UNIVERSITÄT SIEGEN

BACHELORARBEIT
IM FACH INFORMATIK

Transformation und Analyse von parallelen Datenbankprozessen

vorgelegt von

Marin BRESSEL

Betreuer:

Prof. Dr. M. LOCHAU,
Professur für Modellbasierte Entwicklung,
Universität Siegen

M. Sc. SCHÜLER,
Professur für Modellbasierte Entwicklung,
Universität Siegen

Eidesstattliche Erklärung

Ich versichere nach § 39 (1) der Einheitlichen Regelungen, meine Arbeit (bei einer Gruppenarbeit meinen entsprechend gekennzeichneten Anteil der Arbeit) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht zu haben.

Ort, Datum

Unterschrift

UNIVERSITÄT SIEGEN

Zusammenfassung

Transformation und Analyse von parallelen Datenbankprozessen

von Marin BRESSEL

[TODO: Deutsche Zusammenfassung]

UNIVERSITY OF SIEGEN

Abstract

Tranformation and Analysis of Parallel Database Processes

by Marin BRESSEL

[TODO: English abstract]

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

LAH	**L**ist **A**bbreviations **H**ere
WSF	**W**hat (it) **S**tands **F**or

Kapitel 1

Einleitung

1.1 Aufbau der Arbeit

Im Rahmen dieser Arbeit wollen wir ermöglichen parallele Datenbankprozesse sowohl zu analysieren als auch zu Transformieren. Um zu verstehen was genau hier passieren soll und wie eine Mögliche Umsetzung aussehen könnte, müssen wir zunächst einige Begriffe definieren und deren Bedeutung verstehen.
Da wir uns im generellen Bereich von Prozessen befinden, muss zunächst erläutert werden was unter einem Prozess zu verstehen ist und auf welche Art von Prozessen wir uns beschränken wollen. Bei Prozessen handelt es sich um sogenannte reaktive Systeme. Es sind also Systeme, welche auf eine Eingabe warten und je nach Zustand entsprechend reagieren. Hierfür sind in jedem erdenklichen Teil des Alltags Beispiele zu finden. Es kann sich um Komplexe Prozesse in Computerspielen oder Betriebssystemen handeln, oder auch um ganz einfache Abläufe im Alltag. Ein einfaches Beispiel ist in Abbildung 1.1 dargestellt. Hier ist der Tagesablauf eines gewöhnlichen Angestellten einer beliebigen Firma zu sehen.
Wir wollen uns in dieser Arbeit zum größten Teil mit Abläufen von Geschäftsprozessen auseinandersetzen. Damit sind unter anderem jene Abläufe gemeint, die innerhalb von beispielsweise Firmen passieren. Es kann sich hier um viele Verschiedene Abläufe handeln.
Ein passendes Beispiel ist in Abbildung 1.2 zu sehen. Es handelt sich hier um einen Prozess, welcher auf einer beliebigen Internetseite abläuft. Wir beschränken uns hier der Einfachheit halber auf den Prozess des Erstellens eines Accounts. Also auf den Registrierungsprozess. In unserem Beispiel muss der Nutzer auf der Homepage zunächst den Button „Registrieren“ drücken. Er wird im Anschluss auf die korrekte Internetseite weitergeleitet und hat da die Möglichkeit seine Anmeldedaten anzugeben. Nun muss die Korrektheit der Daten geprüft werden. Im Anschluss wird der Nutzer erstellt und wieder auf die Homepage zurück geleitet.
Solche und ähnliche Prozesse laufen in jeder erdenklichen Firma ab und unterscheiden sich sehr stark voneinander in Komplexität und Inhalt.
Bei dem zweiten Begriff, der verstanden werden muss handelt es sich um BPMN (Business Proccess Moddeling Notation). Hierbei handelt es sich um eine Standardisierte Modellierungssprache. Die in den Abbildungen 1.1 und 1.2 dargestellten Beispiele wurden bereits in BPMN dargestellt. Eine Detailliertere Beschreibung erfolgt im Abschnitt 2.1 doch die wichtigsten Features sollen hier schon einmal erwähnt werden. Sie sind, neben den Aktivitäten welche durch abgerundete Rechtecke dargestellt werden und der Konkatenation dieser Aktivitäten durch die Pfeile, die sogenannten Gateways, welche durch Rauten mit einem Entsprechenden Symbol dargestellt werden. Ist ein *X* in die Raute geschrieben so handelt es sich um ein Exklusives OR-Gateway. Hier kann exakt eine der folgenden Pfade ausgeführt werden. Ist ein + in die Raute geschrieben, so liegt ein paralleles Gateway vor. Hier werden alle Pfade

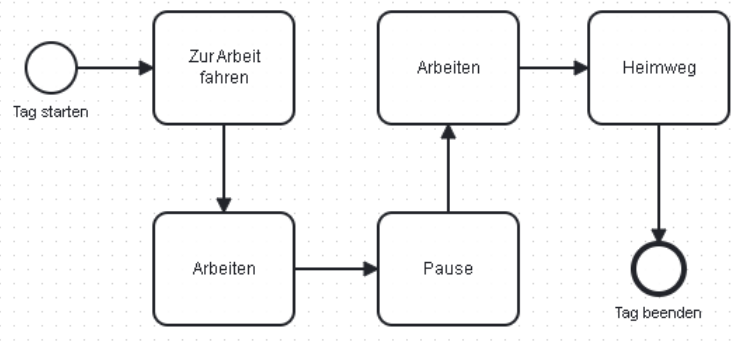

ABBILDUNG 1.1: Alltag eines Mitarbeiters

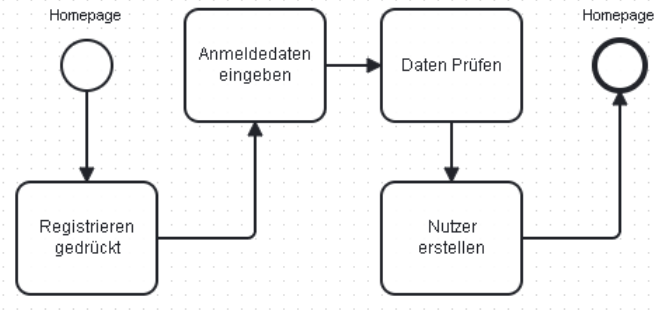

ABBILDUNG 1.2: Einfacher Registrierungsprozess

gleichzeitig ausgeführt. Ein weites paralleles Gateway vereinigt die Pfade dann wieder. An dieser Stelle wird also gewartet, bis alle parallelen Aktivitäten durchgeführt worden sind. Diese Arbeit beschäftigt sich ausschließlich mit Prozessen in BPMN.Es wird allerdings nur eine Teilmenge benutzt.
Zuletzt muss nun noch erläutert werden zu was diese Prozesse transformiert werden sollen. Hierzu müssen wir den Begriff einer Prozessalgebra genauer erläutern. Bei einer Prozessalgebra handelt es sich nun um einen mathematischen Kernkalkül zur Darstellung von Prozessen. Es bietet also eine Möglichkeit die selben Prozesse durch eine Art Formeln darzustellen. Wir wollen in dieser Arbeit eine Prozessalgebra verwenden, welche auf ACP basiert. Zum besseren Verständnis möchte ich auch hier die wichtigsten Features von ACP erläutern. Ähnlich zu BPMN gibt es Möglichkeiten zur Darstellung von Konkatenation, oder-Zweigen und auch Parallelen Zweigen. Eine Aktivität wird durch einfach als Variable dargestellt. Wie diese genannt wird, ist zunächst irrelevant. Eine Konkatenation wird durch ein * dargestellt. Ein Exklusiver oder-Zweig wird durch ein + und ein paralleler Zweig durch || dargestellt. Würde man das Beispiel in Abbildung 1.2 also nun nach ACP überführen, so würde ein Mögliches Ergebnis wie folgt aussehen:
$P := Start * RegestrierenGedruckt * AnmeldeDatenEingeben * DatenPrufen * NutzerErstellen * Homepage$

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist es nun Programm zu entwickeln, welches ein BPMN Diagramm einlesen und dieses zu einer Formel in der Prozessalgebra überführen kann.

Es soll also ein Mapping von BPMN nach ACP erfolgen.
Der Zweck dieser Arbeit wird erkennbar, sobald Daten in form einer relationalen Datenbank für den Prozess relevant werden. BPMN bietet hier nur schwammige und unübersichtliche Methoden die Daten darzustellen. Es erweist sich also als schwierig zu Analysieren und zu verifizieren, was genau mit den Daten passiert. Zudem lässt BPMN zu viel Raum für Interpretation. Es wird viel mit Kommentaren gearbeitet und was genau in einem bestimmten Abschnitt des Prozesses passiert ist in vielerlei Hinsicht offen zur Interpretation. ACP ist etwas Strukturierter. Zudem bietet die Variante die von uns genutzt wird die Möglichkeit mit einer konkreten Datenbank zu kommunizieren und Anweisungen auf dieser auszuführen. Aus diesem Grund ist ein weiteres Ziel dieser Arbeit, die passenden Datenbankanweisungen zu dem jeweiligen Prozess auf einer relationalen Datenbank auszuführen.

Kapitel 2

Grundlagen

Das folgende Kapitel dient dazu die in der Einleitung genannten Begriffe und Prinzipien genauer zu erläutern. Wir wollen also zunächst BPMN und die Bestandteile der Modellierungssprache genauer betrachten. Hierzu werden wir jedes Element einzeln betrachten und dadurch stück für Stück ein passendes Beispiel erstellen. Dieses Beispiel wird auch im Rest der Arbeit Relevanz finden. Im Anschluss wird dann die verwendete Prozessalgebra genauer betrachtet. Auch hier wollen wir jeden Bestandteil des Modells im Detail erläutern.

2.1 BPMN

Wie in der Einleitung bereits erwähnt, dient BPMN der graphischen Darstellung von Business Prozessen. In BPMN werden diese Prozesse in einzelne Aktivitäten oder Aufgaben unterteilt und dann in der richtigen Reihenfolge aufgezeichnet. Es gibt unterschiedliche Möglichkeiten Verzweigungen, Abhängigkeiten und Ähnliches zu Modellieren doch zum größten Teil basiert alles auf der korrekten Aneinanderreihung dieser Aktivitäten.

2.1.1 Erste Schritte - Events und Aktivitäten

Im folgenden Abschnitt schauen wir uns also die von uns verwendete Teilmenge der Modellierungssprache BPMN an. Hierzu möchten wir zunächst einen einführenden Prozess als Beispiel betrachten. Für alle möglichen Homepages und Websites von unterschiedlichen Firmen und Anbietern können Konten erstellt werden. Diese dienen zur Wiedererkennung eines Kunden oder Mitarbeiters. In diesem Abschnitt wollen wir ein BPMN-Diagramm erstellen, welches den Registrierungsprozess auf einer solchen Website darstellen könnte. Wir werden die Modellierungssprache hierzu aufteilen und jeden Bestandteil der Sprache anhand des Beispiels einzeln erläutern, sodass wir am ende dieses Abschnittes ein erstes, vollständiges Diagramm vorfinden.
Die von uns genutzte Teilmenge kann in einzelne Blöcke eingeteilt werden. Diese können in zwei Gruppen unterteilt werden. Die Basic Blocks und die Flow Blocks. Des Weiteren kann unterschieden werden zwischen *Leaf Blocks* und *Nonleaf Blocks*. Alle Nonleaf Blocks bestehen aus beliebig vielen Leaf Blocks. Es existieren genau zwei für uns relevante Leaf Blocks, welche wir hier zunächst erwähnen möchten, um deren Funktion genauer zu erläutern. Die unterschiedlichen Blöcke sind jeweils durch sogenannte *sequnce flows* verbunden. Diese werden dargestellt durch eine durchgezogene Linie mit einem ausgefüllten Pfeil am Ende. Sie verdeutlichen den Fluss des Diagrammes. Jeder Block hat einen eingehenden und einen ausgehenden Flow. Um zu verdeutlichen in welchem Zustand der Prozess sich zu einem bestimmten

ABBILDUNG 2.1: Einfache Task - Anmeldedaten eingeben

Zeitpunkt befindet, verwenden wir sogenannte Token. Diese beinhalten keine Daten, sondern stellen nur dar, welcher Teil des Prozesses gerade aufgeführt wird. Beim Ausführen des Prozesses wird ein Token immer in Richtung der Sequence Flows weitergegeben.
Bei dem *Task Block* und dem *Event Block* handelt es sich um die beiden relevanten leaf-Blocks. Beide Blöcke werden durch den Namen bereits gut erklärt. Der Task Block, welcher durch ein abgerundetes Rechteckt dargestellt wird, ist repräsentativ für eine beliebige Aufgabe. Diese Aufgaben benötigen in jedem Fall Zeit, um ausgeführt zu werden. Er kann durch einen Text in der Mitte des Rechtecks beliebig benannt werden. Diese Aufgaben können jede erdenkliche Form annehmen. Eine Mögliche Aufgabe ist in Abbildung 2.1 dargestellt. Es handelt sich um das Eingeben der Anmeldedaten.Hierbei handelt es sich um eine Aufgabe die Zeit beansprucht und vom Nutzer ausgeführt wird. Sie ist offensichtlich relevant für unser Beispiel.
Bei dem zweiten Leaf Block handelt es sich nun um sogenannte Events. Events werden durch einen Kreis dargestellt. Anders als die Aufgaben passieren Events sofort. Es gibt unterschiedliche Typen, welche durch unterschiedliche Variationen eines Kreises dargestellt werden. Für uns relevant sind allerdings nur sogenannte *catching Events*. Erreicht der Token ein solches Event, wird gewartet bis das erwartete Event auftritt und erst dann wird der Token weitergeschickt. Auch unter den catching Events gibt es drei grundsätzliche Unterscheidungen. Die einfachsten Variationen sind sogenannte Startevents, welche einen Prozess Starten und durch einen einfachen dünn gezeichneten Kreis dargestellt werden und die Endevents, welche den Prozess terminieren und durch einen einfachen dick gezeichneten Kreis dargestellt werden.
Durch die uns nun bekannten Bausteine, ist es uns möglich einen ersten Prozess aufzubauen und in BPMN zu Modellieren. In Abbildung 2.2 sehen wir ein Beispiel für den ersten und einfachsten nonleaf-Block. Den *Prozess-Block*. Dieser besteht aus einem Start- und einem Endevent. Zwischen den beiden Events liegt mindestens ein Task-Block und beliebig viele Event Blocks. Der Prozess in unserem Beispiel startet, indem ein Nutzer die Seite besucht und einen neuen Account anlegen möchte. Dieser wird zunächst auf eine Seite weitergeleitet, welche seine Anmeldedaten abfragt. Diese muss er dann in der ersten Aufgabe eingeben. Die zweite Aufgabe besteht nun darin den neuen Nutzer mit seinen eben angegebenen Daten in die vorhandene Datenbank einzutragen. Sobald diese Aufgabe erledigt ist, endet der Prozess und ein neuer Nutzer wurde erfolgreich angelegt.
Events können allerdings auch während eines Prozesses auftreten. Wollen wir in unserem Beispiel einführen, dass die Startseite besucht werden kann, ohne, dass direkt auf die Nutzer anlegen Seite weitergeleitet wird, so können wir eine neue Aufgabe und ein fangendes Event einführen. Abbildung 2.3 zeigt diesen Prozess. Er startet,

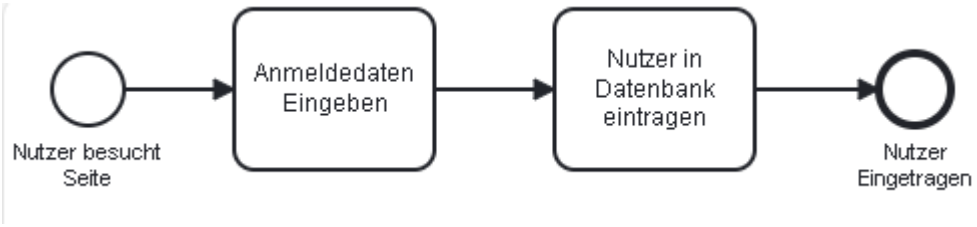

ABBILDUNG 2.2: Erster Vollständiger Prozessblock

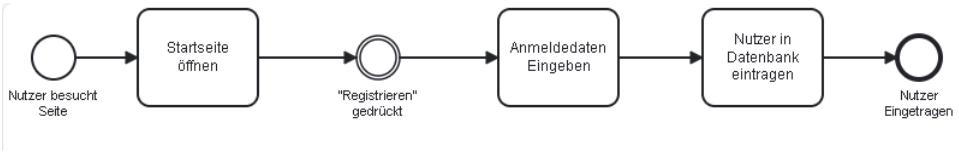

ABBILDUNG 2.3: Beispiel für ein intermediate Event

indem ein Nutzer die von uns erstellte Website besucht. Im nächsten Schritt wartet das catching Event, bis der Nutzer sich dazu entscheidet einen neuen Account anzulegen. In Folge dessen verhält sich der Prozess so, wie im Beispiel aus Abbildung 2.2. Events welche während dem Prozess auftreten werden *Intermediate Events* genannt.

2.1.2 Mögliche Verzweigungen

In nun folgenden Abschnitt wollen wir uns mit Möglichkeiten beschäftigen den Sequenzfluss zu verzweigen. Hierzu bietet BPMN einige sogenannte *Gateways*. Um Gateways darzustellen, werden Rauten verwendet. Unterschiedliche Gateways haben unterschiedliche Symbole in den Rauten eingezeichnet. Sie sind dafür da, den Verlauf des Prozesses aufzuteilen und wieder korrekt zusammenzufügen. Wenn ein Gateway den Verlauf des Prozesses aufspaltet, so muss immer ein zweites Gateway den Verlauf wieder zusammenfügen. Eine Ausnahme hierfür wäre, wenn der Prozess in einer Verzweigung durch ein Endevent terminiert.
Um unser bislang erarbeitetes Beispiel etwas realitätsnaher zu gestalten, wollen wir eine neue Funktion einführen. Offensichtlich soll unsere Seite mehr Funktionen anbieten als neue Nutzer anzulegen. Die nächste logische Erweiterung ist eine Möglichkeit für bereits bestehende Kunden sich mit ihren vorhandenen Anmeldedaten einzuloggen. Hier spielt die erste Verzweigungsmöglichkeit eine Rolle. Da ein Nutzer immer entweder ein bestehendes Konto besitzt oder ein neues erstellen möchte, wird in jedem Durchlauf des Prozesses nur eine dieser Aktionen durchgeführt. Hier kann ein XOR-Gateway genutzt werden. Es ist zu erkennen durch ein *X* in der Raute. Es bietet die Möglichkeit zwischen unterschiedlichen Pfaden zu wählen. Im Beispiel aus Abbildung 2.4 sehen wir eine mögliche Verwendung für dieses Gateway. Nachdem der Nutzer auf der Startseite den entsprechenden Button betätigt, wird er entweder zur Anmeldung oder zur Registrierung weitergeleitet. In beiden Fällen muss er seine Daten angeben. Je nachdem in welchem Teil wir uns befinden muss der Nutzer in der Datenbank eingetragen werden oder die Anmeldedaten müssen geprüft werden. Im Anschluss werden die zwei Äste wieder zusammengeführt und der Nutzer wird mit seinem zugehörigen Account auf der Seite angemeldet. Beim Zusammenführen der Äste, wartet das Gateway auf genau einen Token und gibt diesen dann weiter an das nächste Objekt. In diesem Beispiel ist der erste Flow-Block zu

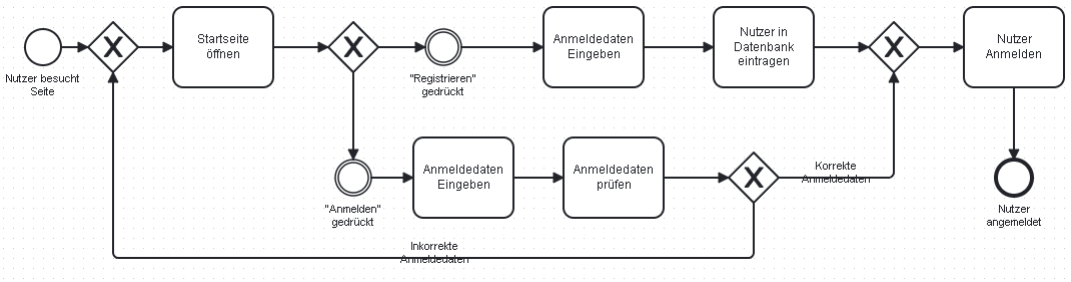

ABBILDUNG 2.4: Exclusive Entscheidung - Das XOR-Gateway

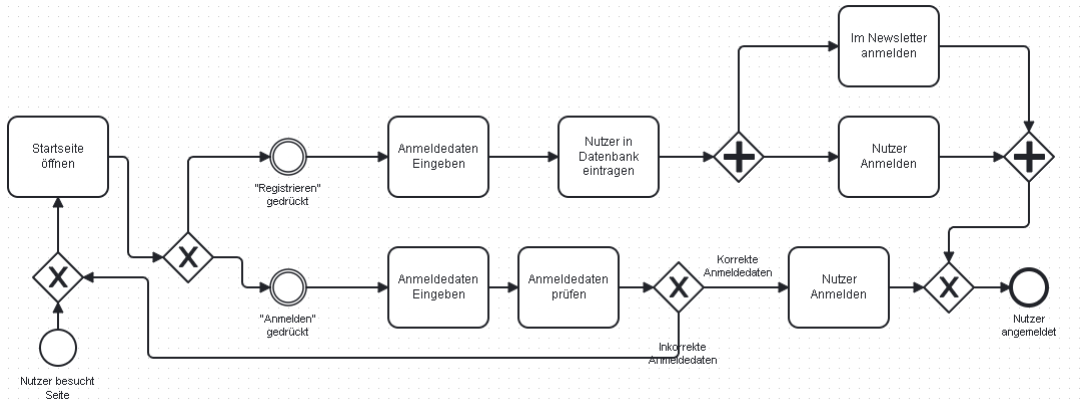

ABBILDUNG 2.5: Parallele Ausführung - Das Paralell-Gateway

erkennen. Wird der Verlauf des Prozess durch ein XOR-Gateway aufgespalten und wieder zusammengeführt, so bilden alle Elemente innerhalb dieser Verzweigung im sogenannen *Exclisive-choice-Block*. Abbildung 2.4 zeigt außerdem eine Mögliche Variante des XOR-Gateways. Wenn der Nutzer einen vorhandenen Account anmelden möchte, aber inkorrekte Anmeldedaten eingibt, so wird er auf die Startseite zurückgeleitet. Das XOR-Gateway kann also auch für Loops verwendet werden. Hier ist zu erkennen, dass das zusammenführende Gateway durchaus auch vor dem aufspaltenden Gateway liegen kann. Es ist zusätzlich möglich auch mehr als zwei Ausgänge an das Gateway anzubinden.
Wir wollen nun ein weiteres Feature in unser Diagramm einfügen. Nachdem ein neuer Account erstellt wurde, soll dieser Nutzer nach wie vor angemeldet und auf die Homepage weitergeleitet werden. Zusätzlich soll er gleichzeitig auch für den Newsletter der Website eingetragen werden. Hierzu können wir ein weiteres Gateway nutzen. Das parallele Gateway wird in Abbildung 2.5 das erste Mal gezeigt. Es wird ebenfalls durch eine Raute dargestellt. Diese enthält allerdings ein + in ihrem Inneren. An diesem Gateway wird der Verlauf des Prozesses wieder aufgespalten. Anstelle von nur einem Strang werden hier aber alle Gelichzeitig ausgeführt. Das zusammenführende Gateway muss desshalb nicht blind den ersten Token weiterschicken, den es erhält, sondern wartet bis an allen Eingängen ein Token vorliegt, fügt diese wieder zusammen und gibt dann den Token weiter an das nächste Objekt. Hier können Schwierigkeiten auftreten, falls ein paralleler Zweig in einem Endevent endet. Bei der Modellierung eines Diagrammes muss dies also verhindert werden. Insgesamt spricht man hier von einem *Parallel Block*

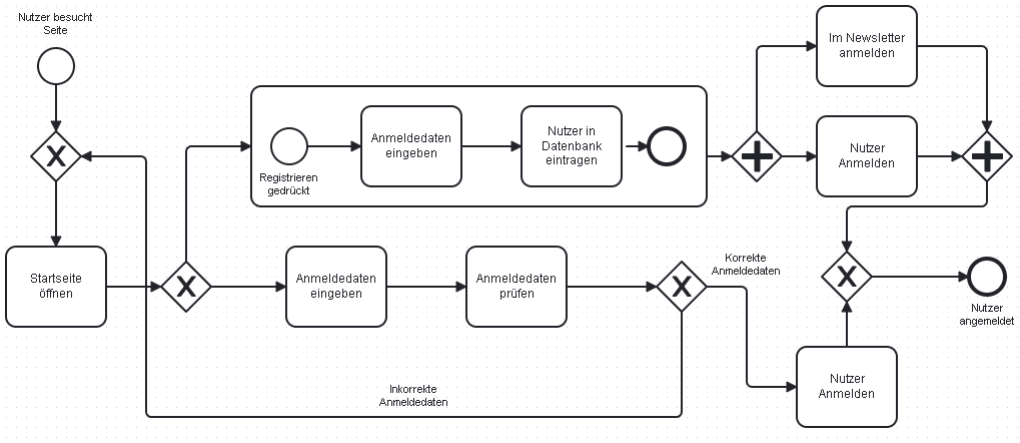

ABBILDUNG 2.6: Subprocess Block - Prozesse innerhalb von Prozessen

2.1.3 Subprozesse und weitere Elemente

Der letzte Block, welcher von uns genauer betrachtet wird ist der *subprocess-Block*. Auch dieser wird bereits durch den Namen gut beschrieben. Zur vereinfachten Darstellung, kann ein Teil von einem Prozess als ein unterprozess zusammengefasst werden. In Abbildung 2.6 ist also der selbe Prozess dargestellt wie der in Abbildung 2.5. Hier wurde der Teil des Prozesses, in dem ein Nutzer registriert wird zu einem Subprocess zusammengefasst. In Abbildung 2.7 findet sich zudem eine Alternative Darstellung dieses Blockes. Hier ist zu sehen, dass der Subprozess auch als einzelnes Elemt auftauchen kann. An der Stelle wird dann eine referenz zu einem Prozess gemacht. Dieser enthält die Elemente die von dem Subprocess ausgeführt werden. In dieser Abbildung ist zustätzlich noch eine Übersicht über alle für uns relevante Elemente zu finden.
Zuletzt möchten wir nun jene Elemente beschreieben, die für unsere Arbeit nicht wichtig werden. Hierbei handelt es sich zum großteil um Varianten der uns bekannten Bausteine. Neben den Exclusive und Parallel Gateways existieren noch so genannte Event-based Gateways. Diese entscheiden anhand eines eintreffenden Events, wleche Aktion ausgefürt wird. Die Task blocks lassen sich durch Symbole in der oberen linken Ecke genauer spezifizieren. Hier wird unterschieden zwischen User task, Service task, Business rule task und Script task. Auch unter den Events kann genauer spezifiziert werden. Für alle catching events sind throwing events vorhanden und auch unter diesen gibt es unterschieldiche Typen wie Message Evens oder timer Events. Zudem gibt es eine Möglichkeit Datenflüsse darzustellen durch eine sogenannte *Date object reference* oder eine *Data store reference*. Zuletzt bietet BPMN eine möglichkeit verschiedene Teilnehmer einer Prozesses darzustellen. Sogenannte *pools* und *lanes* können genutzt werden um Gruppen und auch einzenle Teilnehmer innerhalb dieser Gruppen darzustellen.

2.2 Prozessalgebra - ACP

Nachdem wir die wichtigsten Bestandteile der Modellierungssprache BPMN eingeführt haben, wollen wir uns nun mit der Prozessalgebra ACP beschäftigen und diese mit ihren einzelnen Bestandteilen genauer erläutern.

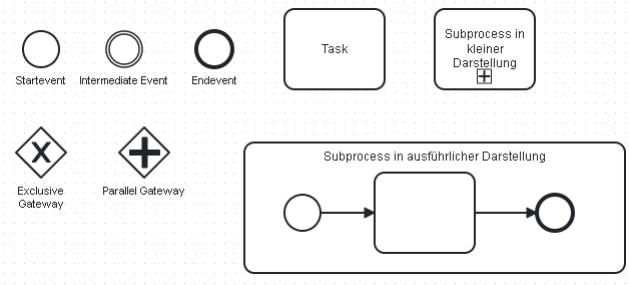

ABBILDUNG 2.7: Alle von uns genutzten Elemente

Parallel zu den Tasks in BPMN sprechen wir hier von sogenannten Aktionen. Diese werden klein geschrieben und können ansonsten beliebig benannt werden. Auch Prozesse und Sub-prozesse werden hier angewendet. Die Syntax für einen Prozess welcher eine beliebige Aktion ausführt ist durch P::=v gegeben. Ein mögliches Beispiel ist auch hier wieder das Registrieren eines neuen Nutzers. P::=nutzerRegistrieren beschreibt also den Prozess P welcher die Aktion nutzerRegistrieren ausführt. Analog dazu können auch Subprozesse dargestellt werden. P::=K beschreibt den Prozess P welcher daraus besteht die Prozess K auszuführen. Dies kann zur vereinfachten Darstellung von ausführlicheren Prozessen oder zur Darstellung von Rekursion genutzt werden.
Ähnlich wie bei BPMN können Prozesse und Aktionen in ACP konkateniert werden. Hierzu wird der *-Operator verwendet. Der Prozess, welcher in Beispiel 2.2 als BPMN Diagramm dargestellt ist, kann also durch P::=anmeldedatenEingeben*nutzerInDbEintragen in ACP dargestellt werden. Hier fällt auf, dass sowohl das Startevent als auch das Endevent nicht in der Formel enthalten sind. In ACP gibt es generell kein Äquivalent zu den Events.
Anders als bei den Events, gibt es für die Gateways welche in Abschnitt 2.1 erläutert wurden auch in ACP Darstellungsmöglichkeiten. Zunächst wollen wir die XOR-verknüpfung erklären. Diese wird durch den +-Operator dargestellt. Auch hier kann ein zuvor durch BPMN modelliertes Beispiel verwendet werden. Wir können hierzu das in Beispiel 2.4 gezeigt Diagramm in eine Formel in ACP umwandeln.
P::=sO*((aE*nE)+(aE*aP))*nA
Die Klammern können so interpretiert werden, dass die umschlossenen Terme zusammenhängen. Es wird in dem Beispiel also entweder (aE*nE) oder (aE*aP) ausgeführt.
Außerdem wurden folgende Abkürzungen genutzt.
sO -> Startseite Öffnen
aE ->Anmeldedaten Eingeben
nE ->Nutzer Eintragen (In DB)
aP ->Anmeldedaten Prüfen
nA ->Nutzer Anmelden
Als nächstes wollen wir nun veranschaulichen wie das Parallel-Gateway aus BPMN in APC umgesetzt wird. Hierzu gibt es drei unterschiedliche sogenannte Merge-Operatoren. Diese sollen hier nun einzeln erklären. Zunächst betrachten wir den regulären Merge-Operator. Dieser wird durch ein || dargestellt und beschreibt das parallele Ausführen zweier Prozesse. Sollte der Prozess P also durch P::=Q||R beschrieben werden, so ist egal welcher der beiden Subprozesse Q oder R zuerst eine Aktion ausführt.
Als nächstes beschreiben wir nun den Left-Merge-Operator. Dieser wird durch ein ⌊⌊

dargestellt. Auch hier werden beide Subprozesse ausgeführt. Es muss allerding zuerst der Linke der beiden eine Aktion ausführen. Im Anschluss verhält sich der Operator genauso wie der reguläre Merge-Operator. Zuletzt betrachten wir den Communication-Merge-Operator, welcher durch ein | dargestellt wird. Hier werden beide Subprozesse gleichzeitig in einem Schritt ausgeführt.

2.3 Datenbanken

Im folgenden Kapitel werden nun Relationale Datenbanken genauer beschrieben. Wir betrachten zunächst ein Relationsschema $\mathcal{R} = \{R_1/a_1, \ldots, R_n/a_n\}$. Dies beinhaltet eine Menge an Relationen R_i, welche jeweils eine zugehörige Stelligkeit a_1 besitzen. Dieses Schema beschreibt unsere Datenbank. Jede Relation R_i aus $\mathcal{R}$ kann als Tabelle der Datenbank angesehen werden.
Auch hier findet das in den Vorherigen Abschnitten verwendete Beispiel Anwendung. Um den Registrierungs- und Anmeldeprozess umsetzen zu können, müssen zumindest die Kontodaten jedes Nutzers gespeichert werden. Diese beinhalten in unserem Beispiel lediglich den Nutzernamen und das Passwort. Unser Beispiel benötigt also nur eine Tabelle. Das bedeutet, dass in unserem Relationsschema nur eine Relation hinterlegt ist. Diese könnte beispielsweise den Namen Nutzer tragen. Da die Tabelle Nutzer aus den zwei Spalten Nutzername und Passwort besteht beträgt die Stelligkeit der Relation zwei. Das Relationsschema aus unserem Beispiel lautet also $\mathcal{R}_1 = \{Nutzer/2\}$.
Zusätzlich definieren wir den Begriff der Domäne Δ einer Datenbank als jene zählbar unendliche Menge, welche sämtliche Datentypen enthält, die in der jeweiligen Datenbank vorkommen können. Sie beschränkt sich in unserem Beispiel also auf alle Strings.
Eine Instanz $\mathcal{I}$ einer Datenbank ist ebenfalls als eine Menge zu betrachten. Diese Menge enthält sämtliche Daten, die zu einem gewissen Zeitpunkt in der Datenbank abgespeichert werden. Diese Daten werden in Form von Tupeln abgespeichert. Ein Tupel enthält jeweils den Namen der jeweiligen Relation, sowie die Daten eines Elements. Auch hier dient das oben bereits verwendete Beispiel gut zu Veranschaulichung. Befindet beispielsweise ein Nutzer mit dem Nutzernamen *Marin* und dem Passwort *Test* in der Datenbank, so ist $\mathcal{I}_1 = \{(Nutzer, Marin, Test)\}$. Wird nun ein weiterer Nutzer mit dem Nutzernamen *Max* und dem Passwort *pw* hinzugefügt, so wird $\mathcal{I}$ um das Tupel (Nutzer, Max, pw) erweitert. Also ist $\mathcal{I}_2 = \{(Nutzer, Marin, Test), (Nutzer, Max, pw)\}$. Eine Alternative Schreibweise ist durch $\mathcal{I}_2 = \{Nutzer(Marin, Test), Nutzer(Max, pw)\}$ gegeben. Der Name der Relation kann also auch außerhalb der Klammern stehen.
In Abbildung 2.8 ist zur veranschaulichung eine Tabelle zu sehen, welche die Inhalte von $\mathcal{I}_2$ überscihtlicher darstellen soll.
Wollen wir die Datenbank nun um eine Weitere Tabelle erweitern, welche beispielsweise zwei Nutzer durch eine Freundschaft verknüpft und zusätzlich speichert seit wie vielen Jahren diese Freundschaft besteht, so wird kann das gegebene Schema $\mathcal{R}_1$ um *Freuzndschaft/3* erweitert werden. Also $\mathcal{R}_2 = \{Nutzer/2, Freundschaft/3\}$ dei drei Attribute der Relation Freundschaft sind hierbei Freund_1, Freund_2 und Dauer. Das Attribut Dauer beschreibt seit wie vielen Jahren eine Freundschaft besteht und wird als Integer angegeben. Durch diese Änderung wird die Domäne Δ zusätzlich um sämtliche Integer erweitert. Fügen wir nun eine Freundschaft zwischen *Max* und *Marin*, welche seit 0 Jahren besteht der Datenbank hinzu, so wird $\mathcal{I}$ um das Tupel (Freundschaft, Max, Marin, 0) erweitert. Also $\mathcal{I}_3$= {Nutzer(Marin, Test), Nutzer(Max,

Nutzer	
Nutzername	Passwort
Marin	Test
Max	pw

ABBILDUNG 2.8: $\mathcal{I}_2$ graphisch dargestellt

Nutzer	
Nutzername	Passwort
Marin	Test
Max	pw

Freundschaft		
Freund_1	Freund_2	Dauer
Max	Marin	0

ABBILDUNG 2.9: $\mathcal{I}_3$ graphisch dargestellt

pw),Freundschaft(Max, Marin, 0)}. Abbildung 2.9 zeigt die Instanz in Form von Tabellen.
Das sogenannte *Universum* enthält alle möglichen Instanzen in Bezug auf das Schema und der Domäne. $\mathcal{I}_1$, $\mathcal{I}_2$ und $\mathcal{I}_3$ sind also Elemente des Universums.
Nun wollen wir betrachten, wie Datenbanken manipuliert werden können.
Hierzu müssen wir zunächst betrachten, wie Daten aus der Datenbank ausgelesen werden können.
Dazu nutzen wir sogenannte Queries. Diese werden in FOL ausgedrückt. Hierzu definieren wir zunächst eine Menge $VARS = \{u1, \ldots, un\}$. Diese Menge besteht aus Variablen, welche alle Werte aus Δ annehmen können. Zusätzlich bildet eine Substitution σ alle Variablen auf Vars auf werte in Δ ab.
Eine Query ϕ ist durch folgende Syntax gegeben:
ϕ ::= true | R(u1, . . . , ua) | $\neg\phi$ | $\phi 1 \wedge \phi 2$ | $\exists$u.ϕ | u1 = u2
Wir wollen nun die Semantik dahinter erläutern.

- Für ϕ ::= true besteht die Query in jedem Fall.
- Für ϕ ::= R(u1, . . . , ua) besteht die Query, falls das Tupel R(e1, … ,ea) in I vorliegt. Hierbei ist ei=σ(ui) für alle i: 1 </= i </= a.
- Für ϕ ::=¬ Q besteht die Query, falls Q nicht bestehen würde.
- Für ϕ ::= Q1 $\wedge$ Q2 besteht die Query, wenn Q1 und Q2 bestehen würden
- Für ϕ ::= u1 = u2 besteht die Query, wenn σ(u1)=σ(u2)
- Für ϕ::= $\exists$u.Q besteht die Query, wenn in der aktuellen Instanz der Datenbank ein Element existiret, welches durch eine Substitution σ' auf u abgebildet werden kann.

Die Manipulation einer Datenbank besteht aus drei Phasen. Zunächst muss ein eine Query, welche wir als Guard bezeichnen bestehen. Im Anschluss kann die Aktion ausgeführt werden. Die Aktion besteht aus dem löschen eines Elements und dem

Hinzufügen eines Elements. Ein Element beschreibt dabei ein Tupel in der Menge $\mathcal{I}$ Hier ist anzumerken, dass es keine Möglichkeit gibt Inhalte der Datenbank zu ändern. Möchte also ein Nutzer sein Passwort ändern, so wird zuerst in einem Schritt der jeweilige Nutzer entfernt und im nächsten Schritt ein neuer Nutzer mit dem veränderten Passwort der Datenbank hinzugefügt.

Kapitel 3

Mapping

3.1 Parallelen und Unterschiede

Im nun folgenden Kapitel wollen wir erläutern in wie fern sich BPMN auf ACP mappen lässt. Hierzu wollen wir zunächst einige parallelen und Unterschiede betrachten.
Da sowohl die Moddelierungssprache BPMN als auch die Prozessalgebra ACP genutzt werden um auf unterschliedliche Art Prozesse darzustellen, finden sich zwischen den beiden viele Parallelen. Die von uns betrachtete Teilmenge von BPMN bietet sehr gute Möglichkeiten ein Mapping durchzuführen. Einige wichtige Aspekte wurden bereits im Abschnitt 2.2 genannt. Die Tasks aus BPMN können als Aktivitäten in ACP angesehen werden. Beispielsweise könnte also der Prozess aus Abbildung 2.1 beschrieben werden durch $P := AnmeldedatenEingeben$. Die Konkatenation von unterschiedlichen Tasks durch die sequence flows können in ACP auch einfach durch den *-Operator dargestellt werden. Auch die Aufspaltung der Diagramme durch die Gateways kann in ACP leicht umgesetzt werden. Der +-Operator und der ||-Operator bieten ein Äquivalent zu den Exclusiven und Parallelen Gateways. Ebenso leicht lassen sich auf Sub-Prozesse umsetzen. Betrachten wir das Beispiel aus Abbildung x.xx, so können wir auch in ACP eine dazu Äquivaltene Menge an Prozessen verfassen. Diese wäre wie folgt definiert **"TODO"**.
Die ersten Komplikationen treten auf bei dem betrachten von Events. In ACP gibt es keine Möglichkeit den Prozess aufzuhalten bis ein Ereignis auftritt. In der hier verwendeten Teilmenge, kann diese Problematik allerdings recht leicht umgangen werden. Einige Objekte wie pools und lanes werden von uns nicht verwenden. Zusätzlich werden auch werfende Events nicht genauer betrachtet. Dadurch ist es ohne größere Konsequenzen möglich, Events genauso wie Tasks zu behandeln und auf die Aktivitäten zu mappen. Da keine werfenden Events auftreten, können die fangenden Events nur von außerhalb des Prozesses getriggert werden. Dadurch ist es möglich das Event durch eine Aktivität zu beschreiben welche Terminiert, sobald das Ereignis auftritt. Startevents und Endevents können genauso durch aktivitäten beschrieben werden wie intermediate Events.
Eingige Beispiele wie das Mapping durchgeführt werden könnte sind in Abbildung x.xx zu finden. **"TODO"** In der Abbildung ist leicht zu erkennen, dass Formeln in ACP sehr schnell unübersichtlich werden. Desshalb wollen wir durch das Mapping den Prozess in BPMN in eine Menge von Prozessen und Subprozessen in ACP aufteilen. In Kapitel 4 wird sich zusätzlich zeigen, dass das aufspalten eines Prozesses in mehrere Subprozesse die Implementierung des Mappings deutlich vereinfacht.

3.2 Aufspaltung in Subprozesse

Im folgenden Abschnitt wollen wir nun genauer betrachten an welchen Stellen ein Prozess am Sinnvollsten aufgespalten werden kann. Zusätzlich müssen noch einige Annahmen und Einschränkungen über die beiden Arten von Gateways gemacht werden.
Im folgenden werden Tasks und Events gleichgesetzt behandelt und es wird nicht mehr zwischen den beiden unterschieden. Beide Elemente werden nur als Aktivitäten benannt. Folgt auf eine gewöhnlich Aktivität genau eine weitere Aktivität, so wird in ACP kein Subprozess erstellt. Es wird eine einfache Konkatenation der beiden Aktivitäten dem Prozess hinzugefügt. Tritt also kein Gateway in dem Prozessdiagramm auf, so wird der Prozess in ACP auch nicht aufgespalten. Das Mapping verhält sich also im Fall der ersten Zeile so wie in Abbildung x.xx.
Für alle weiteren Beispiele aus der Abbildung, werden allerdings Subprozesse erstellt. Die Abbildungs zeigt also nicht die sleben Lösungen wie jene, die von unserem Programm erstellt werden sollen. Hier ist zu beobachten, dass mehrere korrekte Lösungen für das Mapping von BPMN nach ACP existieren.
Ein Subprozess wird in unserem Fall für jeden ausgehenden sequence flow von jedem im Diagramm vorkommenden Gateway erstellt. Zuerst wollen wir das Exclusive Gateway näher betrachten. In Abbildung x ist in einem Beispiel dargestellt wie sich das Mapping verhalten soll. In der dritten Spalte ist eine weite korrekte Lösung vorgezeigt. Diese ist dazu da um zu zeigen wie sich unser Programm nicht verhalten soll. Jeder Prozess listet alle Aktivitäten auf bis das er auf ein Gateway stößt. An dem Gateway selbst wird ein neuer Prozess für jeden ausgehenden sequence Flow erstellt und im Anschluss nicht mehr weitergearbeitet. P0 arbeitet also alle Aktivitäten bis zum Gateway ab, verweist dann auf die Prozesse P1 und P2 und terminiert im Anschluss. Es ist zu sehen, dass für jeden ausgehenden seqeunce Flow ein neuer Subprozess erstellt wurde.
Ebenso ist an dem zusammenführenden Gateway zu kennen, dass auch hier für den ausgehenden Sequenceflow ein neuer Subprozess erstellt wurde. Hier kann nicht einfach eine Konkatenation verwendet werden. Zum einen ist der Abbildung zu sehen, dass es schnell zu unübersichtlichkeit führen kann, da sowohl in P1 als auch in P2 enthalten sein muss wie sich der Prozess nach dem Gateway verhält. Zum anderen hilft das erstellen von Subprozessen bei der Umsetzung von Rückkopplungen und Rekusrionen.
In Abbildung x.xx ist ein weiteres Beispiel zu sehen. Hier wird schnell deutlich, dass es nützlich ist auch für das zusammenführende Gateway zu beginn des Prozesses einen Subprozess zu erstellen, damit der Prozess PX auf den bereits vorhandenen Prozess P1 verweisen kann. Außerdem hilft dieser Ansatz bei komplexeren Beispielen mit komplizierteren verzweigungen die Übersichtlichkeit und lesbarkeit der Formel aufrecht zu erhalten.
Bei den Parallelen Gateways müssen wir zunächst einige einschränkungen machen. Einige verzweigungen die bei den Exclusiven Gateways genutzt werden, dürfen bei den Parallelen Gateways nicht vorkommen.

Hier würde ich gerne die Annahme machen, dass bei parallelen Gateways gleich viele Stränge das zusammenfphrende Gateway treffen wie im aufspaltenden erstellt werden. Außerdem soll kein Strang den bereich zwischen den beiden Gateways verlassen. Der Raum ziwschen einen aufspaltenden und zusammenfphrenden Parallelen Gateway soll also wie geschlossen wirken. Kein Strang kann von außen hinein und keiner kann von innen nach außen gelangen. Sie müssen über die Gateways

gehen. Das erleichtert die Programmierung und zum anderen macht es glaube ich auch von der Semantik her mehr Sinn. Hier bräuchte ich mal deine Meinung. An sich ist es nicht komplett undenkbar, dass ein Prozess zum Beispiel, wenn er an einer gewissen stelle angekommen ist zwar immernoch weiter machen soll aber gleichzeitig auch schon wieder von vorne anfängt. Das wäre mit meinen Einschränkungen nicht erlaubt. Ich habe es leider schon so programmiert und es wären halt ein paar verschweendete Stunden wenn wir sagen, dass wir diese Einschränkung nicht machen. Außerdem hab ich keine Ahnung wie man das mit den Parallelen Gateways dann Programmieren soll :) Deine Meinung wäre hier sehr willkommen...

Kapitel 4

Implementierung

Im folgenden Kapitel wollen wir die Implementierung hinter dem Programm erläutern. Dazu soll zunächst die Camunda Model API beschrieben werden und im Anschluss die genaue Implementierung Schritt für Schritt erklärt werden. Die Programmierung erfolge in Java und es wurden einige Bibliotheken verwendet. Das Programm liest eine .BPMN-Datei ein und gibt einen passenden String über die Konsole aus. Die ausgabe enthält die Prozesse und Subprozesse, welche das BPMN Diagramm beschreiben sollen.

4.1 .BPMN-Dateien

Als eingabe wird ein BPMN Diagramm in form einer Datei eingelsen. Diese Dateien können beispielsweise mit hilfe des Camunda Modeleres erstellt werden. Es gibt allerdings auch viele weitere Tools, welche das Erstellen und Bearbeiten von BPM Diagrammen ermöglichen. Die .BPMN-Datei ähnelt eine .XML Datei. Sie ist aufgebaut wie ein..... Vielleicht lieber in Kapitel 2.1???

4.2 Camunda Model API

Der folgende Abschnitt beschriebt zunächst die Camunda Model API. Camunda ist eine Bibliothek welche Funktionen zum Einlesen, Bearbeiten und Erstellen von Prozessen bietet. Für unser Transformationsprogramm müssen keine Prozesse erstellt oder bearbeitet werden, desshalb beschränken wir uns in diesem Abschnitt auf die Funktionen zum Einlesen von Prozessen und zum Extrahieren der in dem Prozess enthaltenen Daten. Ein Prozessdiagramm kann aus einer Datei eingelsen werden. Hierzu wird eine *BPMNModelInstance* erstellt und iniziiert. Das erfolgt durch folgenden Code.
noch einzufügen
Nun besteht die Möglichkeit ein einzelnes Element aus dem Diagramm mittels der ID zu suchen oder eine Collection an Elementen ausgeben zu lassen, welche alle Elemente mit einem bestimmten Typen beinhalten. Hierzu können die Methoden *modelInstance.getModelElementById(ID)* und *modelInstance.getModelElementsByType(Type)* genutzt werden. Die eingehenden und ausgehenden Sequenceflows einzelner Elemente können mit den Mehtoden *getIncomeing* und *getOutgoing* abgerufen werden. Diese Methoden liefern eine Collection von Sequenceflows. Die Collection enthält alle eingehenden bzw. ausgehenden Sequenceflows des jeweiligen Elements.
Die Ziele und Quellen der Sequenceflows werden mit hilfe der Methoden *getTarget* und *getSource* abgefragt. Die Camunda Model API beinhaltet noch einige weitere Funktionen, welche aber für unser Programm nicht notwenig sind.

4.3 Implementierung

Nun wollen wir uns den genauen Inhalt des Programmes und dessen funktionsweise anschauen. Das Mapping erfolgt über die Methode finalMapping. Diese ist dazu da um eine Problematik, welche durch Parallele Gateways aufkommt zu beheben. In der Methode finalMapping wird als erstes die Methode *public Collection <Prozess> mapping (BpmnModelInstance modelInstance, FlowNode Startelement,int count,int currproc)* aufgerufen. Diese wollen wir zuerst erläutern.

4.3.1 Generelles

Wie im Methodenkopf bereits zu erkennen ist, gibt die Methode eine Collection<Prozess> aus. Die beinhaltet eine Menge an Prozessen und Subprozessen in ACP, welche Äquivalent zum BPMN Diagramm ist. Die Klasse Prozess besteht aus den privaten Attributen String Name, String Content und FlowNode First. Diese speichern den Namen und Inhalt des jeweiligen Prozesses und zusätzlich das erste Element, welches in diesem Prozess abgebildet wird. Der erste Prozess speichert also beispielsweise immer das Startevent. Zu jedem Attribut sind getter und setter Methoden gegeben. Als eingabe erhält die Methode zum einen ein BPMN-Diagramm in Form einer modelInstance, das Startevent in Form einer FlowNode und zwei Integer. Count stellt den Prozess mit der Aktuell höchten Nummer dar. Hierbei geht es nicht nur um die Prozesse, welche bereits fertiggestellt worden sind, sondern um den höchsten Subprozess welcher irgendwo in der Menge an Formeln einmal vorgekommen ist. Currproc speichtert hingegen die Nummer des Aktuell zu bearbeitenden Prozesses.
Einige weitere Variabeln werden zusätzlich Global gespeichert. Der genauere Verwendungszweck wird sich erst im laufe des Kapitels zeigen, allerdings wollen wir den Nutzen hier schon einmal kurz beschreiben.

- Collection <Prozess> prozesse = new ArrayList<Prozess>();
  Diese Collection speichert die fertigen Prozesse. Sie beinhaltet das Endergebnis und wird am ende der Methode zurückgegeben
- Collection <FlowNode> elem = new ArrayList <FlowNode>();
  Die Collection speichert alle elemente die zu beginn eines Prozess vorkommen. Sie dient dazu um doppelte nennungen zu vermeiden, wenn zwei Sequenceflows auf die selbe Aktivität zeigen.
- Collection <Prozess> workInProgress = new ArrayList <Prozess>();
  Diese Collection speichert alle Prozesse die bereits initialisiert wurden, allerdings noch nicht fertig gestellt sind. Sie ebenfalls dazu, doppelte Nennungen zu vermeiden.
- Collection <FlowNode> gates = new ArrayList <FlowNode>();
  Diese Collection speichert alle parallelen Gateways, welche bereits abgearbeitet wurden. Sie dient der Umsetzung des Parallelen Operators.
- Stack <Prozess> prozessStack = new Stack<Prozess>();
  Dieser Stack dient der Umsetzung des Paralleln Operators.
- Stack <Prozess> schritt2Prozesse = new Stack<Prozess>();
  Dieser Stack dient der Umsetzung des Paralleln Operators.

- Stack <Integer> schritt2Zähler = new Stack <Integer>();
  Dieser Stack dient der Umsetzung des Paralleln Operators.

Die Methode Main(String[] args) läd ein BPMN Diagramm aus einer Datei, findet das Startevent und ruft die Methode finalMapping auf und übergibt dabei das Diagramm mit Datentyp modelInstance und das Startevent mit Datentyp FlowNode. Diese Methode ruft dann im ersten Schritt die Methode mapping auf. Hier gibt sie das Diagramm und das Startevent weiter. Zusätzlich werden in der Methode die Variabeln count und currproc als 1 und 0 gesetzt.

4.3.2 Erste Schritte

Im folgenden Abschnitt befassen wir uns mit dem Teil der Methode, welcher sich noch nicht mit Gateways auseinander setzt.
Das Programm erstellt eine zu dem Diagramm Äquivalente ACP-Formel, indem es das Diagramm mithilfe der getOutgoing Methode der FlowNodes und der getTarget Methode der Sequenceflows einmal abläuft.
Im ersten Schritt wird eine Variable FLowNode *curr* erstellt und mit dem wert des Startelements initialisiert. *curr* speichtert immer das aktuell zu bearbeitende Element. Zusätzlich wird das übergebene Startelement in die Menge elem eingeführt.
Nun wird ein neuer Prozess *p* erstellt, dessen Name aus einem P und der Variable currproc besteht. Beim ersten Durchlauf der Methode, würde also der erste Prozess erstellt werden und den Namen P0 tragen. Zudem wird der Wert von *first* des Prozesses auf den Wert des Startelements gesetzt. Im Anschluss wird p der Collection workInProgress hinzugefügt.
Nun geht es darum den Inhalt von *p* zu setzen. Dieser soll alle im Diagramm vorkommenden Aktivitäten bis zu einem Gateway in der richtigen Reihenfolge enthalten. Zuerst soll das Startelement dem Inhalt hinzugefügt werden. Wenn es sich dabei um ein Endevent handelnt, so wird p der Collection Prozesse hinzugefüht und sie wird im return statement zurückgegeben.

4.3.3 Bis zum nächsten Gateway

Falls nicht, wird nun in einer while-Schleife überprüft, ob es sich bei dem nächsten Element um ein Gateway handelt. Solange das nicht der Fall ist, wird die Schleife betreten.Hierbei und in allen weiteren Fällen ist mit dem nächsten Element jenes gemeint, welches durch curr.getOutgoing().getTarget() erhalten wird. Es ist also das Element welches über einen Sequenceflow mit dem aktuellen Element verbunden ist. Die Methode getOutgoing() liefert eine Collection an Sequenceflows, da es möglich ist, dass mehrere Elemente mit beispielsweise dem selben Gateway verbunden sind. Die Methode wird aber nur an Stellen aufgerufen, an denen nur ein Element auf das aktuelle in *curr* gespeicherte Element folgen kann. Daher können wir mit durch die Methoden iterator().next() auf das jeweilige nächste Element zugreifen.
Innerhalb der Schleife wird nun geprüft, ob es sich bei dem nächsten Element um ein Endevent handelt. Ist das der Fall, so wird der Name des Events zusammen mit dem *-Operator dem Inhalt von p hinzugefügt. Dann wird p der Ergebnissmenge Prozesse hinzugefügt und die Collection wird zurückgegeben. Damit Terminiert die Methode
Andernfalls, wird die Variable curr mit dem Wert der nächsten Flownode geladen. Der Name dieses Elements wird dem Inhalt von p hinzugefügt. Auch hier wird der

*-Operator vor den Namen geschrieben. Diese while-Schleife wird ausgeführt, bis sie entweder durch ein Endevent Terminiert oder das nächste vorkommende Element ein beliebiges Gateway ist.

4.3.4 Exclusive Gateways

Da wir zwischen Exclusiven und Parallelen Gateways unterscheiden, müssen wir auch in der Implementierung des Mappings beide fällte getrennt abarbeiten. Als erstes wird also der Typ des Gateways abgefragt. In diesem Abschnitt erläutern wir den Teil des Programmes, welches sich mit der Bearbeitung des Exclusiven Gateways beschäftigt.
Da in der vorherigen Schleife nur das nächste Element überprüft wurde, wird zuerst *curr* mit dem Wert des nächsten Elementes geladen. Hierbei handelt es sich also um das zu bearbeitende Gateway.
Nun müssen wir zwischen aufspaltenden und zusammenführenden Gateways unterscheiden. Dazu wird die größe der Collection abgefragt, welche durch curr.getOutgoing() gegeben wird.

Aufspaltende Exclusive Gateways

Befinden wir uns an einem Anfspaltenden Gateway, so muss curr.getOutgoing().size() einen Wert, welcher größer als 1 ist zurück geben.
Ist das der Fall, so können wir dem Inhalt von p auf jeden Fall die Zeichenkette *(hinzufügen. Innerhalb dieser Klammer, soll nun der jeweils passende Subprozess für jeden ausgehenden Sequenceflow stehen. Diese Prozesse werden mit dem +-Operator von einander getrennt, da es sich um eine XOR aufspalung handelt.
Dies wird durch einen For-Loop realisiert welche über jeden ausgehenden Sequenceflow des Gateways Iteriert. Die Sequenceflows sind durch den for-Loop mit *proz* benannt.
Nun wird das jeweilige Ziel des Sequenceflows betrachtet.
Zuerst wird der Fall betrachtet, dass mehrere Gateways aufeinander folgen. Folgt ein zusammenführendes Gateway auf das zu behandelnde Gateway, so wird es einfach ignoriert. Wir ändern proz also und laden es mit dem Wert des ausgehenden Sequenceflows des nächsten Gateways. Das wird so lange gemacht, bis das nächste Element kein gateway mehr ist.
Nun wird das nächste Element betrachtet. Für den Fall, dass es sich um ein Endevent handelt, so wird der Name der Aktivität dem Inhalt von *p* hinzugefügt. Zusätzlich wird ein + hinzugefügt.
Andernfalls wird geprüft, ob das Element bereits in einem anderem Prozess vorkommt. Da wir Subprozesse immer ausschließlich an Gateways beginnen, muss es also, um bereits abgearbeitet worden zu sein, als Startelement eines anderen Prozesses vorkommen. Zu beginn des Programmes wurde das Startelement der Collection *elem* hinzugefügt. Dies können wir uns nun zu nutze machen um zu prüfen, ob es einen Prozess gibt, welcher mit dem aktuell zu bearbeitenden Element beginnt. Wir prüfen also ob das Ziel des jeweiligen Sequenceflows bereits in der Menge *elem* vorhanden ist.
Ist das der Fall, so wird können wir den Prozess suchen, welcher das passende Element als *first* gespeichert hat. Der Name dieses Prozesses kann dann einfach dem Inhalt von *p* angehangen werden.
Etwas komplizierter wird es, wenn das Ziel des Sequenceflows noch nicht in *elem* zu finden ist. In diesem Fall muss ein neuer Prozess erstellt werden. Hierzu müssen wir

den Prozess finden, dessen Namen die größte nummer beinhaltet. Dazu wird eine Kopie der Variable *count* erstellt. Die *countcopy* wird dann so lange erhöht, bis kein Prozess mehr existiert, der diese Zahl im Namen trägt.
Nun wird der Inhalt von *p* um den String *"P"+countcopy+"+"* erweitert.
Nun folgt ein rekursiver Aufruf der Methode mapping. Als Parameter wird wieder die selbe modelInstance übergeben. Das Startelement ist jetzt das Ziel des jeweiligen Seqenceflows und der Wert des *currproc* muss den der countcopy annehmen. Das Programm definiert nun also den Subprozess für den jeweiligen Ast des Diagrammes.
Im letzten Schritt müssen noch 2 kleine Änderungen an dem Inhalt des Prozesses *t* gemacht werden. Zum einen wird das letzte Zeichen gelöscht, da aktuell ein + zu viel an Ende des Strings hängt. Als letztes wird dann die Klammer geschlossen.
Damit ist ein Prozess fertig defniert, für den Fall, dass er auf auf ein aufspaltendes exclusives Gateway trifft.

Zusammenführende Exclusive Gateways

Falls das Programm auf ein zusammenführendes Gateway trifft, verhält es sich ähnlich zu dem aufspaltenden. Zunächst werden wieder alle folgenden zusammenf+hrenden Gateways ignoriert. Im Anschluss wird wie zuvor geprüft, ob das folgende Element bereits bearbeitet wurde.
Auch hier werden beide Fälle wie im Abschnitt zuvor behandelt.

4.3.5 Parallele Gateways

Für parallele Gateways können einige techniken ähnlich wie im Abschnitt 4.3.4 genutzt werden. Es gibt allerdings einige weitere Merkmale die beachtet werden müssen. Dieser Abschnitt soll die verwendeten Techniken genauer erklären.

Generelle Idee

Wie in Kapitel 3 erläutert muss bei dem Parallelen Operator im ACP muss darauf geachtet werden, dass nicht mehrere der erstellten Subprozesse die folgenden Aktivitäten ausführen. Zusätzlich muss der bereich zwischen einem zusammenführenden und dem dazugehörigen aufspaltenden Gateway immer geschlossen sein. Das erlaubt uns den Subprozess, welcher nach dem zufammenführenden Gateway beginnt, bereits im aufspaltenden Prozess aufzurufen.
Um das umzusetzen wird die Methode *finalMapping* verwendet, welche in einem zweiten Schritt aufgerufen wird. Sie arbeitet mit 3 Stacks, welche in der Methode *mapping* gefüllt werden sollen.

Aufspaltende Parallele Gateways

Die aufspaltenden parallelen Gateways verhalten sich sehr ähnlich zu den exclusiven Gateways. Zu beginn wird der Prozess *p* auf den Stack *prozessStack* gelegt. Dadurch wird der Prozess gespeichert, welcher später um den, auf dem zusammenführenden Gateway folgenden Prozess ergänzt werden soll.
Im Anschluss verhält sich das Programm genau so wie auch für die aufspaltenden exclusiven Gateways. Allerdings werden die einzelnen Prozesse oder Aktivitäten nicht durch +, sondern durch || getrennt.

Zusammenführende Parallele Gateways

Trifft das Programm nun auf ein zusammenführendes paralleles Gateway, so soll der Inhalt des aktuellen Prozesses *p* nicht mehr verändert werden. Hier muss jetzt festgestellt werden, welcher Prozess das aufspaltende Gateway enthält und welcher Prozess auf das zusammenführende Gateway folgt. Dazu wird der Wert der *countcopy*, welcher genauso wir im zuvorigen Abschnitt bestimmt wird, auf einen Stack *schritt2Zähler* gelegt. Zusätzlich wird der oberste Prozess aus dem *prozesseStack* auf einen weiteren Stack *schritt2Prozesse* gelegt. Damit ist sichergestellt, dass in beiden Stack, welche mit *schritt2* gekennzeichnet sind, stehts auf der selben höhe die zusammenhängenden Daten gespeichert werden. Das bedeutet einer Ebene der Stacks liegen immer in einem Stack der Prozess, welcher erweitert werden muss und im anderen Stack die Nummer der Prozesses mit dem er erweitert werden soll.
Im Anschluss wird nun wieder die Methode mapping aufgerufen. Auch hier wird die countcopy wieder als currproc übergeben, damit der rekursive aufruf die bennenung bei der richtigen Zahl anfängt. Außerdem wird das nächste Element was auf das Gateway folgt als Startelement übergeben.
In zusammenführende Gateways treffen immer mehrere Sequenceflows ein. Das bedeutet, dass auch die obige Aktion mehrfach ausgeführt wird. Um das zu verhindern, wird vorher geprüft, ob das zusammenführende Gateway in der Collection *gates* enthalten ist. Nur wenn das nicht der Fall ist, wird die Aktion ausgeführt. Zudem muss das jeweilige Gateway in *gates* gesspeichert werden, um es als abgearbeitet zu markieren.

4.3.6 Ausgabe und Beobachtung

Nun ist der Prozess*p* korrekt abgearbeitet und ausgeschrieben. Er wird der Collection *prozesse* hinzugefügt und diese Collection wird zurückgegeben. Auf Grund der Rekursion ist zu sehen, dass die Nummerierung der Prozesse teilweise willkürlich erscheint. Das Programm startet die Bearbeitung bei P0, allerdings ist der erste fertiggestellte Prozess, welcher der Ergebnissmenge hinzugefügt wird, derjenige, welcher das Endevent beinhaltet.