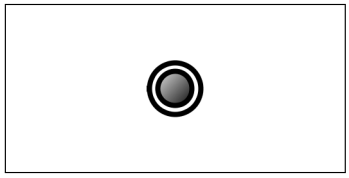


Abbildung C.12: Endknoten (Aktivitätsende)

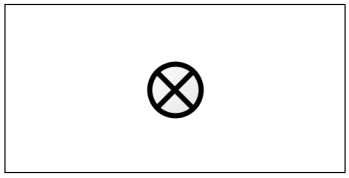


Abbildung C.13: Endknoten (Flussende)

Kontrollflusslenkung

Um den Kontrollfluss im Diagramm zu lenken, gibt es zum einen die Kontrollknoten: Entscheidungs- und Verbindungsknoten. Zum anderen gibt es die Gabelung und Vereinigung.

Ein Entscheidungsknoten (Abbildung C.14 auf der nächsten Seite) hat einen eingehenden Kontrollfluss und beliebig viele ausgehende. Anhand von Wächtern (Guards) wird exklusiv entschieden, welcher ausgehende Kontrollfluss gewählt wird. Aus diesem Grund müssen die Guards einander ausschließend entworfen werden. Somit wird hier die „XOR“-Beziehung modelliert. Die Wächter werden

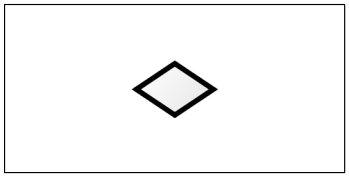


Abbildung C.14: Entscheidungs- und Verbindungsknoten

in eckigen Klammern an die ausgehenden Kontrollflüsse geschrieben. Die Entscheidungsgrundlage kann als Anmerkung an den Knoten modelliert werden. Wenn das Notationselement als Verbindungsknoten (Abbildung C.14) dient, so werden alternative Kontrollflüsse zusammengefasst. Der Knoten kann auch beide Aufgaben zeitgleich ausführen, also sowohl zusammenführen als auch verzweigen. Auf diese Möglichkeit wird im Folgenden verzichtet. Es werden immer zwei Knoten gezeichnet, um dies zu modellieren. Die Entscheidungs- und Verbindungsknoten werden in der UML auch benutzt, um Schleifen zu modellieren.

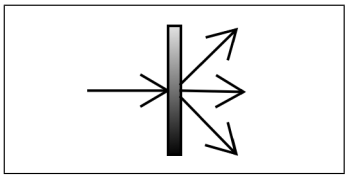


Abbildung C.15: Gabelung

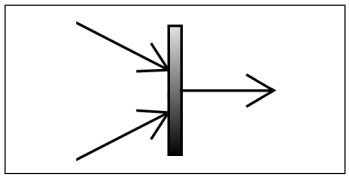


Abbildung C.16: Vereinigung

Alternativ zu den Knoten kann der Kontrollfluss auch mittels Gabelung und Vereinigung (Abbildungen C.15 und C.16) gelenkt werden. Bei der Gabelung wird standardmäßig die „AND“-Beziehung modelliert, bei der ein Kontrollfluss in beliebig viele (abhängig von der Anzahl der ausgehenden Kontrollflüsse) aufgeteilt wird. Die ausgehenden Kanten können jedoch Guards tragen, wodurch der Kontrollfluss nur auf solchen Kanten weiterwandert, deren Guard positiv ausgewertet wird.

Wenn keine anderen Angaben gemacht werden, wird auch bei der Vereinigung die „AND“-Beziehung umgesetzt, es muss also auf alle eingehenden Kontrollflüsse gewartet werden. Allerdings können auch Vereinigungsspezifikationen gemacht werden, sodass beispielsweise nur auf einen von mehreren („OR“-Beziehung) oder auf eine bestimmte Anzahl von Kontrollflüssen („n von m“-Beziehung) gewartet wird. Eine solche Spezifikation wird in geschweiften Klammern an den Balken geschrieben. Wie schon bei den Knoten zur Steuerung des Kontrollflusses kann auch dieses Notationselement beide Aufgaben zeitgleich ausführen. Auch hier wird darauf verzichtet. Es werden immer beide Elemente gezeichnet, um dies zu modellieren.

Im Gegensatz zur UML 1.x ist es in der UML 2.x nicht mehr nötig, dass nach einer Teilung des Kontrollflusses ein entsprechendes Gegenstück vorhanden sein muss. Aus Gründen des einfacheren Verständnisses wird in dieser Arbeit aber nach Möglichkeit immer das Gegenstück gezeichnet.

Anmerkung

Von der Anmerkung (Abbildung C.17) wurde bereits im Zusammenhang mit den Aktionen und den Entscheidungsknoten gesprochen. Eine Anmerkung wird als Rechteck mit umgeknickter, oberer rechter Ecke dargestellt. Sie kann Bedingungen und Erläuterungen darstellen, was durch verschiedene Stereotypen modelliert wird.

Abbildung C.17: Anmerkung

Aktivitätsbereich

Um Aktionen und Aktivitäten im Diagramm zu organisieren, können Aktivitätsbereiche beziehungsweise Partitionen (Abbildung C.18) eingesetzt werden. Diese werden als Rechteck entweder horizontal oder vertikal dargestellt und besitzen entweder links oder oben eine Bezeichnung. Auf diese Weise können zum Beispiel Verantwortlichkeiten festgelegt werden. Aktivitätsbereiche können verschachtelt sein, also weitere Aktivitätsbereiche enthalten. Dies ist in den Abbildungen C.19 und C.20 (auf der nachfolgenden Seite) dargestellt. Eine Aktion kann auch auf der Grenze zwischen zwei Aktivitätsbereichen liegen. Sie ist dann beiden Bereichen zugeordnet.

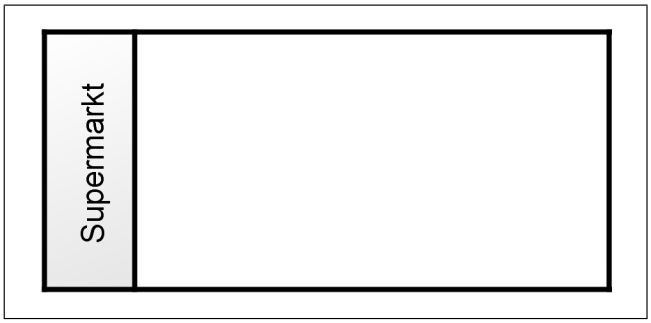


Abbildung C.18: Aktivitätsbereich

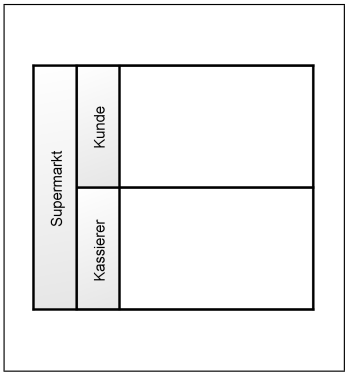


Abbildung C.19: Aktivitätsbereich (verschachtelt) (1)

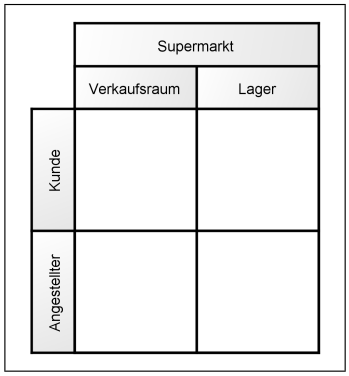


Abbildung C.20: Aktivitätsbereich (verschachtelt) (2)

Schleifen- und Bedingungsknoten

Auf die Verwendung beider Knotenarten wird im Weiteren verzichtet, da sie durchaus auch mit den bisher vorgestellten Notationselementen modelliert werden können und somit nur redundant wären.

Unterbrechungsbereich

Ein Unterbrechungsbereich (Abbildung C.21) umfasst eine Anzahl von Aktivitätsknoten, deren Ausführung zum Beispiel beim Erhalt einer Nachricht sofort abgebrochen wird. Natürlich kann der Abbruch auch durch einen der Aktivitätsknoten ausgelöst werden. Die Unterbrechungskante hat die Form eines Blitzes. Alternativ wird sie wie ein normaler Fluss mit einem Blitz-Symbol dargestellt. Auf letztere Darstellungsform wird im Weiteren verzichtet, um Verwirrung vorzubeugen.

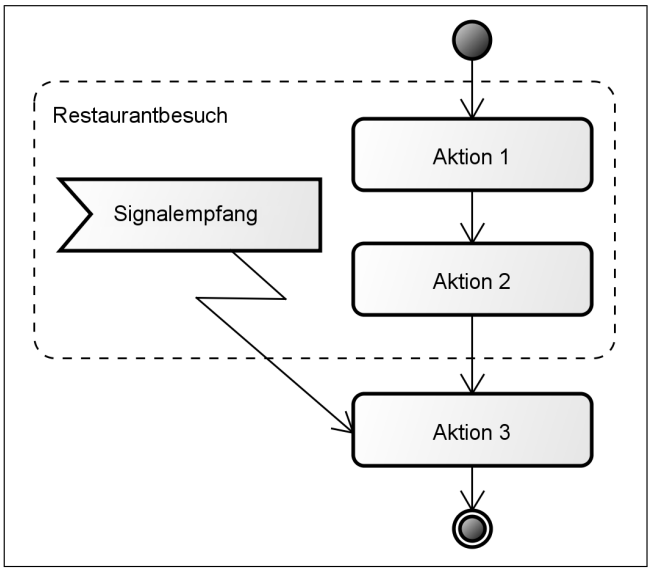


Abbildung C.21: Unterbrechungsbereich

D Ableitung der Bausteine aus der XML-Struktur der BPMN

Auf den folgenden Seiten wird die Ableitung der einzelnen Bausteine aus den jeweiligen XML-Strukturen der BPMN vollständig beschrieben. Dabei wird auf die selbe vereinfachte Darstellungsweise zurückgegriffen wie sie schon in der Arbeit selbst zum Einsatz gekommen ist.

Listing D.1 zeigt das Startereignis. Dieses ist eindeutig identifizierbar, da es als „startEvent“ deklariert wird. Es verwendet unter anderem die Attribute „id“ sowie „name“ und enthält die ID-Nummer der ausgehenden Kante als Element.

```
1 <startEvent id="_2" name="Start Event">
2   <outgoing>_4</outgoing>
3 </startEvent>
```

Listing D.1: Baustein #1a (vereinfachte XML-Struktur - BPMN)

Die BPMN definiert verschiedene Arten von Startereignissen, darunter Zeit- und Nachrichten-Startereignisse (siehe Tabelle B.1 auf Seite 148). Obwohl in dem Abschnitt festgelegt wurde, auf diese Spezialisierungen nach Möglichkeit zu verzichten, können sie als Anmerkungen abgespeichert werden. Um welche Spezialisierung es sich handelt wird durch eine Ereignisdefinition beschrieben. In Tabelle D.1 sind die Ereignisdefinitionen und die dazugehörigen speziellen Anmerkungen aufgeführt.

Ereignisdefinition	Anmerkung
messageEventDefinition	SPEC:MessageEventDefinition
timerEventDefinition	SPEC:TimerEventDefinition
errorEventDefinition	SPEC:ErrorEventDefinition
escalationEventDefinition	SPEC:EscalationEventDefinition
cancelEventDefinition	SPEC:CancelEventDefinition
compensateEventDefinition	SPEC:CompensateEventDefinition
conditionalEventDefinition	SPEC:ConditionalEventDefintion
linkEventDefinition	SPEC:LinkEventDefinition
signalEventDefinition	SPEC:SignalEventDefinition
terminateEventDefinition	SPEC:TerminateEventDefinition

Tabelle D.1: Spezielle Anmerkungen bei Ereignissen

Listing D.2 zeigt ein Start-Ereignis mit Ereignisdefinition. Das heißt: Wenn die Bausteine aus einem bestehenden Diagramm ausgelesen werden, können diese Spezialisierungen als Anmerkungen abgespeichert werden. Bei der Transformation können diese Anmerkungen dann als Anmerkungen auch im Aktivitätsdiagramm der UML stehen.

```
<startEvent id="_25" name="Start Event">
  <outgoing>_27</outgoing>
  <timerEventDefinition id="_25_ED_1"/>
</startEvent>
```

Listing D.2: Baustein #1a (Spezialisierung) (vereinfachte XML-Struktur - BPMN)

Es stellt sich nun die Frage, wie der Baustein #1a konkret aus der XML-Struktur heraus erstellt wird. Zunächst einmal wird hier nach dem Element „startEvent“ gesucht. Die ID-Nummer des entsprechendes Elements wird als ID-Nummer des Bausteins gesetzt. Danach werden die Elemente des Bausteins befüllt: In diesem Beispiel wird „outgoing“ direkt übernommen. Die Bezeichnung („labeling“) entspricht dem Namen des Start-Elements und die Ereignisdefinition wird als Anmerkung vermerkt. Listing D.3 zeigt den aus Listing D.2 abgeleiteten Baustein. Dabei wird die Anmerkung mit den Buchstaben „SPEC“, für „special“, eingeleitet. So wird aus dem Element „timerEventDefinition“ eine eindeutige Anmerkung. Später kann anhand dieser speziellen Anmerkungen die richtige Transformation durchgeführt werden.

```
<brick_01a id="_25">
  <outgoing>_27</outgoing>
  <labeling>Start Event</labeling>
  <annotation>SPEC:TimerEventDefinition</annotation>
</brick_01a>
```

Listing D.3: abgeleiteter Baustein #1a in XML

Ebenso einfach wie Baustein #1a lässt sich Baustein #1c in der XML-Struktur der BPMN finden (Listing D.4). Es handelt sich dabei um das „endEvent“, bei welchem die eingehenden Kanten ebenfalls mit ID-Nummer aufgeführt werden. Auch hier gibt eine Reihe von Spezialisierungen, die als Anmerkungen abgespeichert werden können. Aus diesen Anmerkungen heraus können sie später auch wieder hergestellt werden. Die Ableitung des Bausteins erfolgt auf die gleiche Art und Weise wie zuvor, nur dass dieses Mal „incoming“ auf „incoming“ abgebildet wird.

```
<endEvent id="_7" name="End Event">
  <incoming>_8</incoming>
</endEvent>
```

Listing D.4: Baustein #1c (vereinfachte XML-Struktur - BPMN)

Die Unterscheidung von Baustein #1c und Baustein #1b stellt eine Schwierigkeit dar, denn die BPMN kennt kein eigenständiges Flussende. Beim Vergleich der Workflow Patterns in Abschnitt 4.2.1 wurde an beiden Stellen das Endereignis verwendet. Bei der Ableitung der Bausteine aus einem bestehenden Diagramm kann keine Unterscheidung erfolgen. Wird das Diagramm aber im Gegenzug aus den Bausteinen selbst erstellt, so kann eine Unterscheidung dadurch herbeigeführt werden, dass das Endereignis eine Anmerkung mit eindeutigem Inhalt bekommt, beispielsweise durch ein Schlagwort (hier: „SPEC:FlowFinal“). Dies ist in Listing D.5 zu sehen. Der dazugehörige Baustein sieht in diesem Fall wie in Listing D.6 abgebildet aus.

```
<endEvent id="_7" name="End Event">
  <incoming>_8</incoming>
</endEvent>
<!-- ... -->
<textAnnotation id="_3" textFormat="text/plain">
  <text>SPEC:FlowFinal</text>
</textAnnotation>
<association id="_4" sourceRef="_7" targetRef="_3"/>
```

Listing D.5: Baustein #1b (vereinfachte XML-Struktur - BPMN)

```
<brick_01b id="_7">
  <incoming>_8</incoming>
  <labeling>End Event</labeling>
</brick_01b>
```

Listing D.6: abgeleiteter Baustein #1b in XML

Listing D.5 zeigt auch, wie Anmerkungen in der XML-Struktur der BPMN aussehen: Eine Anmerkung ist ein eigenständiges Element, welches über eine Assoziation an ein anderes Element gebunden ist. Dass es eine Anmerkung zu einem Element gibt, ist allerdings nur durch die Anmerkung an sich erkennbar; das Element selbst enthält keinen Hinweis darauf. Aus diesem Grund muss jedes Mal geprüft werden, ob es ein „association“-Element gibt, bei dem die Quell- oder Ziel-ID-Nummer mit der ID-Nummer des gerade betrachteten Elements übereinstimmen.

```
<task id="_5" name="Task A"/>
  <incoming>_9</incoming>
  <outgoing>_8</outgoing>
</task>
```

Listing D.7: Baustein #2a (vereinfachte XML-Struktur - BPMN)

Eine atomare Einheit der BPMN, also eine Aufgabe, wird in der XML-Struktur als „task“ bezeichnet und führt als Attribute eine ID-Nummer sowie den Namen. Ebenfalls werden die eingehenden und ausgehenden Kanten über deren jeweilige ID-Nummern gelistet. Dies ist in Listing D.7 dargestellt. Die Ableitung des Bausteins erfolgt nach dem bereits erläuterten Muster: Die ID-Nummer der Aufgabe wird als ID-Nummer des Bausteins gesetzt. Der Name entspricht der Bezeichnung.

Die eingehende und ausgehende Kante werden ebenfalls direkt übernommen.

Die sieben unterschiedlichen Arten von Aufgaben in der BPMN wurden kurz angerissen. Obwohl an der Stelle ebenfalls festgelegt wurde, dass auf ihre Verwendung verzichtet und nur die „Blanko-Aufgabe“ verwendet wird, können auch hier die Spezialisierungen als Anmerkungen abgespeichert werden.

In Listing D.8 wird gezeigt, wie die sieben Aufgabentypen in der XML-Struktur dargestellt werden. Es wird nun festgelegt, dass jeder dieser Aufgabentypen als Baustein #2a verwendet werden kann. Tabelle D.2 zeigt alle speziellen Anmerkungen, die gesetzt werden können, um die verschiedenenartigen Aufgabentypen zu vermerken.

```
<sendTask id="_2" name="Send Task">
  <incoming>_11</incoming>
  <outgoing>_18</outgoing>
</sendTask>
<receiveTask id="_3" name="Receive Task">
  <incoming>_12</incoming>
  <outgoing>_19</outgoing>
</receiveTask>
<serviceTask id="_4" name="Service Task">
  <incoming>_13</incoming>
  <outgoing>_20</outgoing>
</serviceTask>
<userTask id="_5" name="User Task">
  <incoming>_14</incoming>
  <outgoing>_21</outgoing>
</userTask>
<manualTask id="_6" name="Manual Task">
  <incoming>_15</incoming>
  <outgoing>_22</outgoing>
</manualTask>
<scriptTask id="_7" name="Script Task">
  <incoming>_16</incoming>
  <outgoing>_23</outgoing>
</scriptTask>
<businessRuleTask id="_8" name="Business Rule Task">
  <incoming>_17</incoming>
  <outgoing>_24</outgoing>
</businessRuleTask>
```

Listing D.8: Aufgabentypen (vereinfachte XML-Struktur - BPMN)

Anmerkung	BPMN-Element
SPEC:SendTask	sendTask
SPEC:ReceiveTask	receiveTask
SPEC:ServiceTask	serviceTask
SPEC:UserTask	userTask
SPEC:ManualTask	manualTask
SPEC:ScriptTask	scriptTask
SPEC:BusinessRuleTask	businessRuleTask

Tabelle D.2: Spezielle Anmerkungen bei Baustein #2a

Listing D.9 zeigt, wie der entsprechende abgeleitete Baustein #2a dann aussieht. Sobald aus den Bausteinen heraus ein Diagramm in der Notation der BPMN generiert wird, können aus den Anmerkungen heraus die Aufgaben in ihre ursprüngliche Form überführt werden. Bei Generierung eines anderen Diagramms, zum Beispiel eines Aktivitätsdiagramms der UML, werden die Anmerkungen selbst abgebildet.

```
<brick_02a id="_3">
  <incoming>_12</incoming>
  <outgoing>_19</outgoing>
  <labeling>Receive Task</labeling>
  <annotation>SPEC:ReceiveTask</annotation>
</brick_02a>
```

Listing D.9: abgeleiteter Baustein #2a in XML

Nicht weniger einfach ist es, die nicht-atomare Einheit (Baustein #2b) zu erkennen. Sie ist eindeutig über das Element „subProcess“ mit allen seinen Kindern zu identifizieren. Dieses Element führt unter anderem die Attribute „id“ und „name“ und listet alle enthaltenen Elemente auf, so zum Beispiel ihre eingehenden und ausgehenden Kanten, das Startereignis, das Endereignis sowie atomare Einheiten. In Listing D.10 enthält die Untereinheit zwei atomare Einheiten.

```
<subProcess id="_12" name="Sub-Process">
  <incoming>_13</incoming>
  <outgoing>_14</outgoing>
  <startEvent id="_15" name="Start Event">
    <outgoing>_19</outgoing>
  </startEvent>
  <endEvent id="_16" name="End Event">
    <incoming>_21</incoming>
  </endEvent>
  <task id="_17" name="Task A">
    <incoming>_19</incoming>
    <outgoing>_20</outgoing>
  </task>
  <task id="_18" name="Task B">
    <incoming>_20</incoming>
    <outgoing>_21</outgoing>
  </task>
  <sequenceFlow id="_19" sourceRef="_15" targetRef="_17"/>
  <sequenceFlow id="_20" sourceRef="_17" targetRef="_18"/>
  <sequenceFlow id="_21" sourceRef="_18" targetRef="_16"/>
</subProcess>
```

Listing D.10: Baustein #2b (vereinfachte XML-Struktur - BPMN)

Der abgeleitete Baustein sieht dann wie in Listing D.11 (auf der nächsten Seite) dargestellt aus. Die ID-Nummer des Sub-Prozesses wird wieder direkt übernommen, sein Name wird zur Bezeichnung. Die eingehende und ausgehende Kante werden ebenfalls direkt übernommen. Danach wird begonnen, alle Bausteine innerhalb des Sub-Prozesses nach den bekannten Regeln abzuleiten.

```
<brick_02b id="_12">
  <incoming>_13</incoming>
  <outgoing>_14</outgoing>
  <labeling>Sub-Process</labeling>
  <brick_01a id="15"> <!-- ... --> </brick_01a>
  <brick_02a id="17"> <!-- ... --> </brick_02a>
  <brick_02a id="18"> <!-- ... --> </brick_02a>
  <brick_03 id="19"> <!-- ... --> </brick_03>
  <brick_03 id="20"> <!-- ... --> </brick_03>
  <brick_03 id="21"> <!-- ... --> </brick_03>
  <brick_01c id="16"> <!-- ... --> </brick_01c>
</brick_02b>
```

Listing D.11: abgeleiterer Baustein #2b in XML

In Listing D.10 (auf der vorherigen Seite) ist bereits das Element mit aufgeführt, welches Baustein #3 darstellt: „sequenceFlow“. Als Attribute werden die eigene ID-Nummer sowie sie ID-Nummern des Source-Elements sowie des Target-Elements geführt. Listing D.12 zeigt, wie sich die Notation verändert, wenn ein Name und eine Bedingung hinzugefügt werden.

```
<sequenceFlow id="_26" name="Sequence Flow" sourceRef="_17" targetRef="_25">
  <conditionExpression><![CDATA[i=4]]></conditionExpression>
</sequenceFlow>
```

Listing D.12: Baustein #3 (vereinfachte XML-Struktur - BPMN)

Listing D.13 zeigt, wie der abgeleitete Baustein aussieht. Auch hier gilt: Die ID-Nummer des Elements wird zur ID-Nummer des Bausteins. Der Name wird als Bezeichnung übernommen. Als „source_id“-Element wird das Attribut „sourceRef“ des Sequenzflusses benutzt, das „target_id“-Element wird mit dem Wert des Attributs „targetRef“ befüllt. Verfügt der Sequenzfluss über ein „conditionExpression“-Element, so wird die Bedingung in das „condition“-Element des Bausteins übernommen.

```
<brick_03 id="_26">
  <source_id>_17</source_id>
  <target_id>_25</target_id>
  <labeling>Sequence Flow</labeling>
  <condition>i=4</condition>
</brick_03>
```

Listing D.13: abgeleiterer Baustein #3 in XML

Der vierte Baustein ist ebenso einfach zu identifizieren wie die vorangegangenen. In der BPMN wird die parallele Aufspaltung mittels eines parallelen Gateways modelliert. Dieses ist in der XML-Struktur einfach zu identifizieren wie in Listing D.14 (auf der folgenden Seite) zu sehen ist. Es werden sämtliche eingehenden Kanten sowie die ausgehenden mit aufgeführt. Zusätzlich ist Baustein #4 am Attribut „gatewayDirection“ gut zu erkennen, da diese als divergierend angegeben wird.

Dem entgegen wird das Attribut beim Baustein #5a als konvergierend angegeben, wie man in Listing D.15 sehen kann. Vom Attribut abgesehen, kann die Art des Bausteins (#4 oder #5a) auch anhand der eingehenden beziehungsweise ausgehenden Kanten bestimmt werden.

```
<parallelGateway gatewayDirection="Diverging" id="_6" name="Parallel Gateway">
  <incoming>_7</incoming>
  <outgoing>_8</outgoing>
  <outgoing>_9</outgoing>
  <outgoing>_10</outgoing>
</parallelGateway>
```

Listing D.14: Baustein #4 (vereinfachte XML-Struktur - BPMN)

```
<parallelGateway gatewayDirection="Converging" id="_6" name="Parallel Gateway">
  <incoming>_7</incoming>
  <incoming>_8</incoming>
  <incoming>_9</incoming>
  <outgoing>_10</outgoing>
</parallelGateway>
```

Listing D.15: Baustein #5a (vereinfachte XML-Struktur - BPMN)

Die Ableitung erfolgt dann nach dem bekannten Muster: Die ID-Nummer des Elements wird als ID-Nummer des Bausteins übernommen. Ebenso werden die eingehenden und ausgehenden Kanten übernommen. Als Bezeichnung wird der Name des Elements verwendet. Das Attribut „gatewayDirection“ muss nicht übernommen werden, da der Baustein aufgrund seines Namens eindeutig zurücktransformiert werden kann.

Komplizierter wird die Identifizierung von Baustein #5b. Wird von diesem das BPMN-Diagramm abgeleitet, so ergibt sich kein Problem. Soll dieser Baustein jedoch in einem bestehenden Diagramm identifiziert werden, so ist die Aufgabe komplexer, die korrespondierenden Gateways zu ermitteln. An dieser Stelle müssen die ID-Nummern und dazugehörigen Sequenzflüsse untersucht werden: Es müssen alle ID-Nummern überprüft werden, die vom aufspaltendem Element (Baustein #4) fortführen. Weiterhin müssen die Wege dieser ID-Nummern bis zum zusammenführenden Element (Baustein #5a) verfolgt werden. Es handelt sich genau dann um Baustein #5b, wenn alle ID-Nummern, die vom aufspaltendem Element fortführen, im zusammenführenden Element zusammenlaufen. Das gilt insbesondere auch dann, wenn zwischen diesen beiden Elementen weitere aufspaltende oder zusammenführende Bausteine gesetzt wurden. Listing D.16 (auf der nächsten Seite) zeigt diesen Baustein. Die Reihenfolge der Elemente ist intuitiv geordnet worden. Zwischen den beiden Gateways liegen zwei parallel abzuarbeitende Aufgaben.

```
<parallelGateway gatewayDirection="Diverging" id="_2" name="Parallel Gateway">
  <incoming>_20</incoming>
  <outgoing>_6</outgoing>
  <outgoing>_8</outgoing>
</parallelGateway>
<task id="_3" name="Task A">
  <incoming>_6</incoming>
  <outgoing>_7</outgoing>
</task>
<task id="_4" name="Task B">
  <incoming>_8</incoming>
  <outgoing>_9</outgoing>
</task>
<parallelGateway gatewayDirection="Converging" id="_5" name="Parallel Gateway">
  <incoming>_7</incoming>
  <incoming>_9</incoming>
</parallelGateway>
<sequenceFlow id="_6" sourceRef="_2" targetRef="_3"/>
<sequenceFlow id="_7" sourceRef="_3" targetRef="_5"/>
<sequenceFlow id="_8" sourceRef="_2" targetRef="_4"/>
<sequenceFlow id="_9" sourceRef="_4" targetRef="_5"/>
```

Listing D.16: Baustein #5b (vereinfachte XML-Struktur - BPMN)

Das parallele Gateway (Baustein #4) besitzt hier genau zwei ausgehende Kanten. Diese führen zu den „sequenceFlow“-Elementen „_6“ und „_8“. Nun kann über die Source- und Target-ID-Nummern geprüft werden, ob die Sequenzflüsse in einem gemeinsamen Element enden. Dies ist in diesem Fall das Element mit der ID-Nummer „_5“. Das dazugehörige Element ist ebenfalls ein paralleles Gateway und als Baustein #5a identifizierbar. Zu beachten ist an dieser Stelle noch die Anzahl eingehender Kanten dieses Elements, denn sie muss mit der Anzahl ausgehender Kanten des ersten parallelen Gateways übereinstimmen. Das ist hier der Fall. Auf diese Weise kann hier Baustein #5b ermittelt werden. Er ist in Listing D.17 zu sehen. Es ist zu beachten, dass der Baustein eine bisher nicht vergebene ID-Nummer erhalten muss, da er keine vorhandene übernehmen kann. Die eingehende Kante entspricht der eingehenden Kante von Baustein #4. Die ausgehende Kante entspricht der ausgehenden Kante von Baustein #5a. Die Bausteine, die zwischen #4 und #5a auftauchen, werden dann nach den bekannten Regeln abgeleitet.

```
<brick05b id="_N01">
  <incoming>_20</incoming>
  <outgoing>_21</outgoing>
  <brick04 id="_2"> <!-- ... --> </brick04>
  <brick02a id="_3"> <!-- ... --> </brick02a>
  <brick02a id="_4"> <!-- ... --> </brick02a>
  <brick03 id="_6"> <!-- ... --> </brick03>
  <brick03 id="_7"> <!-- ... --> </brick03>
  <brick03 id="_8"> <!-- ... --> </brick03>
  <brick03 id="_9"> <!-- ... --> </brick03>
  <brick05a id="_5"> <!-- ... --> </brick05a>
</brick05b>
```

Listing D.17: abgeleiteter Baustein #5b in XML

Baustein #6 ist ebenso einfach in der XML-Struktur zu identifizieren wie schon Baustein #4. In der BPMN wird die exklusive Auswahl mittels eines exklusiven Gateways modelliert. Das Gateway ist auch als solches gekennzeichnet, wie in Listing D.18 zu sehen ist. Wie schon das parallele Gateway kann auch dieses Gateway entweder über die Anzahl eingehender und ausgehender Kanten identifiziert werden oder über das Attribut „gatewayDirection“, welches als divergierend angegeben wird. Die Erstellung des Bausteins an sich erfolgt nach den gleichen Prinzipien wie bei Baustein #4.

```
<exclusiveGateway gatewayDirection="Diverging" id="_4" name="Exclusive G.">
  <incoming>_9</incoming>
  <outgoing>_10</outgoing>
  <outgoing>_11</outgoing>
</exclusiveGateway>
```

Listing D.18: Baustein #6 (vereinfachte XML-Struktur - BPMN)

Das Gegenstück ähnelt wiederum dem Baustein #5a und ist in Listing D.19 abgebildet. Die Richtung des Gateways ist hier als konvergierend angegeben. Die Ableitung des konkreten Bausteins folgt den gleichen Prinzipien wie schon bei Baustein #5a.

```
<exclusiveGateway gatewayDirection="Converging" id="_7" name="Exclusive G.">
  <incoming>_12</incoming>
  <incoming>_13</incoming>
  <outgoing>_14</outgoing>
</exclusiveGateway>
```

Listing D.19: Baustein #7a (vereinfachte XML-Struktur - BPMN)

Die korrespondierende Version des Bausteins zu identifizieren, ist wie schon Baustein #5b komplexer, folgt allerdings den selben Regeln, die schon bei Baustein #5b Anwendung fanden. Listing D.20 (auf der nächsten Seite) zeigt die Elemente, die betrachtet werden müssen. Der Unterschied zu Baustein #5b besteht darin, dass die „sequenceFlow“-Elemente nun teilweise Bedingungen beinhalten. Diese sind für die Ableitung von Baustein #7b aber von untergeordneter Bedeutung. Der Baustein erhält eine neue ID-Nummer, weil er keine bestehende übernehmen kann, und als eingehende und ausgehende Kanten werden die eingehende Kante von Baustein #6 beziehungsweise die ausgehende Kante von Baustein #7a gesetzt (Listing D.21 auf der nachfolgenden Seite).

```
<exclusiveGateway gatewayDirection="Diverging" id="_4" name="Exclusive G.">
  <incoming>_9</incoming>
  <outgoing>_10</outgoing>
  <outgoing>_11</outgoing>
</exclusiveGateway>
<!-- beteiligte Elemente -->
<exclusiveGateway gatewayDirection="Converging" id="_7" name="Exclusive G.">
  <incoming>_12</incoming>
  <incoming>_13</incoming>
  <outgoing>_14</outgoing>
</exclusiveGateway>
<sequenceFlow id="_10" sourceRef="_4" targetRef="_5">
  <conditionExpression><![CDATA[i<4]]></conditionExpression>
</sequenceFlow>
<sequenceFlow id="_11" sourceRef="_4" targetRef="_6">
  <conditionExpression><![CDATA[i>=4]]></conditionExpression>
</sequenceFlow>
<sequenceFlow id="_12" sourceRef="_5" targetRef="_7"/>
<sequenceFlow id="_13" sourceRef="_6" targetRef="_7"/>
```

Listing D.20: Baustein #7b (vereinfachte XML-Struktur - BPMN)

```
<brick07b id="_N02">
  <incoming>_9</incoming>
  <outgoing>_14</outgoing>
  <brick06 id="_4"> <!-- ... --> </brick06>
  <!-- ... -->
  <brick03 id="_10"> <!-- ... --> </brick03>
  <brick03 id="_11"> <!-- ... --> </brick03>
  <brick03 id="_12"> <!-- ... --> </brick03>
  <brick03 id="_13"> <!-- ... --> </brick03>
  <brick07a id="_7"> <!-- ... --> </brick07a>
</brick07b>
```

Listing D.21: abgeleiteter Baustein #7b in XML

Den Baustein #8 in der XML-Struktur zu finden ist ebenfalls nicht schwierig. Da die BPMN hierfür das inklusive Gateway benutzt, kann nach diesem Element gesucht werden (Listing D.22). Allerdings wird die Richtung des Gateways in der XML-Struktur nicht angegeben, sodass das Attribut nicht als Indikator dienen kann. Stattdessen muss man sich auf die Anzahl der eingehenden und ausgehenden Kanten als Indikator verlassen. Aufgrund von Einschränkungen, die in Abschnitt B bemacht wurden, ist der Indikator eindeutig. Das gleiche gilt bei Baustein #9a (Listing D.23 auf der nächsten Seite). Die Ableitung der konkreten Bausteine erfolgt in beiden Fällen nach den bereits bekannten Mustern.

```
<inclusiveGateway gatewayDirection="Unspecified" id="_3" name="Inclusive G.">
  <incoming>_8</incoming>
  <outgoing>_9</outgoing>
  <outgoing>_10</outgoing>
</inclusiveGateway>
```

Listing D.22: Baustein #8 (vereinfachte XML-Struktur - BPMN)

```
<inclusiveGateway gatewayDirection="Unspecified" id="_6" name="Inclusive G.">
  <incoming>_11</incoming>
  <incoming>_12</incoming>
  <outgoing>_13</outgoing>
</inclusiveGateway>
```

Listing D.23: Baustein #9a (vereinfachte XML-Struktur - BPMN)

Wie schon bei den Bausteine #5b und #7b ist die Identifizierung von Baustein #9b (Listing D.24) komplexer, unterliegt aber den gleichen Regeln. Der Baustein erhält eine neue ID-Nummer, weil er keine übernehmen kann, und die Einträge für seine eingehende und ausgehende Kante erhält er von der eingehenden Kante vom inkludierten Baustein #8 beziehungsweise vom inkludierten Baustein #9a.

```
<inclusiveGateway id="_4" name="Inclusive Gateway">
  <incoming>_9</incoming>
  <outgoing>_10</outgoing>
  <outgoing>_11</outgoing>
</inclusiveGateway>
<!-- beteiligte Elemente -->
<inclusiveGateway id="_7" name="Inclusive Gateway">
  <incoming>_12</incoming>
  <incoming>_13</incoming>
  <outgoing>_14</outgoing>
</inclusiveGateway>
<sequenceFlow id="_10" sourceRef="_4" targetRef="_5">
  <conditionExpression><![CDATA[i<4]]></conditionExpression>
</sequenceFlow>
<sequenceFlow id="_11" sourceRef="_4" targetRef="_6">
  <conditionExpression><![CDATA[i>=8]]></conditionExpression>
</sequenceFlow>
<sequenceFlow id="_12" sourceRef="_5" targetRef="_7"/>
<sequenceFlow id="_13" sourceRef="_6" targetRef="_7"/>
```

Listing D.24: Baustein #9b (vereinfachte XML-Struktur - BPMN)

In der BPMN wird der Diskriminator in einem Sub-Prozess dargestellt, der von einem speziellen Endereignis terminiert wird. Dieses schickt ein Signal zu einem angehefteten Zwischenereignis, von welchem aus es weitergeleitet wird. In Listing D.25 (auf der nächsten Seite) ist der Baustein in XML-Struktur aus der Datei dargestellt und enthält dabei zwei konkurrierende Aufgaben.

Baustein #10 wird über das angeheftete Zwischenereignis identifiziert, in diesem Fall also über das „boundaryEvent“-Element mit seinem Kind-Element „errorEventDefinition“. Durch das Attribut „attachedToRef“ kann der dazugehörige Sub-Prozess ebenfalls identifiziert werden.

An dieser Stelle wird deutlich, dass die Bausteine in bestimmter Reihenfolge im Diagramm gesucht werden müssen, da ansonsten der Sub-Prozess leicht als Baustein #2b identifiziert werden könnte. Er enthält ein Startereignis sowie ein paralleles Gateway, um die konkurrierenden Aufgaben zu aktivieren. Dies ist aufgrund von zuvor festgelegten Einschränkungen nötig. Anstelle der „task“-Elemente könnten dort auch „subProcess“-Elemente eingebettet sein. Über das exklusive Gateway werden die Einheiten wieder zusammengeführt und enden in einem

```
<subProcess id="_2" name="Sub-Process">
  <incoming>_30</incoming>
  <outgoing>_31</outgoing>
  <startEvent id="_3" name="Start Event">
    <outgoing>_11</outgoing>
  </startEvent>
  <parallelGateway gatewayDirection="Diverging" id="_4" name="Parallel G.">
    <incoming>_11</incoming>
    <outgoing>_12</outgoing>
    <outgoing>_13</outgoing>
    <!-- weitere -->
  </parallelGateway>
  <task id="_5" name="Task A">
    <incoming>_12</incoming>
    <outgoing>_15</outgoing>
  </task>
  <task id="_6" name="Task B">
    <incoming>_13</incoming>
    <outgoing>_16</outgoing>
  </task>
  <!-- weitere Aufgaben -->
  <exclusiveGateway gatewayDirection="Converging" id="_8" name="Exclusive G.">
    <incoming>_15</incoming>
    <incoming>_16</incoming>
    <incoming>_17</incoming>
    <!-- weitere -->
    <outgoing>_18</outgoing>
  </exclusiveGateway>
  <endEvent id="_9" name="End Event">
    <incoming>_18</incoming>
    <errorEventDefinition id="_9_ED_1"/>
      <ref>_10</ref>
    </errorEventDefinition>
  </endEvent>
  <sequenceFlow id="_11" sourceRef="_3" targetRef="_4"/>
  <sequenceFlow id="_12" sourceRef="_4" targetRef="_5"/>
  <sequenceFlow id="_13" sourceRef="_4" targetRef="_6"/>
  <!-- weitere Sequenzflüsse -->
  <sequenceFlow id="_15" sourceRef="_5" targetRef="_8"/>
  <sequenceFlow id="_16" sourceRef="_6" targetRef="_8"/>
  <sequenceFlow id="_18" sourceRef="_8" targetRef="_9"/>
</subProcess>
<boundaryEvent attachedToRef="_2" cancelActivity="true"
                                              id="_10" name="Boundary Event">
  <errorEventDefinition id="_10_ED_1"/>
    <ref>_9</ref>
  </errorEventDefinition>
</boundaryEvent>
```

Listing D.25: Baustein #10 (vereinfachte XML-Struktur - BPMN)

```
<brick10 id="_2">
  <incoming>_30</incoming>
  <outgoing>_31</outgoing>
  <labeling>Sub-Process</labeling>
  <brick02a id="_5"> <!-- ... --> </brick02a>
  <brick02a id="_6"> <!-- ... --> </brick02a>
  <!-- weitere -->
</brick10>
```

Listing D.26: abgeleiteter Baustein #10 in XML

Fehler-Endereignis. Dieses löst dann das angeheftete Ereignis aus. Listing D.26 (auf der vorherigen Seite) zeigt, wie der abgeleitete Baustein aussieht.

Als ID-Nummer wird die ID-Nummer des Sub-Prozesses verwendet, als Bezeichnung der Name des Sub-Prozesses. Die eingehende und ausgehende Kante werden direkt übernommen. Im Anschluss werden noch die „task"-Elemente als Bausteine #2a übernommen. Dabei werden ihre eingehenden und ausgehenden Kanten auf Pseudo-ID-Nummern gesetzt. Alle weiteren Elemente wie das parallele und das exklusive Gateway können im Baustein völlig vernachlässigt werden. Diese werden bei der Erstellung eines Diagramms neu erzeugt.

Baustein #11 stellt die verzögerte Auswahl dar. Diese wird in der BPMN mittels eines ereignis-basierten Gateways dargestellt. Dieses ist als „eventBasedGateway" in der XML-Struktur identifizierbar. Das Gateway ist mit mindestens zwei fangenden Zwischenereignissen verbunden, was anhand der ausgehenden Kanten erkennbar ist. Diese Verbindungen lassen sich wie zuvor anhand der ID-Nummern verfolgen. Der Baustein hat genau so viele Ausgänge wie Zwischenereignisse. In Listing D.27 sind die beteiligten Elemente zu sehen.

```
<eventBasedGateway id="_2" instantiate="false" name="Event Gateway">
  <incoming>_10</incoming>
  <outgoing>_5</outgoing>
  <outgoing>_6</outgoing>
</eventBasedGateway>
<intermediateCatchEvent id="_3" name="Intermediate Catch Event A">
  <incoming>_5</incoming>
  <outgoing>_11</outgoing>
</intermediateCatchEvent>
<intermediateCatchEvent id="_4" name="Intermediate Catch Event B">
  <incoming>_6</incoming>
  <outgoing>_12</outgoing>
</intermediateCatchEvent>
<!-- weitere Ereignisse -->
<sequenceFlow id="_5" sourceRef="_2" targetRef="_3"/>
<sequenceFlow id="_6" sourceRef="_2" targetRef="_4"/>
<!-- weitere -->
<sequenceFlow id="_10" sourceRef="_20" targetRef="_2"/>
```

Listing D.27: Baustein #11 (vereinfachte XML-Struktur - BPMN)

Die konkrete Ableitung des Bausteins ist ein wenig komplexer. Zunächst einmal erhält der Baustein die ID-Nummer des ereignis-basierten Gateways. Die eingehende Kante ergibt sich aus der eingehenden Kante des Gateways, die ausgehenden Kanten ergeben sich aus den ausgehenden Kanten der Zwischenereignisse. Die Verbindungen zwischen Gateway und Zwischenereignis kann über die entsprechenden „sequenceFlow"-Elemente nachvollzogen werden. Die Bezeichnung des Bausteins ergibt sich aus dem Namen des Gateways. Nun erfolgt die Befüllung der „event"-Elemente. Je ein „event"-Element ergibt sich aus einem Zwischenereignis. Das heißt, das Kind-Element „condition" wird mit dem Namen des Zwischenereignisses gefüllt, als Target-ID-Nummer wird nochmals die ausgehende Kante des Zwischenereignisses verwendet. Auf diese Weise ist später nachvollziehbar, welches Zwischenereignis mit welchem Folgebaustein verbunden werden muss. Der

so abgeleitete Baustein sieht dann wie in Listing D.28 dargestellt aus.

```
<brick11 id="_2">
  <incoming>_10</incoming>
  <outgoing>_11</outgoing>
  <outgoing>_12</outgoing>
  <labeling>Event Gateway</labeling>
  <event>
    <condition>Intermediate Catch Event A</condition>
    <target_id>_11</target_id>
  </event>
  <event>
    <condition>Intermediate Catch Event B</condition>
    <target_id>_12</target_id>
  </event>
</brick11>
```

Listing D.28: abgeleiteter Baustein #11 in XML

Beim Abbruch einer Aktivität (Baustein #12) kommt wieder ein angeheftetes Zwischenereignis zum Einsatz, also ein „boundaryEvent“-Element mit seinem Kind-Element „errorEventDefinition“. Dieses Mal ist es jedoch nicht mit einem Sub-Prozess verbunden, sondern mit einer Aufgabe. Um diesen Baustein also von Baustein #10 zu unterscheiden, muss überprüft werden, ob das angeheftete Ereignis mit einem Sub-Prozess oder einer Aufgabe verknüpft ist, was über das Attribut „attachedToRef“ geschieht. Listing D.29 zeigt die XML-Struktur des Bausteins in der BPMN. In Listing D.30 (auf der nachfolgenden Seite) ist der abgeleitete Baustein dargestellt. Er übernimmt die ID-Nummer der Aufgabe sowie als Bezeichnung deren Name. Die eingehende Kante wird ebenfalls direkt von der Aufgabe übernommen. Die zwei ausgehenden Kanten ergeben sich aus den ausgehenden Kanten der Aufgabe sowie des „boundaryEvent“-Elements.

```
<task id="_5" name="Task A">
  <incoming>_9</incoming>
  <outgoing>_10</outgoing>
</task>
<boundaryEvent attachedToRef="_5" cancelActivity="true" id="_6"
                                                  name="Boundary Event">
  <outgoing>_11</outgoing>
  <errorEventDefinition id="_5_ED_1"/>
</boundaryEvent>
```

Listing D.29: Baustein #12 (vereinfachte XML-Struktur - BPMN)

```
<brick12 id="_5">
  <incoming>_9</incoming>
  <outgoing>_10</outgoing>
  <outgoing>_11</outgoing>
  <labeling>Task A</labeling>
  <exit_condition>
    <condition>Boundary Event</condition>
    <target_id>_11</target_id>
  </exit_condition>
</brick12>
```

Listing D.30: abgeleiteter Baustein #12 in XML

Die Bausteine #13 und #14 sind wie bereits erwähnt nicht mit den Elementen der BPMN modellierbar.

E Ableitung der Bausteine aus der XML-Struktur der UML AD

Auf den folgenden Seiten wird die Ableitung der einzelnen Bausteine aus den jeweiligen XML-Strukturen der UML Aktivitätsdiagramm vollständig beschrieben. Dabei wird auf die selbe vereinfachte Darstellungsweise zurückgegriffen wie sie schon in der Arbeit selbst zum Einsatz gekommen ist.

Listing E.1 zeigt die XML-Notation des Startknotens (Baustein #1a). Er ist eindeutig identifizierbar, da er als „Pseudostate“ mit dem Attribut „kind=‘initial‘“ deklariert wird. Er verwendet unter anderem die Attribute „id“ sowie „name“ und enthält eine Referenz auf die ID-Nummer der ausgehenden Kante.

```
<UML:Pseudostate xmi.id="_02" name="Initial Node" kind="initial">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_03"/>
  </UML:StateVertex.outgoing>
</UML:Pseudostate>
```

Listing E.1: Baustein #1a (vereinfachte XML-Struktur - UML AD)

Listing E.2 zeigt, wie der daraus abgeleitete Baustein aussieht. Er übernimmt die ID-Nummer des Pseudo-Status als eigene ID-Nummer. Weiterhin wird der Name als Bezeichnung gesetzt und die ausgehende Kante direkt übernommen. Zu erkennen ist diese Kante auch hier durch das Wort „outgoing“.

```
<brick01a id="_2">
  <outgoing>_03</outgoing>
  <labeling>Initial Node</labeling>
</brick01a>
```

Listing E.2: abgeleiteter Baustein #1a in XML

Sein direktes Gegenstück (Baustein #1c) ist ebenfalls einfach zu finden, denn der Endknoten ist als „FinalState“ gekennzeichnet (Listing E.3 auf der nachfolgenden Seite). Die Abteilung erfolgt genau wie beim Startknoten, nur dass hier die eingehenden Kanten als eingehende Kanten übernommen werden, zu erkennen am Wort „incoming“.

```
<UML:FinalState xmi.id="_04" name="Activity Final">
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_05"/>
  </UML:StateVertex.incoming>
</UML:FinalState>
```

Listing E.3: Baustein #1c (vereinfachte XML-Struktur - UML AD)

Während es in der BPMN schwierig war, Baustein #1b zu identifizieren, so ist es hier relativ gesehen einfacher. In Listing E.4 ist die XML-Struktur dargestellt, die für Baustein #1b wichtig ist. Ein Flussende ist hier ein „ActionState“ mit einem Stereotyp. Über eine Referenz wird die genaue Art des Stereotypen festgelegt, in diesem Fall „flow_final_node“. Im Listing ist das Rahmenkonstrukt „Activity-Graph“ mit abgebildet, um zu verdeutlichen, dass sich die Stereotypendefinierung außerhalb desselben liegt.

```
<UML:ActivityGraph xmi.id="_01">
<!-- ... -->
  <UML:ActionState xmi.id="_06" name="Flow Final">
    <UML:ModelElement.stereotype>
      <UML:Stereotype xmi.idref="_07"/>
    </UML:ModelElement.stereotype>
    <UML:StateVertex.incoming>
      <UML:Transition xmi.idref="_08"/>
    </UML:StateVertex.incoming>
  </UML:ActionState>
<!-- ... -->
</UML:ActivityGraph>
<!-- ... -->
<UML:Stereotype xmi.id="_07" name="flow_final_node">
  <UML:Stereotype.baseClass>ActionState</UML:Stereotype.baseClass>
  <UML:Stereotype.extendedElement>
    <JUDE:ModelElement xmi.idref="_06"/>
  </UML:Stereotype.extendedElement>
</UML:Stereotype>
```

Listing E.4: Baustein #1b (vereinfachte XML-Struktur - UML AD)

Der aus der XML-Struktur abgeleitete Baustein ist in Listing E.5 zu sehen. Er übernimmt die ID-Nummer des „ActionState“-Elements sowie die eingehende Kante. Vom Stereotyp muss nichts übernommen werden, da er lediglich für die Identifizierung des Bausteins erforderlich war.

```
<brick01b id="_06">
  <incoming>_08</incoming>
  <labeling>Flow Final</labeling>
</brick01b>
```

Listing E.5: abgeleiteter Baustein #1b in XML

Eine andere Art „ActionState“ stellt die Aktion dar. Listing E.6 (auf der nachfolgenden Seite) zeigt die XML-Struktur, die für die Identifizierung des Bausteins #2a wichtig ist. Das „ActionState“-Element verfügt hier nicht über einen Stereotypen, dafür aber sowohl über eingehende als auch ausgehende Kanten.

```
<UML:ActionState xmi.id="_10" name="Action A">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_11"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_12"/>
  </UML:StateVertex.incoming>
</UML:ActionState>
```

Listing E.6: Baustein #2a (vereinfachte XML-Struktur - UML AD)

Der abgeleitete Baustein übernimmt die ID-Nummer als eigene sowie die Angaben zu eingehenden und ausgehenden Kanten. Den Namen des „ActionState“-Elements übernimmt er als Bezeichung (Listing E.7).

```
<brick02a id="_10">
  <incoming>_12</incoming>
  <outgoing>_11</outgoing>
  <labeling>Action A</labeling>
</brick02a>
```

Listing E.7: abgeleiteter Baustein #2a in XML

Die nicht-atomare Einheit (Baustein #2b) lässt sich ebenfalls erkennen. Listing E.8 zeigt die XML-Struktur. An dieser Stelle wurden die ID-Nummern mit Buchstaben ergänzt. Dies macht es einfacher zu erkennen, worauf sie sich beziehen. Die ID-Nummern ohne Buchstaben gehören zu Elementen der übergeordneten Einheit. Der Buchstabe „T“ zeigt eine Transition an, der Buchstabe „A“ die Untereinheit. Das „A“ steht in diesem Fall für „ActivityGraph“.

```
<UML:ActivityGraph xmi.id="_01">
<!-- ... -->
  <UML:SubactivityState xmi.id="_03" name="Activity A">
    <UML:StateVertex.outgoing>
      <UML:Transition xmi.idref="_T02"/>
    </UML:StateVertex.outgoing>
    <UML:StateVertex.incoming>
      <UML:Transition xmi.idref="_T01"/>
    </UML:StateVertex.incoming>
    <UML:SubmachineState.submachine>
      <UML:StateMachine xmi.idref="_A01"/>
    </UML:SubmachineState.submachine>
  </UML:SubactivityState>
<!-- ... -->
</UML:ActivityGraph>
<UML:ActivityGraph xmi.id="_A01" name="Activity A">
<!-- ... -->
  <UML:StateMachine.submachineState>
    <JUDE:SubmachineState xmi.idref="_03"/>
  </UML:StateMachine.submachineState>
<!-- ... -->
</UML:ActivityGraph>
```

Listing E.8: Baustein #2b (vereinfachte XML-Struktur - UML AD)

Eine Untereinheit wird durch „SubactivityState“ gekennzeichnet. Die üblichen Referenzierungen auf eingehende und ausgehende Kanten sowie eine weitere Referenz sind vermerkt. Diese führt zu einem weiteren Aktivitätsgraphen, der die Elemente der Untereinheit enthält. Die Untereinheit an sich enthält im Gegenzug eine Referenz auf die übergeordnete Einheit. Die XML-Struktur der UML Aktivitätsdiagramme ist an dieser Stelle nicht ganz so intuitiv nachzuvollziehen wie die der BPMN, dennoch ist die konkrete Ableitung des Bausteins #2b kein Problem (Listing E.9). Der Baustein übernimmt die ID-Nummer des „SubactivityState“-Elements, also hier „_03“. Weiterhin übernimmt der Baustein dessen eingehenden und ausgehenden Kanten. Als Bezeichnung wird der Name verwendet. Danach wird begonnen, alle Bausteine innerhalb der Aktivität nach den bekannten Regeln abzuleiten, beginnend mit Baustein #1a und endend mit Baustein #1c. Diese Bausteine werden allerdings im referenzierten zweiten „activityGraph“-Element gesucht.

```
<brick02b id="_03">
  <incoming>_T01</incoming>
  <outgoing>_T02</outgoing>
  <labeling>Activity A</labeling>
  <brick01a> <!-- ... --> </brick01a>
  <!-- ... -->
  <brick01c> <!-- ... --> </brick01c>
</brick02b>
```

Listing E.9: abgeleiteter Baustein #2b in XML

Die Struktur der Verbindungen zwischen zwei Elementen hingegen ist der XML-Struktur bei der BPMN wieder ähnlich. Listing E.10 zeigt diese. Eine Verbindung verfügt über eine eigene ID-Nummer und enthält die ID-Nummern des Source- und des Target-Elements. Eine Bedingung in Form eines Guards kann ebenso hinzugefügt werden.

```
<UML:Transition xmi.id="_T01">
  <UML:Transition.guard>
    <UML:Guard xmi.id="_G01" name="i=4"></UML:Guard>
  </UML:Transition.guard>
  <UML:Transition.source>
    <UML:StateVertex xmi.idref="_02"/>
  </UML:Transition.source>
  <UML:Transition.target>
     <UML:StateVertex xmi.idref="_03"/>
  </UML:Transition.target>
</UML:Transition>
```

Listing E.10: Baustein #3 (vereinfachte XML-Struktur - UML AD)

In diesem Fall sieht der abgeleitete Baustein wie in Listing E.11 (auf der nächsten Seite) dargestellt aus. Der Baustein übernimmt die ID-Nummer der Transition. Aus dem „source“-Element beziehungsweise dem „target“-Element ergeben sich die Werte „source_id“ und „target_id“. Die Bedingung wird vom Namensattribut des „guard“-Elements übenommen. Eine Bezeichnung gibt es in diesem Fall nicht.

```
<brick_03 id="_T01">
  <source_id>_02</source_id>
  <target_id>_03</target_id>
  <condition>i=4</condition>
</brick_03>
```

Listing E.11: abgeleiterer Baustein #3 in XML

Listing E.12 zeigt, wie die XML-Struktur für die parallele Aufteilung bei den UML Aktivitätsdiagrammen aussieht. Wie beim Startknoten handelt es sich um einen Pseudo-Status, dieses mal allerdings von der Art „fork“ statt von der Art „initial“. Davon jedoch abgesehen, sind die gleichen Dinge vermerkt wie bei der Struktur der BPMN. Aus diesem Grund ist auch die Ableitung des Bausteins kein Problem: Die ID-Nummer wird übernommen, als Bezeichnung wird der Name verwendet. Die eingehenden und ausgehen Kanten ergeben sich aus den Einträgen im „incoming“-Element beziehungsweise im „outgoing“-Element.

```
<UML:Pseudostate xmi.id="_10" name="Fork Node" kind="fork">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_T02"/>
    <UML:Transition xmi.idref="_T03"/>
    <UML:Transition xmi.idref="_T04"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_T01"/>
  </UML:StateVertex.incoming>
</UML:Pseudostate>
```

Listing E.12: Baustein #4 (vereinfachte XML-Struktur - UML AD)

Für das Gegenstück (Baustein #5a) wird wie bei der BPMN das gleiche Element mit einem anderen Attributwert verwendet. So ist die Art des Pseudo-Status hier „join“. Dies ist in Listing E.13 zu sehen. Die Abteilung des Bausteins folgt nun nach den üblichen Regeln.

```
<UML:Pseudostate xmi.id="_15" name="Join Node" kind="join">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_T13"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_T10"/>
    <UML:Transition xmi.idref="_T11"/>
    <UML:Transition xmi.idref="_T12"/>
  </UML:StateVertex.incoming>
</UML:Pseudostate>
```

Listing E.13: Baustein #5a (vereinfachte XML-Struktur - UML AD)

Den Baustein #5b in einem bestehenden Diagramm zu identifizieren wird ebenso komplex wie bei der BPMN. Alle vom aufspaltenden Element ausgehenden Verbindungen müssen im zusammenführenden Element enden. Dafür müssen die Transitionen betrachtet werden. Listing E.14 (auf der nächsten Seite) zeigt beispielhaft einen Baustein #5b.

```
<UML:Pseudostate xmi.id="_05" name="Fork Node" kind="fork">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_T06"/>
    <UML:Transition xmi.idref="_T09"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_T05"/>
  </UML:StateVertex.incoming>
</UML:Pseudostate>
<UML:Pseudostate xmi.id="_08" name="Join Node" kind="join">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_T08"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_T07"/>
    <UML:Transition xmi.idref="_T10"/>
  </UML:StateVertex.incoming>
</UML:Pseudostate>
<UML:ActionState xmi.id="_06" name="Action A">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_T07"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_T06"/>
  </UML:StateVertex.incoming>
</UML:ActionState>
<UML:ActionState xmi.id="_07" name="Action B">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_T10"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_T09"/>
  </UML:StateVertex.incoming>
</UML:ActionState>
<UML:Transition xmi.id="_T06">
  <UML:Transition.source>
    <UML:StateVertex xmi.idref="_05"/>
  </UML:Transition.source>
  <UML:Transition.target>
    <UML:StateVertex xmi.idref="_06"/>
  </UML:Transition.target>
</UML:Transition>
<UML:Transition xmi.id="_T07">
  <UML:Transition.source>
    <UML:StateVertex xmi.idref="_06"/>
  </UML:Transition.source>
  <UML:Transition.target>
    <UML:StateVertex xmi.idref="_08"/>
  </UML:Transition.target>
</UML:Transition>
<UML:Transition xmi.id="_T09">
  <UML:Transition.source>
    <UML:StateVertex xmi.idref="_05"/>
  </UML:Transition.source>
  <UML:Transition.target>
    <UML:StateVertex xmi.idref="_07"/>
  </UML:Transition.target>
</UML:Transition>
<UML:Transition xmi.id="_T10">
  <UML:Transition.source>
    <UML:StateVertex xmi.idref="_07"/>
  </UML:Transition.source>
  <UML:Transition.target>
    <UML:StateVertex xmi.idref="_08"/>
  </UML:Transition.target>
</UML:Transition>
```

Listing E.14: Baustein #5b (vereinfachte XML-Struktur - UML AD)

Trotz des enormen Umfangs von Listing E.14 ist der abgeleitete Baustein ziemlich klein (Listing E.15). Zunächst einmal weist Baustein #4 auf die potenzielle Existenz von Baustein #5b hin. Die ausgehenden Kanten werden betrachtet. In diesem Fall also die Transitionen mit den ID-Nummern „_T06“ und „_T09“. Über die „source“-Elemente und die „target“-Elemente wird nun geprüft, ob die weiterführenden Transitionen in einem gemeinsamen Element enden. Dies ist das Element mit der ID-Nummer „_08“, welches sich als Baustein #5a mit passender Anzahl eingehender Kanten erweist. Somit handelt es sich hierbei um Baustein #5b. Baustein #4 wird. Nun werden die Bausteine #4 und #5a sowie die, die dazwischen auftauchen, nach den bekannten Regeln abgeleitet. Die eingehende Kante ergibt sich aus der eingehenden Kante von Baustein #4, die ausgehende Kante ergibt sich aus der ausgehenden Kante von Baustein #5a. Der Baustein an sich erhält eine neue ID-Nummer, weil er keine vorhandene übernehmen kann.

```
<brick05b id="_N01">
  <incoming>_T05</incoming>
  <outgoing>_T08</outgoing>
  <brick04 id="_05"> <!-- ... --> </brick04>
  <brick02a id="_06"> <!-- ... --> </brick02a>
  <brick02a id="_07"> <!-- ... --> </brick02a>
  <brick03 id="_T06"> <!-- ... --> </brick03>
  <brick03 id="_T07"> <!-- ... --> </brick03>
  <brick03 id="_T09"> <!-- ... --> </brick03>
  <brick03 id="_T10"> <!-- ... --> </brick03>
  <brick05a id="_08"> <!-- ... --> </brick05a>
</brick05b>
```

Listing E.15: abgeleiteter Baustein #5b in XML

Während bei der BPMN eine deutliche Unterscheidung zwischen den Bausteinen #6 und #7a über das Attribut „gatewayDirection“ möglich war, gleichen sich bei der XML-Struktur der UML Aktivitätsdiagramme die beiden Elemente. Das Merkmal „kind“, welches bisher bei der Unterscheidung der verschiedenen Pseudo-Stati geholfen hat, ist bei beiden mit „junction“ angegeben. Aus diesem Grund müssen die eingehenden beziehungsweise ausgehenden Kanten an dieser Stelle als Indikator verwendet werden, denn ein Entscheidungsknoten darf nur eine eingehende Kante haben und mindestens zwei ausgehende. Ein Verbindungsknoten hingegen hat mindestens zwei eingehende Kanten und nur eine ausgehende. Die Listings E.16 und E.17 (auf der nächsten Seite) zeigen die beiden Bausteine. Die für die Entscheidung bei der exklusiven Auswahl benötigten Bedingungen stehen an den Verbindungen, also an den dazugehörigen Bausteinen #3.

Abgeleitet werden die Bausteine wie bei Baustein #4 beziehungsweise Baustein #5a. Um Baustein #7b zu identifizieren, werden die gleichen Schritte wie bei Baustein #5b angewandt, sodass auch hier der abgeleitete Baustein deutlich schlanker wird, als die dazugehörige zu durchsuchende XML-Struktur.

Graphisch betrachtet unterscheidet sich Baustein #8 bei den UML Aktivitätsdiagrammen kaum von Baustein #4. Es stehen allerdings Bedingungen an den ausgehenden Kanten des Knotens. In der XML-Struktur des

```
<UML:Pseudostate xmi.id="_20" name="Decision Node" kind="junction">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_T21"/>
    <UML:Transition xmi.idref="_T22"/>
    <UML:Transition xmi.idref="_T23"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_T20"/>
  </UML:StateVertex.incoming>
</UML:Pseudostate>
```

Listing E.16: Baustein #6 (vereinfachte XML-Struktur - UML AD)

```
<UML:Pseudostate xmi.id="_25" name="Merge Node" kind="junction">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_T28"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_T25"/>
    <UML:Transition xmi.idref="_T26"/>
    <UML:Transition xmi.idref="_T27"/>
  </UML:StateVertex.incoming>
</UML:Pseudostate>
```

Listing E.17: Baustein #7a (vereinfachte XML-Struktur - UML AD)

Knotens selbst ist dies allerdings nicht vermerkt, da er nur die Referenz auf die ausgehenden Kanten enthält. Daher ist die XML-Struktur identisch zu der von Baustein #4, wie in Listing E.18 zu sehen ist. An dieser Stelle muss also anhand der dazugehörigen Bausteine #3 eine Identifizierung im bestehenden Diagramm vorgenommen werden. An dieser Stelle wird deutlich, dass später eine eindeutige Reihenfolge für die Identifizierung der einzelnen Bausteine definiert werden muss. Davon abgesehen, ist die Ableitung der Bausteine aus der XML-Struktur hier genauso einfach wie schon bei Baustein #4.

```
<UML:Pseudostate xmi.id="_10" name="Fork Node" kind="fork">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_T02"/>
    <UML:Transition xmi.idref="_T03"/>
    <UML:Transition xmi.idref="_T04"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_T01"/>
  </UML:StateVertex.incoming>
</UML:Pseudostate>
```

Listing E.18: Baustein #8 (vereinfachte XML-Struktur - UML AD)

Baustein #9a kann allein nicht für die UML Aktivitätsdiagramme stehen. Die synchronisierende Verschmelzung verlangt eine Korrespondenz zu einem aufspaltendem Element, wie es in dieser Modellsprache nicht vorgesehen ist. Aus diesem Grund wird der Baustein allein auch nicht in einem bestehenden Diagramm auftauchen. Baustein #9b hingegen kann bestehen, da bei ihm die Korrespondenz vorausgesetzt wird. Der inkludierte Baustein #09a bedient sich dann eines Tricks: Es wird eine eindeutige Anmerkung am zusammenführenden Element gesetzt, sodass das

gewünschte Verhalten wenigstens graphisch besteht.

Listing E.19 zeigt die entsprechende Struktur. Die Anmerkung unterscheidet Baustein #9a von Baustein #5a. Baustein #9b umfasst nun die Bausteine #8 und #9a wie definiert und lässt sich mit den gleichen Schritten wie bei Baustein #5b identifizieren.

```
<UML:Comment xmi.id="_C01" name="" body="SPEC:brick09a">
  <UML:Comment.annotatedElement>
    <JUDE:ModelElement xmi.idref="_30"/>
  </UML:Comment.annotatedElement>
</UML:Comment>
<!-- ... -->
  <UML:ActivityGraph xmi.id="_01">
  <!-- ... -->
    <UML:Pseudostate xmi.id="_30" name="Join Node" kind="join">
      <UML:ModelElement.comment>
        <UML:Comment xmi.idref="_C01"/>
      </UML:ModelElement.comment>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_T33"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_T30"/>
        <UML:Transition xmi.idref="_T31"/>
        <UML:Transition xmi.idref="_T32"/>
      </UML:StateVertex.incoming>
    </UML:Pseudostate>
  <!-- ... -->
  </UML:ActivityGraph>
```

Listing E.19: Baustein #9a (vereinfachte XML-Struktur - UML AD)

Das Listing E.19 zeigt auch, wie Anmerkungen allgemein in der XML-Struktur aussehen. Solche „comment“-Elemente können nicht nur verwendet werden, um besondere Markierungen zu setzen, sondern auch um echte Anmerkungen anzubringen. In diesem Fall wird die Einleitung „SPEC“ weggelassen.

Um den Diskriminator (Baustein #10) in der Notation der UML Aktivitätsdiagramme darzustellen, benötigt es einen Unterbrechungsbereich. Dieses Notationselement wird von Astah Professional nicht angeboten, ist aber im XMI[161] für UML-Diagramme angegeben. Listing E.20 (auf der nächsten Seite) zeigt daher keinen korrekten XML-Code aus einer Export-Datei, sondern vielmehr eine Pseudo-Variante, um die Idee darzustellen. Die Elemente, die zum Unterbrechungsbereich gehören, erhalten eine entsprechende Referenz auf diesen. Umgekehrt sind alle beteiligten Elemente im Unterbrechungsbereich aufgeführt.

[161] Vgl. OMG [2011c], Zeile 6968ff

```
<UML:InterruptibleActivityRegion xmi.id="_I01" name="Inter. Activity Region">
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
    <JUDE:ModelElement xmi.idref="_T47"/>
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
    <JUDE:ModelElement xmi.idref="_41"/>
    <JUDE:ModelElement xmi.idref="_42"/>
    <!-- weitere -->
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
</UML:InterruptibleActivityRegion>
<!-- ... -->
  <UML:ActivityGraph xmi.id="_01">
  <!-- ... -->
    <UML:Pseudostate xmi.id="_41" name="Fork Node" kind="fork">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_I01"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_T41"/>
        <UML:Transition xmi.idref="_T42"/>
        <UML:Transition xmi.idref="_T43"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_T40"/>
      </UML:StateVertex.incoming>
    </UML:Pseudostate>
    <UML:ActionState xmi.id="_42" name="Action A">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_I01"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_T44"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_T41"/>
      </UML:StateVertex.incoming>
    </UML:ActionState>
    <!-- weitere Aktionen oder Aktivitäten -->
    <UML:Pseudostate xmi.id="_45" name="Merge Node" kind="junction">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_I01"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_T47"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_T44"/>
        <UML:Transition xmi.idref="_T45"/>
        <UML:Transition xmi.idref="_T46"/>
      </UML:StateVertex.incoming>
    </UML:Pseudostate>
  <!-- ... -->
  </UML:ActivityGraph>
```

Listing E.20: Baustein #10 (vereinfachte XML-Struktur - UML AD)

Wie es schon bei einigen anderen abgeleiteten Bausteinen der Fall war, so wird auch dieser deutlich schlanker. Listing E.21 (auf der folgenden Seite) zeigt den abgeleiteten Baustein. Dabei wurde die ID-Nummer des Unterbrechnungsbereiches sowie der Name als Bezeichnung übernommen. Die eingehende Kante der Gabelung („Fork Node“) wird als eingehende Kante verwendet. Die ausgehende Kante der Vereinigung („Merge Node“) wird als ausgehende Kante gesetzt. Alle „ActionState“-Elemente werden je zu einem Baustein #2a überführt. Dabei werden ihre eingehenden und ausgehenden Kanten auf Pseudo-ID-Nummern gesetzt. Alle

weiteren Elemente wie die Gabelung und die Vereinigung können im Baustein vernachlässigt werden. Diese werden bei der Erstellung eines Diagramms neu erzeugt.

```
<brick10 id="_I01">
  <incoming>_T40</incoming>
  <outgoing>_T47</outgoing>
  <labeling>Interruptible Activity Region</labeling>
  <brick02a id="_42"> <!-- ... --> </brick02a>
  <!-- weitere -->
</brick10>
```

Listing E.21: abgeleiteter Baustein #10 in XML

Auch in den beiden folgenden Bausteinen kommt bei den UML Aktivitätsdiagrammen der Unterbrechungsbereich zum Einsatz. Aus diesem Grund muss bei der Identifizierung der Bausteine im Diagramm später auf eine Reihenfolge geachtet werden. Bei Baustein #11 verzweigt eine Gabelung auf mehrere Signale, die im Unterbrechungsbereich liegen. Das Signal ist als ein besonderer „ActionState“ definiert und hat den Stereotypen „signal receipt“. Sobald ein Signal ausgelöst wird, werden alle anderen als Optionen verworfen. Auf diese Weise erfolgt die Auswahl aufgrund von äußeren Einflüssen. In Listing E.22 (auf der nachfolgenden Seite) ist die XML-Struktur zu sehen.

Daraus wird der Baustein wie folgt abgeleitet: Zunächst wird die ID-Nummer des Unterbrechungsbereichs sowie der Name als Bezeichnung übernommen. Als eingehende Kante wird die eingehende Kante der Gabelung („Fork Node“) ausgewählt, als ausgehende Kanten werden die ausgehenden Kanten der Signale ausgewählt. Danach werden alle Signale in „event“-Elemente überführt. Das heißt, das „condition“-Element wird mit dem Namen des Signals befüllt und als „target_id“ wird nochmals die ausgehende Kante des Signals angegeben. Auf diese Weise kann später nachvollzogen werden, welches Ereignis welchen Folgebaustein besitzt. Alle weiteren Elemente können ignoriert werden. Der so abgeleitete Baustein ist in Listing E.23 (auf der nächsten Seite) abgebildet.

```
<UML:InterruptibleActivityRegion xmi.id="_I02" name="Inter. Activity Region">
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
    <JUDE:ModelElement xmi.idref="_T53"/>
    <!-- weitere -->
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
    <JUDE:ModelElement xmi.idref="_50"/>
    <!-- weitere -->
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
</UML:InterruptibleActivityRegion>
<!-- ... -->
  <UML:ActivityGraph xmi.id="_01">
  <!-- ... -->
    <UML:Pseudostate xmi.id="_50" name="Fork Node" kind="fork">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_I02"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_T51"/>
        <!-- weitere -->
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_T50"/>
      </UML:StateVertex.incoming>
    </UML:Pseudostate>
    <UML:ActionState xmi.id="_51" name="Signal A">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_I02"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:ModelElement.stereotype>
        <UML:Stereotype xmi.idref="_S50"/>
      </UML:ModelElement.stereotype>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_T53"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_T51"/>
      </UML:StateVertex.incoming>
    </UML:ActionState>
    <!-- weitere Signale -->
  <!-- ... -->
  </UML:ActivityGraph>
  <!-- ... -->
  <UML:Stereotype xmi.id="_S50" name="signal receipt">
    <UML:Stereotype.baseClass>ActionState</UML:Stereotype.baseClass>
    <UML:Stereotype.extendedElement>
      <JUDE:ModelElement xmi.idref="_51"/>
    </UML:Stereotype.extendedElement>
  </UML:Stereotype>
  <!-- weitere Stereotypen -->
```

Listing E.22: Baustein #11 (vereinfachte XML-Struktur - UML AD)

```
<brick11 id="_I02">
  <incoming>_T50</incoming>
  <outgoing>_T53</outgoing>
  <!-- weitere -->
  <labeling>Interruptible Activity Region</labeling>
  <event>
    <condition>Signal A</condition>
    <target_id>_T53</target_id>
  </event>
  <!-- weitere -->
</brick11>
```

Listing E.23: abgeleiteter Baustein #11 in XML

Bei Baustein #12 liegen genau eine Aktion und ein Signal im Unterbrechungsbereich. Die Aktion kann auf diese Weise jederzeit bei Signaleingang abgebrochen werden. Listing E.24 zeigt die entsprechende XML-Struktur.

```
<UML:InterruptibleActivityRegion xmi.id="_I02" name="Interrup. Activity Region">
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
    <JUDE:ModelElement xmi.idref="_T51"/>
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
    <JUDE:ModelElement xmi.idref="_50"/>
    <JUDE:ModelElement xmi.idref="_51"/>
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
</UML:InterruptibleActivityRegion>
<!-- ... -->
  <UML:ActivityGraph xmi.id="_01">
  <!-- ... -->
    <UML:ActionState xmi.id="_50" name="Cancellation Signal">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_I02"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:ModelElement.stereotype>
        <UML:Stereotype xmi.idref="_S01"/>
      </UML:ModelElement.stereotype>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_T51"/>
      </UML:StateVertex.outgoing>
    </UML:ActionState>
    <UML:ActionState xmi.id="_51" name="Action A">
    <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_I02"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_T53"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_T52"/>
      </UML:StateVertex.incoming>
    </UML:ActionState>
  <!-- ... -->
  </UML:ActivityGraph>
  <!-- ... -->
  <UML:Stereotype xmi.id="_S01" name="signal receipt">
    <UML:Stereotype.baseClass>ActionState</UML:Stereotype.baseClass>
    <UML:Stereotype.extendedElement>
      <JUDE:ModelElement xmi.idref="_50"/>
    </UML:Stereotype.extendedElement>
  </UML:Stereotype>
```

Listing E.24: Baustein #12 (vereinfachte XML-Struktur - UML AD)

Der Baustein wird daraus wie folgt abgeleitet: Die ID-Nummer des Unterbrechungsbereiches wird erneut übernommen und der Name der Aktion wird als Bezeichnung gesetzt. Die eingehende Kante ergibt sich aus der eingehende Kante der Aktion. Die beiden ausgehenden Kanten ergeben sich aus der ausgehenden Kante der Aktion sowie der ausgehenden Kante des Signals. Das „condition“-Element im „exit_condition“-Element wird mit dem Namen des Signals befüllt und als „target_id“ wird nochmals die ausgehende Kante des Signals angegeben. Alle weiteren Elemente können ignoriert werden. Der so abgeleitete Baustein ist in Listing E.25 (auf der nachfolgenden Seite) dargestellt.

```
<brick12 id="_I03">
  <incoming>_T52</incoming>
  <outgoing>_T53</outgoing>
  <outgoing>_T51</outgoing>
  <labeling>Action A</labeling>
  <exit_condition>
    <condition>Cancellation Signal</condition>
    <target_id>_T51</target_id>
  </exit_condition>
</brick12>
```

Listing E.25: abgeleiteter Baustein #12 in XML

Die Bausteine #13 und #14 sind wie bereits erwähnt nicht mit den Elementen der UML Aktivitätsdiagramme modellierbar.

F Ableitung der XML-Strukturen aus einem Baustein

Auf den folgenden Seiten wird die Ableitung der XML-Strukturen für die BPMN beziehungsweise für die UML Aktivitätsdiagramme aus dem jeweiligen Baustein vollständig beschrieben. Dabei wird auf die selbe vereinfachte Darstellungsweise zurückgegriffen wie sie schon in der Arbeit selbst zum Einsatz gekommen ist.

Die Schritte, die zuvor zur Ableitung der Bausteine aus der XML-Struktur heraus gegangen worden sind, werden nun praktisch in umgekehrter Reihenfolge ausgeführt. Listing F.1 zeigt den Baustein #1a.

```
<brick_01a id="_02">
  <outgoing>_05</outgoing>
  <labeling>Start Event</labeling>
  <annotation>SPEC:TimerEventDefinition</annotation>
</brick_01a>
```

Listing F.1: Baustein #1a

Welche Elemente in der BPMN benötigt werden, um diesen Baustein darzustellen, ist aus den vorangegangenen Abschnitten klar. Für diesen Baustein wird das „startEvent“-Element (mit seinen Kind-Elementen) benötigt. So wird die ID-Nummer des Bausteins als ID-Nummer des Elements benutzt. Die Bezeichnung wird wieder zum Namen. Die ausgehende Kante wird ebenfalls direkt übernommen. Aus dem Inhalt der Anmerkung heraus ist erkennbar, dass es sich um eine besondere Anmerkung handelt. Das Schlagwort verdeutlicht, dass das Startereignis vom Typ Zeit-Startereignis ist. Dementsprechend wird eine Ereignisdefinition hinzugefügt, welche eine eigene ID-Nummer erhält. Die erhaltene XML-Struktur sieht dann wie in Listing F.2 dargestellt aus.

```
<startEvent id="_02" name="Start Event">
  <outgoing>_05</outgoing>
  <timerEventDefinition id="_05_ED_1"/>
</startEvent>
```

Listing F.2: abgeleitete XML-Struktur - BPMN (Baustein #1a) (1)

Handelt es sich um eine Anmerkung ohne spezielles Schlagwort, so wird diese wie in Listing F.3 dargestellt angefügt.

```
<startEvent id="_02" name="Start Event">
  <outgoing>_05</outgoing>
</startEvent>
<!-- ... -->
<textAnnotation id="_03" textFormat="text/plain">
  <text>Annotation</text>
</textAnnotation>
<association id="_04" sourceRef="_02" targetRef="_03"/>
```

Listing F.3: abgeleitete XML-Struktur - BPMN (Baustein #1a) (2)

Genauso funktioniert die Ableitung der XML-Struktur für die UML Aktivitätsdiagramme. Es ist bekannt, dass das entsprechende Element ein „PseudoState“-Element ist mit dem „kind“-Attribut mit Wert „initial“. Die ID-Nummer wird wieder als ID-Nummer des Elements übernommen, die Bezeichnung wird zum Namen-Attribut. Die ausgehende Kante wird ebenfalls direkt übernommen. Obwohl die Anmerkung auf eine Besonderheit hinweist, gibt es bei den UML Aktivitätsdiagrammen keine Entsprechung dafür, sodass die Anmerkung unverändert bestehen bleibt. Die erhaltene XML-Struktur sieht dann wie in Listing F.4 dargestellt aus.

```
<UML:Comment xmi.id="_C01" name="" body="SPEC:TimerEventDefinition">
  <UML:Comment.annotatedElement>
    <JUDE:ModelElement xmi.idref="_02"/>
  </UML:Comment.annotatedElement>
</UML:Comment>
<!-- ... -->
  <UML:ActivityGraph xmi.id="_01">
    <!-- ... -->
    <UML:Pseudostate xmi.id="_02" name="Start Event" kind="initial">
      <UML:ModelElement.comment>
        <UML:Comment xmi.idref ="_C01"/>
      </UML:ModelElement.comment>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_05"/>
      </UML:StateVertex.outgoing>
    </UML:Pseudostate>
    <!-- ... -->
  </UML:ActivityGraph>
```

Listing F.4: abgeleitete XML-Struktur - UML AD (Baustein #1a)

Auch die anderen Bausteine lassen sich auf diese Weise in die XML-Struktur überführen. In Listing F.5 ist der Baustein #1b abgebildet.

```
<brick_01b id="_02">
  <incoming>_05</incoming>
  <labeling>Flow Final</labeling>
</brick_01b>
```

Listing F.5: Baustein #1b

Die BPMN kennt kein eigenständiges Flussende, sodass dieses hier mit einer Anmerkung modelliert werden muss. Ansonsten werden die Werte aus dem Baustein ähnlich wie beim Baustein #1a in die Struktur eingesetzt: Die ID-Nummer des Bausteins wird zur ID-Nummer des „endEvent“-Elements, die Bezeichnung wird zum Namen-Attribut. Die eingehende Kante wird ebenfalls direkt übernommen. Um nun zu kennzeichnen, dass es sich um Baustein #1b handelt und nicht um Baustein #1c, wird nun noch eine eindeutige Anmerkung gesetzt. Dies ist in Listing F.6 zu sehen.

```
<endEvent id="_02" name="Flow Final">
  <incoming>_05</incoming>
</endEvent>
<!-- ... -->
<textAnnotation id="_03" textFormat="text/plain">
  <text>SPEC:FlowFinal</text>
</textAnnotation>
<association id="_04" sourceRef="_02" targetRef="_03"/>
```

Listing F.6: abgeleitete XML-Struktur - BPMN (Baustein #1b)

Bei den UML Aktivitätsdiagrammen gibt es ein eigenes Notationselement für das Flussende. Es handelt sich dabei um ein „ActionState“-Element mit Stereotyp. In diesem werden die Werte aus dem Baustein wie folgt eingesetzt: Die ID-Nummer des Bausteins wird zur ID-Nummer des „ActionState“-Elements. Die Bezeichnung wird zu dessen Name. Die eingehende Kante wird ebenfalls direkt übernommen. Die ID-Nummer des Bausteins muss noch einmal im Stereotyp eingesetzt werden, um die Verbindung zwischen diesem und dem Status zu verdeutlichen. Alle anderen Werte werden automatisch erzeugt. Die so erzeugte XML-Struktur ist in Listing F.7 zu sehen.

```
<UML:ActivityGraph xmi.id="_01">
<!-- ... -->
  <UML:ActionState xmi.id="_02" name="Flow Final">
    <UML:ModelElement.stereotype>
      <UML:Stereotype xmi.idref="_07"/>
    </UML:ModelElement.stereotype>
    <UML:StateVertex.incoming>
      <UML:Transition xmi.idref="_05"/>
    </UML:StateVertex.incoming>
  </UML:ActionState>
<!-- ... -->
</UML:ActivityGraph>
<!-- ... -->
<UML:Stereotype xmi.id="_07" name="flow_final_node">
  <UML:Stereotype.baseClass>ActionState</UML:Stereotype.baseClass>
  <UML:Stereotype.extendedElement>
    <JUDE:ModelElement xmi.idref="_02"/>
  </UML:Stereotype.extendedElement>
</UML:Stereotype>
```

Listing F.7: abgeleitete XML-Struktur - UML AD (Baustein #1b)

Baustein #1c gleicht dem Baustein #1b bis auf den Namen (Listing F.8). Die Ableitung der XML-Struktur der BPMN ist identisch bis auf die Anmerkung, die hier nicht gesetzt werden muss. Bei den Aktivitätsdiagrammen der UML ist dies anders. Hier wird nicht das gleiche Element genutzt, sondern ein eigenes: das „FinalState"-Element. Die ID-Nummer des Bausteins wird zur ID-Nummer dieses Elements. Die Bezeichnung wird als Name übernommen, die eingehende Kante wird direkt übernommen. Daraus ergibt sich die XML-Struktur aus Listing F.9.

```
<brick_01c id="_02">
  <incoming>_05</incoming>
  <labeling>End Node</labeling>
</brick_01c>
```

Listing F.8: Baustein #1c

```
<UML:FinalState xmi.id="_02" name="End Node">
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_05"/>
  </UML:StateVertex.incoming>
</UML:FinalState>
```

Listing F.9: abgeleitete XML-Struktur - UML AD (Baustein #1c)

Eine atomare Einheit im Prozess wird durch Baustein #2a dargestellt. Die BPMN kennt mehrere Arten dieser Einheit, weshalb im Baustein diese Art mit abgespeichert wird. In Listing F.10 ist beispielsweise die Sende-Aufgabe durch eine spezielle Anmerkung vermerkt worden.

```
<brick_02a id="_08">
  <incoming>_11</incoming>
  <outgoing>_12</outgoing>
  <labeling>Send Task</labeling>
  <annotation>SPEC:SendTask</annotation>
</brick_02a>
```

Listing F.10: Baustein #2a

Daraus wird die in Listing F.11 (auf der nächsten Seite) dargestellte XML-Struktur ermittelt. Die Anmerkung verweist auf das „sendTask"-Element mit dessen Kindern. Die ID-Nummer des Bausteins wird wieder als ID-Nummer des Elements übernommen. Die Bezeichnung wird ins Namen-Attribut übernommen und die eingehenden und ausgehenden Kanten werden direkt übernommen. Tabelle F.1 (auf der nachfolgenden Seite) zeigt, welche weiteren speziellen Anmerkungen bei Baustein #2a auftreten können.

```
<sendTask id="_08" name="Send Task">
  <incoming>_11</incoming>
  <outgoing>_12</outgoing>
</sendTask>
```

Listing F.11: abgeleitete XML-Struktur - BPMN (Baustein #2a)

Anmerkung	BPMN-Element
SPEC:SendTask	sendTask
SPEC:ReceiveTask	receiveTask
SPEC:ServiceTask	serviceTask
SPEC:UserTask	userTask
SPEC:ManualTask	manualTask
SPEC:ScriptTask	scriptTask
SPEC:BusinessTask	businessTask

Tabelle F.1: Spezielle Anmerkungen bei Baustein #2a

Die UML Aktivitätsdiagramme kennen keine speziellen Aufgaben wie sie in der BPMN vorkommen. Aus diesem Grund wird bei der Ableitung der XML-Struktur die Anmerkung angefügt. Die ID-Nummer des Bausteins wird als ID-Nummer des „ActionState“-Elements übernommen, die Bezeichnung wird zu dessen Name. Die eingehende und ausgehende Kante werden an den entsprechenden Stellen eingetragen. Daraus ergibt sich dann die in Listing F.12 dargestellte XML-Struktur.

```
<UML:Comment xmi.id="_C01" name="" body="SPEC:SendTask">
  <UML:Comment.annotatedElement>
    <JUDE:ModelElement xmi.idref="_08"/>
  </UML:Comment.annotatedElement>
</UML:Comment>
<!-- ... -->
  <UML:ActivityGraph xmi.id="_01">
  <!-- ... -->
    <UML:ActionState xmi.id="_08" name="Send Task">
      <UML:ModelElement.comment>
        <UML:Comment xmi.idref="_C01"/>
      </UML:ModelElement.comment>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_12"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_11"/>
      </UML:StateVertex.incoming>
    </UML:ActionState>
  <!-- ... -->
  </UML:ActivityGraph>
```

Listing F.12: abgeleitete XML-Struktur - UML AD (Baustein #2a)

Weil Baustein #2b andere Bausteine enthält, ist seine Ableitung etwas schwieriger. In Listing F.13 ist er beispielhaft dargestellt. Er enthält dort zwei Bausteine #2a sowie drei Bausteine #3, welche alles verbinden. Die Bausteine #1a und #1c sind obligatorisch.

```
<brick_02b id="_08">
  <incoming>_11</incoming>
  <outgoing>_12</outgoing>
  <labeling>Sub-Activity</labeling>
  <brick01a> <!-- ... --> </brick01a>
  <brick02a> <!-- ... --> </brick02a>
  <brick02a> <!-- ... --> </brick02a>
  <brick03> <!-- ... --> </brick03>
  <brick03> <!-- ... --> </brick03>
  <brick03> <!-- ... --> </brick03>
  <brick01c> <!-- ... --> </brick01c>
</brick_02b>
```

Listing F.13: Baustein #2b

Daraus wird die XML-Struktur in der BPMN wie folgt abgeleitet: Die ID-Nummer des Bausteins wird zur ID-Nummer des „subProcess“-Elements. Die Bezeichnung wird zum Namen-Attribut. Die eingehende und ausgehende Kante werden direkt übernommen. Die inkludierten Bausteine werden danach nach den bekannten Regeln ebenfalls abgeleitet. Daraus ergibt sich die in Listing F.14 dargestellte XML-Struktur.

```
<subProcess id="_08" name="Sub-Activity">
  <incoming>_11</incoming>
  <outgoing>_12</outgoing>
  <!-- ... -->
</subProcess>
```

Listing F.14: abgeleitete XML-Struktur - BPMN (Baustein #2b)

Bei den UML Aktivitätsdiagrammen sieht die ganze Struktur wieder deutlich komplexer aus (Listing F.15 auf der nächsten Seite). Zunächst einmal wird ein „SubactivityState“-Element erzeugt, welches die ID-Nummer des Bausteins erhält. Auch die Bezeichnung wird in das Namen-Attribut dieses Elements übertragen. Die eingehenden und ausgehenden Kanten werden in die entsprechenden Elemente geschrieben. Nun gibt es in dem „SubactivityState“-Element den Verweis auf einen anderen Aktivitätsgraphen. In diesem werden alle inkludierten Bausteine abgeleitet. Dieser zweite Aktivitätsgraph enthält zusätzlich einen Rückverweis auf das „SubactivityState“-Element, um die Verbindung zu verdeutlichen. Dort wird nochmals die ID-Nummer des Bausteins eingetragen.

```
<UML:ActivityGraph xmi.id="_01">
<!-- ... -->
  <UML:SubactivityState xmi.id="_08" name="Sub-Activity">
    <UML:StateVertex.outgoing>
      <UML:Transition xmi.idref="_12"/>
    </UML:StateVertex.outgoing>
    <UML:StateVertex.incoming>
      <UML:Transition xmi.idref="_11"/>
    </UML:StateVertex.incoming>
    <UML:SubmachineState.submachine>
      <UML:StateMachine xmi.idref="_A01"/>
    </UML:SubmachineState.submachine>
  </UML:SubactivityState>
<!-- ... -->
</UML:ActivityGraph>
<UML:ActivityGraph xmi.id="_A01" name="Sub-Activity">
<!-- ... -->
  <UML:StateMachine.submachineState>
    <JUDE:SubmachineState xmi.idref="_08"/>
  </UML:StateMachine.submachineState>
<!-- ... -->
</UML:ActivityGraph>
```

Listing F.15: abgeleitete XML-Struktur - UML AD (Baustein #2b)

Wieder deutlich einfacher wird die Ableitung von Baustein #3, denn in beiden Modellsprachen wird dafür je nur ein Element mit seinen Kindern benötigt. Listing F.16 zeigt den Baustein mit Bedingung.

```
<brick_03 id="_26">
  <source_id>_17</source_id>
  <target_id>_25</target_id>
  <labeling>Sequence Flow</labeling>
  <condition>i=4</condition>
</brick_03>
```

Listing F.16: Baustein #3

Wie üblich wird die ID-Nummer des Bausteins zur ID-Nummer des Elements. Die Bezeichnung wird als Name übernommen. Die Elemente „source_id“ und „target_id“ werden in die beiden Attribute „sourceRef„ beziehungsweise „targetRef“ geschrieben und die Bedingung wird in das „conditionExpression“-Element eingefügt. Die daraus erzeugte XML-Struktur sieht dann wie in Listing F.17 gezeigt aus. Sollte Baustein #3 ohne das „condition“-Element auftreten, so wird dort das Kind-Element weggelassen und die Struktur sieht wie in Listing F.18 (auf der folgenden Seite) dargestellt aus.

```
<sequenceFlow id="_26" name="Sequence Flow" sourceRef="_17" targetRef="_25">
  <conditionExpression><![CDATA[i=4]]></conditionExpression>
</sequenceFlow>
```

Listing F.17: abgeleitete XML-Struktur - BPMN (Baustein #3) (1)

```
<sequenceFlow id="_26" name="Sequence Flow" sourceRef="_17" targetRef="_25"\>
```

Listing F.18: abgeleitete XML-Struktur - BPMN (Baustein #3) (2)

Auch beim UML Aktivitätsdiagramm wird die ID-Nummer des Bausteins zur ID-Nummer des verwendeten Elements. Die Bezeichnung wird nicht übernommen, da es keine Entsprechung dafür in der Notation gibt. Sie kann allerdings als Anmerkung übernommen werden. Die Elemente „source_id“ und „target_id“ werden in den beiden Elementen „UML:Transition.source“ und „UML:Transition.target“ gespeichert, und die Bedingung wird im Namen-Attribut des „UML:Transition.guard“-Elements gespeichert. Die abgeleitete Struktur sieht dann wie in Listing F.19 dargestellt aus.

```
<UML:Transition xmi.id="_26">
  <UML:Transition.guard>
    <UML:Guard xmi.id="_G01" name="i=4"></UML:Guard>
  </UML:Transition.guard>
  <UML:Transition.source>
    <UML:StateVertex xmi.idref="_17"/>
  </UML:Transition.source>
  <UML:Transition.target>
     <UML:StateVertex xmi.idref="_25"/>
  </UML:Transition.target>
</UML:Transition>
```

Listing F.19: abgeleitete XML-Struktur - UML AD (Baustein #3)

Die XML-Strukturen der Bausteine #4 und #5a ähneln einander sowohl bei der BPMN als auch bei den Aktivitätsdiagrammen der UML. Die Listings F.20 und F.21 zeigen die beiden Bausteine.

```
<brick04 id="_23">
  <incoming>_24</incoming>
  <outgoing>_25</outgoing>
  <outgoing>_26</outgoing>
  <labeling>Split</labeling>
</brick04>
```

Listing F.20: Baustein #4

In der BPMN werden beide Bausteine mittels eines Parallelen Gateways modelliert. Dabei ist zum einen das „gatewayDirection“-Attribut unterschiedlich, zum anderen kann der Unterschied anhand der Anzahl der eingehenden und ausgehenden Kanten ermittelt werden. Beim Baustein #4 hat das Attribut den Wert „diverging“. Die ID-Nummer entspricht der ID-Nummer des Bausteins und die Bezeichnung wird als Name übernommen. Die eingehende Kante sowie die ausgehenden Kanten werden direkt übernommen (Listing F.22 auf der nächsten Seite).

```
<brick05a id="_27">
  <incoming>_28</incoming>
  <incoming>_29</incoming>
  <outgoing>_30</outgoing>
  <labeling>Join</labeling>
</brick05a>
```

Listing F.21: Baustein #5a

```
<parallelGateway gatewayDirection="Diverging" id="_23" name="Split">
  <incoming>_24</incoming>
  <outgoing>_25</outgoing>
  <outgoing>_26</outgoing>
</parallelGateway>
```

Listing F.22: abgeleitete XML-Struktur - BPMN (Baustein #4)

Bei Baustein #5a hat das Attribut den Wert „converging“, ansonsten sind alle Überführungen gleich (Listing F.23). Auch bei den Aktivitätsdiagrammen der UML sind die abgeleiteten XML-Strukturen zu den beiden Bausteinen einander sehr ähnlich. Hier werden „PseudoState“-Elemente verwendet, bei denen das „kind“-Attribut einmal den Wert „fork“ und einmal den Wert „join“ hat. Ansonsten folgt die Ableitung den gleichen Regeln wie zuvor: Die ID-Nummer des Bausteins wird zur ID-Nummer des Elements. Das Namen-Attribut wird mit dem Wert aus der Bezeichnung gefüllt und die eingehenden beziehungsweise ausgehenden Kanten werden entsprechend übertragen. Daraus ergeben sich die XML-Strukturen aus den Listings F.24 beziehungsweise F.25 (auf der nächsten Seite).

Der Baustein #5b ist schwieriger abzuleiten, da er wie Baustein #2b andere Bausteine beinhaltet. Im Listing F.26 (auf der folgenden Seite) ist ein solcher Baustein exemplarisch dargestellt. Er beinhaltet obligatorisch die Bausteine #4 und #5a sowie hier zwei parallel ausgeführte Bausteine #2a und die Verbindungsbausteine #3.

Daraus ergibt sich die XML-Struktur der BPMN wie folgt: Es werden alle inkludierten Bausteine nach den bekannten Regeln abgeleitet. Die ID-Nummer des Bausteins wird nicht weiterverwendet, da sie bei der Ableitung des Bausteins neu dazugeschrieben wurde. Auch die eingehende und ausgehende Kante des Bausteins werden nicht übernommen, da sie in Baustein #4 beziehungsweise #5a enthalten sind. Das gleiche gilt bei der Ableitung der XML-Struktur der UML Aktivitätsdiagramme.

```
<parallelGateway gatewayDirection="Converging" id="_27" name="Join">
  <incoming>_28</incoming>
  <incoming>_29</incoming>
  <outgoing>_30</outgoing>
</parallelGateway>
```

Listing F.23: abgeleitete XML-Struktur - BPMN (Baustein #5a)

```
<UML:Pseudostate xmi.id="_23" name="Split" kind="fork">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_25"/>
    <UML:Transition xmi.idref="_26"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_24"/>
  </UML:StateVertex.incoming>
</UML:Pseudostate>
```

Listing F.24: abgeleitete XML-Struktur - UML AD (Baustein #4)

```
<UML:Pseudostate xmi.id="_27" name="Join" kind="join">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_30"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_28"/>
    <UML:Transition xmi.idref="_29"/>
  </UML:StateVertex.incoming>
</UML:Pseudostate>
```

Listing F.25: abgeleitete XML-Struktur - UML AD (Baustein #5a)

```
<brick05b id="_N01">
  <incoming>_T05</incoming>
  <outgoing>_T08</outgoing>
  <brick04 id="_05"> <!-- ... --> </brick04>
  <brick02a id="_06"> <!-- ... --> </brick02a>
  <brick02a id="_07"> <!-- ... --> </brick02a>
  <brick03 id="_T06"> <!-- ... --> </brick03>
  <brick03 id="_T07"> <!-- ... --> </brick03>
  <brick03 id="_T09"> <!-- ... --> </brick03>
  <brick03 id="_T10"> <!-- ... --> </brick03>
  <brick05a id="_08"> <!-- ... --> </brick05a>
</brick05b>
```

Listing F.26: Baustein #5b

```
<brick06 id="_23">
  <incoming>_24</incoming>
  <outgoing>_25</outgoing>
  <outgoing>_26</outgoing>
  <labeling>Choice</labeling>
</brick06>
```

Listing F.27: Baustein #6

```
<brick07a id="_27">
  <incoming>_28</incoming>
  <incoming>_29</incoming>
  <outgoing>_30</outgoing>
  <labeling>Merge</labeling>
</brick07a>
```

Listing F.28: Baustein #7a

Baustein #6 und #7a gleichen einander wie zuvor Baustein #4 und #5a. Die Listings F.27 und F.28 (auf der vorherigen Seite) zeigen die Bausteine.

Die Ableitung in die XML-Struktur der BPMN erfolgt wie bei den Bausteinen #4 und #5a, nur dass statt einem „parallelGateway“-Element in beiden Fällen ein „exclusiveGateway“-Element verwendet wird. Bei den UML Aktivitätsdiagrammen wird ebenfalls fast die gleiche Struktur verwendet wie bei Baustein #4 und #5a. Nur hier ist das „kind“-Attribut in beiden Fällen mit dem Wert „junction“ belegt. Baustein #7b folgt den gleichen Regeln wie Baustein #5b: Es werden einfach nur alle inkludierten Bausteine nach den bereits bekannten Regeln abgeleitet.

Der Baustein für die Mehrfach-Auswahl (Listing F.29) orientiert sich ebenfalls an der Ableitung des Bausteins #4. In der Notation der BPMN kommt an dieser Stelle das „inclusiveGateway“-Element zum Einsatz. Bei diesem ist das „gatewayDirection“-Attribut unspezifiziert. Die ID-Nummer des Bausteins wird von dem Element übernommen, die Bezeichnung wird zum Namen-Attribut. Die eingehenden und ausgehenden Kanten werden in die entsprechenden Elemente eingefügt. Auf diese Weise ergibt sich die in Listing F.30 dargestellte XML-Struktur. Bei den UML Aktivitätsdiagrammen ist die abgeleitete Struktur identisch zu der von Baustein #4 (Listing F.31). Die Unterscheidung erfolgt über die dazugehörigen Verbindungsbausteine #3, welche die Bedingungen enthalten.

```
<brick08 id="_23">
  <incoming>_24</incoming>
  <outgoing>_25</outgoing>
  <outgoing>_26</outgoing>
  <labeling>Multi Choice</labeling>
</brick08>
```

Listing F.29: Baustein #8

```
<inclusiveGateway gatewayDirection="Unspecified" id="_23" name="Multi Choice">
  <incoming>_24</incoming>
  <outgoing>_25</outgoing>
  <outgoing>_26</outgoing>
</inclusiveGateway>
```

Listing F.30: abgeleitete XML-Struktur - BPMN (Baustein #8)

```
<UML:Pseudostate xmi.id="_23" name="Multi Choice" kind="fork">
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref="_25"/>
    <UML:Transition xmi.idref="_26"/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref="_24"/>
  </UML:StateVertex.incoming>
</UML:Pseudostate>
```

Listing F.31: abgeleitete XML-Struktur - UML AD (Baustein #8)

Listing F.32 zeigt Baustein #9a. Es ist bekannt, dass dieser Baustein in der Notation der BPMN durch das „inclusiveGateway“-Element (mit dessen Kind-Elementen) beschrieben wird. Das Attribut „gatewayDirection“ wird als unspezifiziert angegeben wie es auch schon in der Gegenrichtung der Fall gewesen ist. Die ID-Nummer des Bausteins wird als ID-Nummer des „inclusiveGateway“-Elements genommen und die Bezeichnung wird zum Namen-Attribut. Die eingehenden sowie ausgehenden Kanten werden bei den „incoming“ beziehungsweise „outgoing“-Elementen eingesetzt. Daraus ergibt sich die in Listing F.33 dargestellte Struktur.

```
<brick_09a id="_10">
  <incoming>_11</incoming>
  <incoming>_12</incoming>
  <incoming>_13</incoming>
  <outgoing>_14</outgoing>
  <labeling>Join</labeling>
</brick09a>
```

Listing F.32: Baustein #9a

```
<inclusiveGateway gatewayDirection="Unspecified" id="_10" name="Join">
  <incoming>_11</incoming>
  <incoming>_12</incoming>
  <incoming>_13</incoming>
  <outgoing>_14</outgoing>
</inclusiveGateway>
```

Listing F.33: abgeleitete XML-Struktur - BPMN (Baustein #9a)

Die abgeleitete Struktur für die UML Aktivitätsdiagramme sieht dagegen komplexer aus, ist aber ebenso einfach zu erzeugen. In der XML-Struktur wird der Baustein mittels eines „PseudoState“-Elements dargestellt. Dieses Mal ist die Art „join“. Die ID-Nummer des Bausteins wird zur ID-Nummer des Status-Elements, die Bezeichnung wird zum Namen-Attribut. Auch hier werden die eingehenden und ausgehenden Kanten in die entsprechenden Kind-Elemente eingefügt. Zum Schluss kommt die Besonderheit bei diesem Baustein: Da Baustein #9a eigentlich nicht in der Notation der UML Aktivitätsdiagramme erscheinen kann, wird eine Anmerkung gesetzt, die wenigstens optisch dafür sorgt, dass das gewünschte Verhalten modellierbar wird. Bei einer späteren Rücktransformation kann anhand dieser Anmerkung der richtige Baustein abgeleitet werden. Die abgeleitete XML-Struktur ist in Listing F.34 (auf der nächsten Seite) zu sehen.

Bei Baustein #9b werden wie schon bei den Bausteinen #5a und #7a alle inkludierten Bausteine abgeleitet. Listing F.35 (auf der nachfolgenden Seite) zeigt den Baustein #10, welcher den Diskriminator beschreibt.

Der Diskriminator wird in der BPMN mittels eines Sub-Prozesses mit angeheftetem Fehler-Zwischenereignis beschrieben. In diesem Sub-Prozess laufen mehrere Aufgaben (oder wiederum Sub-Prozesse) konkurrierend ab. Dementsprechend muss die XML-Struktur eines Sub-Prozesses inklusive Startereignis und parallelem Gateway erzeugt werden. Das Startereignis ist mit dem parallelem Gateway verbunden.

```
<UML:Comment xmi.id="_C01" name="" body="SPEC:brick09a">
  <UML:Comment.annotatedElement>
    <JUDE:ModelElement xmi.idref="_10"/>
  </UML:Comment.annotatedElement>
</UML:Comment>
<!-- ... -->
  <UML:ActivityGraph xmi.id="_01">
  <!-- ... -->
    <UML:Pseudostate xmi.id="_10" name="Join" kind="join">
      <UML:ModelElement.comment>
        <UML:Comment xmi.idref="_C01"/>
      </UML:ModelElement.comment>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_14"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_11"/>
        <UML:Transition xmi.idref="_12"/>
        <UML:Transition xmi.idref="_13"/>
      </UML:StateVertex.incoming>
    </UML:Pseudostate>
  <!-- ... -->
  </UML:ActivityGraph>
```

Listing F.34: abgeleitete XML-Struktur - UML AD (Baustein #9a)

```
<brick10 id="_2">
  <incoming>_30</incoming>
  <outgoing>_31</outgoing>
  <labeling>Discriminator</labeling>
  <brick02a id="_5"> <!-- ... --> </brick02a>
  <brick02a id="_6"> <!-- ... --> </brick02a>
</brick10>
```

Listing F.35: Baustein #10

Ferner muss ein exklusives Gateway erzeugt werden sowie ein Fehler-Endereignis, welches mit dem angehefteten Zwischenereignis in Verbindung gebracht werden kann. Das exklusive Gateway ist mit dem Endereignis verbunden. In diese Struktur werden die Werte aus dem Baustein wie folgt eingesetzt: Die ID-Nummer des Bausteins wird zur ID-Nummer des „subProcess“-Elements. Diese ID-Nummer wird auch beim „attachedToRef“-Attribut des angehefteten Zwischenereignisses eingesetzt. Die Bezeichnung wird zum Namen-Attribut des Sub-Prozesses. Die eingehende Kante sowie die ausgehende Kante werden bei den entsprechenden Elementen des Sub-Prozesses eingesetzt. Danach werden die inkludierten Bausteine #2a (oder #2b) abgeleitet. Abschließend werden Verbindungen vom parallelen Gateway zu jedem „task“-Element (oder „subProcess“-Element) gesetzt sowie von diesen zum exklusiven Gateway. Daraus ergibt sich eine XML-Struktur wie sie in Listing F.36 (auf der nächsten Seite) dargestellt ist.

```
<subProcess id="_2" name="Discriminator">
  <incoming>_30</incoming>
  <outgoing>_31</outgoing>
  <startEvent id="_3" name="Start Event">
    <outgoing>_11</outgoing>
  </startEvent>
  <parallelGateway gatewayDirection="Diverging" id="_4" name="P. Gateway">
    <incoming>_11</incoming>
    <outgoing>_12</outgoing>
    <outgoing>_13</outgoing>
  </parallelGateway>
  <task id="_5" name="Task A">
    <incoming>_12</incoming>
    <outgoing>_15</outgoing>
  </task>
  <task id="_6" name="Task B">
    <incoming>_13</incoming>
    <outgoing>_16</outgoing>
  </task>
  <exclusiveGateway gatewayDirection="Converging" id="_8" name="E. Gateway">
    <incoming>_15</incoming>
    <incoming>_16</incoming>
    <incoming>_17</incoming>
    <outgoing>_18</outgoing>
  </exclusiveGateway>
  <endEvent id="_9">
    <incoming>_18</incoming>
    <errorEventDefinition id="_9_ED_1" name="End Event"/>
      <ref>_10</ref>
    </errorEventDefinition>
  </endEvent>
  <sequenceFlow id="_11" sourceRef="_3" targetRef="_4"/>
  <sequenceFlow id="_12" sourceRef="_4" targetRef="_5"/>
  <sequenceFlow id="_13" sourceRef="_4" targetRef="_6"/>
  <sequenceFlow id="_15" sourceRef="_5" targetRef="_8"/>
  <sequenceFlow id="_16" sourceRef="_6" targetRef="_8"/>
  <sequenceFlow id="_18" sourceRef="_8" targetRef="_9"/>
</subProcess>
<boundaryEvent attachedToRef="_2" cancelActivity="true" name="Boundary Event"
                                                    id="_10">
  <errorEventDefinition id="_10_ED_1"/>
    <ref>_9</ref>
  </errorEventDefinition>
</boundaryEvent>
```

Listing F.36: abgeleitete XML-Struktur - BPMN (Baustein #10)

Bei den UML Aktivitätsdiagrammen kommt für den Baustein#10 der Unterbrechungsbereich ins Spiel. Die ID-Nummer des Bausteins wird zur ID-Nummer des Unterbrechungsbereichs. Die Bezeichnung wird zu dessen Namen-Attribut. Dann wird innerhalb des Bereichs eine Gabelung erzeugt sowie ein Verbindungsknoten. Die eingehende Kante des Bausteins wird zur eingehenden Kante der Gabelung. Die ausgehende Kante des Bausteins wird zur ausgehenden Kante des Verbindungsknotens. Diese ausgehende Kante wird auch als Unterbrechungskante im Unterbrechungsbereich eingesetzt. Dann wird jeder im Baustein inkludierte Baustein #2a (oder #2b) abgeleitet. Abschließend werden Verbindungen zwischen der Gabelung und jedem „ActionState“-Element (oder „SubactivityState“-Element) gesetzt. Ebenso wird eine Verbindung von jedem dieser Element zum Verbindungsknoten gesetzt. Auf diese Weise ergibt sich eine XML-Struktur wie sie in Listing F.37 (auf der folgenden Seite) stark vereinfacht dargestellt ist.

```
<UML:InterruptibleActivityRegion xmi.id="_02" name="Discriminator">
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
    <JUDE:ModelElement xmi.idref="_31"/>
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
    <JUDE:ModelElement xmi.idref="_41"/>
    <JUDE:ModelElement xmi.idref="_05"/>
    <JUDE:ModelElement xmi.idref="_06"/>
    <!-- ... -->
    <JUDE:ModelElement xmi.idref="_45"/>
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
</UML:InterruptibleActivityRegion>
<!-- ... -->
  <UML:ActivityGraph xmi.id="_01">
  <!-- ... -->
    <UML:Pseudostate xmi.id="_41" name="Fork Node" kind="fork">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_02"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:StateVertex.outgoing>
        <!-- ... -->
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_30"/>
      </UML:StateVertex.incoming>
    </UML:Pseudostate>
    <UML:ActionState xmi.id="_05" name="Action A">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_02"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <!-- ... -->
    </UML:ActionState>
    <UML:ActionState xmi.id="_05" name="Action B">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_02"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <!-- ... -->
    </UML:ActionState>
    <UML:Pseudostate xmi.id="_45" name="Merge Node" kind="junction">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_02"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_31"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <!-- ... -->
      </UML:StateVertex.incoming>
    </UML:Pseudostate>
  <!-- ... -->
  </UML:ActivityGraph>
```

Listing F.37: abgeleitete XML-Struktur - UML AD (Baustein #10)

Der vorletzte Baustein, der betrachtet werden muss, ist Baustein #11. Er ist in Listing F.38 (auf der nachfolgenden Seite) dargestellt und enthält dort zwei „event“-Elemente, also die minimale Anzahl.

In der BPMN wird dieser Baustein durch das „eventBasedGateway“-Element dargstellt, welchem mindestens zwei „intermediateCatchEvent“-Elemente folgen. Folglich müssen diese drei Elemente für die Darstellung erzeugt werden sowie die Verbindungen zwischen dem Gateway und den Ereignissen. Die ID-Nummer des Bausteins wird zur ID-Nummer des Gateways. Dessen Namen-Attribut nimmt die Bezeichnung auf. Die eingehende Kante des Bausteins wird zur eingehenden

```
<brick11 id="_2">
  <incoming>_10</incoming>
  <outgoing>_11</outgoing>
  <outgoing>_12</outgoing>
  <labeling>Event Gateway</labeling>
  <event>
    <condition>Event A</condition>
    <target_id>_11</target_id>
  </event>
  <event>
    <condition>Event B</condition>
    <target_id>_12</target_id>
  </event>
</brick11>
```

Listing F.38: Baustein #11

Kante des Gateways. Danach werden die „event“-Elemente aus dem Baustein auf die Ereignisse übertragen. Das „condition“-Element übergibt seinen Inhalt an das Namen-Attribut des Ereignisses. Die „target_id“ wird zur ausgehenden Kante. Daraus ergibt sich dann die in Listing F.39 dargestellt XML-Struktur.

```
<eventBasedGateway id="_2" instantiate="false" name="Event Gateway">
  <incoming>_10</incoming>
  <outgoing>_5</outgoing>
  <outgoing>_6</outgoing>
</eventBasedGateway>
<intermediateCatchEvent id="_3" name="Event A">
  <incoming>_5</incoming>
  <outgoing>_11</outgoing>
</intermediateCatchEvent>
<intermediateCatchEvent id="_4" name="Event B">
  <incoming>_6</incoming>
  <outgoing>_12</outgoing>
</intermediateCatchEvent>
<sequenceFlow id="_5" sourceRef="_2" targetRef="_3"/>
<sequenceFlow id="_6" sourceRef="_2" targetRef="_4"/>
```

Listing F.39: abgeleitete XML-Struktur - BPMN (Baustein #11)

Die XML-Struktur der UML Aktivitätsdiagramme ist deutlich größer. Hier wird wieder ein Unterbrechnungsbereich verwendet, der hier mehrere Signale beinhaltet. Eine Gabelung spaltet den Fluss zuvor auf alle Signale auf. Die ID-Nummer des Bausteins wird zur ID-Nummer des Unterbrechnungsbereiches, die Bezeichnung wird zu dessen Namen-Attribut. Als Unterbrechnungskanten werden alle ausgehenden Kanten des Bausteins eingetragen. Die eingehende Kante des Bausteins wird zur eingehenden Kante der Gabelung. Danach werden die „event“-Elemente übertragen. Je eines dieser Elemente wird zu einem „ActionState“-Element mit Signal-Stereotyp. Die Namen dieser Elemente ergeben sich aus dem „condition“-Element, ihre ausgehenden Kanten aus den „target_id“-Elementen. Von der Gabelung ausgehend werden Verbindungen zu den Stati gesetzt. Auf diese Weise ergibt sich die in Listing F.40 (auf der nächsten Seite) dargestellte XML-Struktur.

```
<UML:InterruptibleActivityRegion xmi.id="_2" name="Inter. Activity Region">
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
    <JUDE:ModelElement xmi.idref="_11"/>
    <JUDE:ModelElement xmi.idref="_12"/>
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
    <JUDE:ModelElement xmi.idref="_50"/>
    <JUDE:ModelElement xmi.idref="_5"/>
    <JUDE:ModelElement xmi.idref="_6"/>
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
</UML:InterruptibleActivityRegion>
<!-- ... -->
  <UML:ActivityGraph xmi.id="_01">
  <!-- ... -->
    <UML:Pseudostate xmi.id="_50" name="Fork Node" kind="fork">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_2"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_51"/>
        <UML:Transition xmi.idref="_52"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_10"/>
      </UML:StateVertex.incoming>
    </UML:Pseudostate>
    <UML:ActionState xmi.id="_5" name="Event A">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_2"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:ModelElement.stereotype>
        <UML:Stereotype xmi.idref="_S41"/>
      </UML:ModelElement.stereotype>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_11"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_51"/>
      </UML:StateVertex.incoming>
    </UML:ActionState>
    <UML:ActionState xmi.id="_6" name="Event B">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_2"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:ModelElement.stereotype>
        <UML:Stereotype xmi.idref="_S42"/>
      </UML:ModelElement.stereotype>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_12"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_52"/>
      </UML:StateVertex.incoming>
    </UML:ActionState>
  <!-- ... -->
  </UML:ActivityGraph>
  <!-- Signal-Stereotypen -->
```

Listing F.40: abgeleitete XML-Struktur - UML AD (Baustein #11)

Als letztes wird nun noch die Ableitung der XML-Struktur für Baustein #12 gezeigt. In Listing F.41 ist dieser exemplarisch dargestellt.

```
<brick12 id="_5">
  <incoming>_9</incoming>
  <outgoing>_10</outgoing>
  <outgoing>_11</outgoing>
  <labeling>Action Task</labeling>
  <exit_condition>
    <condition>Exit Condition</condition>
    <target_id>_11</target_id>
  </exit_condition>
</brick12>
```

Listing F.41: Baustein #12

In der BPMN werden für den Baustein zwei Notationselemente für die Darstellung benötigt. Zum einen eine Aufgabe, zum anderen ein spezielles angeheftetes Zwischenereignis. Letzteres wird durch das „boundaryEvent“-Element mit seinem Kind-Element „errorEventDefinition“ dargestellt. Nun wird wie folgt der Baustein auf die XML-Struktur übertragen: Die ID-Nummer des Bausteins wird zur ID-Nummer der Aufgabe. Die Bezeichnung wird zum Namen-Attribut. Die eingehende sowie die ausgehenden Kanten werden ebenfalls direkt übernommen. Der Inhalt des „condition“-Elements wird zum Namen des angehefteten Ereignisses. Dessen Attribut „attachedToRef“ erhält die ID-Nummer des Bausteins. Auf diese Weise wird die Verbindung der beiden Elemente hergestellt. Das Kind-Element „target_id“ gibt seinen Inhalt an das „outgoing“-Element des Zwischenereignisses weiter. Im gleichen Zug wird das zuvor gesetzte „outgoing“-Element wieder aus der Aufgabe entfernt. Dann entspricht es den festgelegten Modellierungsregeln, nach denen eine Aufgabe nur eine ausgehende Kante haben darf. Listing F.42 zeigt die aus dem Baustein generierte XML-Struktur.

```
<task id="_5" name="Action Task">
  <incoming>_9</incoming>
  <outgoing>_10</outgoing>
</task>
<boundaryEvent attachedToRef="_5" cancelActivity="true" id="_6"
                                                    name="Exit Condition">
  <outgoing>_11</outgoing>
  <errorEventDefinition id="_5_ED_1"/>
</boundaryEvent>
```

Listing F.42: abgeleitete XML-Struktur - BPMN (Baustein #12)

Bei den UML Aktivitätsdiagrammen werden ebenfalls mehrere Elemente benötigt, um Baustein #12 darzustellen: Es wird ein Unterbrechnungsbereich benötigt sowie eine Aktion und ein Signal. Danach werden die Werte aus dem Baustein wie folgt in die XML-Struktur eingetragen: Die ID-Nummer des Bausteins wird zur ID-Nummer des Unterbrechnungsbereichs. Die Bezeichnung wird in das Namen-Attribut der Aktion eingetragen. Die eingehende Kante wird zur eingehenden Kante der Aktion. Die ausgehenden Kanten werden als ausgehende Kanten der Aktion gesetzt. Der Inhalt des „target_id“-Elementes wird sowohl bei der Unterbrechnungskante des

Unterbrechnungsbereiches eingesetzt als auch als ausgehende Kante beim Signal. Ebenso wird die ausgehende Kante mit der gleichen Nummer wieder aus der Aktion gelöscht. Auf diese Weise wird den festgelegten Modellierungsregeln entsprochen, bei denen eine Aktion nur eine ausgehende Kante haben darf. Listing F.43 zeigt die generierte XML-Struktur.

```
<UML:InterruptibleActivityRegion xmi.id="_5" name="Interruptible Activity R.">
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
    <JUDE:ModelElement xmi.idref="_11"/>
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-interruptingEdge>
  <UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
    <JUDE:ModelElement xmi.idref="_50"/>
    <JUDE:ModelElement xmi.idref="_51"/>
  </UML:InterruptibleActivityRegion.InterruptibleActivityRegion-node>
</UML:InterruptibleActivityRegion>
<!-- ... -->
  <UML:ActivityGraph xmi.id="_01">
  <!-- ... -->
    <UML:ActionState xmi.id="_50" name="Exit Condition">
      <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_5"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:ModelElement.stereotype>
        <UML:Stereotype xmi.idref="_S01"/>
      </UML:ModelElement.stereotype>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_11"/>
      </UML:StateVertex.outgoing>
    </UML:ActionState>
    <UML:ActionState xmi.id="_51" name="Action Task">
    <UML:ModelElement.interruptibleActivityRegion>
        <UML:InterruptibleActivityRegion xmi.idref="_5"/>
      </UML:ModelElement.interruptibleActivityRegion>
      <UML:StateVertex.outgoing>
        <UML:Transition xmi.idref="_10"/>
      </UML:StateVertex.outgoing>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref="_9"/>
      </UML:StateVertex.incoming>
    </UML:ActionState>
  <!-- ... -->
  </UML:ActivityGraph>
  <!-- ... -->
  <UML:Stereotype xmi.id="_S01" name="signal receipt">
    <UML:Stereotype.baseClass>ActionState</UML:Stereotype.baseClass>
    <UML:Stereotype.extendedElement>
      <JUDE:ModelElement xmi.idref="_50"/>
    </UML:Stereotype.extendedElement>
  </UML:Stereotype>
```

Listing F.43: abgeleitete XML-Struktur - UML AD (Baustein #12)