FH JOANNEUM - University of Applied Sciences

BPMN zu PASS

Übersetzung von Geschäftsprozessmodellen

Masterarbeit

zur Erlangung des akademischen Grades eines

„Diplomingenieurs für technisch-wissenschaftliche Berufe“

eingereicht am Master-Studiengang IT Architecture

Verfasser:

Tim Tobias Braunauer

Betreuer:

Robert Singer

Graz, 2022

FH | JOANNEUM
University of Applied Sciences

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende Masterarbeit selbstständig angefertigt und die mit ihr verbundenen Tätigkeiten selbst erbracht habe. Ich erkläre weiters, dass ich keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle aus gedruckten, ungedruckten oder dem Internet im Wortlaut oder im wesentlichen Inhalt übernommenen Formulierungen und Konzepte sind gemäß den Regeln für gutes wissenschaftliches Arbeiten zitiert und durch Fußnoten bzw. durch andere genaue Quellenangaben gekennzeichnet.

Die vorliegende Originalarbeit ist in dieser Form zur Erreichung eines akademischen Grades noch keiner anderen Hochschule vorgelegt worden. Diese Arbeit wurde in gedruckter und elektronischer Form abgegeben. Ich bestätige, dass der Inhalt der digitalen Version vollständig mit dem der gedruckten Version übereinstimmt.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

SBD	Subject Behavior Diagram
SID	Subject Interaction Diagram
DDD	Data Definition Diagram
BPMN	Business Process Model and Notation
S-BPM	Subject-Oriented Business Process Management
OWL	Web Ontology Language
WfMS	Workflow Management System
XML	Extensible Markup Language
URL	Uniform Resource Locator
PASS	Parallel Activity Specification Schema
BPM	Business Process Management

Code-Snippet-Verzeichnis

Kurzfassung

Die Business Process Model and Notation (BPMN) ist der am weitesten verbreitete Standard für die Modellierung von Geschäftsprozessen. Es gibt jedoch auch andere Standards, wie das subjektorientierte Parallel Activity Specification Schema (PASS). Aufgrund der wenigen Werkzeuge, die für den PASS Standard zur Verfügung stehen, ist dieser jedoch nicht weit verbreitet, obwohl er aufgrund seiner Maschinenlesbarkeit und der Möglichkeit zur Umwandlung in einen ausführbaren Code einige Vorteile hat. Das Ziel dieser Arbeit ist es, eine Lösung zur Modellierung von PASS Modellen mit BPMN Tools zu erarbeiten. Zum einen wird ein Modellierungsleitfaden entwickelt, der angibt, wie mit BPMN Tools subjektorientiert modelliert werden kann. Zum anderen wird eine Software erstellt und beschrieben, welche BPMN Modelle in PASS Modelle übersetzen kann.

Der erste Teil dieser Arbeit beschäftige sich mit der Analyse des PASS Standards. Hier wird auch auf den Aufbau des Standards als Ontologie eingegangen. Im nächsten Schritt wird eine Elementeinschränkung für den BPMN Standard erstellt, die nur dessen übersetzbare Elemente enthält. Auch diese Elemente werden näher analysiert. Danach wird der Modellierungsleitfaden beschrieben.

Im nächsten Teil der Arbeit wird erklärt, wie die Übersetzung von BPMN nach PASS durchgeführt wird. Dabei wird der Übersetzungsprozess für jedes Element beschrieben. Danach wird erklärt, wie die übersetzten Modelle auf Vollständigkeit und Richtigkeit überprüft werden. Außerdem wird mit Hilfe der Service Interaction Patterns beschrieben, welche BPMN Modelle übersetzt werden können und welche nicht.

Dabei wurde gezeigt, dass alle Service Interaction Pattern, sofern diese mit dem ausgearbeiteten Modellierungsleitfaden erstellt wurden, übersetzbar sind. Sieben der neun übersetzten Modelle sind auch ausführbar. In der Arbeit wird auch genau beschrieben, warum die restlichen zwei Modelle nicht ohne Probleme ausführbar sind.

Am Ende der Masterarbeit wird ein Ausblick gegeben, wie das Projekt weitergeführt werden kann und wie weitere BPMN-Elemente unterstützt und übersetzt werden können.

Abstract

The Business Process Model and Notation (BPMN) is the most widely used standard for modeling business processes. However, there are also other standards such as the subject-oriented Parallel Activity Specification Schema (PASS). However, due to the few tools available for the PASS standard, it is not widely used, although it has some advantages due to its machine readability and ability to be converted into executable code. The goal of this work is to work out a solution to model PASS models with BPMN tools. On the one hand a modeling guideline is developed, which specifies how to model subject-oriented with BPMN tools. On the other hand a software is created and described, which can translate BPMN models into PASS models.

The first part of this thesis deals with the analysis of the PASS standard. Here the structure of the standard as an ontology is discussed. In the next step an element restriction for the BPMN standard is created, which contains only its translatable elements. These elements are also analyzed in more detail. After that, the modeling guideline is described.

The next part of the paper explains how the translation from BPMN to PASS is performed. In this process, the translation process for each element is described. After that, it is explained how the translated models are checked for completeness and correctness. Furthermore, with the support of the Service Interaction Patterns, it is described which BPMN models can be translated and which cannot.

It was shown that all service interaction patterns, as long as they were created with the elaborated modeling guide, are translatable. Seven of the nine translated models are also executable. The paper also describes in detail why the remaining two models are not executable without problems.

At the end of the master thesis, an outlook is given on how the project can be continued and how more BPMN elements could be supported and translated.

1 Einleitung

1.1 Problem und Motivation

In einem Unternehmen gibt es viele verschiedene Geschäftsprozesse. Einige sind Kernprozesse. Das sind die Prozesse, die für die Wertschöpfung des Unternehmens zuständig sind. Zusätzlich dazu gibt es noch unterstützende Prozesse. Damit ein Unternehmen erfolgreich und effizient ist, ist es unter anderem wichtig, dass die Geschäftsprozesse gut funktionieren. Hierfür werden diese oft als Modell dargestellt, um sie zu analysieren und zu optimieren und um sie gegebenenfalls an neue Anforderungen anpassen zu können.

Seit vielen Jahren ist der Business Process Model and Notation (BPMN) Standard etabliert, um Geschäftsprozesse zu modellieren. Es gibt unzählige Editoren mit denen BPMN Modelle erstellt und verändert werden können. Bei diesem Standard wird der Prozess als Ganzes modelliert. Das Modell ähnelt einem Flussdiagramm, bei dem genau definiert wird, welche Prozessschritte in welcher Reihenfolge ausgeführt werden. Dabei muss aber nicht zwingend festgelegt werden, von welchen Personen die Aufgaben erledigt werden. Die Aufgabenzuteilung wird dadurch erst bei der Prozessumsetzung relevant.

In den letzten Jahrzehnten gab es Bestrebungen sich von diesem Prozessmodellierungsparadigma zu distanzieren und sich mehr auf die einzelnen Subjekte und deren Aufgaben beim Modellieren zu fokussieren. Dadurch entstand das Konzept des Subject-Oriented Business Process Management (S-BPM). Mit BPMN kann unter anderem auch subjektorientiert modelliert werden, wenn gewisse Vorgaben eingehalten werden. Es gibt jedoch auch bereits einen Modellierungsstandard, der den S-BPM Ansatz als zentrales Konzept in den Mittelpunkt stellt. Dieser Standard nennt sich Parallel Activity Specification Schema (PASS).

Der PASS Standard fokussiert sich auf die Aufgaben der einzelnen Subjekte und besonders auf den Nachrichtenaustausch zwischen diesen. Dabei gibt es in einem PASS Model einen Nachrichtenpool, der dafür sorgt, dass Nachrichten asynchron verschickt werden können. Der PASS Standard ist als Ontologie definiert und auch das fertige Modell wird als Ontologie in einem Web Ontology Language (OWL) File gespeichert. Außerdem ist der Standard so aufgebaut, dass die Modelle anschließend von entsprechender Software gelesen, interpretiert und ausgeführt werden können. Damit ist der PASS Standard vor allem im Bereich der automatisierten Ausführung von Prozessen relevant.

Der große Nachteil von PASS ist allerdings, dass es bisher noch wenige Editoren gibt, mit denen PASS Modelle generiert und verändert werden können (Stand September 2022). Im Gegensatz dazu ist der BPMN Standard, mit dem auch subjektorientiert modelliert werden kann, in der Industrie weit verbreitet und hat viele verschiedene Editoren. Daher stellt sich die Frage, ob Prozesse subjektorientiert mit einem BPMN Editor er-

stellt werden können und diese dann umgewandelt und als PASS Modell ausgeführt werden können. Dadurch müssten Prozessmanager keinen neuen Modellierungsstandard lernen und können mit den bekannten Tools vorgehen. Außerdem wäre es eine große Hilfestellung, um das Konzept S-BPM in der Industrie zu verbreiten und alle Vorteile dieses Konzepts zu nutzen. Dabei stellt sich die Frage, wie die beiden Standards aufgebaut sind. Welchen Syntax besitzen sie und welche Konzepte sind innerhalb der beiden Standards umgesetzt.

Einige dieser Fragen waren bereits Inhalt der Arbeit von Frau Reiter [2]. Dadurch wurde bewiesen, dass eine Umwandlung von BPMN in PASS möglich ist. Gezeigt wurde dies anhand eines Proof Of Concept mit ein paar Beispielprozessen. In der Arbeit wurde jedoch keine systematische Umwandlung beschrieben und auch die Übersetzungssoftware funktioniert nur für die zuvor definierten Prozesse. Außerdem wurde seitdem der PASS Standard aktualisiert.

Diese Arbeit fokussiert sich daher darauf, die beiden Standards BPMN und PASS zu analysieren. Aufgrund dieser Analyse wird ein Leitfaden entstehen, der vorgibt wie PASS Prozesse mit einem BPMN Editor erstellt werden können. Danach wird eine systematische Umwandlung der Prozesse beschrieben und schlussendlich auch umgesetzt werden.

1.2 Stand der Literatur

In [3] wird das S-BPM Konzept vorgestellt. Dabei wird auf dessen Vorteile eingegangen und wie das Konzept entstanden und aufgebaut ist.

Die Arbeit [4] ist ein umfassendes Werk, in dem die S-BPM Herangehensweise für Business Process Management (BPM) beschrieben wird. Es wird beschrieben welche Konzepte dahinter stecken, welche Vorteile sich daraus ergeben und wie die Herangehensweise entstanden ist. Abgeleitet davon wird der PASS Standard vorgestellt. Es wird gezeigt woraus dieser besteht und wie mit ihm Prozesse modelliert werden können. Außerdem wird beschrieben, wie Prozesse validiert, optimiert und implementiert werden können.

[5] zeigt, wie der subjektorientierte PASS Standard und dessen Tools eingesetzt werden können, um Kommunikations- und Synchronisationsprobleme in Unternehmen zu lösen. Im Buch wird mit der Modellierung eines einfachen Prozesses gestartet und dieser wird immer wieder angepasst und erweitert. Damit wird ein reales Szenario simuliert und dabei werden die Vorteile der Modellierung mit dem PASS Standard praxisnah verdeutlicht.

[6] ist die praktische Umsetzung von [2], welche als Vorgängerprojekt dieser Arbeit dient. Mit diesem Projekt war es möglich, ein paar vorgegebene BPMN Prozesse in PASS Prozesse umzuwandeln.

Ein weiteres Projekt, das das Ziel hatte BPMN Prozesse in Ontologien umzuwandeln und damit auf Richtigkeit und Vollständigkeit zu überprüfen, wurde bisher erfolgreich abgeschlossen. Es kann unter [7] gefunden werden. Dabei wurde der Prozess aber in eine eigens dafür entworfene BPMN Ontologie umgewandelt und nicht wie in dieser Arbeit in eine PASS Ontologie.

[8] beschreibt, was der BPMN Standard ist und wie er grundsätzlich aufgebaut ist. Dabei wird gezeigt aus welchen Elementen er besteht und wie damit einfache Geschäftsprozesse modelliert werden können.

In der Arbeit [9] wird gezeigt, dass jedes BPMN-Modell als OWL Ontologie serialisiert werden kann. In der Arbeit wird dies durch eine Akteur- oder Subjekt-basierte Sicht auf Geschäftsprozesse gezeigt. Unter anderem wird auch eine Referenzarchitektur eines Workflow Management System (WfMS), das aus Microservices besteht, diskutiert, mit dem akteurbasierte Geschäftsprozesse ausgeführt werden können.

Bruce Silver erklärt in seinem Buch [10] wie korrekte, vollständige und klare BPMN-Modelle erstellt werden können. Außerdem wird gezeigt welche BPMN-Elemente ein Prozessmodellierer verstehen muss. Diese werden sowohl in der Descriptive modeling Subclass als auch in der Analytic Subclass erklärt. Auch auf die Extensible Markup Language (XML) Serialiserung wird im Buch genauer eingegangen, weshalb es eine gute Basis für diese Arbeit darstellt.

In der Arbeit [11] wird darauf eingegangen, wie die Konzepte von BPMN und PASS gemappt werden können um die Elemente miteinander zu vergleichen. Damit dient sie als theoretische Basis dieser Arbeit.

[12] ist das aktuelle BPMN Standard Dokument.

[13] ist das aktuelle PASS Standard Dokument.

1.3 Ziele und Fragestellungen

Das Ziel dieser Arbeit ist es die beiden Modellierungsstandards BPMN und PASS miteinander zu vergleichen und in weiterer Folge eine Übersetzungssoftware zu schreiben, die BPMN Modelle systematisch in PASS Elemente umwandeln kann. Dieses Ziel kann in mehrere Teilschritte und Teilziele heruntergebrochen werden.

Zuallererst wird darauf eingegangen, wie der PASS Standard aufgebaut ist. Dabei wird unter anderem beschrieben, aus welchen Teilen ein PASS Prozess besteht und wie eine fertige OWL Datei aufgebaut ist.

Resultierend darauf wird als nächstes der BPMN Standard analysiert. Dabei geht es vor allem darum jene Elemente ausfindig zu machen, die benötigt werden, um die Konzepte des PASS Standards nachzubilden. Hierbei gibt es zwei große Ziele die erreicht werden. Zum einen wird eine Elementeinschränkung definiert, die vorgibt welche BPMN Elemente in PASS Elemente übersetzt werden können. Zum anderen wird ein Modellierungsleitfaden entstehen, der beschreibt, wie mit den eingeschränkten Elementen modelliert werden muss, damit das fertige Modell dann in ein PASS Modell umgewandelt werden kann.

Im nächsten Schritt geht es darum zu beschreiben, wie die Umwandlung von BPMN Modelle in PASS Modelle genau funktioniert. Diese Analyse dient als Grundlage der ersten praktischen Umsetzung dieser Arbeit.

Anhand der bis dahin erreichten Ziele kann dann der erste praktische Teil dieser Arbeit umgesetzt werden. Das Ziel hierbei ist die Programmierung einer Software, die BPMN Modelle einliest, versteht, in PASS Modelle umwandelt und anschließend wieder exportiert.

Das nächste Ziel ist, zum einen zu zeigen, dass die Umwandlung erfolgreich war und zum anderen zu zeigen, dass die Umwandlung mit jedem Prozess, der nach dem zuvor erstellten Leitfaden modelliert wurde, funktioniert.

Nachdem gezeigt wurde, dass die Umwandlung funktioniert, muss noch bewiesen werden, dass diese auch semantisch richtig ist. Hierfür wird ein weiterer Übersetzer programmiert, der die Prozesse wieder in BPMN Prozesse zurückkonvertiert. Dies stellt das zweite praktische Ziel dieser Arbeit dar.

Anschließend müssen das ursprüngliche Modell und das zurückübersetzte Modell miteinander verglichen werden. Hierfür wird ebenfalls eine Software erstellt, die zeigt, ob die beiden Modelle inhaltlich identisch sind. Dies stellt das letzte praktische Ziel dieser Arbeit dar.

Die Fragestellung mit der sich diese Arbeit beschäftigt ist, ob die gerade beschriebene Übersetzung systematisch möglich ist. Außerdem wird bewiesen, dass keine wichtigen Prozessinformationen beim Übersetzungsvorgang verloren gehen. Um dies zu beweisen dient die Rückübersetzung und Validierung des Prozesses.

Das Ziel dieser Arbeit ist es nicht eine Software zu schreiben, die alle möglichen BPMN Elemente übersetzen kann, sondern einen funktionierenden Übersetzungsprototypen zu entwickeln, mit dem man viele Prozesse umwandeln kann.

Zum Abschluss der Arbeit wird noch ein kurzer Ausblick gegeben, der zeigt, wie in Zukunft in diesem Bereich weitergearbeitet werden und wie und ob die Software weiter verbessert werden kann. Außerdem wird gezeigt inwiefern das Ergebnis in der Praxis weiterverwendet werden kann.

1.4 Methodik

Der erste große Schritt dieser Arbeit besteht darin, möglichst viel Wissen im Bereich BPM und vor allem S-BPM anzusammeln. Hierfür werden zuallererst Standarddokumente zu PASS und BPMN studiert und in weiterer Folge vor allem Arbeiten herangezogen, die sich bisher mit dem Vergleich der beiden Standards und Konzepte beschäftigt haben.

Nachdem man sich mit den Grundlagen vertraut gemacht hat, wird im nächsten Schritt damit begonnen, dass Prozesse in beiden Standards abgebildet werden. Dabei wird zweimal der selbe Prozess in beiden Standards abgebildet. Dadurch kann in diesem Schritt erkannt werden, welche Konzepte sich decken und bei welchen Elementen Workarounds eingebaut werden müssen, oder welche Konzepte sogar nur in einer der beiden Technologien funktionieren oder vorhanden sind.

Nachdem einige Prozesse entworfen wurden, werden die so entstandenen Dateien analysiert und miteinander verglichen. Dadurch kann erkannt werden, wie ähnlich oder unterschiedlich die einzelnen Elemente definiert sind und es dient als Basis für die spätere praktische Umsetzung.

Die so erhaltenen Erkenntnisse können im nächsten Schritt niedergeschrieben werden und es kann damit begonnen werden, die Elementeinschränkung zu definieren. Nachdem die Elementeinschränkung definiert wurde, kann im nächsten Schritt der Modellierungsleitfaden geschrieben werden. Auch die Vorgehensweise beim Erstellen der Übersetzungssoftware kann definiert und niedergeschrieben werden.

Im nächsten Schritt geht es darum, die Übersetzungssoftware umzusetzen. Während dieser Arbeit können die bisherigen Kapitel nochmal geringfügig überarbeitet werden, falls erkannt wird, dass die Übersetzung nicht so möglich ist wie sie davor definiert wurde.

Nachdem die Übersetzung funktioniert, kann im nächsten Schritt die Rückübersetzungs- und Validierungssoftware erstellt werden. Mit Hilfe dieser können viele unterschiedliche Prozesse getestet und Fehler können damit identifiziert und ausgebessert werden.

Im letzten Schritt geht es darum, alle gesammelten Erkenntnisse bei der Umsetzung niederzuschreiben und die Arbeit fertigzustellen. Dabei werden zum einen die Ergebnisse der Arbeit evaluiert und bewertet. Zum anderen werden mögliche Verbesserungen genannt und Möglichkeiten gezeigt, wie in diesem Gebiet weitergearbeitet werden könnte.

2 Aufbau von PASS

In diesem Kapitel wird ein Überblick über den PASS Standard gegeben, welcher nach dem S-BPM Konzept aufgebaut ist. Zuerst wird darauf eingegangen, wie er aufgebaut ist und aus welchen Elementen der Standard besteht. Danach wird gezeigt wie die OWL-Datei eines PASS Prozesses aufgebaut ist. Schlussendlich werden alle zuvor aufgezählten Elemente genau beschrieben. Dabei wird darauf eingegangen, wofür jedes Element benötigt wird und wie der OWL-Aufbau bei jedem Element aussieht.

2.1 Übersicht über den PASS Aufbau

Grundsätzlich besteht der PASS Standard aus einem Subject Interaction Diagram (SID), bei dem mehrere Subjekte dargestellt werden und je einem Subject Behavior Diagram (SBD) pro Subjekt.

Im SID werden alle Subjekte dargestellt, die eine Rolle im Prozess spielen. Eines oder mehrere der Subjekte müssen als Start-Subjekt definiert sein, welche den Prozess initialisieren und auch dementsprechend gekennzeichnet sind. Neben den Subjekten wird der Nachrichtenaustausch dargestellt, der zwischen den Subjekten stattfindet. Es wird jeder mögliche Nachrichtenaustausch mit allen möglichen Nachrichten dargestellt.

Jedes Subjekt, das im SID dargestellt wird hat ein eigenes Diagramm, in dem das Verhalten des Subjekts innerhalb des Prozesses dargestellt wird. Dieses Diagramm nennt man SBD. Grundsätzlich besteht ein SBD aus States und Transitions. Die States sind die Zustände in denen sich ein Subjekt befinden kann und die Transitions, die Zustandsübergänge. Es gibt immer genau einen Start-State und ein oder mehrere End-States. Außerdem gibt es Do-States, in denen das Subjekt eine Aufgabe erfüllt, Send-States in denen das Subjekt eine Nachricht verschickt und Receive-States, in denen das Subjekt auf das Empfangen einer Nachricht wartet. Genau wie bei den States gibt es auch Do-Transitions, Send-Transitions und Receive-Transitions, bei denen das Subjekt in einen anderen State wechselt und je nach Transition Nachrichten empfangen oder versendet werden. Außerdem gibt es noch die DayTimeTimer-Transitions, bei denen das Subjekt eine bestimmte Zeit wartet, bevor der Zustand gewechselt wird. Diese Transitions können auch dafür verwendet werden, um einen alternativen Weg einzuschlagen, falls eine Nachricht zu viel Zeit in Anspruch nimmt oder gar nicht ankommt.

Neben dem SID und den SBD gibt es noch jeweils ein Data Definition Diagram (DDD) pro Nachrichtenaustausch. In diesem ist definiert, wie die Daten aufgebaut sind, welche beim Datenaustausch verschickt werden. Diese verschickten Daten werden auch Business Objects genannt. Es wird definiert welche Felder verschickt werden und welchen Datentyp jedes Feld hat. Da es solch eine Definition im BPMN Standard nicht gibt und es somit auch nicht übersetzt werden kann, wird das DDD in dieser Arbeit nicht näher beschrieben.

Mit diesen Elementen kann man beinahe alle Prozesse modellieren. Auch in der Arbeit von Herrn Fleischmann [3] werden alle diese Elemente genannt. Daneben gibt es noch die Möglichkeit einzelne Prozesse miteinander zu verknüpfen oder Multi-Prozesse zu modellieren. Da dies den Rahmen dieser Arbeit sprengen würde und viele Modelle auch als einzelner Prozess abgebildet werden können, wird darauf aber nicht weiter eingegangen. Es gibt noch weitere PASS Elemente, jedoch werden diese in der Arbeit von Herrn Fleischmann nicht erwähnt und daher werden sie auch in dieser Arbeit nicht behandelt. Außerdem würden sie zum einen den Rahmen der Arbeit sprengen und zum anderen werden sie nur selten, zum Modellieren von bestimmten Prozesskonstellationen, gebraucht.

In Abbildung 2.1 sieht man einen PASS Prozess mit allen Elementen, die in den vorherigen Absätzen vorgestellt wurden.

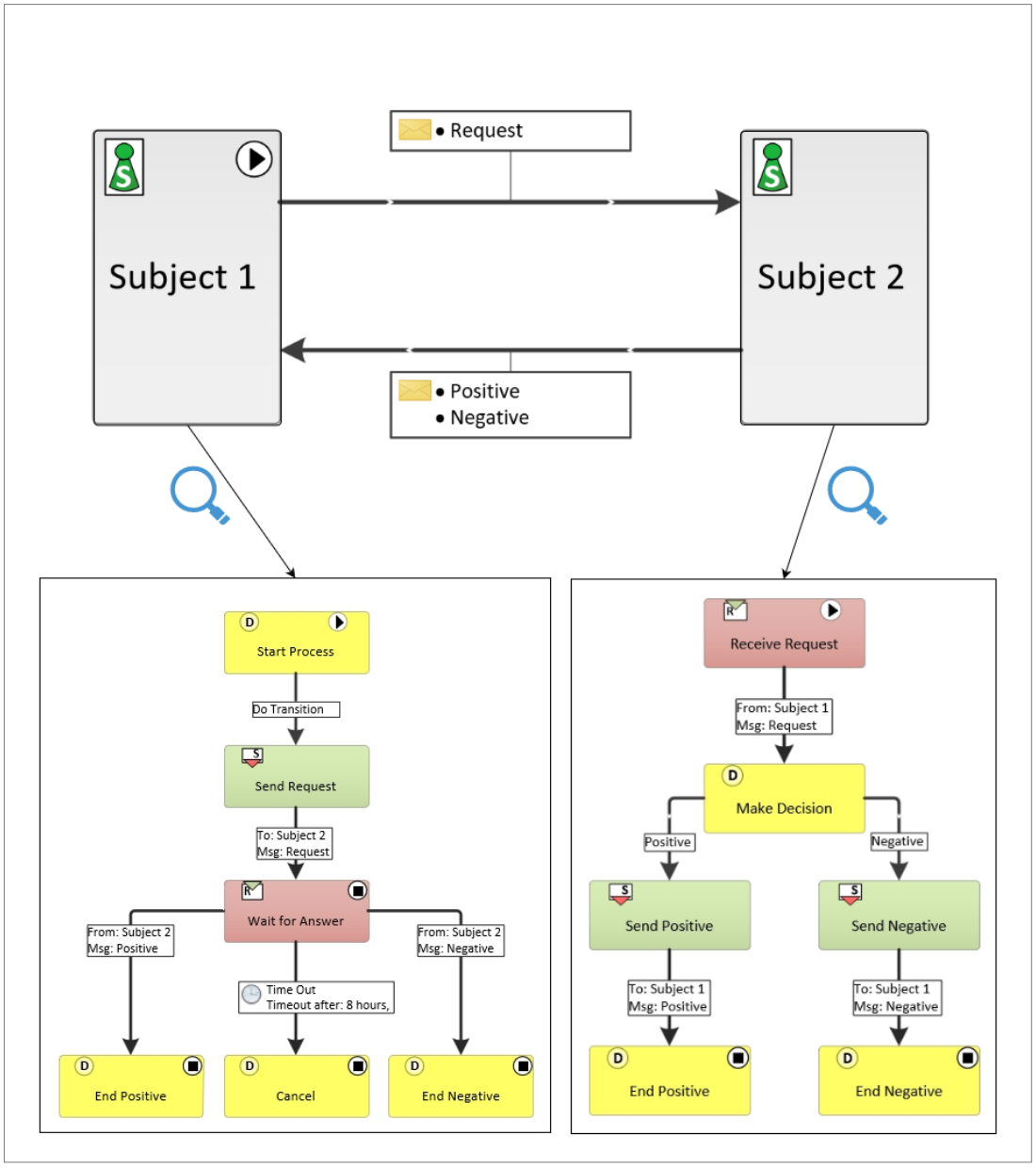

Abbildung 2.1: PASS Beispielprozess mit allen Kernelementen

2.2 Ontologie als Basis für PASS

Auch wenn das subjektorientierte Konzept um Prozesse zu modellieren bereits im Jahr 2017 relativ weit ausgereift war, so hatte man sich noch nicht darauf geeinigt, wie die Modelle und deren Struktur gespeichert werden sollen. Es gab bereits mehrere Tools zum Erstellen und Bearbeiten von PASS Modellen, aber nicht alle hatten den selben Aufbau. Herr Elstermann hat aus diesem Grund eine Arbeit verfasst, bei der er vorschlägt, OWL als Standardsprache für PASS Modelle zu verwenden. Somit sollte der Modellierungsstandard als Ontologie gespeichert werden. Außerdem hat er vorgeschlagen, dass es eine `standard-pass-ont` Datei gibt, die den Standard definiert und mehrere Ontologie Dateien, die diesen Standard erweitern und somit als Basis für unterschiedliche Tools verwendet werden können [14].

Als Folge auf diese Arbeit, wurde noch im selben Jahr ein Konsortium gegründet, das die Standardisierung der PASS Modellierungssprache als zentrales Ziel hatte. Es wurden sowohl Menschen aus der Forschung, der Beratung, als auch aus der Entwicklung beteiligt, um eine möglichst umfangreiche Sicht auf das Thema zu ermöglichen. Die Arbeit von Herrn Elstermann diente als Richtlinie zum Erstellen einer neuen Ontologie. Aus dieser Arbeit entstand die heutige Standardontologiedatei `standard-pass-ont` [15].

Nachdem der PASS Standard als Ontologie aufgebaut ist, muss kurz geklärt werden was eine Ontologie ist. Herr Elstermann beschreibt eine Ontologie in seiner Arbeit von 2017 folgendermaßen:

> "In that definition an ontology is a knowledge graph structure that describes concepts and instances in a machine-readable form. The knowledge graph is combination of taxonomies, axioms, relationship and data definitions, restrictions, and rules that apply to so-called individuals/instances and their relationships, that may also be stored in an ontology." [14]

Grundsätzlich ist eine Ontologie einem Baumdiagramm sehr ähnlich. Es gibt viele verschiedene Knoten, die miteinander verbunden sind. Anders als bei einem Baumdiagramm gibt es zwar einen Wurzelknoten, von dem alle anderen Knoten ausgehen (hier ist es immer der `owl:Thing` Knoten), jedoch können sich die Knoten auch untereinander verzweigen und somit kann ein Knoten mehr als einen Parent-Knoten haben.

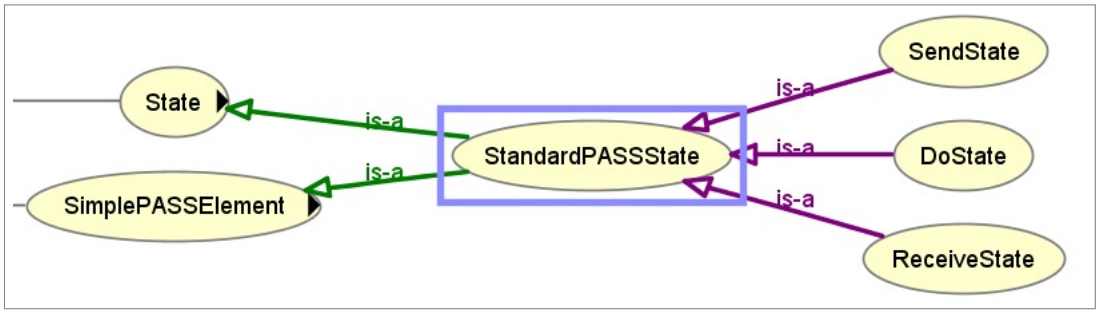

Abbildung 2.2: Ontologie des StandardPASSState aus der standard-pass-ont

Eine Ontologie wird dafür genutzt, Daten strukturiert abzuspeichern. Jeder Knoten steht für ein Datenobjekt und die Kanten zwischen den Knoten stehen für deren Vererbungen untereinander. So gibt es etwa einen `StandardPASSState` und einen `SendState`. Deren Verbindung zeigt, dass der `SendState` ein `StandardPASSState` ist. Dies sieht man in Abbildung 2.2.

Zusätzlich hat jeder Knoten auch noch Attribute die ihn näher beschreiben und Verbindungen zu anderen Knoten. In Abbildung 2.3 sieht man alle Informationen und Attribute, die der `StandardPASSState` Knoten besitzt. Neben den Parent-Knoten, die man bereits in der graphischen Darstellung erkennen konnte, sind alle weiteren Attribute und Verbindungen abgebildet, die dieser Knoten besitzen kann.

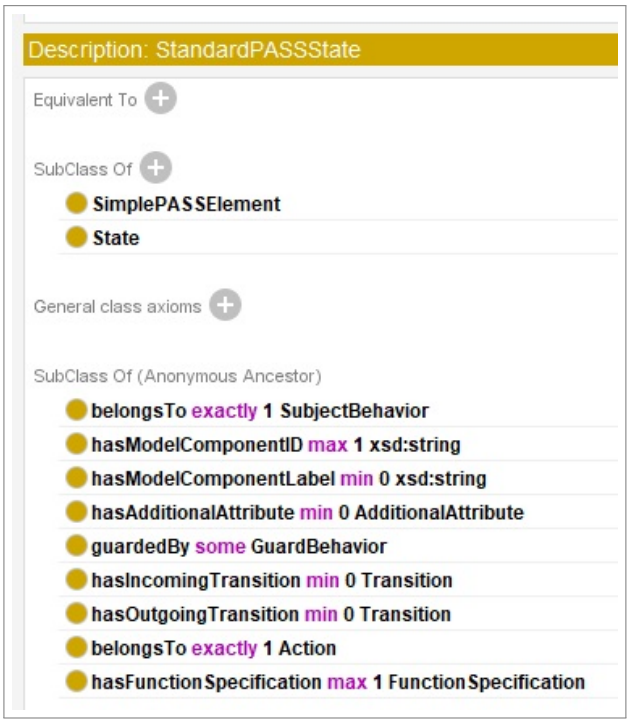

Abbildung 2.3: Detailierter Aufbau des StandardPASSState aus der standard-pass-ont

Damit ist es möglich mit Hilfe einer Ontologie komplexe Systeme extrem detailliert abzubilden. Dies ist einer der Hauptgründe warum die Entscheidung getroffen wurde, den PASS Standard als Ontologie zu definieren. Vor allem im Vergleich zu XML, welches nur in der Lage ist einfache Baumdiagramme abzubilden, haben Ontologien und die damit verbundene OWL Sprache einen entscheidenden Vorteil.

2.3 Übersicht über den PASS OWL-Aufbau

Grundsätzlich besteht eine OWL-Datei aus mehreren `NamedIndividual` Elementen, die sich alle auf der selben Ebene befinden. Darin enthalten sind mehrere Unterelemente, die genauer bestimmen um welches Element es sich dabei handelt. Neben einem Label und einer eindeutigen ID enthält jedes `NamedIndividual` Element mindestens

ein `rdf:type` Element. Darin steht die Klasse (Class) des OWL Elements. Die Class gibt an, um welches PASS Element es sich bei dem OWL Element handelt. Ein OWL Element kann mehrere Classes beinhalten, so kann etwa ein Element sowohl ein `DoState`, als auch ein `EndState` sein.

In der folgenden Aufzählung sieht man alle PASS Classes, die die PASS Kernelemente besitzen können.

- PASSProcessModel
- ModelLayer
- SubjectBehavior
- FullySpecifiedSubject
- MessageExchangeList
- MessageSpecification
- MessageExchange
- Action
- DoState
- SendState
- ReceiveState
- InitialStateOfBehavior
- EndState
- DoTransition
- SendTransition
- ReceiveTransition
- DayTimeTimerTransition
- SendTransitionCondition
- ReceiveTransitionCondition
- DayTimeTimerTransitionCondition

Damit die einzelnen PASS Classes übersichtlicher sind, wurden sie logisch in sechs verschiedene Kategorien unterteilt. Diese sind *Prozessebenen*, *Subjekte*, *Nachrichtenaustausch*, *States*, *Transitions* und *Transition Conditions*. Auch das Unterkapitel über den OWL Aufbau der PASS Elemente wurde auf die gleiche Weise unterteilt. In Abbildung 2.4 sieht man diesen Aufbau als Mindmap.

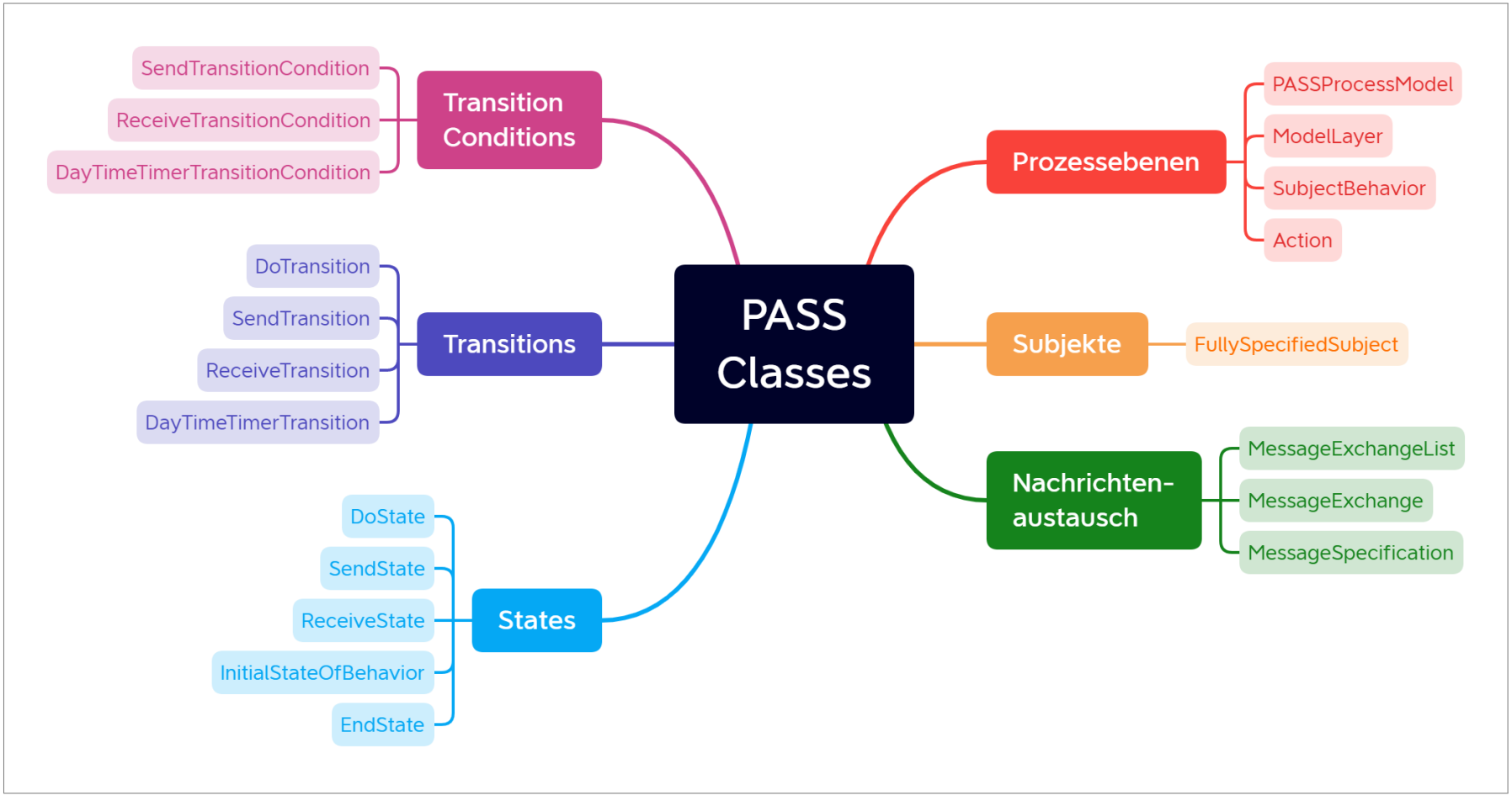

Abbildung 2.4: Mindmap der einzelnen PASS Classes

2.4 OWL Aufbau der einzelnen PASS Elemente

In diesem Kapitel werden die einzelnen PASS Elemente beschrieben und welchen Zweck diese innerhalb des Standards erfüllen. Dabei wird vor allem auf deren OWL-Aufbau eingegangen.

Jedes Element im PASS Standard hat eine eindeutige ID. Diese ist im `NamedIndividual` Tag im `rdf:about` Attribut gespeichert. Die ID setzt sich zusammen aus einem Uniform Resource Locator (URL), der bei jeder ID gleich ist und aus einem zweiten Teil, der das PASS Element genauer beschreibt und eindeutig ist. Die beiden Elemente werden durch ein # Symbol getrennt. Damit die Code-Snippets leichter zu lesen sind, wurde bei allen IDs in dieser Arbeit die URL durch "URL" ersetzt. Die URL für alle Code-Snippets dieser Arbeit lautet:

http://subjective-me.jimdo.com/s-bpm/processmodels/2022-07-07/Page-1

Jedes `NamedIndividual` Tag im PASS Standard besitzt eine `hasModelComponentID` und eine `hasModelComponentLabel` Komponente. Diese Komponenten sind dazu da, um die PASS OWL Elemente zu beschreiben und um ihnen eine `ComponentID` zu geben. Bei der `PASSProcessModel` PASS Class bildet das `ComponentLabel` den zweiten Teil der eindeutigen ID des Elements. Bei allen anderen Elementen bildet die `ComponentID` den zweiten Teil der eindeutigen ID.

In Code-Snippet 2.1 sieht man zwei `NamedIndividual` Tags mit jeweils einer `hasModelComponentID` und `hasModelComponentLabel` Komponente. Zur Vereinfachung werden bei allen weiteren Code-Snippets in diesem Kapitel die beiden Komponenten weggelassen.

```
<owl:NamedIndividual rdf:about="URL#Beispielprozess">
   <standard-pass-ont:hasModelComponentID rdf:datatype="&xsd;string">
      URL
   </standard-pass-ont:hasModelComponentID>
   <standard-pass-ont:hasModelComponentLabel xml:lang="en">
      Beispielprozess
   </standard-pass-ont:hasModelComponentLabel>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="URL#SID_1">
   <standard-pass-ont:hasModelComponentID rdf:datatype="&xsd;string">
       SID_1
    </standard-pass-ont:hasModelComponentID>
   <standard-pass-ont:hasModelComponentLabel xml:lang="en">
       SID_1
    </standard-pass-ont:hasModelComponentLabel>
</owl:NamedIndividual>
```

Code-Snippet 2.1: OWL-Aufbau von ModelComponent und ModelComponentLabel

2.4.1 Prozessebenen

Die PASS Classes in diesem Kapitel sind allesamt Ebenen innerhalb eines PASS Diagramms. Sie bilden eine Oberkategorie, die mehrere PASS Elemente beinhalten um diese zu gliedern.

2.4.1.1 PASSProcessModel

Die `PASSProcessModel` PASS Class kennzeichnet die oberste Ebene eines PASS Prozesses. Er steht somit für das gesamte PASS Diagram und beinhaltet ein `ModelLayer`, alle Subjekte und deren Nachrichtenflüsse sowie alle `SubjectBehavior` Elemente des Prozesses. Den `PASSProcessModel` Tag gibt es genau einmal pro PASS OWL Datei. In Code-Snippet 2.2 sieht man den OWL Aufbau einer `PASSProcessModel` Class.

```
<owl:NamedIndividual rdf:about="URL#Beispielprozess">
  <rdf:type rdf:resource="&standard-pass-ont;PASSProcessModel">
   </rdf:type>
  <standard-pass-ont:contains rdf:resource="URL#SID_1">
  </standard-pass-ont:contains>
  <standard-pass-ont:contains
      rdf:resource="URL#SID_1_FullSpecSubj_2">
  </standard-pass-ont:contains>
  <standard-pass-ont:contains
      rdf:resource="URL#MsgExchLst_on_SID_1_StdMsgConn_32">
  </standard-pass-ont:contains>
  <standard-pass-ont:contains rdf:resource="URL#MsgSpec_2">
  </standard-pass-ont:contains>
  <standard-pass-ont:contains
      rdf:resource="URL#SID_1_StdMsgConn_32_MsgSpec_2">
  </standard-pass-ont:contains>
  <standard-pass-ont:contains
      rdf:resource="URL#SBD_4_SID_1_FullSpecSubj_2">
  </standard-pass-ont:contains>
</owl:NamedIndividual>
```

Code-Snippet 2.2: OWL-Aufbau von der PASSProcessModel Class

2.4.1.2 ModelLayer (SID)

Das SID eines PASS Prozesses wird als `ModelLayer` PASS Class gespeichert. In einem SID werden alle Subjekte eines Prozesses dargestellt. Es gibt ein oder mehrere Start-Subjekte, die mit einem *Play* Symbol gekennzeichnet sind. Außerdem sieht man die Kommunikation zwischen allen Subjekten. Dabei werden nicht nur die Richtungen der Nachrichtenflüsse dargestellt, sondern auch alle einzelnen Nachrichten. In Abbildung 2.5 sieht man ein SID.

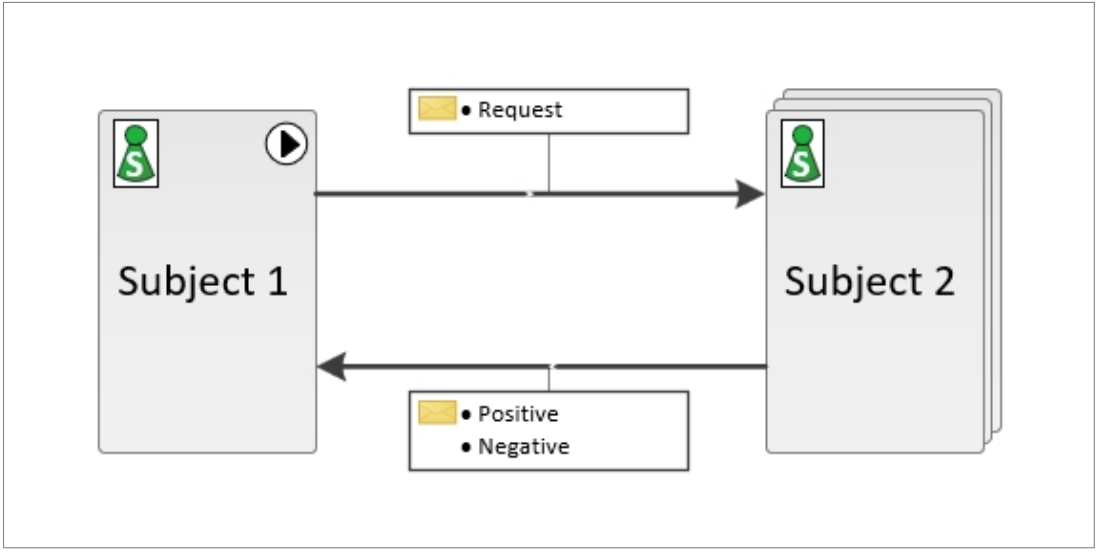

Abbildung 2.5: PASS Subject Interaction Diagram (SID)

Die `ModelLayer` PASS Class beinhaltet ein `hasPriorityNumber` Element. Außerdem werden alle Subjekte, Nachrichtenfllüsse und alle `SubjectBehavior` Elemente des Prozesses als `contains` Elemente dargestellt. Die `PASSProcessModel` Class gibt es genau einmal pro PASS OWL Datei. In Code-Snippet 2.3 sieht man den OWL Aufbau einer `ModelLayer` Class.

```
<owl:NamedIndividual rdf:about="URL#SID_1">
   <rdf:type rdf:resource="&abstract-pass-ont;ModelLayer">
    </rdf:type>
   <standard-pass-ont:hasPriorityNumber
      rdf:datatype="http://www.w3.org/2001/XMLSchema#positiveInteger">
       1
    </standard-pass-ont:hasPriorityNumber>
   <standard-pass-ont:contains
      rdf:resource="URL#SID_1_FullSpecSubj_2">
    </standard-pass-ont:contains>
   <standard-pass-ont:contains
      rdf:resource="URL#MsgExchLst_on_SID_1_StdMsgConn_32">
    </standard-pass-ont:contains>
   <standard-pass-ont:contains rdf:resource="URL#MsgSpec_2">
    </standard-pass-ont:contains>
   <standard-pass-ont:contains
      rdf:resource="URL#SID_1_StdMsgConn_32_MsgSpec_2">
    </standard-pass-ont:contains>
   <standard-pass-ont:contains
      rdf:resource="URL#SBD_4_SID_1_FullSpecSubj_2">
    </standard-pass-ont:contains>
</owl:NamedIndividual>
```

Code-Snippet 2.3: OWL-Aufbau von der ModelLayer Class

2.4.1.3 SubjectBehavior (SBD)

Die SBD eines PASS Prozesses werden als `SubjectBehavior` PASS Class gespeichert. Ein SBD beschreibt alle Aufgaben, die ein Subjekt innerhalb eines Prozesses erledigen muss und alle erlaubten Pfade, die das Subjekt beschreiten kann. Das SBD besteht aus mehreren Actions, welche wiederum je aus einem State und mehreren Transitions bestehen. Es gibt immer genau einen Start-State, der mit einem *Play* Symbol und mindestens einen End-State, der mit einem *Stop* Symbol gekennzeichnet ist. In Abbildung 2.6 sieht man ein einfaches SBD.

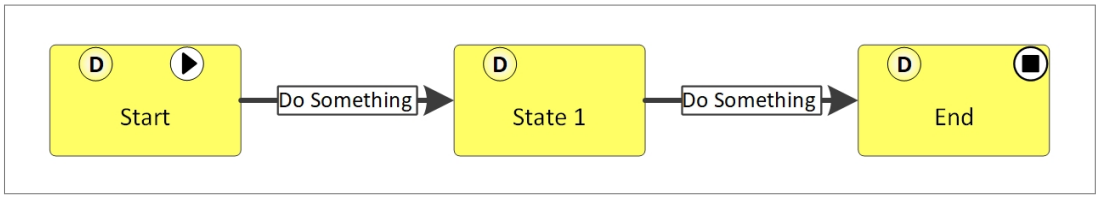

Abbildung 2.6: PASS Subject Behavior Diagram (SBD)

Die `SubjectBehavior` PASS Class beinhaltet ein `hasPriorityNumber` Element. Außerdem werden alle States, Transitions und Actions innerhalb des abgebildeten SBD als `contains` Elemente dargestellt. Falls ein State auch ein Start-State ist, wird er zusätzlich als `hasInitialState` Element gespeichert. Falls ein State auch ein End-State ist, wird er zusätzlich als `hasEndState` Element gespeichert. Die `SubjectBehavior` Class gibt es genau so oft wie es Subjekte im PASS Prozess gibt. In Code-Snippet 2.4 sieht man den OWL Aufbau der `SubjectBehavior` Class aus Abbildung 2.6.

```
1  <owl:NamedIndividual
       rdf:about="URL#SBD_4_SID_1_FullySpecifiedSubject_2">
2    <rdf:type rdf:resource="&standard-pass-ont;SubjectBehavior">
3    </rdf:type>
4    <standard-pass-ont:hasPriorityNumber
       rdf:datatype="http://www.w3.org/2001/XMLSchema#positiveInteger">
5      1
6    </standard-pass-ont:hasPriorityNumber>
7    <standard-pass-ont:hasInitialState
       rdf:resource="URL#SBD_4_DoState_2">
8    </standard-pass-ont:hasInitialState>
9    <standard-pass-ont:contains rdf:resource="URL#SBD_4_DoState_2">
10   </standard-pass-ont:contains>
11   <standard-pass-ont:contains
       rdf:resource="URL#action_of_SBD_4_DoState_2">
12   </standard-pass-ont:contains>
13   <standard-pass-ont:contains
       rdf:resource="URL#SBD_4_DoTransition_17">
14   </standard-pass-ont:contains>
15   <standard-pass-ont:contains rdf:resource="URL#SBD_4_DoState_20">
16   </standard-pass-ont:contains>
17   <standard-pass-ont:contains
       rdf:resource="URL#action_of_SBD_4_DoState_20">
```

```
18    </standard-pass-ont:contains>
19    <standard-pass-ont:contains
          rdf:resource="URL#SBD_4_DoTransition_35">
20    </standard-pass-ont:contains>
21    <standard-pass-ont:hasEndState
          rdf:resource="URL#SBD_4_DoState_38">
22    </standard-pass-ont:hasEndState>
23    <standard-pass-ont:contains rdf:resource="URL#SBD_4_DoState_38">
24    </standard-pass-ont:contains>
25    <standard-pass-ont:contains
          rdf:resource="URL#action_of_SBD_4_DoState_38">
26    </standard-pass-ont:contains>
27 </owl:NamedIndividual>
```

Code-Snippet 2.4: OWL-Aufbau von der SubjectBehavior Class

2.4.1.4 Action

Eine `Action` PASS Class ist eine weitere Prozessebene des PASS Standards. Sie beinhaltet ein State Element mit allen davon ausgehenden Transitions. Der State und die Transitions innerhalb der `Action` PASS Class werden jeweils als `contains` Elemente dargestellt. Das SBD besteht aus mehreren `Action` PASS Classes. Es gibt genauso viele `Action` Classes wie State Classes. In Code-Snippet 2.5 sieht man den OWL Aufbau einer `Action` Class mit einem `DoState` und zwei `DoTransition` Classes.

```
1  <owl:NamedIndividual rdf:about="URL#action_of_SBD_6_DoState_15">
2    <rdf:type rdf:resource="&standard-pass-ont;Action">
3     </rdf:type>
4    <standard-pass-ont:contains rdf:resource="URL#SBD_6_DoState_15">
5     </standard-pass-ont:contains>
6    <standard-pass-ont:contains rdf:resource="URL#SBD_6_DoTrns_61">
7     </standard-pass-ont:contains>
8    <standard-pass-ont:contains rdf:resource="URL#SBD_6_DoTrns_64">
9     </standard-pass-ont:contains>
10 </owl:NamedIndividual>
```

Code-Snippet 2.5: OWL-Aufbau einer Action

2.4.2 Subjekte

Die Subjekte eines PASS Prozesses werden als `FullySpecifiedSubject` PASS Class gespeichert. Ein Subjekt ist eine abstrakte Beschreibung eines Prozessausführenden. Jedes Subjekt hat ein SBD. Es gibt beliebig viele Subjekte in einem PASS Prozess. In Abbildung 2.7 sieht man ein `FullySpecifiedSubject`.

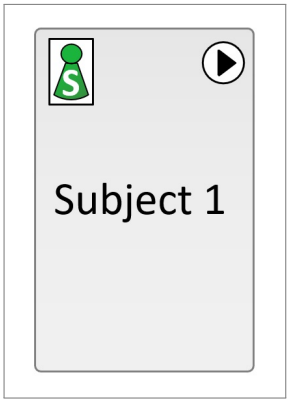

Abbildung 2.7: PASS Subjekt

Jedes Subjekt Element in einer OWL Datei besitzt die `FullySpecifiedSubject` PASS Class. Falls das Subjekt den Prozess startet, beinhaltet es auch die `StartSubject` PASS Class. Außerdem beinhaltet es noch einen Verweis auf den SBD des Subjekts als `containsBehavior` und `containsBaseBehavior` Elemente. Schlussendlich beinhaltet ein `FullySpecifiedSubject` noch eine `hasMaximumSubjectInstanceRestriction`, die auf den Wert 1 gesetzt werden muss. In Code-Snippet 2.6 sieht man den OWL Aufbau des Subjektes aus Abbildung 2.7.

```
<owl:NamedIndividual rdf:about="URL#SID_1_FullSpecSubj_1">
  <rdf:type
     rdf:resource="&standard-pass-ont;FullySpecifiedSubject"/>
   <rdf:type rdf:resource="&standard-pass-ont;StartSubject"/>
  <standard-pass-ont:containsBaseBehavior
     rdf:resource="URL#SBD_4_SID_1_FullSpecSubj_2"/>
   <standard-pass-ont:containsBehavior
      rdf:resource="URL#SBD_4_SID_1_FullSpecSubj_2"/>
   <standard-pass-ont:hasMaximumSubjectInstanceRestriction
      rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
      1
   </standard-pass-ont:hasMaximumSubjectInstanceRestriction>
</owl:NamedIndividual>
```

Code-Snippet 2.6: OWL-Aufbau eines Subjekts

Neben der `FullySpecifiedSubject` Class gibt es noch die `MultiSubject` PASS Class. Diese wird für Subjekte benutzt, welche mehr als einmal mit dem selben SBD vorkommen. Ursprünglich war geplant, dass diese auch in dieser Arbeit genauer beschrieben wird. Bei der Umsetzung stellte sich jedoch heraus, dass das Erstellen eines MultiSubjekts zwar kein Problem ist, jedoch muss auch der gesamte Nachrichtenaustausch dahingehend angepasst werden. Da der Nachrichtenaustausch zu diesem Zeitpunkt jedoch bereits umgesetzt war und eine Änderung von diesem sehr viel Aufwand bedeutet hätte, wurde die Entscheidung getroffen, die MultiSubjekte aus Zeitgründen wegzulassen. Dies bietet jedoch die Chance die `MultiSubject` PASS Class in einer möglichen weiterführenden Arbeit einzubauen und somit die Umwandlungssoftware dieser Arbeit weiter zu verbessern.

2.4.3 Nachrichtenaustausch

Der Nachrichtenaustausch in einem PASS Prozess beinhaltet alle Nachrichten, die zwischen den Subjekten ausgetauscht werden können. Er beinhaltet die `MessageExchangeList`, `MessageSpecification` und `MessageExchange` PASS Classes.

2.4.3.1 MessageExchangeList

Das `MessageExchangeList` Element eines PASS Prozesses ist eine Liste aller `MessageExchange` Elemente zwischen zwei Subjekten in eine Richtung. Es wird im SID als gerichteter Pfeil zwischen zwei Subjekten dargestellt. Auf dem Pfeil ist eine Box mit der Auflistung der `MessageExchange` Elemente. In einem PASS Prozess kann es so viele `MessageExchangeList` Elemente geben, wie es Möglichkeiten gibt, die Subjekte zu verbinden. In Abbildung 2.8 sieht man eine `MessageExchangeList` mit zwei `MessageExchange` Elementen.

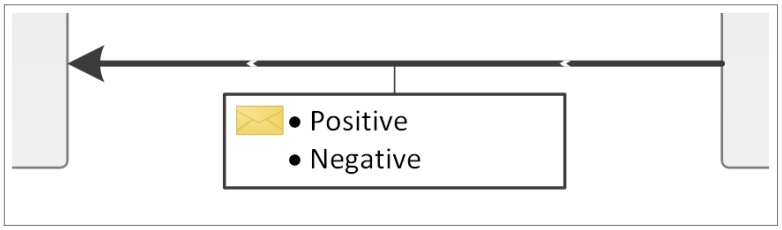

Abbildung 2.8: PASS MessageExchangeList

Die `MessageExchangeList` PASS Class beinhaltet alle `MessageExchange` Elemente mit dem selben Absender und Empfänger. Diese werden als `contains` Elemente dargestellt. Die `MessageExchangeList` Class gibt es genau so oft, wie es `MessageExchange` Elemente mit verschiedenen Absendern und Empfängern gibt. In Code-Snippet 2.7 sieht man den OWL Aufbau der `MessageExchangeList` Class aus Abbildung 2.8.

```
1 <owl:NamedIndividual
     rdf:about="URL#MsgExchLst_on_SID_1_StdMsgConn_32">
2    <rdf:type rdf:resource="&standard-pass-ont;MessageExchangeList">
3     </rdf:type>
4    <standard-pass-ont:contains
        rdf:resource="URL#SID_1_StdMsgConn_32_MsgSpec_2">
5     </standard-pass-ont:contains>
6    <standard-pass-ont:contains
        rdf:resource="URL#SID_1_StdMsgConn_32_MsgSpec_3">
7     </standard-pass-ont:contains>
8 </owl:NamedIndividual>
```

Code-Snippet 2.7: OWL-Aufbau von der MessageExchangeList Class

2.4.3.2 MessageExchange

Die MessageExchange Elemente eines PASS Prozesses sind die Nachrichten, die zwischen zwei Subjekten verschickt werden. Sie werden im SID als Auflistung auf einem MessageExchangeList Pfeil dargestellt. In einem PASS Prozess kann es beliebig viele MessageExchangeList Elemente geben. In Abbildung 2.8 sieht man zwei MessageExchange Elemente auf einem MessageExchangeList Pfeil.

Die MessageExchange PASS Class beinhaltet ein hasSender Element, welches die ID des Sendersubjekts beinhaltet und ein hasReceiver Element, welches die ID des Empfängersubjekts beinhaltet. Außerdem besitzt sie ein hasMessageType Element, welches die MessageSpecification beinhaltet. In Code-Snippet 2.8 sieht man den OWL Aufbau eines der MessageExchange Elemente aus Abbildung 2.8.

```
<owl:NamedIndividual rdf:about="URL#SID_1_StdMsgConn_32_MsgSpec_2">
  <rdf:type rdf:resource="&standard-pass-ont;MessageExchange">
   </rdf:type>
  <standard-pass-ont:hasSender
     rdf:resource="URL#SID_1_FullSpecSubj_22">
   </standard-pass-ont:hasSender>
  <standard-pass-ont:hasReceiver
     rdf:resource="URL#SID_1_FullSpecSubj_2">
   </standard-pass-ont:hasReceiver>
  <standard-pass-ont:hasMessageType rdf:resource="URL#MsgSpec_2">
   </standard-pass-ont:hasMessageType>
</owl:NamedIndividual>
```

Code-Snippet 2.8: OWL-Aufbau von der MessageExchange Class

2.4.3.3 MessageSpecification (PayloadDefinition)

Das MessageSpecification Element eines PASS Prozesses beinhaltet die tatsächliche Nachricht eines MessageExchange Elements. Das Label des MessageSpecification Elements wird im SID als Name in der Auflistung der MessageExchange Elemente auf einem MessageExchangeList Pfeil dargestellt. In einem PASS Prozess gibt es genau so viele MessageSpecification Elemente wie es MessageExchange Elemente gibt.

Neben der Nachricht, können sich innerhalb eines MessageSpecification Elements noch viele weitere Attribute und Informationen befinden. Diese werden in einem eigenen Diagramm als Message Payload gespeichert. In Abbildung 2.9 wird der Payload einer der MessageSpecification Elemente gezeigt, die in Abbildung 2.8 zu sehen sind.

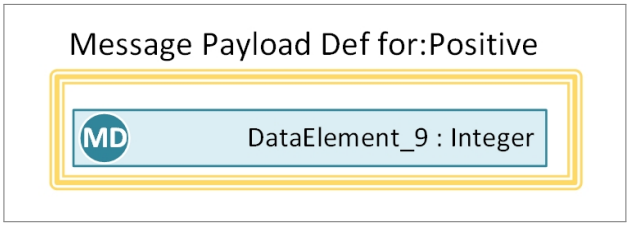

Abbildung 2.9: PASS Standard Message Payload

Die `MessageSpecification` PASS Class beinhaltet ein `hasModelComponentLabel`, welches die Nachricht, die zwischen den Subjekten verschickt wurde beinhaltet. Außerdem besitzt sie ein `containsPayloadDescription` Element, welches auf ein `PayloadDefinition` Element verweist. In Code-Snippet 2.9 sieht man den OWL Aufbau eines der `MessageSpecification` Elemente aus Abbildung 2.8.

Das `PayloadDefinition` Element beinhaltet alle Attribute und Informationen, die zusammen mit der Nachricht verschickt werden können. In dieser Arbeit wird nicht genau darauf eingegangen welche Attribute das sein können und wie der genaue OWL Aufbau des `PayloadDefinition` Elements aussieht, da diese Informationen im BPMN Standard nicht vorkommen und somit auch nicht übersetzt werden können.

```
<owl:NamedIndividual rdf:about="URL#MsgSpec_2">
  <rdf:type rdf:resource="&standard-pass-ont;MessageSpecification">
   </rdf:type>
   <standard-pass-ont:hasModelComponentLabel xml:lang="en" >
     Positive
  </standard-pass-ont:hasModelComponentLabel>
  <standard-pass-ont:containsPayloadDescription
      rdf:resource="URL#PayloadDefinition_of_MsgSpec_2">
   </standard-pass-ont:containsPayloadDescription>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="URL#PayloadDefinition_of_MsgSpec_2">
  <rdf:type rdf:resource="URL#PayloadDefinition_of_MsgSpec_2">
   </rdf:type>
</owl:NamedIndividual>
```

Code-Snippet 2.9: OWL-Aufbau von der MessageSpecification Class

2.4.4 States

Es gibt fünf verschiedene PASS Classes von States. Diese sind `DoState`, `SendState`, `ReceiveState`, `InitialStateOfBehavior` und `EndState`. Wobei nur die ersten drei Classes im Prozess als States dargestellt werden. Die anderen beiden Classes sind Zusatzinformationen für die ersten drei Classes. Die States werden im SBD als Boxen mit abgerundeten Ecken dargestellt. Jede State Class hat eine andere Farbe. Sie beschreiben alle Zustände in denen sich ein Subjekt befinden kann. Ein Subjekt darf sich immer nur in einem State befinden. Zwischen den States gibt es die Transititions. Diese beschreiben

die Zustandsübergänge von einem State zum nächsten. Es gibt beliebig viele States in einem SBD. In Abbildung 2.10 sieht man alle Arten von States in einem SBD.

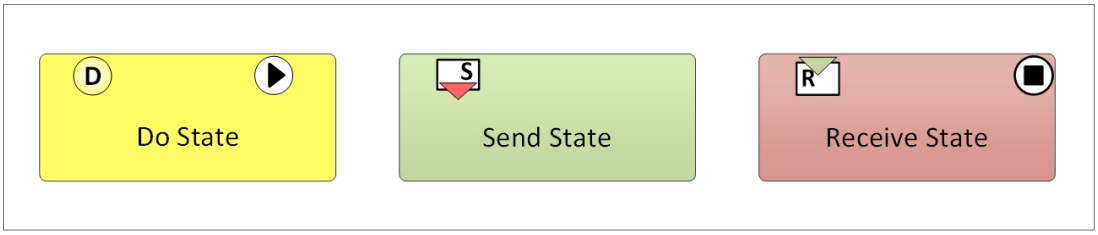

Abbildung 2.10: PASS States

Jedes State Element in einer OWL Datei beinhaltet das `hasFunctionSpecification` Element. In Code-Snippet 2.10 sieht man einen vereinfachten OWL Aufbau eines States mit diesem Element. In den nächsten Unterkapiteln wird beschrieben welche Details einen `DoState`, `SendState`, `ReceiveState`, `InitialStateOfBehavior` und einen `EndState` unterscheiden.

```
1 <owl:NamedIndividual rdf:about="URL#SBD_6_DoState_15">
2   <standard-pass-ont:hasFunctionSpecification
      rdf:resource="URL#functionDefinition_of_SBD_6_DoState_15">
3    </standard-pass-ont:hasFunctionSpecification>
4 </owl:NamedIndividual>
```

Code-Snippet 2.10: Grundsätzlicher OWL-Aufbau eines States

2.4.4.1 DoState

Ein `DoState` ist ein State, bei dem ein Subjekt mit keinem anderen Subjekt interagiert. Im SBD wird er als gelbe Box mit einem *D* Symbol in der linken oberen Ecke dargestellt. In Abbildung 2.10 sieht man einen solchen State auf der linken Seite.

Neben dem grundsätzlichen OWL Aufbau eines States beinhaltet ein `DoState` noch die `DoState` PASS Class. In Code-Snippet 2.11 sieht man diesen Aufbau.

```
1 <owl:NamedIndividual rdf:about="URL#SBD_6_DoState_15">
2   <rdf:type rdf:resource="&standard-pass-ont;DoState">
3    </rdf:type>
4 </owl:NamedIndividual>
```

Code-Snippet 2.11: OWL-Aufbau von der DoState Class

2.4.4.2 SendState

Ein `SendState` ist ein State, bei dem ein Subjekt eine Nachricht an ein anderes Subjekt schickt. Im SBD wird er als grüne Box mit einem *S* Symbol in der linken oberen Ecke dargestellt. In Abbildung 2.10 sieht man einen solchen State in der Mitte.

Neben dem grundsätzlichen OWL Aufbau eines States beinhaltet ein `SendState` noch die `SendState` PASS Class. In Code-Snippet 2.12 sieht man diesen Aufbau.

```
<owl:NamedIndividual rdf:about="URL#SBD_4_SendState_30">
   <rdf:type rdf:resource="&standard-pass-ont;SendState">
    </rdf:type>
</owl:NamedIndividual>
```

Code-Snippet 2.12: OWL-Aufbau von der SendState Class

2.4.4.3 ReceiveState

Ein `ReceiveState` ist ein State, bei dem ein Subjekt eine Nachricht von einem anderen Subjekt empfängt. Im SBD wird er als rote Box mit einem *R* Symbol in der linken oberen Ecke dargestellt. In Abbildung 2.10 sieht man einen solchen State auf der rechten Seite.

Neben dem grundsätzlichen OWL Aufbau eines States beinhaltet ein `ReceiveState` noch die `ReceiveState` PASS Class. In Code-Snippet 2.13 sieht man diesen Aufbau.

```
<owl:NamedIndividual rdf:about="URL#SBD_4_RcvState_17">
   <rdf:type rdf:resource="&standard-pass-ont;ReceiveState">
    </rdf:type>
</owl:NamedIndividual>
```

Code-Snippet 2.13: OWL-Aufbau von der ReceiveState Class

2.4.4.4 InitialStateOfBehavior

Ein `InitialStateOfBehavior` kennzeichnet einen State als den ersten State eines SBD. Er kann entweder ein `DoState`, `SendState` oder ein `ReceiveState` sein. Es gibt immer nur einen `InitialStateOfBehavior` State pro SBD. Im SBD wird er als *Play* Symbol in der rechten oberen Ecke eines States dargestellt. In Abbildung 2.10 sieht man einen solchen State auf der linken Seite.

Neben dem grundsätzlichen OWL Aufbau eines States beinhaltet ein `InitialStateOfBehavior` noch die entsprechende PASS Class und eine `DoState`, `SendState` oder eine `ReceiveState` Class. Eine `InitialStateOfBehavior` Class kann sich nie ohne eine andere Class in einem State Element befinden. Code-Snippet 2.14 zeigt diesen Aufbau mit einer `DoState` Class.

```
<owl:NamedIndividual rdf:about="URL#SBD_4_DoState_2">
    <rdf:type rdf:resource="&standard-pass-ont;DoState">
    </rdf:type>
   <rdf:type
      rdf:resource="&standard-pass-ont;InitialStateOfBehavior">
    </rdf:type>
</owl:NamedIndividual>
```

Code-Snippet 2.14: OWL-Aufbau von der InitialStateOfBehavior Class

2.4.4.5 EndState

Ein `EndState` kennzeichnet einen State als den letzten State eines SBD in dem sich das Subjekt ganz am Ende befindet. Er kann entweder ein `DoState`, `SendState` oder ein `ReceiveState` sein. Es kann beliebig viele `EndState` States pro SBD geben. Im SBD wird er als *Stop* Symbol in der rechten oberen Ecke eines States dargestellt. In Abbildung 2.10 sieht man einen solchen State auf der rechten Seite.

Neben dem grundsätzlichen OWL Aufbau eines States beinhaltet ein `EndState` noch die `EndState` PASS Class und eine `DoState`, `SendState` oder eine `ReceiveState` Class. Eine `EndState` Class kann sich nie ohne eine andere Class in einem State Element befinden. In Code-Snippet 2.15 sieht man diesen Aufbau mit einer `DoState` Class.

```
<owl:NamedIndividual rdf:about="URL#SBD_4_DoState_138">
    <rdf:type rdf:resource="&standard-pass-ont;DoState">
    </rdf:type>
   <rdf:type rdf:resource="&standard-pass-ont;EndState">
    </rdf:type>
</owl:NamedIndividual>
```

Code-Snippet 2.15: OWL-Aufbau von der EndState Class

2.4.5 Transitions

Es gibt vier verschiedene PASS Classes von Transitions. Diese sind `DoTransition`, `SendTransition`, `ReceiveTransition` und `DayTimeTimerTransition`. Die Transitions werden im SBD als gerichtete Pfeile mit einer Box darauf, zwischen zwei States, dargestellt. Sie beschreiben alle Zustandsübergänge zwischen den States, die ein Subjekt benutzen darf. Es gibt beliebig viele Transitions, abhängig von der Anzahl der States in einem SBD. In Abbildung 2.11 sieht man alle Arten von Transitions in einem SBD.

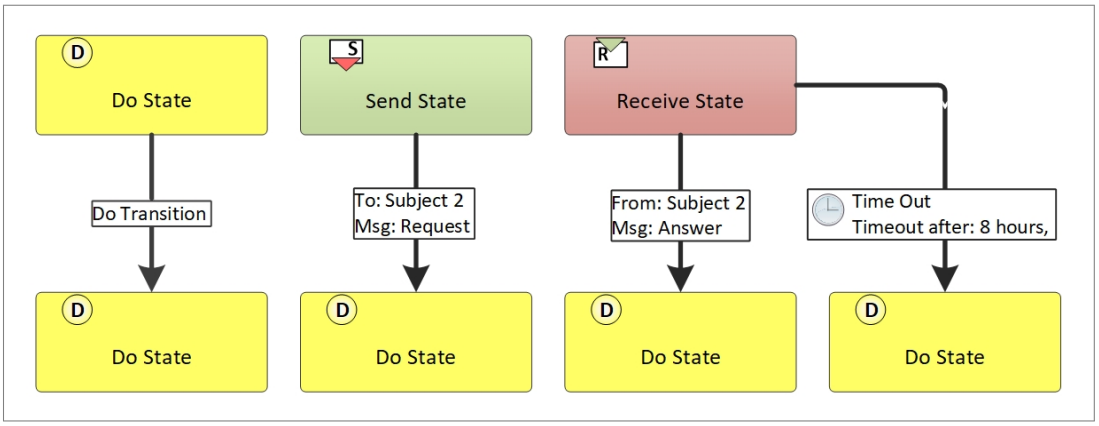

Abbildung 2.11: PASS Transitions

Jedes Transition Element in einer OWL Datei beinhaltet das `hasPriorityNumber` Element. Außerdem beinhaltet es ein `hasSourceState` Element, welches die ID des

Ursprungsstates beinhaltet und ein `hasTargetState` Element, welches die ID des Zielstates beinhaltet. In Code-Snippet 2.16 sieht man einen vereinfachten OWL Aufbau einer Transition mit diesen Elementen. In den nächsten Unterkapiteln wird beschrieben, welche Details eine `DoTransition`, `SendTransition`, `ReceiveTransition` und eine `DayTimeTimerTransition` unterscheiden.

```
<owl:NamedIndividual rdf:about="URL#SBD_6_DoTrns_61">
  <standard-pass-ont:hasSourceState
     rdf:resource="URL#SBD_6_DoState_15">
   </standard-pass-ont:hasSourceState>
  <standard-pass-ont:hasTargetState
     rdf:resource="URL#SBD_6_SndState_35">
   </standard-pass-ont:hasTargetState>
  <standard-pass-ont:hasPriorityNumber
     rdf:datatype="http://www.w3.org/2001/XMLSchema#positiveInteger">
      1
   </standard-pass-ont:hasPriorityNumber>
</owl:NamedIndividual>
```

Code-Snippet 2.16: Grundsätzlicher OWL-Aufbau einer Transition

2.4.5.1 DoTransition

Eine `DoTransition` ist eine Transition, die immer einen `DoState` als `SourceState` hat. Sie ist dafür zuständig einen Zustandsübergang zu beschreiben. Im SBD wird sie als Pfeil mit einer Box darauf dargestellt. In der Box befindet sich die Bezeichnung der Transition. In Abbildung 2.11 sieht man eine solche Transition auf der linken Seite.

Neben dem grundsätzlichen OWL Aufbau einer Transition beinhaltet eine `DoTransition` noch die `DoTransition` PASS Class. In Code-Snippet 2.17 sieht man diesen Aufbau.

```
<owl:NamedIndividual rdf:about="URL#SBD_6_DoTrns_61">
  <rdf:type rdf:resource="&standard-pass-ont;DoTransition">
   </rdf:type>
</owl:NamedIndividual>
```

Code-Snippet 2.17: OWL-Aufbau von der DoTransition Class

2.4.5.2 SendTransition

Eine `SendTransition` ist eine Transition, die immer einen `SendState` als `SourceState` hat. Sie ist dafür zuständig einen Zustandsübergang zu beschreiben und eine Nachricht zu verschicken. Im SBD wird sie als Pfeil mit einer Box darauf dargestellt. In der Box befindet sich eine Nachricht und das Subjekt an das sie geschickt werden soll. In Abbildung 2.11 sieht man eine solche Transition als zweites Element von links.

Neben dem grundsätzlichen OWL Aufbau einer Transition beinhaltet eine `SendTransition` noch die `SendTransition` PASS Class. Außerdem beinhaltet sie ein `hasTransitionCondition` Element welches die ID der zuständigen TransitionCondition beinhaltet. Im Falle einer `SendTransition` ist das eine `SendTransitionCondition`. In Code-Snippet 2.18 sieht man diesen Aufbau.

```
<owl:NamedIndividual rdf:about="URL#SBD_4_SndTrns_1">
  <rdf:type rdf:resource="&standard-pass-ont;SendTransition">
   </rdf:type>
  <standard-pass-ont:hasTransitionCondition
     rdf:resource="URL#SBD_4_SndTrns_1_sndTrnsCnd">
   </standard-pass-ont:hasTransitionCondition>
</owl:NamedIndividual>
```

Code-Snippet 2.18: OWL-Aufbau von der SendTransition Class

2.4.5.3 ReceiveTransition

Eine `ReceiveTransition` ist eine Transition, die immer einen `ReceiveState` als `SourceState` hat. Sie ist dafür zuständig einen Zustandsübergang zu beschreiben und eine Nachricht zu empfangen. Im SBD wird sie als Pfeil mit einer Box darauf dargestellt. In der Box befindet sich eine Nachricht und das Subjekt von dem sie empfangen werden soll. In Abbildung 2.11 sieht man eine solche Transition als drittes Element von links.

Neben dem grundsätzlichen OWL Aufbau einer Transition beinhaltet eine `ReceiveTransition` noch die `ReceiveTransition` PASS Class. Außerdem beinhaltet sie ein `hasTransitionCondition` Element welches die ID der zuständigen TransitionCondition beinhaltet. Im Falle einer `ReceiveTransition` ist das eine `ReceiveTransitionCondition`. In Code-Snippet 2.19 sieht man diesen Aufbau.

```
<owl:NamedIndividual rdf:about="URL#SBD_4_RcvTrns_2">
  <rdf:type rdf:resource="&standard-pass-ont;ReceiveTransition">
   </rdf:type>
  <standard-pass-ont:hasTransitionCondition
     rdf:resource="URL#SBD_4_RcvTrns_2_rcvTrnsCnd">
   </standard-pass-ont:hasTransitionCondition>
</owl:NamedIndividual>
```

Code-Snippet 2.19: OWL-Aufbau von der ReceiveTransition Class

2.4.5.4 DayTimeTimerTransition

Eine DayTimeTimerTransition ist eine Transition, die jeden State als SourceState haben kann. Sie ist dafür zuständig einen Zustandsübergang zu beschreiben und diesen das Subjekt auch beschreiten zu lassen, wenn eine definierte Zeit verstrichen ist. Im SBD wird sie als Pfeil mit einer Box darauf dargestellt. In der Box befindet sich die Zeit, die vergehen muss, um die Transition auszulösen. In Abbildung 2.11 sieht man eine solche Transition auf der rechten Seite.

Neben dem grundsätzlichen OWL Aufbau einer Transition beinhaltet eine DayTimeTimerTransition noch die DayTimeTimerTransition PASS Class. Außerdem beinhaltet sie ein hasTransitionCondition Element welches die ID der zuständigen TransitionCondition beinhaltet. Im Falle einer DayTimeTimerTransition ist das eine DayTimeTimerTransitionCondition. In Code-Snippet 2.20 sieht man diesen Aufbau.

```
<owl:NamedIndividual rdf:about="URL#SBD_4_DayTimeTimerTrns_132">
   <rdf:type
      rdf:resource="&standard-pass-ont;DayTimeTimerTransition">
   </rdf:type>
  <standard-pass-ont:hasTransitionCondition
     rdf:resource="URL#SBD_4_DayTimeTimerTrns_132_TimeTrnsCnd">
   </standard-pass-ont:hasTransitionCondition>
</owl:NamedIndividual>
```

Code-Snippet 2.20: OWL-Aufbau von der DayTimeTimerTransition Class

2.4.6 Transition Conditions

Es gibt drei verschiedene PASS Classes von TransitionConditions. Diese sind SendTransitionCondition, ReceiveTransitionCondition und DayTimeTimerTransitionCondition. Die TransitionConditions werden im SBD als Box auf einer Transition dargestellt. Sie beschreiben alle Transitions bis auf die DoTransition näher und sind bei diesen auch verpflichtend. In Abbildung 2.11 sieht man alle Arten von TransitionConditions in einem SBD.

2.4.6.1 SendTransitionCondition

Eine SendTransitionCondition ist eine TransitionCondition, die ein SendTransition Element näher beschreibt. Sie beschreibt welche Nachricht verschickt werden soll und wer das Empfängersubjekt ist. In der Box im SBD befinden sich die Nachricht und das Empfängersubjekt. In Abbildung 2.11 sieht man eine solche TransitionCondition als zweites Element von links.

Die SendTransitionCondition PASS Class beinhaltet ein requiresPerformedMessageExchange Element, welches die ID des verbundenen MessageExchange Elements beinhaltet. Außerdem beinhaltet sie ein requiresMessageSentTo Element, welches

die ID des Empfängersubjekts beinhaltet und ein `requiresSendingOfMessage` Element, welches die ID des verbundenen `MessageSpecification` Elements beinhaltet. Schlussendlich besitzt sie auch noch ein `hasSendType` Element. In Code-Snippet 2.21 sieht man diesen Aufbau.

```
<owl:NamedIndividual rdf:about="URL#SBD_4_SndTrns_1_sndTrnsCnd">
   <rdf:type
      rdf:resource="&standard-pass-ont;SendTransitionCondition">
    </rdf:type>
   <standard-pass-ont:requiresPerformedMessageExchange
      rdf:resource="URL#SID_1_StdMsgConn_12_MsgSpec_1">
    </standard-pass-ont:requiresPerformedMessageExchange>
   <standard-pass-ont:requiresMessageSentTo
      rdf:resource="URL#SID_1_FullSpecSubj_22">
    </standard-pass-ont:requiresMessageSentTo>
   <standard-pass-ont:requiresSendingOfMessage
      rdf:resource="URL#MsgSpec_1">
    </standard-pass-ont:requiresSendingOfMessage>
   <standard-pass-ont:hasSendType
      rdf:resource="http://www.i2pm.net/standard-pass-ont#SendTypeStandard">
    </standard-pass-ont:hasSendType>
</owl:NamedIndividual>
```

Code-Snippet 2.21: OWL-Aufbau von der SendTransitionCondition Class

2.4.6.2 ReceiveTransitionCondition

Eine `ReceiveTransitionCondition` ist eine TransitionCondition, die ein `ReceiveTransition` Element näher beschreibt. Sie beschreibt, welche Nachricht empfangen werden soll und wer das Sendersubjekt ist. In der Box im SBD befinden sich die Nachricht und das Sendersubjekt. In Abbildung 2.11 sieht man eine solche TransitionCondition als drittes Element von links.

Die `ReceiveTransitionCondition` PASS Class beinhaltet ein `requiresPerformedMessageExchange` Element, welches die ID des verbundenen `MessageExchange` Elements beinhaltet. Außerdem beinhaltet sie ein `requiresMessageSentFrom` Element, welches die ID des Sendersubjekts beinhaltet und ein `requiresReceptionOfMessage` Element, welches die ID des verbundenen `MessageSpecification` Elements beinhaltet. Schlussendlich besitzt sie auch noch ein `hasReceiveType` Element. In Code-Snippet 2.22 sieht man diesen Aufbau.

```
<owl:NamedIndividual rdf:about="URL#SBD_4_RcvTrns_2_rcvTrnsCnd">
   <rdf:type
      rdf:resource="&standard-pass-ont;ReceiveTransitionCondition">
    </rdf:type>
   <standard-pass-ont:requiresMessageSentFrom
      rdf:resource="URL#SID_1_FullSpecSubj_22">
    </standard-pass-ont:requiresMessageSentFrom>
   <standard-pass-ont:requiresReceptionOfMessage
      rdf:resource="URL#MsgSpec_2">
    </standard-pass-ont:requiresReceptionOfMessage>
   <standard-pass-ont:requiresPerformedMessageExchange
      rdf:resource="URL#SID_1_StdMsgConn_32_MsgSpec_2">
    </standard-pass-ont:requiresPerformedMessageExchange>
   <standard-pass-ont:hasReceiveType
      rdf:resource="http://www.i2pm.net/standard-pass-ont#ReceiveTypeStandard">
    </standard-pass-ont:hasReceiveType>
</owl:NamedIndividual>
```

Code-Snippet 2.22: OWL-Aufbau von der ReceiveTransitionCondition Class

2.4.6.3 DayTimeTimerTransitionCondition

Eine `DayTimeTimerTransitionCondition` ist eine TransitionCondition, die ein `DayTimeTimerTransition` Element näher beschreibt. Sie beschreibt wie lange die Zeit läuft, bis die Transition ausgelöst wird. In der Box im SBD befindet sich diese Zeit. In Abbildung 2.11 sieht man eine solche TransitionCondition auf der rechten Seite.

Die `DayTimeTimerTransitionCondition` PASS Class beinhaltet ein `hasDayTimeDurationTimeOutTime` Element, welches die Zeit beinhaltet, die verstreichen muss, um die Transition auszulösen. In Code-Snippet 2.23 sieht man diesen Aufbau.

```
<owl:NamedIndividual
    rdf:about="URL#SBD_4_DayTimeTimerTrns_132_TimeTrnsCnd">
   <rdf:type
      rdf:resource="&standard-pass-ont;DayTimeTimerTransitionCondition">
    </rdf:type>
   <standard-pass-ont:hasDayTimeDurationTimeOutTime
      rdf:datatype="http://www.w3.org/2001/XMLSchema#dayTimeDuration">
       P0DT8H0M0S
    </standard-pass-ont:hasDayTimeDurationTimeOutTime>
</owl:NamedIndividual>
```

Code-Snippet 2.23: OWL-Aufbau von der DayTimeTimerTransitionCondition Class

3 Aufbau von BPMN

Nachdem im vorherigen Kapitel der Aufbau des PASS Standards beschrieben wurde, geht es jetzt mit dem grundsätzlichen Aufbau des BPMN Standards weiter.

3.1 Elementeinschränkung und sonstige Vereinfachungen

Der BPMN Standard ist wesentlich umfangreicher als der PASS Standard. Es gibt über 100 verschiedene Symbole. Aber nicht alle Symbole sind dafür geeignet ausgeführt zu werden und noch weniger kann man in PASS Elemente umwandeln. Aufgrund dessen wird nicht der gesamte BPMN Standard beschrieben, sondern nur jene Elemente, die sich ohne Probleme und ohne den Verlust von Informationen übersetzen lassen.

Ziel hierbei war es für jedes PASS Element, das im vorherigen Kapitel beschrieben wurde, ein Äquivalent zu finden. Zu diesem Zweck wurde die ”BPMN Kernelement Palette” herangezogen und alle Elemente die nicht übersetzbar sind herausgestrichen. Zusätzlich wurden dann weitere Elemente gesucht, die in die übrigen PASS Elemente übersetzt werden können.

In dieser Arbeit wurde nicht versucht alle möglichen BPMN Elemente zu unterstützen und zu übersetzen. Das Ziel war, dass man die übersetzbaren PASS Konzepte mit BPMN Elementen abdecken kann und somit ohne große Einschränkungen subjektorientiert mit einem BPMN Tool modellieren kann. Außerdem ging es hierbei um die Umsetzung eines Prototypen und nicht darum ein fertiges Produkt zu entwickeln.

Im Laufe der Arbeit wurde entschieden sich auf diese BPMN Elemente zu fokussieren:

- Collaboration
- Pool (Participant)
- Message Flow
- Process
- Start Event
- Task
- Exclusive Gateway
- Event Based Gateway
- Throw Event
  - Message Throw Event

- Catch Event
  - Message Catch Event
  - Timer Catch Event
- End Event
- Sequence Flow

In Abbildung 3.1 sieht man einen Prozess mit allen Elementen, die sich innerhalb der Elementeinschränkung befinden. Es gibt zwei Participants. Firma A und Firma B. Firma A trifft zuerst eine Entscheidung und führt danach einen von zwei Tasks aus. Danach wird eine Nachricht an Firma B geschickt und der Prozess beendet. Firma B führt zuerst einen Task aus und wartet danach auf die Nachricht von Firma A. Wenn diese nicht innerhalb von 8 Stunden eintrifft wird der Prozess abgebrochen. Wenn die Nachricht rechtzeitig eintrifft wird ein weiterer Task ausgeführt und der Prozess anschließend beendet.

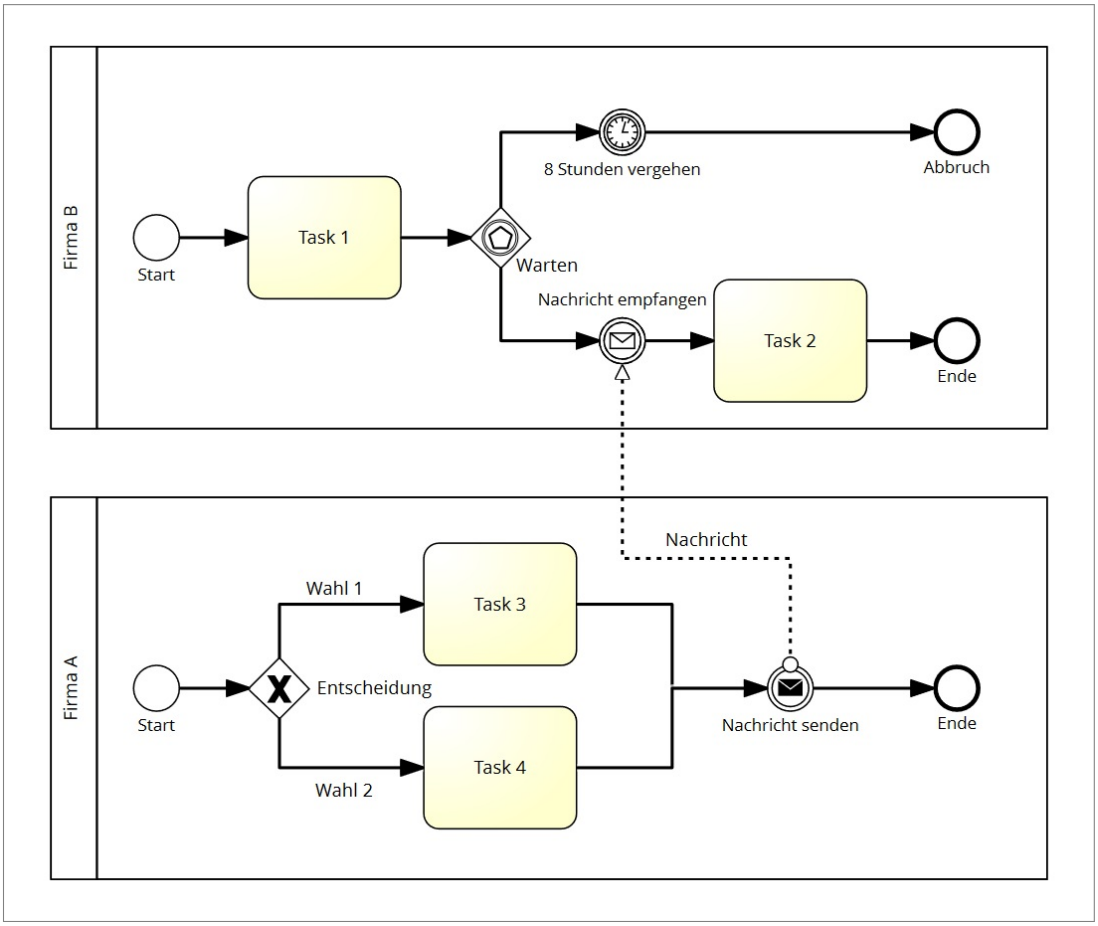

Abbildung 3.1: Alle BPMN Elemente innerhalb der Elementeinschränkung

Neben den BPMN Elementen, die benötigt werden, um die Prozesse zu beschreiben, gibt es noch sogenannte Extension Elements. Diese können bei jedem anderen Element eingebaut werden, um weitere Informationen zu speichern. Dies wird meistens von den BPMN Editoren verwendet, um den Standard um zusätzliche Informationen und Formatierungen zu erweitern. Jeder Editor verwendet andere Extension Elements, aber diese tragen nichts zu der Prozesslogik bei. Daher werden für die Umwandlung alle Extension Elements aus der XML Datei entfernt. Auch in der Arbeit wird nicht weiter auf diese Elemente eingegangen.

Schlussendlich befindet sich innerhalb des BPMN Standards noch das `bpmndi:BPMNDiagram` Element. Darin wird auf jedes Element des Prozesses verwiesen und es werden zusätzliche Informationen gespeichert. Dabei handelt es sich um die Form der Elemente, deren Abmessungen und Position im Modell, sowie die Formatierung der Beschriftung der Elemente. Diese Informationen sind zwar für den BPMN Standard und um das Modell darzustellen sehr wichtig, allerdings für die Transformation in PASS werden sie nicht benötigt. Daher werden auch diese Informationen aus dem Modell gelöscht und auch in der Arbeit wird nicht genauer darauf eingegangen.

3.2 Übersicht über den BPMN XML-Aufbau

In diesem Kapitel wird ein Überblick über den BPMN-Standard gegeben. Es wird darauf eingegangen, wie er aufgebaut ist und aus welchen Elementen der Standard besteht. Für diesen Zweck wurde ein bestehendes Modell genommen und alle Modellinformationen wurden daraus gelöscht, sodass nur mehr der Tag-Aufbau des XML Dokuments übrig geblieben ist. In Code-Snippet 3.1 sieht man diesen Aufbau.

```
<definitions>
  <collaboration>
    <participant/>
    <messageFlow/>
  </collaboration>
  <process>
    <laneSet>
      <lane>
        <flowNodeRef/>
      </lane>
    </laneSet>
    <startEvent>
      <outgoing/>
    </startEvent>
    <task>
      <incoming/>
      <outgoing/>
    </task>
    <intermediateThrowEvent>
      <incoming/>
      <outgoing/>
      <messageEventDefinition/>
```

```
23      </intermediateThrowEvent>
24      <intermediateCatchEvent>
25        <incoming/>
26        <outgoing/>
27        <messageEventDefinition/>
28        <timerEventDefinition>
29          <timeDuration/>
30        </timerEventDefinition>
31      </intermediateCatchEvent>
32      <eventBasedGateway>
33        <incoming/>
34        <outgoing/>
35      </eventBasedGateway>
36      <exclusiveGateway>
37        <incoming/>
38        <outgoing/>
39      </exclusiveGateway>
40      <endEvent>
41        <incoming/>
42      </endEvent>
43      <sequenceFlow/>
44    </process>
45  </definitions>
```

Code-Snippet 3.1: Aufbau von BPMN

Der gesamte Inhalt des BPMN Modells befindet sich innerhalb des `definitions` Tag. Dieser ist wiederum in zwei Hauptelemente aufgeteilt. Dem `collaboration` Tag und einem oder mehreren `process` Tags.

Im `collaboration` Tag befinden sich null, ein oder mehrere `participant` und `messageFlow` Tags.

Innerhalb des `process` Tag befindet sich der `laneSet` und der `startEvent` Tag, sowie ein oder mehrere `endEvent` und `sequenceFlow` Tags , sowie null, ein oder mehrere `task`, `intermediateThrowEvent`, `intermediateCatchEvent`, `eventBasedGateway` und `exclusiveGateway` Tags.

Im `laneSet` Tag befindet sich ein `lane` Tag und darin mehrere `flowNodeRef` Tags.

Innerhalb des `startEvent` Tag befinden sich ein oder mehrere `outgoing` Tags.

Innerhalb des `task` Tag befinden sich ein oder mehrere `incoming` und `outgoing` Tags.

Innerhalb des `intermediateThrowEvent` Tag befinden sich ein oder mehrere `incoming` und `outgoing` Tags, sowie ein `messageEventDefinition` Tag.

Innerhalb des `intermediateCatchEvent` Tag befinden sich ein oder mehrere `incoming` und `outgoing` Tags, sowie entwededer ein `messageEventDefinition` oder ein `timerEventDefinition` Tag.

Innerhalb des `timerEventDefinition` Tag befindet sich ein `timeDuration` Tag.

Innerhalb des `eventBasedGateway` Tag befinden sich ein oder mehrere `incoming` und `outgoing` Tags.

Innerhalb des `exclusiveGateway` Tag befinden sich ein oder mehrere `incoming` und `outgoing` Tags.

Innerhalb des `endEvent` Tag befinden sich ein oder mehrere `incoming` Tags.

3.3 XML Aufbau der einzelnen BPMN Elemente

In diesem Kapitel werden die einzelnen BPMN Elemente beschrieben und welchen Zweck diese innerhalb des Standards erfüllen. Dabei wird vor allem auf deren XML-Aufbau eingegangen.

3.3.1 Collaboration

Am Anfang der BPMN XML Datei gibt es den `collaboration` Tag. Der Tag besitzt immer eine eindeutige ID und innerhalb des Tags befinden sich die `participant` (Prozessteilnehmer) und `messageFlow` (Nachrichtenfluss) Tags. Den `collaboration` Tag gibt es genau einmal pro Modell, sollte es allerdings keine `participant` und `messageFlow` Tags geben, gibt es diesen Tag auch nicht. In Code-Snippet 3.2 sieht man den XML Aufbau des `collaboration` Tag.

```
1 <collaboration id="sid-00dc18e2-3a4d-47e3-a962-8709d56512e6">
2     <participant id="sid-AB3DD4EB-ABAD-40F9-85CD-E863EC6D3ADE"
          name="Prozessteilnehmer 1"
          processRef="sid-520165BC-25F3-49D3-A038-615BC07383AB">
3     </participant>
4     <messageFlow id="sid-25B20548-7EF9-473B-8ECB-609B0074A6DA"
          name="Nachricht 1"
          sourceRef="sid-501D155F-A056-45A0-8EDF-F7501891A569"
          targetRef="sid-9A311107-B44E-42B3-B8FE-A11202D337EC">
5     </messageFlow>
6 </collaboration>
```

Code-Snippet 3.2: XML-Aufbau der BPMN Collaboration

3.3.2 BPMN Pools

Die Pool Elemente im BPMN Standard sind dazu da, um Prozessteilnehmer abzubilden. In einem BPMN Modell erkennt man sie als Rahmen in dem sich ein Prozess befindet. Jeder Prozessteilnehmer besitzt einen eigenen Pool, dadurch kann man die einzelnen Prozessteile voneinander abgrenzen. In Abbildung 3.2 sieht man das Modell eines Pools.

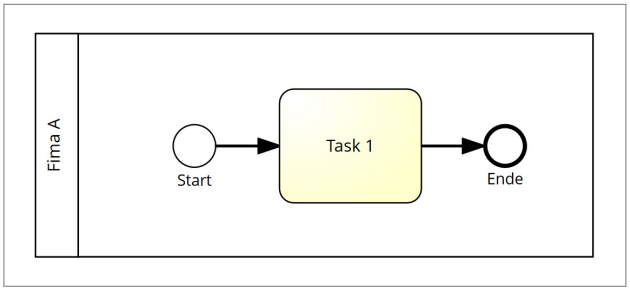

Abbildung 3.2: BPMN Pool

Innerhalb der BPMN XML Datei wird ein Pool als `participant` Tag gespeichert. Der Tag besitzt immer eine eindeutige ID, einen Namen und eine Prozessreferenz. Die Prozessreferenz beinhaltet die ID des `process` Tag, der sich innerhalb des Pools befindet. Den `participant` Tag gibt es genau so oft wie es Pools im Prozessmodel gibt. In Code-Snippet 3.3 sieht man den XML Aufbau des Pools aus Abbildung 3.2.

```
1 <participant id="sid-2D12E177-B1E9-4B2A-9C70-97F1899E9B56"
     name="Fima A"
     processRef="sid-68ADF24A-FF4A-455B-9443-F8E223E5BDF7">
2 </participant>
```

Code-Snippet 3.3: XML-Aufbau von BPMN Pools

3.3.3 Nachrichtenfluss

Die Nachrichtenfluss Elemente im BPMN Standard sind dazu da, um abzubilden welche Nachrichten wann und zwischen welchen Prozessteilnehmern verschickt werden. In einem BPMN Modell erkennt man sie als gerichtete, punktierte Linien. Dabei zeigt der Pfeil immer zum Empfänger und die Seite ohne Pfeil zum Sender. In Abbildung 3.3 sieht man einen einfachen Prozess mit zwei Nachrichtenfluss Elementen.

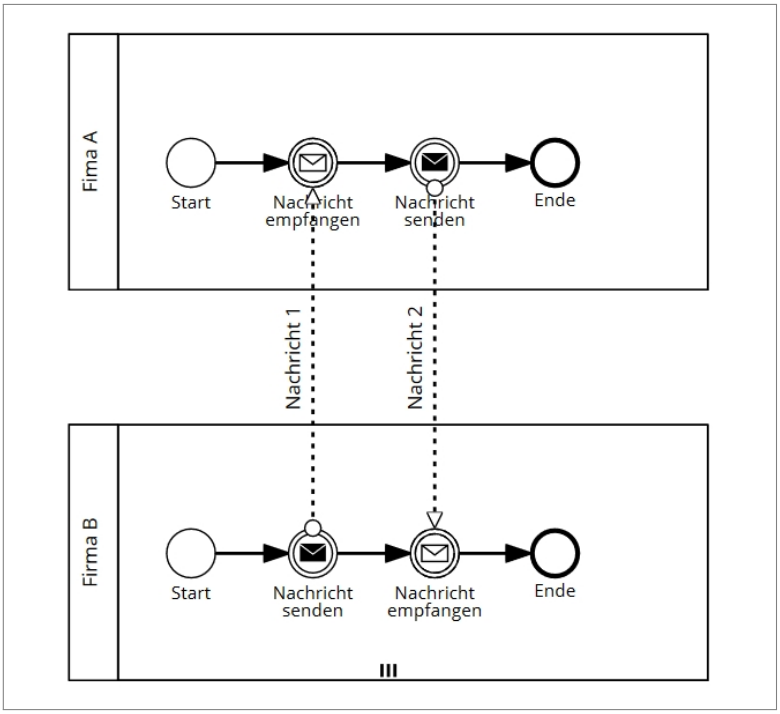

Abbildung 3.3: BPMN Nachrichtenfluss

Innerhalb der BPMN XML Datei wird ein Nachrichtenfluss als `messageFlow` Tag gespeichert. Der Tag besitzt immer eine eindeutige ID, einen Namen, eine Ursprungsreferenz und eine Zielreferenz. Die Ursprungsreferenz beinhaltet die ID des Tag von dem aus die Nachricht gesendet wurde. Die Zielreferenz beinhaltet die ID des Tag an den die Nachricht versendet wurde. Den `messageFlow` Tag gibt es genau so oft wie es Nachrichtenflüsse im Prozessmodell gibt. In Code-Snippet 3.4 sieht man den XML Aufbau der beiden Nachrichtenflüsse aus Abbildung 3.3.

```
1 <messageFlow id="sid-185477F3-F3B2-4238-A2BE-90AF300B7281"
    name="Nachricht 1"
    sourceRef="sid-C1CD3877-AA3A-4368-AF88-0B6E42F6578B"
    targetRef="sid-70EE080C-B776-4481-84A2-4CBBDCD23D2C">
2 </messageFlow>
3 <messageFlow id="sid-7EFDB475-9014-4984-B606-2CA784C4E998"
    name="Nachricht 2"
    sourceRef="sid-7DC1F423-8DCD-4497-A547-B71CAAD9B1F1"
    targetRef="sid-4DB2CF5C-9AF4-4760-96B7-6D8EE0158750">
4 </messageFlow>
```

Code-Snippet 3.4: XML-Aufbau von BPMN Pools

3.3.4 Process

Nach dem `collaboration` Tag gibt es die `process` Tags. Die Tags besitzen immer eine eindeutige ID, einen Namen, einen Prozesstyp und jeweils ein Attribut, ob er geöffnet oder ausführbar ist. Für diese Arbeit sind nur die ID und Namen Attribute relevant. Innerhalb eines `process` Tag befinden sich immer ein `laneSet` Tag und die Tags aus denen der Prozess besteht. Den `process` Tag gibt es so oft wie es Prozessteilnehmer im Modell gibt. Sollte es allerdings keinen `participant` Tag geben, gibt es trotzdem genau einen `process` Tag. In Code-Snippet 3.5 sieht man den XML Aufbau des `process` Tag.

```
<process id="sid-EEF6C328-C3D8-4086-B2A0-7CC24CC5E73F"
    isClosed="false" isExecutable="false" name="Firma B"
    processType="None">
    <laneSet id="sid-a5e47367-25dd-4827-acd4-c6ff5509beb5">
      <lane id="sid-77BB0357-C7A1-45CB-A757-B54A62841E61">
        <flowNodeRef>
           sid-2E26EB8D-0447-4E9A-8FBB-D0239299A6EB
        </flowNodeRef>
      </lane>
    </laneSet>
    <task completionQuantity="1"
       id="sid-2E26EB8D-0447-4E9A-8FBB-D0239299A6EB"
       isForCompensation="false" name="Task 2" startQuantity="1">
      <incoming>sid-C861AE94-138B-447A-8ADC-3F9B040AD30A</incoming>
      <outgoing>sid-7E6A602F-B204-41BD-9716-61EBC786ACB5</outgoing>
    </task>
</process>
```

Code-Snippet 3.5: XML-Aufbau des BPMN Process

3.3.5 Lane Set

Innerhalb des `process` Tag gibt es immer genau ein `laneSet` Tag. Der Tag besitzt immer eine eindeutige ID. Innerhalb des `laneSet` Tag befinden sich mindestens ein `lane` Tag. In Code-Snippet 3.6 sieht man den XML Aufbau des `laneSet` Tag.

```
<laneSet id="sid-a5e47367-25dd-4827-acd4-c6ff5509beb5">
  <lane id="sid-77BB0357-C7A1-45CB-A757-B54A62841E61">
    <flowNodeRef>
       sid-2E26EB8D-0447-4E9A-8FBB-D0239299A6EB
    </flowNodeRef>
  </lane>
</laneSet>
```

Code-Snippet 3.6: XML-Aufbau des BPMN Lane Sets

3.3.6 Lane

Innerhalb des `laneSet` Tag gibt es die `lane` Tags. Die Tags besitzen immer eine eindeutige ID. Innerhalb eines `lane` Tag befinden sich ein oder mehrere `flowNodeRef` Tags. Es gibt genau so viele `flowNodeRef` Tags wie es Prozesselemente in der entsprechenden Lane gibt. Innerhalb eines `flowNodeRef` Tag befindet sich die ID des entsprechenden Prozesselements. Aufgrund der Elementeinschränkung dieser Arbeit gibt es immer nur ein `lane` Tag innerhalb eines `laneSet` Tag. In Code-Snippet 3.7 sieht man den XML Aufbau eines `lane` Tag.

```
<lane id="sid-77BB0357-C7A1-45CB-A757-B54A62841E61">
    <flowNodeRef>
        sid-2E26EB8D-0447-4E9A-8FBB-D0239299A6EB
    </flowNodeRef>
    <flowNodeRef>
        sid-1E194D9E-3F56-4BCF-A874-46D0371A6069
    </flowNodeRef>
    <flowNodeRef>
        sid-5C507581-5DB3-40E7-AAB3-5CB05D1956EF
    </flowNodeRef>
</lane>
```

Code-Snippet 3.7: XML-Aufbau einer BPMN Lane

3.3.7 Startereignis

Die Startereignisse im BPMN Standard sind dazu da, um zu zeigen, wo ein Prozess beginnt. In einem BPMN Modell erkennt man sie als Kreis mit dünner Kontur. Es gibt viele verschiedene Arten von Startereignissen mit verschiedenen Symbolen innerhalb des Kreises, für diese Arbeit sind aber nur die Standard Elemente ohne Symbol relevant, weil nur diese übersetzt werden. Jeder Pool innerhalb eines Prozesses besitzt mindestens ein Startereignis. Aufgrund der Elementeinschränkung dieser Arbeit darf er nur genau ein Startereignis besitzen. In Abbildung 3.4 sieht man einen einfachen Prozess mit einem Startereignis.

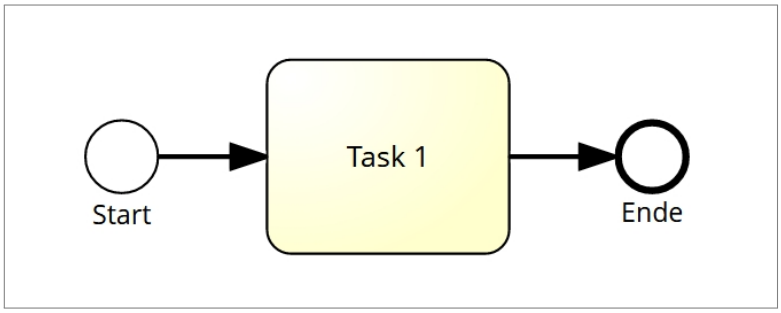

Abbildung 3.4: Einfacher BPMN Prozess - Startereignis und Task

Innerhalb der BPMN XML Datei wird ein Startereignis als `startEvent` Tag gespeichert. Der Tag besitzt immer eine eindeutige ID und einen Namen. Innerhalb eines `startEvent` Tag befindet sich ein `outgoing` Tag mit der ID des nachfolgenden Prozesselements. Da es das erste Element ist, gibt es keine `incoming` Tags. Den `startEvent` Tag gibt es genau einmal pro `process` Tag. In Code-Snippet 3.8 sieht man den XML Aufbau des Startereignisses aus Abbildung 3.4.

```
<startEvent id="sid-AE1C6B69-9AAF-4FAD-BE44-2B9D92A90A98"
    name="Start">
    <outgoing>sid-28868FCC-B920-4F92-9527-09EA79F4072B</outgoing>
</startEvent>
```

Code-Snippet 3.8: XML-Aufbau eines BPMN Startereignisses

3.3.8 Task

Die Task Elemente im BPMN Standard sind dazu da, um zu zeigen, welche Arbeitsschritte und Aufgaben in einem Prozess erledigt werden müssen. In einem BPMN Modell erkennt man sie als Rechteck mit abgerundeten Ecken. Es gibt mehrere Arten von Task Elementen, die sich alle durch ein eigenes Symbol in der linken oberen Ecke des Rechtecks unterscheiden. Aufgrund der Elementeinschränkung dieser Arbeit dürfen nur die Standard Task Elemente verwendet werden. In einem Pool innerhalb eines Prozessesmodells können sich beliebig viele Task Elemente befinden, es können aber auch gar keine vorhanden sein. In Abbildung 3.4 sieht man einen einfachen Prozess mit einem Task Element.

Innerhalb der BPMN XML Datei wird ein Task als `task` Tag gespeichert. Der Tag besitzt immer eine eindeutige ID, einen Namen, ein `startQuantity`, ein `completionQuantity` und ein `isForCompensation` Attribut. Innerhalb eines `task` Tag befinden sich ein oder mehrere `incoming` Tags mit den IDs der vorhergehenden Prozesselemente und ein `outgoing` Tag mit der ID des nachfolgenden Prozesselements. Den `task` Tag gibt es beliebig oft pro `process` Tag. In Code-Snippet 3.9 sieht man den XML Aufbau des Task Elements aus Abbildung 3.4.

```
<task completionQuantity="1"
    id="sid-8E81EC43-6066-4795-96F3-ED96BBA292BD"
    isForCompensation="false" name="Task 1&#xA;Normal"
    startQuantity="1">
      <incoming>sid-622A9BAE-2265-4915-A8B7-2D72667142DA</incoming>
      <outgoing>sid-CD3077EA-953D-4535-809C-D6387AD1E5F6</outgoing>
</task>
```

Code-Snippet 3.9: XML-Aufbau eines BPMN Task Elements

3.3.9 Auslösende Zwischenereignisse

Die auslösenden Zwischenereignisse im BPMN Standard sind dazu da, um Informationen von einem Teil des Prozessmodells zu einem anderen zu schicken. In einem BPMN Modell erkennt man sie als Kreis mit doppelter Kontur. Innerhalb des Kreises befindet sich ein Symbol, das schwarz ausgemalt wurde. Es gibt viele verschiedene Arten von auslösenden Zwischenereignis Elementen mit verschiedenen Symbolen, für diese Arbeit sind aber nur die auslösenden Nachrichten-Zwischenereignis Elemente relevant, welche mit einem Brief-Symbol gekennzeichnet sind. In einem Pool innerhalb eines Prozessmodells können sich beliebig viele auslösende Zwischenereignisse befinden, es können aber auch gar keine vorhanden sein. In Abbildung 3.5 sieht man einen Prozess mit einem auslösenden Zwischenereignis.

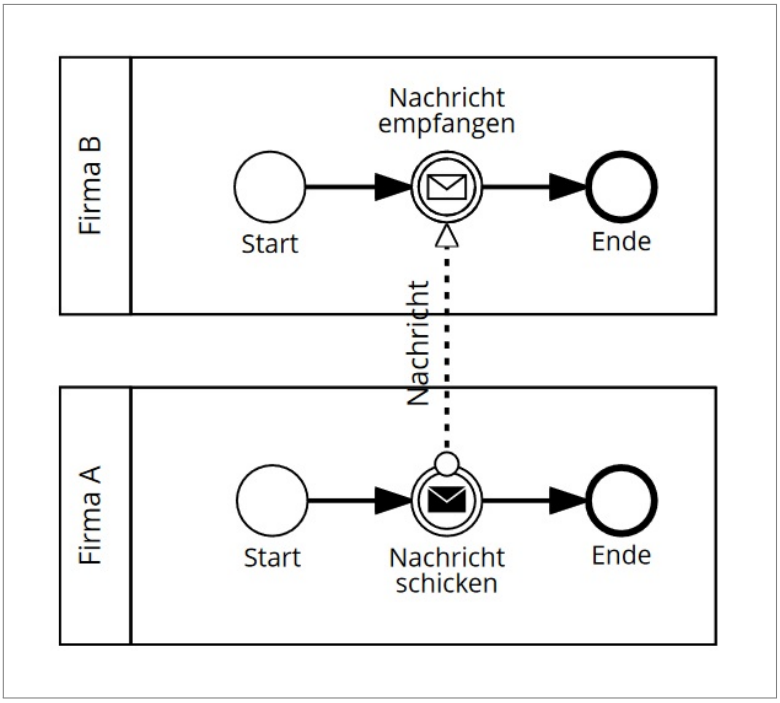

Abbildung 3.5: BPMN Prozess mit Throw und Catch Events

Innerhalb der BPMN XML Datei wird ein auslösendes Zwischenereignis als `intermediateThrowEvent` Tag gespeichert. Der Tag besitzt immer eine eindeutige ID und einen Namen. Innerhalb eines `intermediateThrowEvent` Tag befinden sich ein oder mehrere `incoming` Tags mit den IDs der vorhergehenden Prozesselemente und ein `outgoing` Tag mit der ID des nachfolgenden Prozesselements. Außerdem befindet sich darin noch der `messageEventDefinition` Tag mit einer ID, welcher kennzeichnet, dass es sich hierbei um ein auslösendes Nachrichten-Zwischenereignis handelt. Den `intermediateThrowEvent` Tag gibt es beliebig oft pro `process` Tag. In Code-Snippet 3.10 sieht man den XML Aufbau des auslösenden Zwischenereignis Elements aus Abbildung 3.5.

```
<intermediateThrowEvent
    id="sid-501D155F-A056-45A0-8EDF-F7501891A569"
    name="Nachricht&#xA;schicken">
  <incoming>sid-89D384DB-67F1-4AB5-A964-A8E341449E3E</incoming>
  <outgoing>sid-4031A25A-4664-405C-A8CB-0F819E5327F8</outgoing>
  <messageEventDefinition
     id="sid-eaaf9a7c-a786-4f83-bbd2-a77df8547315"/>
</intermediateThrowEvent>
```

Code-Snippet 3.10: XML-Aufbau eines BPMN auslösenden Zwischenereignis Elements

3.3.10 Eintretende Zwischenereignisse

Die eintretenden Zwischenereignisse im BPMN Standard sind dazu da, um Informationen aufzufangen und darauf zu reagieren, die entweder von einem anderen Teil des Prozessmodells oder von der Umwelt verschickt wurden. In einem BPMN Modell erkennt man sie als Kreis mit doppelter Kontur. Innerhalb des Kreises befindet sich ein Symbol, das nicht ausgemalt wurde. Es gibt viele verschiedene Arten von eintretenden Zwischenereignis Elementen mit verschiedenen Symbolen, für diese Arbeit sind aber nur die eintretenden Nachrichten- und die eintretenden Zeit-Zwischenereignis Elemente relevant, welche mit einem Brief- und einem Uhr-Symbol gekennzeichnet sind. In einem Pool innerhalb eines Prozessmodells können sich beliebig viele eintretende Zwischenereignisse befinden, es können aber auch gar keine vorhanden sein. In Abbildung 3.5 sieht man einen Prozess mit beiden eintretenden Zwischenereignissen.

Innerhalb der BPMN XML Datei wird ein eintretendes Zwischenereignis als `intermediateCatchEvent` Tag gespeichert. Der Tag besitzt immer eine eindeutige ID und einen Namen. Innerhalb eines `intermediateCatchEvent` Tag befinden sich ein oder mehrere `incoming` Tags mit den IDs der vorhergehenden Prozesselemente und ein `outgoing` Tag mit der ID des nachfolgenden Prozesselements. Außerdem befindet sich darin noch entweder der `messageEventDefinition` oder der `timerEventDefinition` Tag mit jeweils einer ID, welcher kennzeichnet, ob es sich hierbei um ein eintretendes Nachrichten- oder ein eintretendes Zeit-Zwischenereignis handelt.

Innerhalb des `timerEventDefinition` Tag befindet sich noch der `timeDuration` Tag. In diesem befindet sich die Zeitspanne, nach der das Zeit-Zwischenereignis ausgelöst wird. Damit der Prozess ordnungsgemäß in ein PASS Element übersetzt werden kann muss diese Zeitdauer im ISO 8601 Format angegeben werden. Den `intermediateCatchEvent` Tag gibt es beliebig oft pro `process` Tag. In Code-Snippet 3.11 sieht man den XML Aufbau der eintretenden Zwischenereignis Elemente aus Abbildung 3.5.

```
<intermediateCatchEvent
    id="sid-9A311107-B44E-42B3-B8FE-A11202D337EC"
    name="Nachricht empfangen">
  <incoming>sid-9505002D-978F-40F5-9E0A-544E1AEFE2E2</incoming>
  <outgoing>sid-24329F52-0B8F-4667-886E-D858C63D1750</outgoing>
  <messageEventDefinition
      id="sid-d9841057-61f0-4cd1-900c-6defa5593f31"/>
</intermediateCatchEvent>

<intermediateCatchEvent
    id="sid-607F93D2-0B71-4030-AC6C-03928A428631"
    name="8 Stunden vergehen">
    <incoming>sid-C6287ABE-DD47-4229-83C9-6E2AC9F807FC</incoming>
    <outgoing>sid-FD95EDB4-BFEE-4058-9E7D-CA7886694FC3</outgoing>
    <timerEventDefinition
        id="sid-315f56eb-ab6e-47b9-ad86-43e6812b787a">
        <timeDuration id="sid-becdb391-b35e-46a2-904b-0d1ed555a0ea"
            xsi:type="tFormalExpression">
            P0DT8H0M0S
        </timeDuration>
    </timerEventDefinition>
</intermediateCatchEvent>
```

Code-Snippet 3.11: XML-Aufbau von BPMN eintretenden Zwischenereignis Elementen

3.3.11 Exklusives Gateway

Die exklusiven Gateways im BPMN Standard sind dazu da, um mehrere alternative Pfade in einem Prozessablauf zu ermöglichen. Das Gateway kann entweder den Prozess aufsplitten oder die Pfade wieder zusammenführen. Es kann immer nur einer der entstehenden Pfade gewählt werden. Es gibt eine Entscheidung und je nachdem welche Wahl getroffen wird, wird ein Pfad ausgewählt. In einem BPMN Modell erkennt man die exklusiven Gateways als Raute mit einem **X** darin. In einem Pool innerhalb eines Prozessmodells können sich beliebig viele exklusive Gateways befinden, es können aber auch gar keine vorhanden sein. In Abbildung 3.6 sieht man einen Prozess mit einem exklusiven Gateway.

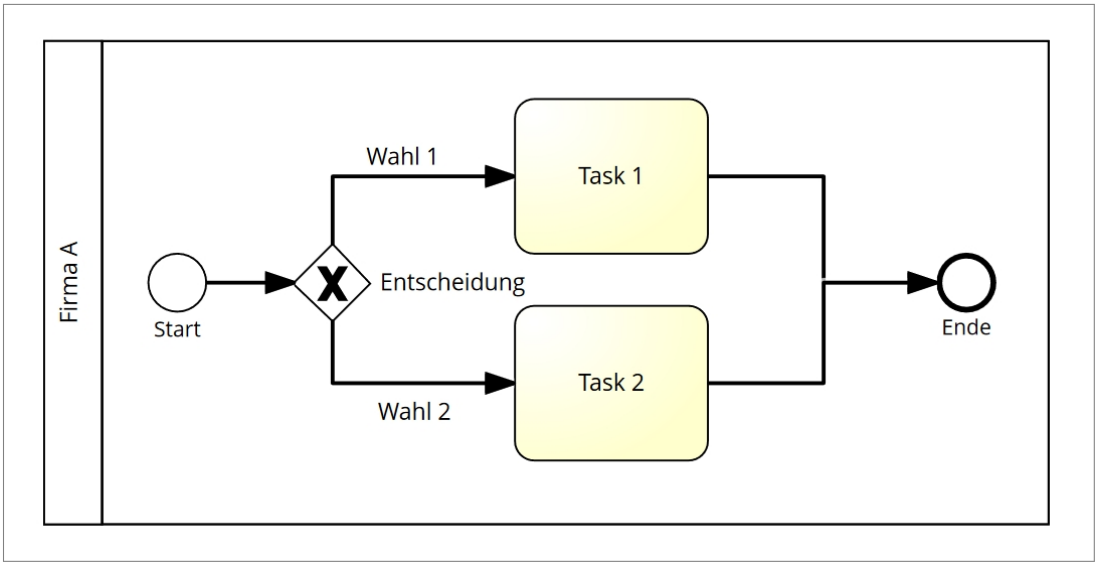

Abbildung 3.6: BPMN Prozess mit einem exklusiven Gateway

Innerhalb der BPMN XML Datei wird ein exklusives Gateway als `exclusiveGateway` Tag gespeichert. Der Tag besitzt immer eine eindeutige ID, einen Namen und eine Gateway Richtung, die angibt ob das Gateway den Prozess in alternative Pfade aufteilt oder diese wieder zusammenführt. Innerhalb eines `exclusiveGateway` Tag befinden sich ein oder mehrere `incoming` Tags mit den IDs der vorhergehenden Prozesselemente und ein oder mehrere `outgoing` Tags mit den IDs der nachfolgenden Prozesselemente. Den `exclusiveGateway` Tag gibt es beliebig oft pro `process` Tag. In Code-Snippet 3.12 sieht man den XML Aufbau des exklusiven Gateways aus Abbildung 3.6.

```
1 <exclusiveGateway gatewayDirection="Diverging"
      id="sid-4B6AE0F9-465B-430B-BCF1-4572F59DF730"
      name="Entscheidung">
2   <incoming>sid-40F914BE-FB2D-4BC9-B6FF-B38D174D2F3F</incoming>
3   <outgoing>sid-CCD4846B-1F98-48EF-B1AF-BA25EBF21A5B</outgoing>
4   <outgoing>sid-A0ECE9FA-5C24-41AD-8B9D-3F931FEBCA6D</outgoing>
5 </exclusiveGateway>
```

Code-Snippet 3.12: XML-Aufbau eines BPMN exklusiven Gateways

3.3.12 Ereignisbasiertes Gateway

Die ereignisbasierten Gateways im BPMN Standard sind dazu da, um mehrere alternative Pfade in einem Prozessablauf zu ermöglichen. Das Gateway kann entweder den Prozess aufsplitten oder die Pfade wieder zusammenführen. Es kann immer nur einer der entstehenden Pfade gewählt werden. Bei jedem der ausgehenden Pfade gibt es ein eintretendes Zwischenereignis. Je nachdem welches Ereignis zuerst eintritt, wird der jeweilige Pfad gewählt. In einem BPMN Modell erkennt man die ereignisbasierten Gateways als Raute mit einem Fünfeck in einem Kreis darin. In einem Pool innerhalb eines Prozessmodells können sich beliebig viele ereignisbasierte Gateways befinden, es können aber auch gar keine vorhanden sein. In Abbildung 3.7 sieht man einen Prozess mit einem eventbasierten Gateway.

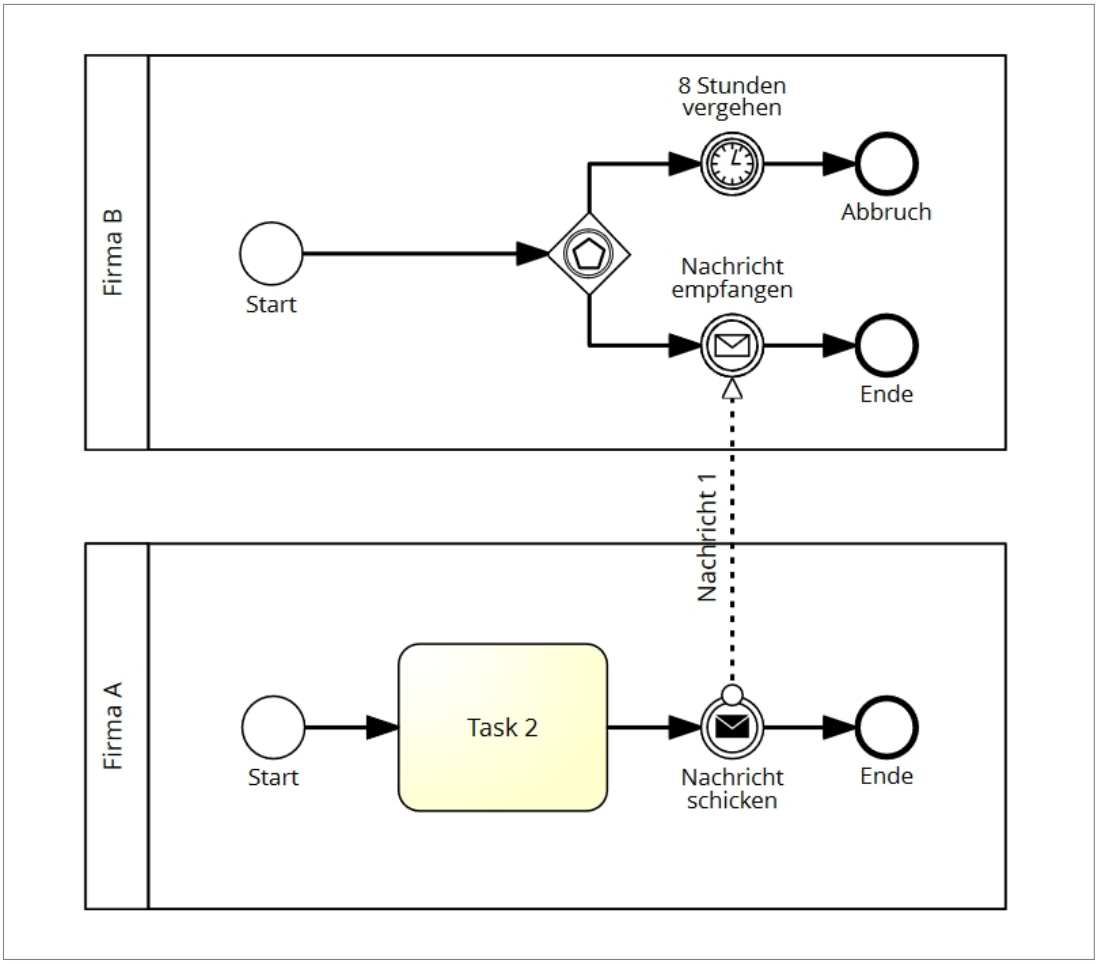

Abbildung 3.7: BPMN Prozess mit einem eventbasiertem Gateway

Innerhalb der BPMN XML Datei wird ein eventbasiertes Gateway als `eventBasedGateway` Tag gespeichert. Der Tag besitzt immer eine eindeutige ID, einen Namen, eine Gateway Richtung, die angibt ob das Gateway den Prozess in alternative Pfade aufteilt oder diese wieder zusammenführt, einen Ereignis-Gateway-Typ, welcher aufgrund der Elementeinschränkung immer `Exclusive` ist und ein `instantiate` Attribut. Innerhalb eines `eventBasedGateway` Tag befinden sich ein oder mehrere `incoming` Tags mit

den IDs der vorhergehenden Prozesselemente und ein oder mehrere `outgoing` Tags mit den IDs der nachfolgenden Prozesselemente. Den `eventBasedGateway` Tag gibt es beliebig oft pro `process` Tag. In Code-Snippet 3.13 sieht man den XML Aufbau des ereignisbasierten Gateways aus Abbildung 3.7.

```
<eventBasedGateway eventGatewayType="Exclusive" name=""
    gatewayDirection="Diverging" instantiate="false"
    id="sid-28611AF9-8637-4F3F-8364-EA6233CB5D4E">
  <incoming>sid-9B18DD6E-F0D8-4A74-82E2-01BE13FCDBA5</incoming>
  <outgoing>sid-9505002D-978F-40F5-9E0A-544E1AEFE2E2</outgoing>
  <outgoing>sid-C6287ABE-DD47-4229-83C9-6E2AC9F807FC</outgoing>
</eventBasedGateway>
```

Code-Snippet 3.13: XML-Aufbau eines BPMN eventbasierten Gateways

3.3.13 Endereignis

Die Endereignis Elemente im BPMN Standard sind dazu da, um zu zeigen wo ein Prozess endet. In einem BPMN Modell erkennt man sie als Kreis mit fett gedruckter Kontur. Es gibt viele verschiedene Arten von Endereignissen mit verschiedenen Symbolen innerhalb des Kreises, für diese Arbeit sind aber nur die Standard Elemente ohne Symbol relevant. Jeder Pool innerhalb eines Prozesses besitzt mindestens ein Endereignis. Es können aber auch mehrere Endereignisse vorhanden sein, sodass der Prozess alternative Enden besitzt. In Abbildung 3.8 sieht man einen einfachen Prozess mit einem Endereignis.

Innerhalb der BPMN XML Datei wird ein Endereignis als `endEvent` Tag gespeichert. Der Tag besitzt immer eine eindeutige ID und einen Namen. Innerhalb eines `endEvent` Tag befinden sich ein oder mehrere `incoming` Tags mit den IDs der vorhergehenden Prozesselemente. Den `endEvent` Tag gibt es mindestens einmal pro `process` Tag.

In Code-Snippet 3.14 sieht man den XML Aufbau des Endereignisses aus Abbildung 3.8.

```
<endEvent id="sid-4DDE8F65-A3E8-4E2B-8D79-2D22449D5B2E" name="Ende">
     <incoming>sid-1A37A455-7539-47E2-8ADC-9BE109FEA42D</incoming>
</endEvent>
```

Code-Snippet 3.14: XML-Aufbau eines BPMN Endereignisses

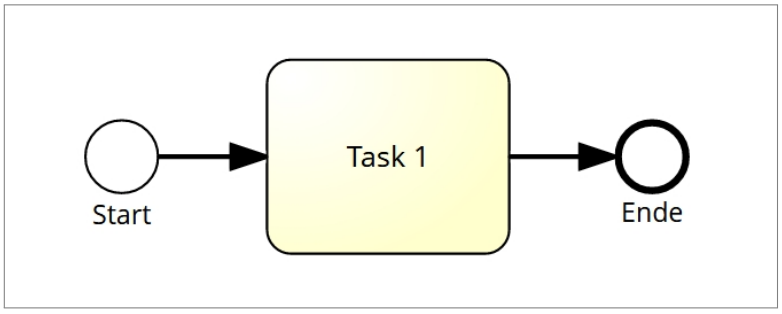

Abbildung 3.8: Einfacher BPMN Prozess - Endereignis und Sequenzfluss

3.3.14 Sequenzfluss

Die Sequenzfluss Elemente im BPMN Standard sind dazu da, um zu zeigen wie die einzelnen Elemente miteinander verbunden sind und welche Wege ein Prozessteilnehmer nehmen kann. In einem BPMN Modell erkennt man sie als durchgezogene Linie mit einem Pfeil an einem Ende. Jeder Pool innerhalb eines Prozesses besitzt genau so viele Sequenzflüsse, wie es Übergänge zwischen den Prozesselementen gibt. In Abbildung 3.8 sieht man einen einfachen Prozess mit zwei Sequenzflüssen.

Innerhalb der BPMN XML Datei wird ein Sequenzfluss als `sequenceFlow` Tag gespeichert. Der Tag besitzt immer eine eindeutige ID, einen Namen, eine Ursprungsreferenz und eine Zielreferenz. Die letzten beiden Attribute geben an, wo der Sequenzfluss beginnt und wohin er zeigt. Den `sequenceFlow` Tag gibt es beliebig oft pro `process` Tag. Code-Snippet 3.15 zeigt den XML Aufbau der beiden Sequenzflüsse aus Abbildung 3.8.

```
1 <sequenceFlow id="sid-28868FCC-B920-4F92-9527-09EA79F4072B" name=""
     sourceRef="sid-AE1C6B69-9AAF-4FAD-BE44-2B9D92A90A98"
     targetRef="sid-0F6E0045-0DED-46CA-A1CD-8DBF54672B23">
2 </sequenceFlow>
3 <sequenceFlow id="sid-1A37A455-7539-47E2-8ADC-9BE109FEA42D" name=""
     sourceRef="sid-0F6E0045-0DED-46CA-A1CD-8DBF54672B23"
     targetRef="sid-4DDE8F65-A3E8-4E2B-8D79-2D22449D5B2E">
4 </sequenceFlow>
```

Code-Snippet 3.15: XML-Aufbau von BPMN Sequenzfluss Elementen

3.4 Modellierungsleitfaden

Zusätzlich zu der Elementeinschränkung wird ein Modellierungsleitfaden benötigt, der regelt, wie modelliert werden muss, damit das Modell am Ende in ein PASS Modell umgewandelt werden kann.

3.4.1 BPMN Pools und Lanes

Beim Erstellen und Benennen der Pools sollte man darauf achten, dass die Pools später in Subjekte umgewandelt werden. Es treten keine Fehler auf, wenn man die Pools nicht richtig benennt, aber um nach der Umwandlung ein sauberes PASS Modell zu erhalten, sollte man dies beim modellieren beachten.

Es werden nur Pools ohne Lanes richtig erkannt. Sollte beim Modellieren das Bedürfnis auftreten, dass man eine weitere Lane benötigt, muss stattdessen ein weiteres Pool hinzugefügt werden. Außerdem dürfen keine zugeklappten Pools modelliert werden. Abbildung 3.9 zeigt zwei richtig modellierte Pools.

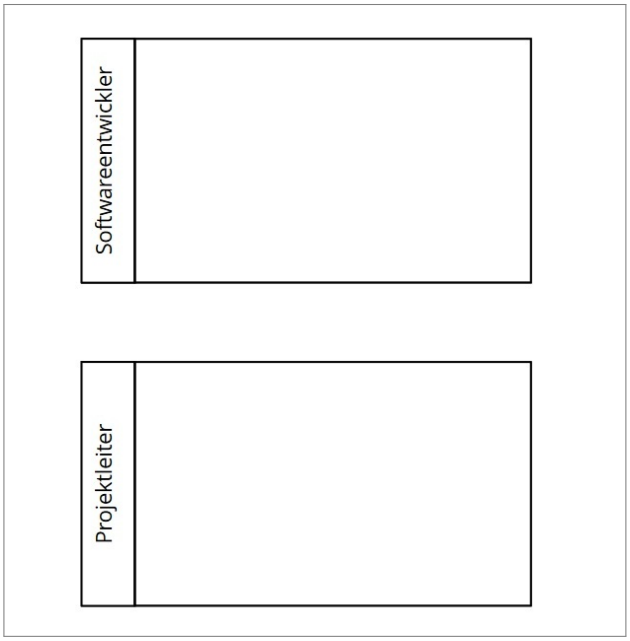

Abbildung 3.9: Richtig modellierte Pools

3.4.2 Nachrichtenfluss und Nachrichten-Zwischenereignisse

Der Nachrichtenfluss ist im PASS Standard eines der Kernelemente. Daher sollte beim Erstellen des Nachrichtenflusses auch besonders sorgsam umgegangen werden.

Ein Nachrichtenfluss muss immer zwischen einem auslösenden Nachrichten-Zwischenereignis und einem eintretenden Nachrichten-Zwischenereignis modelliert werden. Der Nachrichtenaustausch zwischen anderen Elementen ist nicht erlaubt. Diese Ereignisse müssen sich auch in verschiedenen Pools befinden. Außerdem sollten die Nachricht

und die Zwischenereignisse auch immer sinnvoll benannt werden. Abbildung 3.10 zeigt einen richtig modellierten Nachrichtenfluss zwischen zwei Pools.

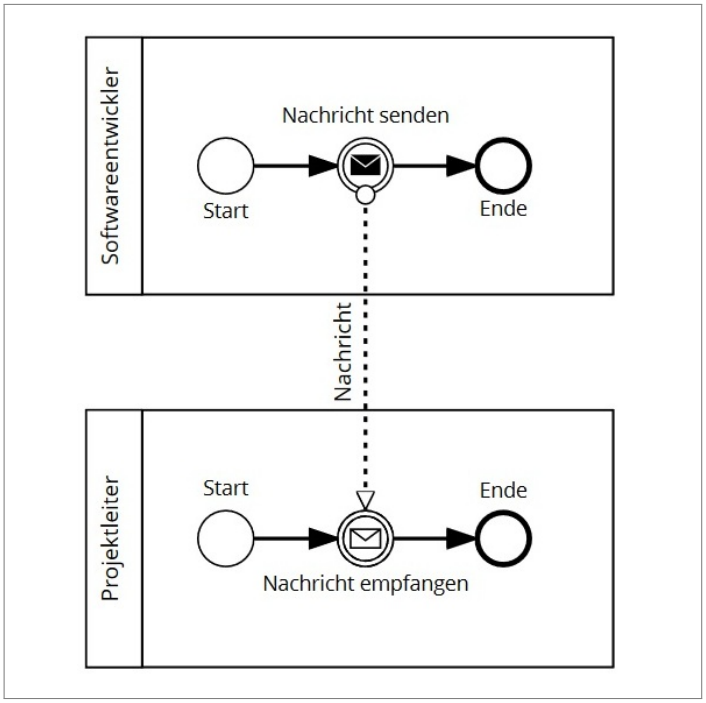

Abbildung 3.10: Richtig modellierter Nachrichtenfluss

3.4.3 Zeit-Zwischenereignis

Neben den beiden Nachrichten-Zwischenereignissen gibt es noch das eintretende Zeit-Zwischenereignis, das erlaubt ist. Alle anderen Zwischenereignisse werden nicht unterstützt.

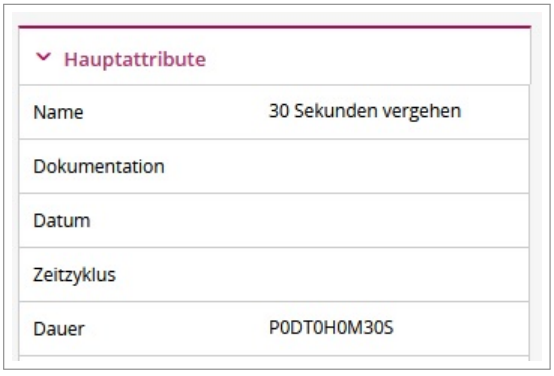

Abbildung 3.11: Attribute eines richtig modellierten Zeit-Zwischenereignisses

Beim Modellieren vom eintretenden Zeit-Zwischenereignis muss besonders darauf geachtet werden, dass das Dauer Attribut richtig gesetzt wird. Diese Zeitdauer muss im ISO 8601 Format angegeben werden. Dieses Format sieht folgendermaßen aus:

PODTOHOMOS. Die Zahlen stehen, von links nach rechts, für die Tage, Stunden, Minuten und Sekunden. Außerdem sollte dem Zwischenereignis eine sinnvolle Benennung gegeben werden. Abbildung 3.11 zeigt die Attribute eines richtig modellierten Zeit-Zwischenereignisses.

3.4.4 Start- und Endereignis

Bei den Start- und Endereignissen werden nur die normalen Ereignisse ohne weitere Informationen unterstützt. Beim umwandeln in ein PASS Modell werden sie in Do-States verwandelt. Daher sollten sie, wenn möglich, sinnvoll benannt werden. Abbildung 3.12 zeigt je ein richtig modelliertes Start- und Endereignis.

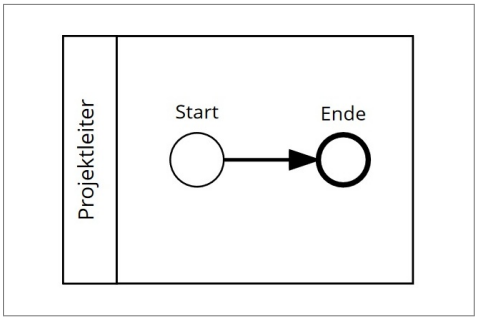

Abbildung 3.12: Richtig modelliertes Start- und Endereignis

3.4.5 Task

Bei den Tasks werden nur die normalen Tasks unterstützt und keine weiteren Typen. Auch die Schleifentypen werden nicht unterstützt. Bei der Umwandlung von BPMN zu PASS werden Tasks in States umgewandelt und sie sollten auch dementsprechend modelliert werden. Es treten keine Fehler auf, wenn man die Tasks nicht richtig benennt, aber um nach der Umwandlung ein sauberes PASS Modell zu erhalten, sollte man dies beim Modellieren beachten. Abbildung 3.13 zeigt einen Prozess mit richtig modellierten Tasks.

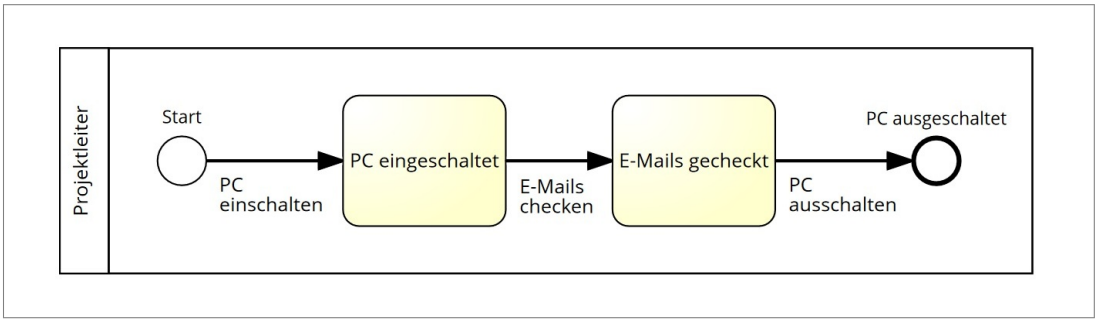

Abbildung 3.13: Richtig modellierte Tasks

3.4.6 Sequenzfluss

Bei der Umwandlung von BPMN zu PASS werden Sequenzflüsse in Transitions umgewandelt und sie sollten auch dementsprechend modelliert werden. Vor allem sollten alle Sequenzflüsse einen Namen erhalten. Es treten keine Fehler auf, wenn die Sequenzflüsse nicht richtig oder gar nicht benannt werden, aber um nach der Umwandlung ein sauberes PASS Modell zu erhalten, sollte dies beim Modellieren beachtet werden. Abbildung 3.13 zeigt einen Prozess mit richtig modellierten Sequenzflüssen.

3.4.7 Exklusives Gateway

Bei der Umwandlung von BPMN zu PASS werden exklusive Gateways in Do-States umgewandelt. Dabei muss beachtet werden, dass nur das Gateway modelliert werden darf, welches den Prozess aufsplittet. Ein zweites Gateway, welches den Prozess wieder zusammenführt, darf nicht modelliert werden und würde einen Fehler verursachen. Stattdessen sollten alle Sequenzflüsse, die zusammengeführt werden sollen, zum selben, nachfolgenden BPMN Element führen. Abbildung 3.14 zeigt einen Prozess mit einem richtig modellierten exklusiven Gateway.

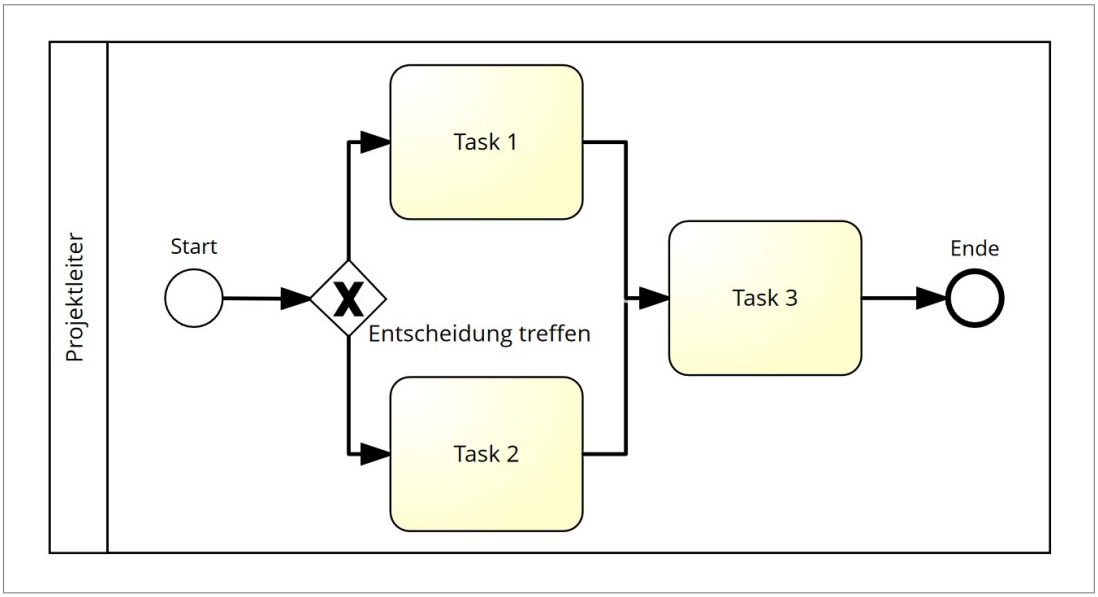

Abbildung 3.14: Richtig modelliertes exklusives Gateway

3.4.8 Ereignisbasiertes Gateway

Bei der Umwandlung von BPMN zu PASS werden ereignisbasierte Gateways in Receive-States umgewandelt. Dabei muss beachtet werden, dass nur das Gateway modelliert werden darf, welches den Prozess aufsplittet. Ein zweites Gateway, welches den Prozess wieder zusammenführt, darf nicht modelliert werden und würde einen Fehler verursachen. Stattdessen sollten alle Sequenzflüsse, die zusammengeführt werden sollen, zum selben, nachfolgenden BPMN Element führen.

Nach einem eventbasierten Gateway können sich ein oder mehrere Nachrichten-Zwischenereignisse befinden und bei Bedarf auch ein Zeit-Zwischenereignis. Außerdem muss beachtet werden, dass ein ereignisbasiertes Gateway auch immer exklusiv ist. Das bedeutet, dass nur einer der nachfolgenden Pfade beschritten werden kann. Dies ist immer der Pfad, dessen Ereignis als erstes eintritt. Abbildung 3.15 zeigt einen Prozess mit einem richtig modellierten ereignisbasierten Gateway.

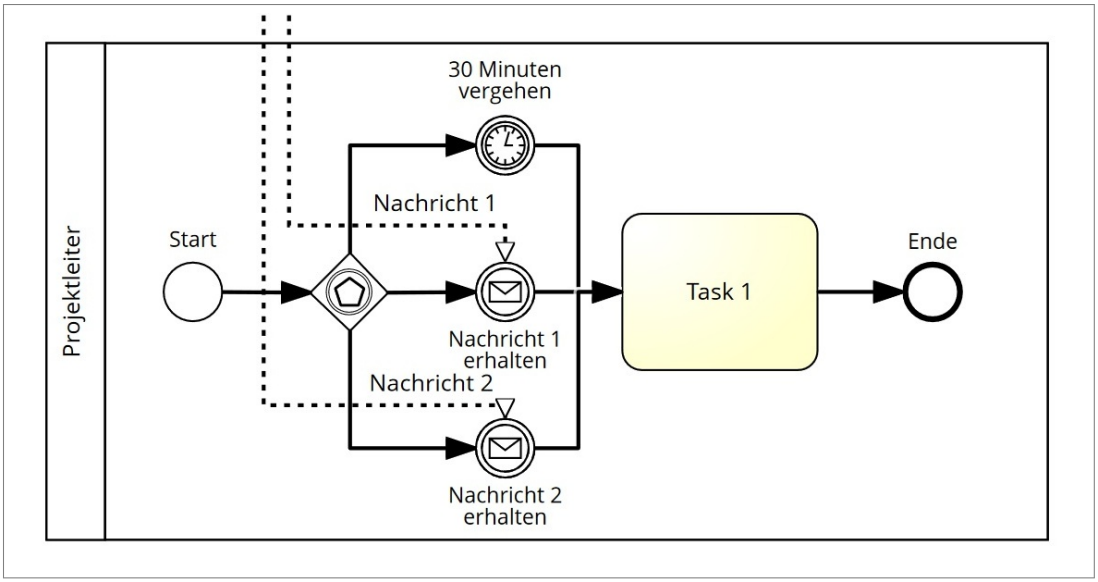

Abbildung 3.15: Richtig modelliertes ereignisbasiertes Gateway

4 Umwandlung von BPMN in PASS

In diesem Kapitel wird beschrieben, wie BPMN Prozesse in PASS Prozesse umgewandelt werden. Dies geschieht in zwei Schritten.

Zuerst wird das BPMN Modell in ein SimpleBPMN Modell umgewandelt. Hierbei werden nicht benötigte Informationen und Elemente herausgefiltert. Außerdem werden teilweise Informationen zu den Elementen hinzugefügt, um die spätere Umwandlung in ein PASS Modell zu vereinfachen.

Anschließend werden die SimpleBPMN Elemente in PASS Elemente umgewandelt. Dabei werden die Namen und IDs der BPMN Elemente benutzt, um das `ComponentLabel` und die `ComponentId` der PASS Elemente zu generieren. Dies wird zum einen deshalb gemacht, um den Prozess möglichst unverändert zu lassen und zum andern, um den Prozess mit den selben IDs und Namen wieder zurückübersetzen zu können.

4.1 Umwandlung von BPMN in SimpleBPMN

In diesem Kapitel wird gezeigt wie die Umwandlung von BPMN in SimpleBPMN in C# funktioniert. Hierfür kommen im Code immer wieder die Variablen `newProcess` und `oldProcess` vor. `newProcess` steht hierbei für das SimpleBPMN Modell und `oldProcess` steht für das BPMN Modell.

4.1.1 Collaboration

Für den `collaboration` Tag gibt es kein entsprechendes Element in PASS. Daher wird es auch nicht umgewandelt.

4.1.2 Participants

Zuerst wird eine Liste aller `Participants` erstellt. Danach wird jedes alte `Participant` in ein neues umgewandelt. Dabei wird nur die ID, der Name und die `ProcessRef` beibehalten. Diese Umwandlung wird in Code-Snippet 4.1 gezeigt.

```
1  // Liste der Participants erstellen
2  newProcess.participants = new List<SimpleBPMN.Participant>();
3  // Liste Befüllen
4  foreach (BPMN.Participant pOld in
       oldProcess.Collaboration.Participant)
5  {
6      SimpleBPMN.Participant pNew = new SimpleBPMN.Participant();
7      pNew.name = pOld.Name;
8      pNew.processRef = pOld.ProcessRef;
9      pNew.id = pOld.Id;
10     newProcess.participants.Add(pNew);
11 }
```

Code-Snippet 4.1: C# Umwandlung Participant

4.1.3 Nachrichtenfluss

Zuerst wird eine Liste aller `MessageFlows` erstellt. Danach wird jeder alte `MessageFlow` in einen neuen umgewandelt. Dabei wird die ID, der Name, die `SourceRef` und die `TargetRef` beibehalten. Zusätzlich wird noch, sowohl für die `SourceRef`, als auch für die `TargetRef`, der entsprechende `Participant` mitgespeichert. Diese Umwandlung wird in Code-Snippet 4.2 gezeigt.

```
1  // Liste der messageFlows erstellen
2  newProcess.messageFlows = new List<SimpleBPMN.MessageFlow>();
3  foreach (BPMN.MessageFlow mfOld in
       oldProcess.Collaboration.MessageFlow)
4  {
5      SimpleBPMN.MessageFlow mfNew = new SimpleBPMN.MessageFlow();
6      mfNew.id = mfOld.Id;
7      mfNew.name = mfOld.Name;
8      mfNew.sourceRef = mfOld.SourceRef;
9      mfNew.sourceParticipant =
           getParticipantFromElementId(mfOld.SourceRef, oldProcess,
           newProcess.participants);
10     mfNew.targetRef = mfOld.TargetRef;
11     mfNew.targetParticipant =
           getParticipantFromElementId(mfOld.TargetRef, oldProcess,
           newProcess.participants);
12     newProcess.messageFlows.Add(mfNew);
13 }
```

Code-Snippet 4.2: C# Umwandlung MessageFlow

4.1.4 Process

Zuerst wird eine Liste aller `Processes` erstellt. Danach wird jeder alte `Process` in einen neuen umgewandelt. Dabei wird die ID und der Name beibehalten. Zusätzlich wird noch der entsprechende `Participant` mitgespeichert. Diese Umwandlung wird in Code-Snippet 4.3 gezeigt.

```
1  // Liste der Processes erstellen
2  newProcess.processes = new List<SimpleBPMN.Process>();
3  // Liste Befüllen
4  foreach (BPMN.Process pOld in oldProcess.Process)
5  {
6      SimpleBPMN.Process pNew = new SimpleBPMN.Process();
7      pNew.id = pOld.Id;
8      pNew.name = pOld.Name;
9      pNew.participant = getParticipantFromProcess(pOld.Id,
           newProcess.participants);
10     newProcess.processes.Add(pNew);
11 }
```

Code-Snippet 4.3: C# Umwandlung Process

4.1.5 Lane Set

Für den `laneSet` Tag gibt es kein entsprechendes Element in PASS. Daher wird es auch nicht umgewandelt.

4.1.6 Lane

Für den `lane` Tag gibt es kein entsprechendes Element in PASS. Daher wird es auch nicht umgewandelt.

4.1.7 Startereignis

Als Nächstes wird das alte `StartEvent` in eine neues umgewandelt. Dabei wird nur die ID, der Name und die `outgoing` ID beibehalten. Diese Umwandlung wird in Code-Snippet 4.4 gezeigt.

```
1  //StartEvent erstellen
2  SimpleBPMN.StartEvent startNew = new SimpleBPMN.StartEvent();
3  startNew.id = pOld.StartEvent.Id;
4  startNew.name = pOld.StartEvent.Name;
5  startNew.outgoing = pOld.StartEvent.Outgoing;
6
7  pNew.startEvent = startNew;
```

Code-Snippet 4.4: C# Umwandlung StartEvent

4.1.8 Task

Zuerst wird eine Liste aller `Tasks` erstellt. Danach wird jeder alte `Task` in einen neuen umgewandelt. Dabei wird nur die ID, der Name, die `incoming` IDs und die `outgoing` ID beibehalten. Diese Umwandlung wird in Code-Snippet 4.5 gezeigt.

```
1  // Liste der tasks erstellen
2  pNew.tasks = new List<SimpleBPMN.Task>();
3  // Liste Befüllen
4  foreach (BPMN.Task tOld in pOld.Task)
5  {
6      SimpleBPMN.Task tNew = new SimpleBPMN.Task();
7
8      tNew.id = tOld.Id;
9      tNew.name = tOld.Name;
10     tNew.incomings = tOld.Incoming;
11     tNew.outgoings = new List<string>();
12     tNew.outgoings.Add(tOld.Outgoing);
13
14     pNew.tasks.Add(tNew);
15 }
```

Code-Snippet 4.5: C# Umwandlung Task

4.1.9 Auslösende Zwischenereignisse

Zuerst wird eine Liste aller `IntermediateThrowEvents` erstellt. Danach wird jedes alte `IntermediateThrowEvent` in ein neues umgewandelt. Dabei wird die ID, der Name, die `incoming` IDs und die `outgoing` ID beibehalten. Zusätzlich wird noch der entsprechende `MessageFlow` mitgespeichert. Diese Umwandlung wird in Code-Snippet 4.6 gezeigt.

```
1  // Liste der intermediateThrowEvents erstellen
2  pNew.intermediateThrowEvents = new
       List<SimpleBPMN.IntermediateThrowEvent>();
3
4  // Liste Befüllen
5  foreach (BPMN.IntermediateThrowEvent itOld in
       pOld.IntermediateThrowEvent)
6  {
7      SimpleBPMN.IntermediateThrowEvent itNew = new
           SimpleBPMN.IntermediateThrowEvent();
8
9      itNew.id = itOld.Id;
10     itNew.name = itOld.Name;
11
12     itNew.incomings = itOld.Incoming;
13     itNew.outgoing = itOld.Outgoing;
```

```
14      itNew.messageFlow =
            getMessageFlowFromSourceOrTargetRef(itOld.Id,
            newProcess.messageFlows);
15
16      pNew.intermediateThrowEvents.Add(itNew);
17  }
```

Code-Snippet 4.6: C# Umwandlung IntermediateThrowEvents

4.1.10 Eintretende Zwischenereignisse

Zuerst wird je eine Liste für die `intermediateCatchMessageEvents`, als auch für die `intermediateCatchTimeEvents` erstellt. Danach wird jedes alte `IntermediateThrowEvent` je nachdem ob es eine `MessageEventDefinition` besitzt, in ein `intermediateCatchTimeEvent`, oder wenn nicht, in ein `intermediateCatchMessageEvent` umgewandelt. Dabei wird in beiden Fällen die ID, der Name, die `incoming` ID und die `outgoing` ID beibehalten. Zusätzlich wird beim `intermediateCatchTimeEvent` noch die `timeDuration` beibehalten und beim `intermediateCatchMessageEvent` wird noch der entsprechende `MessageFlow` mitgespeichert. Diese Umwandlung wird in Code-Snippet 4.7 gezeigt.

```
1   // Liste der intermediateCatchEvent für Message und Time erstellen
2   pNew.intermediateCatchMessageEvents = new
        List<SimpleBPMN.IntermediateCatchMessageEvent>();
3   pNew.intermediateCatchTimeEvents = new
        List<SimpleBPMN.IntermediateCatchTimeEvent>();
4   // Listen Befüllen
5   foreach (BPMN.IntermediateCatchEvent icOld in
        pOld.IntermediateCatchEvent)
6   {
7       if (icOld.MessageEventDefinition != null)
8       {
9           SimpleBPMN.IntermediateCatchMessageEvent icmNew = new
                SimpleBPMN.IntermediateCatchMessageEvent();
10          icmNew.id = icOld.Id;
11          icmNew.name = icOld.Name;
12          icmNew.incoming = icOld.Incoming;
13          icmNew.outgoing = icOld.Outgoing;
14          icmNew.messageFlow =
                getMessageFlowFromSourceOrTargetRef(icOld.Id,
                newProcess.messageFlows);
15          pNew.intermediateCatchMessageEvents.Add(icmNew);
16      }
17      else
18      {
19          SimpleBPMN.IntermediateCatchTimeEvent ictNew = new
                SimpleBPMN.IntermediateCatchTimeEvent();
20          ictNew.id = icOld.Id;
```

```
21          ictNew.name = icOld.Name;
22          ictNew.incoming = icOld.Incoming;
23          ictNew.outgoing = icOld.Outgoing;
24          ictNew.timeDuration =
                icOld.TimerEventDefinition.TimeDuration.Text;
25          pNew.intermediateCatchTimeEvents.Add(ictNew);
26      }
27  }
```

Code-Snippet 4.7: C# Umwandlung IntermediateCatchEvents

4.1.11 Exklusives Gateway

Als Nächstes wird jedes alte `ExclusiveGateway` in einen `Task` umgewandelt. Dabei wird die ID, der Name, die `incoming` IDs und die `outgoing` IDs beibehalten. Diese Umwandlung wird in Code-Snippet 4.8 gezeigt.

```
1  // Liste der tasks mit exclusiveGateway Elementen Befüllen
2  foreach (BPMN.ExclusiveGateway egOld in pOld.ExclusiveGateway)
3  {
4      SimpleBPMN.Task tNew = new SimpleBPMN.Task();
5
6      tNew.id = egOld.Id;
7      tNew.name = egOld.Name;
8      tNew.incomings = egOld.Incoming;
9      tNew.outgoings = egOld.Outgoing;
10
11     pNew.tasks.Add(tNew);
12 }
```

Code-Snippet 4.8: C# Umwandlung ExclusiveGateways

4.1.12 Ereignisbasiertes Gateway

Zuerst wird eine Liste aller `EventBasedGateways` erstellt. Danach wird jedes alte `EventBasedGateway` in ein neues umgewandelt. Dabei wird die ID, der Name, die `incoming` IDs, die `outgoing` IDs und die `gatewayDirection` beibehalten. Diese Umwandlung wird in Code-Snippet 4.9 gezeigt.

```
// Liste der eventBasedGateways erstellen
pNew.eventBasedGateways = new List<SimpleBPMN.EventBasedGateway>();
// Liste Befüllen
foreach (BPMN.EventBasedGateway ebgOld in pOld.EventBasedGateway)
{
    SimpleBPMN.EventBasedGateway ebgNew = new
        SimpleBPMN.EventBasedGateway();

    ebgNew.id = ebgOld.Id;
    ebgNew.name = ebgOld.Name;
    ebgNew.incomings = ebgOld.Incoming;
    ebgNew.outgoings = ebgOld.Outgoing;
    ebgNew.gatewayDirection = ebgOld.GatewayDirection;

    pNew.eventBasedGateways.Add(ebgNew);
}
```

Code-Snippet 4.9: C# Umwandlung EventBasedGateways

4.1.13 Endereignis

Zuerst wird eine Liste aller `EndEvents` erstellt. Danach wird jedes alte `EndEvent` in ein neues umgewandelt. Dabei wird die ID, der Name und die `incoming` IDs beibehalten. Diese Umwandlung wird in Code-Snippet 4.10 gezeigt.

```
// Liste der EndEvents erstellen
pNew.endEvents = new List<SimpleBPMN.EndEvent>();
// Liste Befüllen
foreach (BPMN.EndEvent eeOld in pOld.EndEvent)
{
    SimpleBPMN.EndEvent eeNew = new SimpleBPMN.EndEvent();

    eeNew.id = eeOld.Id;
    eeNew.name = eeOld.Name;
    eeNew.incomings = eeOld.Incoming;

    pNew.endEvents.Add(eeNew);
}
```

Code-Snippet 4.10: C# Umwandlung EndEvents

4.1.14 Sequenzfluss

Zuerst wird eine Liste aller `SequenceFlows` erstellt. Danach wird jeder alte `SequenceFlow` in einen neuen umgewandelt. Dabei wird die ID, der Name, die `sourceRef` ID und die `targetRef` ID beibehalten. Diese Umwandlung wird in Code-Snippet 4.11 gezeigt.

```
// Liste der SequenceFlows erstellen
pNew.sequenceFlows = new List<SimpleBPMN.SequenceFlow>();
// Liste Befüllen
foreach (BPMN.SequenceFlow sfOld in pOld.SequenceFlow)
{
    SimpleBPMN.SequenceFlow sfNew = new SimpleBPMN.SequenceFlow();

    sfNew.id = sfOld.Id;
    sfNew.name = sfOld.Name;
    sfNew.sourceRef = sfOld.SourceRef;
    sfNew.targetRef = sfOld.TargetRef;

    pNew.sequenceFlows.Add(sfNew);
}
```

Code-Snippet 4.11: C# Umwandlung SequenceFlows

4.2 Umwandlung von SimpleBPMN in PASS

In diesem Kapitel wird gezeigt, wie die Umwandlung von SimpleBPMN in PASS in C# funktioniert. Hierfür kommen im Code immer wieder die Variablen `newProcess` und `oldProcess` vor. `newProcess` steht hierbei für das PASS Modell und `oldProcess` steht für das SimpleBPMN Modell.

Zuallererst wird für ein PASS Modell das `PASSProcessModel` Element und das `ModelLayer` Element benötigt. Beide Elemente werden immer gleich benannt und mit statischen Werten befüllt. Danach wird das `ModelLayer` Element zum `PASSProcessModel` Element hinzugefügt. Alle weiteren Elemente werden zum `ModelLayer` Element hinzugefügt. Diese Umwandlung wird in Code-Snippet 4.12 gezeigt.

```
// neues PASSProcessModel erstellen
PASS.PASSProcessModel passProcessModel = new
    PASS.PASSProcessModel();
passProcessModel.componentId = "ID";
passProcessModel.componentLabel = "Neuer S-BPM Prozess";
newProcess.passProcessModel = passProcessModel;

// SID erstellen
PASS.ModelLayer sid = new PASS.ModelLayer();
sid.componentId = "SID_1";
sid.componentLabel = "SID_1";
newProcess.modelLayer = sid;
```

Code-Snippet 4.12: C# Umwandlung PASSProcessModel und ModelLayer

4.2.1 Participants

Eines der wichtigsten Elemente eines PASS Prozesses sind die Subjekte. Diese werden aus den `Participants` generiert.

Zuerst wird eine Liste der `Subjects` erstellt. Danach wird jeder `Participant` in ein `Subject` umgewandelt. Dabei wird die ID des `Participants` in die `ComponentID` des `Subjects` umgewandelt. Der Name des `Participants` wird zum `ComponentLabel` des `Subjects`. Und die ID und die `processRef` des `Participants` wird zur `ComponentID` des `subjectBehavior` Elements umgewandelt. Diese Umwandlung wird in Code-Snippet 4.13 gezeigt.

```
1  // Liste der Subjects erstellen
2  newProcess.subjects = new List<PASS.FullySpecifiedSubject>();
3  // Liste befüllen
4  foreach (SimpleBPMN.Participant pOld in oldProcess.participants)
5  {
6      // Subject erstellen
7      PASS.FullySpecifiedSubject sNew = new
           PASS.FullySpecifiedSubject();
8      sNew.bpmnId = pOld.id;
9      sNew.componentId = "SID_1_FullySpecifiedSubject_" + pOld.id;
10     sNew.componentLabel = pOld.name;
11
12     sNew.subjectBehavior = "SBD_" + pOld.processRef +
           "_SID_1_FullySpecifiedSubject_" + pOld.id;
13
14     newProcess.subjects.Add(sNew);
15 }
```

Code-Snippet 4.13: C# Umwandlung der SimpleBPMN Participants

4.2.2 Nachrichtenfluss

Neben den Subjekten ist der Nachrichtenfluss einer der Hauptbestandteile eines PASS Prozesses. Dieser wird aus den `MessageFlows` generiert.

Zuerst wird eine Liste der `MessageExchangeLists` erstellt. Danach wird jeder `MessageFlow` in eine `MessageSpecification` und einen `MessageExchange` umgewandelt. Diese werden dann miteinander verknüpft und in einer `MessageExchangeList` gespeichert. Danach wird die `MessageExchangeList` in der zuvor erstellten Liste gespeichert.

Für die `MessageSpecification` wird die ID des `MessageFlows` in eine `ComponentID` umgewandelt. Außerdem wird der Name des `MessageFlows` als `ComponentLabel` der `MessageSpecification` genutzt.

Für den `MessageExchange` wird die ID des `MessageFlows`, die ID des `sourceParticipant` des MessageFlows und die ID des targetParticipant des MessageFlows in eine `ComponentID` umgewandelt. Außerdem wird der Name des `MessageFlows`, der Name des `sourceParticipant` des `MessageFlows` und der Name des `targetParticipant`

des MessageFlows in das ComponentLabel des MessageExchange umgewandelt. Danach wird die ID des sourceParticipant des MessageFlows in den sender des MessageExchange umgewandelt. Danach wird die ID des targetParticipant des MessageFlows in den receiver des MessageExchange umgewandelt. Schlussendlich wird die zuvor erstellte MessageSpecification zur MessageExchange hinzugefügt.

Am Ende wird der MessageExchange in die richtige MessageExchangeList hinzugefügt. Hierfür wird die ID des sourceParticipant des MessageFlows und die ID des targetParticipant des MessageFlows in eine ComponentID umgewandelt. Mit dieser ID wird in der Liste der MessageExchangeLists das entsprechende Element gesucht. Danach wird dieses Element aus der Liste gelöscht, damit es später keine doppelten Elemente gibt. Falls keine MessageExchangeList gefunden wurde, wird eine neue mit der zuvor generierten ID erstellt. Danach wird mit der ID des sourceParticipant des MessageFlows und der ID des targetParticipant des MessageFlows ein ComponentLabel erstellt. Außerdem wird eine Liste von MessageExchanges erstellt. Am Ende wird, egal ob eine MessageExchangeList gefunden oder neu erstellt wurde, der MessageExchange in die MessageExchangeList hinzugefügt und die MessageExchangeList zum PASS Prozess hinzugefügt.

Diese Umwandlungen werden in Code-Snippet 4.14 gezeigt.

```
1  // MessageFlow Umwandeln
2  newProcess.messageExchangeLists = new
       List<PASS.MessageExchangeList>();
3  // Liste Befüllen
4  foreach (SimpleBPMN.MessageFlow mfOld in oldProcess.messageFlows)
5  {
6  // MessageSpecification erstellen
7  PASS.MessageSpecification msNew = new PASS.MessageSpecification();
8  msNew.componentId = "MsgSpec_" + mfOld.id;
9  msNew.componentLabel = mfOld.name;
10
11 // MessageExchange erstellen
12 PASS.MessageExchange meNew = new PASS.MessageExchange();
13 meNew.bpmnId = mfOld.id;
14 meNew.componentId = "SID_1_StdMsgConn_" +
       mfOld.sourceParticipant.id + "+" + mfOld.targetParticipant.id +
       "_MsgSpec_" + mfOld.id;
15 meNew.componentLabel = "Message: " + mfOld.name + " From: " +
       mfOld.sourceParticipant.name + " To: " +
       mfOld.targetParticipant.name;
16 meNew.sender = "SID_1_FullySpecifiedSubject_" +
       mfOld.sourceParticipant.id;
17 meNew.receiver = "SID_1_FullySpecifiedSubject_" +
       mfOld.targetParticipant.id;
18 meNew.messageSpecification = msNew;
19
20 // MessageExchangeList erstellen
21 string melNewComponentId =
       "MessageExchangeList_on_SID_1_StdMsgConn_" +
```

```
    mfOld.sourceParticipant.id + "+" + mfOld.targetParticipant.id;
PASS.MessageExchangeList melNew =
    findMessageExchangeList(melNewComponentId,
    newProcess.messageExchangeLists);
newProcess.messageExchangeLists.Remove(melNew);
if (melNew == null)
{
    melNew = new PASS.MessageExchangeList();
    melNew.componentId = melNewComponentId;
    melNew.componentLabel = "SID_1_StdMsgConn_" +
        mfOld.sourceParticipant.id + "+" +
        mfOld.targetParticipant.id;

    melNew.messageExchanges = new List<PASS.MessageExchange>();
}
melNew.messageExchanges.Add(meNew);
newProcess.messageExchangeLists.Add(melNew);
}
```

Code-Snippet 4.14: C# Umwandlung der SimpleBPMN MessageFlows

4.2.3 Process

Jedes Subjekt in einem PASS Prozess benötigt ein SBD, das als SubjectBehavior gespeichert wird. Diese Elemente werden aus den `Processes` generiert.

Zuerst wird eine Liste der `SubjectBehaviors` erstellt. Danach wird jeder `Process` in ein `SubjectBehavior` umgewandelt. Dabei wird die ID des `Processes` und die ID des zugehörigen `Participants` in die `ComponentID` des `SubjectBehaviors` umgewandelt. Der Name des `Processes` wird zum `ComponentLabel` des `Subjects` umgewandelt. Außerdem wird die ID des zugehörigen `Participants` in die `SubjectComponentID` umgewandelt. Schlussendlich wird noch eine Liste von `Actions` erstellt. Danach werden alle weiteren Elemente, die sich im `Process` befinden, umgewandelt und als `Action` in der Liste gespeichert. Diese Umwandlung wird in Code-Snippet 4.15 gezeigt.

```
// Liste der SubjectBehaviors erstellen
newProcess.subjectBehaviors = new List<PASS.SubjectBehavior>();
// Liste befüllen
foreach (SimpleBPMN.Process pOld in oldProcess.processes)
{
    // SubjectBehavior erstellen
    PASS.SubjectBehavior sbNew = new PASS.SubjectBehavior();

    sbNew.componentId = "SBD_" + pOld.id +
        "_SID_1_FullySpecifiedSubject_" + pOld.participant.id;
    sbNew.componentLabel = "SBD: " + pOld.participant.name;

```

```
12      sbNew.subjectComponentId = "SID_1_FullySpecifiedSubject_" +
            pOld.participant.id;
13
14      // Liste der Actions erstellen
15      sbNew.actions = new List<PASS.Action>();
16
17      newProcess.subjectBehaviors.Add(sbNew);
18  }
```

Code-Snippet 4.15: C# Umwandlung der SimpleBPMN Processes

4.2.4 Startereignis

In einem PASS Prozess gibt es immer einen State, der ein InitialStateOfBehavior ist. Dieser wird aus dem `StartEvent` erzeugt. Normalerweise kann jeder State ein InitialStateOfBehavior sein. Bei der Umwandlung von BPMN Prozessen wird dieser State immer als `DoState` gespeichert.

Zuerst wird die ID des `StartEvents` und die ID des zugehörigen `Processes` in die `ComponentID` des `DoStates` umgewandelt. Der Name des `StartEvents` wird zum `ComponentLabel` des `DoStates`. Danach wird das `isStartState` Attribut noch auf `true` gesetzt.

Als Nächstes wird die zugehörige `Action` erstellt. Hierfür wird die ID des `StartEvents` und die ID des zugehörigen `Processes` in die `ComponentID` und das `ComponentLabel` der Action umgewandelt. Anschließend wird der `DoState` und eine leere Liste von `Transitions` hinzugefügt. Diese Umwandlung wird in Code-Snippet 4.16 gezeigt.

```
1  // StartEvenet umwandeln
2  PASS.DoState dssNew = new PASS.DoState();
3
4  dssNew.componentId = "SBD_" + pOld.id + "_DoState_" +
       pOld.startEvent.id;
5  dssNew.componentLabel = pOld.startEvent.name;
6
7  dssNew.isStartState = true;
8
9  PASS.Action asNew = new PASS.Action();
10
11 asNew.bpmnId = pOld.startEvent.id;
12 asNew.componentId = "action_of_SBD_" + pOld.id + "_DoState_" +
       pOld.startEvent.id;
13 asNew.componentLabel = "action_of_SBD_" + pOld.id + "_DoState_" +
       pOld.startEvent.id;
14 asNew.state = dssNew;
15 asNew.transitions = new List<PASS.Transition>();
16 sbNew.actions.Add(asNew);
```

Code-Snippet 4.16: C# Umwandlung der SimpleBPMN StartEvents

4.2.5 Task

In einem SBD eines PASS Prozesses gibt es meistens ein oder mehrere DoStates. Diese Elemente werden unter anderem aus den `Tasks` erzeugt.

Zuerst wird die ID des `Tasks` und die ID des zugehörigen `Processes` in die `ComponentID` des `DoStates` umgewandelt. Der Name des `Tasks` wird zum `ComponentLabel` des `DoStates`.

Als Nächstes wird die zugehörige `Action` erstellt. Hierfür wird die ID des `Tasks` und die ID des zugehörigen `Processes` in die `ComponentID` und das `ComponentLabel` der `Action` umgewandelt. Anschließend wird der `DoState` und eine leere Liste von `Transitions` hinzugefügt. Diese Umwandlung wird in Code-Snippet 4.17 gezeigt.

```
1  // tasks umwandeln
2  foreach (SimpleBPMN.Task tOld in pOld.tasks)
3  {
4      // DoState erstellen
5      PASS.DoState dsNew = new PASS.DoState();
6
7      dsNew.componentId = "SBD_" + pOld.id + "_DoState_" + tOld.id;
8      dsNew.componentLabel = tOld.name;
9
10     PASS.Action aNew = new PASS.Action();
11
12     aNew.bpmnId = tOld.id;
13     aNew.componentId = "action_of_SBD_" + pOld.id + "_DoState_" +
           tOld.id;
14     aNew.componentLabel = "action_of_SBD_" + pOld.id + "_DoState_" +
           tOld.id;
15
16     aNew.state = dsNew;
17     aNew.transitions = new List<PASS.Transition>();
18
19     sbNew.actions.Add(aNew);
20 }
```

Code-Snippet 4.17: C# Umwandlung der SimpleBPMN Tasks

4.2.6 Endereignis

In einem SBD eines PASS Prozesses gibt es immer einen oder mehrere States, die auch ein EndState sind. Diese werden aus den `EndEvents` erzeugt. Normalerweise kann jeder State ein EndState sein. Bei der Umwandlung von BPMN Prozessen wird dieser State immer als `DoState` gespeichert.

Zuerst wird die ID des `EndEvents` und die ID des zugehörigen `Processes` in die `ComponentID` des `DoStates` umgewandelt. Der Name des `EndEvents` wird zum `ComponentLabel` des `DoStates`. Danach wird das `isEndState` Attribut noch auf `true` gesetzt.

Als Nächstes wird die zugehörige `Action` erstellt. Hierfür wird die ID des `EndEvents` und die ID des zugehörigen `Processes` in die `ComponentID` und das `ComponentLabel` der `Action` umgewandelt. Anschließend wird der `DoState` und eine leere Liste von `Transitions` hinzugefügt. Diese Umwandlung wird in Code-Snippet 4.18 gezeigt.

```
// endEvents umwandeln
foreach (SimpleBPMN.EndEvent eeOld in pOld.endEvents)
{
    // DoState erstellen
    PASS.DoState dsNew = new PASS.DoState();

    dsNew.componentId = "SBD_" + pOld.id + "_DoState_" + eeOld.id;
    dsNew.componentLabel = eeOld.name;

    dsNew.isEndState = true;

    PASS.Action aNew = new PASS.Action();

    aNew.bpmnId = eeOld.id;
    aNew.componentId = "action_of_SBD_" + pOld.id + "_DoState_" +
        eeOld.id;
    aNew.componentLabel = "action_of_SBD_" + pOld.id + "_DoState_" +
        eeOld.id;

    aNew.state = dsNew;

    sbNew.actions.Add(aNew);
}
```

Code-Snippet 4.18: C# Umwandlung der SimpleBPMN EndEvents

4.2.7 Ereignisbasiertes Gateway

Um auf Nachrichten von andere Subjekte zu warten gibt, es in einem SBD eines PASS Prozesses ein oder mehrere ReceiveStates. Diese Elemente werden aus den `EventBasedGateways` erzeugt.

Zuerst wird die ID des `EventBasedGateways` und die ID des zugehörigen `Processes` in die `ComponentID` des `ReceiveStates` umgewandelt. Der Name des `EventBasedGateways` wird zum `ComponentLabel` des `ReceiveStates`.

Als Nächstes wird die zugehörige `Action` erstellt. Hierfür wird die ID des `EventBasedGateways` und die ID des zugehörigen `Processes` in die `ComponentID` und das `ComponentLabel` der `Action` umgewandelt. Anschließend wird der `ReceiveStates` und eine leere Liste von `Transitions` hinzugefügt. Diese Umwandlung wird in Code-Snippet 4.19 gezeigt.

```
// ereignisbasiertes Gateway umwandeln
foreach (SimpleBPMN.EventBasedGateway ebgOld in
    pOld.eventBasedGateways)
{
    // ReceiveState erstellen
    PASS.ReceiveState rsNew = new PASS.ReceiveState();

    rsNew.componentId = "SBD_" + pOld.id + "_ReceiveState_" +
        ebgOld.id;
    rsNew.componentLabel = ebgOld.name;

    PASS.Action aNew = new PASS.Action();
    aNew.bpmnId = ebgOld.id;
    aNew.componentId = "action_of_SBD_" + pOld.id + "_ReceiveState_"
        + ebgOld.id;
    aNew.componentLabel = "action_of_SBD_" + pOld.id +
        "_ReceiveState_" + ebgOld.id;

    aNew.state = rsNew;
    aNew.transitions = new List<PASS.Transition>();

    sbNew.actions.Add(aNew);
}
```

Code-Snippet 4.19: C# Umwandlung der SimpleBPMN EventBasedGateways

4.2.8 Eintretende Zeit Zwischenereignisse

Um die Zeit zu messen und dadurch Zustandsübergänge auszulösen, gibt es in einem PASS Prozess ein oder mehrere DayTimeTimerTransitions. Diese Elemente werden aus den `IntermediateCatchTimeEvents` erzeugt.

Wenn sich das `IntermediateCatchTimeEvent` nach einem `EventBasedGateway` befindet, wird die `DayTimeTimerTransition` direkt an den daraus erstellten `ReceiveState` angehängt. Sollte dies nicht der Fall sein, muss ein eigener `DoState` hierfür erstellt werden.

Zuerst wird die ID des `IntermediateCatchTimeEvents` und die ID des zugehörigen `Processes` in die `ComponentID` des `DoStates` umgewandelt. Der Name des `IntermediateCatchTimeEvents` wird zum `ComponentLabel` des `DoStates`.

Als Nächstes wird die zugehörige `Action` erstellt. Hierfür wird die ID des `IntermediateCatchTimeEvents` und die ID des zugehörigen `Processes` in die `ComponentID` und das `ComponentLabel` der `Action` umgewandelt. Anschließend wird der `DoState` und eine leere Liste von `Transitions` hinzugefügt. Diese Umwandlung wird in Code-Snippet 4.20 gezeigt.

```
// imtermediateCatchTimeEvents umwandeln (State)
foreach (SimpleBPMN.IntermediateCatchTimeEvent icetOld in
    pOld.intermediateCatchTimeEvents)
{
    if (!EventGatewayWasBefore(icetOld.incoming, pOld.sequenceFlows,
        pOld.eventBasedGateways))
    {
        // DoState erstellen
        PASS.DoState dsNew = new PASS.DoState();

        dsNew.componentId = "SBD_" + pOld.id + "_DoState_" +
            icetOld.id;
        dsNew.componentLabel = icetOld.name;

        PASS.Action aNew = new PASS.Action();

        aNew.bpmnId = icetOld.id;
        aNew.componentId = "action_of_SBD_" + pOld.id + "_DoState_"
            + icetOld.id;
        aNew.componentLabel = "action_of_SBD_" + pOld.id +
            "_DoState_" + icetOld.id;

        aNew.state = dsNew;

        sbNew.actions.Add(aNew);
    }
}
```

Code-Snippet 4.20: C# Umwandlung der SimpleBPMN ImtermediateCatchTimeEvents (State)

Als Nächstes wird die `DayTimeTimerTransition` und die zugehörige `DayTimeTimerTransitionCondition` erstellt.

Zuerst wird die `DayTimeTimerTransitionCondition` erstellt. Hierfür wird die `outgoing` ID des `IntermediateCatchTimeEvents` und die ID des zugehörigen `Processes` in die `ComponentID` der `DayTimeTimerTransitionCondition` umgewandelt. Der Name des `IntermediateCatchTimeEvents` wird zum `ComponentLabel` der `DayTimeTimerTransitionCondition`. Außerdem wird die `timeDuration` des `IntermediateCatchTimeEvents` zur `duration` der `DayTimeTimerTransitionCondition`.

Als Nächstes wird die `DayTimeTimerTransition` erstellt. Hierfür wird der Name des `IntermediateCatchTimeEvents` zum `ComponentLabel` der `DayTimeTimerTransition`.

Wenn sich das `IntermediateCatchTimeEvent` nach einem `EventBasedGateway` befindet wird die ID, die `incoming` ID, die `outgoing` ID und die `process` ID des `IntermediateCatchTimeEvents` in die `ComponentID` der `DayTimeTimerTransition` umgewandelt. Danach wird mit Hilfe der `incoming` ID des `IntermediateCatchTimeEvents` der `sourceState` gefunden und hinzugefügt.

Wenn dies nicht der Fall ist, wird nur die `outgoing` ID und die `process` ID des `IntermediateCatchTimeEvents` in die `ComponentID` der `DayTimeTimerTransition` umgewandelt. Danach wird mit Hilfe der ID des `IntermediateCatchTimeEvents` der `sourceState` gefunden und hinzugefügt.

In beiden Fällen wird als Nächstes mit Hilfe der `outgoing` ID des `IntermediateCatchTimeEvents` der `targetState` gefunden und hinzugefügt. Zum Schluss wird noch die zuvor erstellte `DayTimeTimerTransitionCondition` hinzugefügt.

Nachdem die `DayTimeTimerTransition` erstellt wurde, muss sie noch zur richtigen `Action` hinzugefügt werden. Je nachdem, ob sich vor dem `IntermediateCatchTimeEvent` ein `EventBasedGateway` befindet oder nicht, wird zum Suchen der `Action` entweder die ID oder die `incoming` ID des `IntermediateCatchTimeEvents` benutzt. Danach wird die gefundene `Action` aus der Liste gelöscht, damit es später keine Duplikate gibt. Anschließend wird die `DayTimeTimerTransition` zur `Action` hinzugefügt und schlussendlich wird die so erweiterte `Action` wieder zur Liste hinzugefügt.

Diese Umwandlungen werden in Code-Snippet 4.21 gezeigt.

```
1  // imtermediateCatchTimeEvents umwandeln (Transition)
2  foreach (SimpleBPMN.IntermediateCatchTimeEvent icetOld in
       pOld.intermediateCatchTimeEvents)
3  {
4      // DayTimeTimerTransitionCondition erstellen erstellen
5      PASS.DayTimeTimerTransitionCondition ttcNew = new
           PASS.DayTimeTimerTransitionCondition();
6      ttcNew.componentId = "SBD_" + pOld.id +
           "_DayTimeTimerTransition_" + icetOld.outgoing +
           "_DayTimeTimerTransitionCondition";
7      ttcNew.componentLabel = icetOld.name;
8      ttcNew.duration = icetOld.timeDuration;
9
10     // DayTimeTimerTransition erstellen erstellen
11     PASS.DayTimeTimerTransition ttNew = new
           PASS.DayTimeTimerTransition();
12     ttNew.componentLabel = icetOld.name;
13
14     if (EventGatewayWasBefore(icetOld.incoming, pOld.sequenceFlows,
           pOld.eventBasedGateways))
15     {
16         ttNew.componentId = "SBD_" + pOld.id +
               "_DayTimeTimerTransition_" + icetOld.incoming + "+" +
               icetOld.id + "+" + icetOld.outgoing;
17         string sourceId = FindBPMNIdOfSourceState(icetOld.incoming,
               pOld.sequenceFlows);
18         ttNew.sourceState = FindActionWithId(sourceId,
               sbNew.actions).state;
19     }
20     else
21     {
22         ttNew.componentId = "SBD_" + pOld.id +
```

```
            "_DayTimeTimerTransition_" + icetOld.outgoing;
23          ttNew.sourceState = FindActionWithId(icetOld.id,
                sbNew.actions).state;
24      }
25      string targetId = FindBPMNIdOfTargetState(icetOld.outgoing,
            pOld.sequenceFlows);
26      ttNew.targetState = FindActionWithId(targetId,
            sbNew.actions).state;
27      ttNew.transitionCondition = ttcNew;
28
29      // Action finden
30      PASS.Action aNew;
31      if (EventGatewayWasBefore(icetOld.incoming, pOld.sequenceFlows,
            pOld.eventBasedGateways))
32      {
33          string sourceId = FindBPMNIdOfSourceState(icetOld.incoming,
                pOld.sequenceFlows);
34          aNew = FindActionWithId(sourceId, sbNew.actions);
35      }
36      else
37      {
38          aNew = FindActionWithId(icetOld.id, sbNew.actions);
39      }
40      sbNew.actions.Remove(aNew);
41      aNew.transitions.Add(ttNew);
42      sbNew.actions.Add(aNew);
43  }
```

Code-Snippet 4.21: C# Umwandlung der SimpleBPMN ImtermediateCatchTimeEvents (Transition)

4.2.9 Eintretende Nachrichten Zwischenereignisse

Um Nachrichten von anderen Subjekten zu empfangen, gibt es in einem PASS Prozess ein oder mehrere ReceiveTransitions. Diese Elemente werden aus den `IntermediateCatchMessageEvents` erzeugt.

Wenn sich das `IntermediateCatchMessageEvent` nach einem `EventBasedGateway` befindet, wird die `ReceiveTransitions` direkt an den daraus erstellten `ReceiveState` angehängt. Sollte dies nicht der Fall sein, muss ein eigener `ReceiveState` hierfür erstellt werden.

Zuerst wird die ID des `IntermediateCatchMessageEvents` und die ID des zugehörigen `Processes` in die `ComponentID` des `ReceiveStates` umgewandelt. Der Name des `IntermediateCatchMessageEvents` wird zum `ComponentLabel` des `ReceiveStates`.

Als Nächstes wird die zugehörige `Action` erstellt. Hierfür wird die ID des `IntermediateCatchMessageEvents` und die ID des zugehörigen `Processes` in die `ComponentID`

und das ComponentLabel der Action umgewandelt. Anschließend wird der ReceiveState und eine leere Liste von Transitions hinzugefügt. Diese Umwandlung wird in Code-Snippet 4.22 gezeigt.

```
// imtermediateCatchMessageEvents umwandeln (State)
foreach (SimpleBPMN.IntermediateCatchMessageEvent icemOld in
    pOld.intermediateCatchMessageEvents)
{
    if (!EventGatewayWasBefore(icemOld.incoming, pOld.sequenceFlows,
        pOld.eventBasedGateways))
    {
        // ReceiveState erstellen
        PASS.ReceiveState rsNew = new PASS.ReceiveState();
        rsNew.componentId = "SBD_" + pOld.id + "_ReceiveState_" +
            icemOld.id;
        rsNew.componentLabel = icemOld.name;

        PASS.Action aNew = new PASS.Action();

        aNew.bpmnId = icemOld.id;
        aNew.componentId = "action_of_SBD_" + pOld.id +
            "_ReceiveState_" + icemOld.id;
        aNew.componentLabel = "action_of_SBD_" + pOld.id +
            "_ReceiveState_" + icemOld.id;

        aNew.state = rsNew;
        aNew.transitions = new List<PASS.Transition>();

        sbNew.actions.Add(aNew);
    }
}
```

Code-Snippet 4.22: C# Umwandlung der SimpleBPMN IntermediateCatch-MessageEvents (State)

Im nächsten Schritt wird die ReceiveTransitionCondition erstellt. Hierfür wird die outgoing ID des IntermediateCatchMessageEvents und die ID des zugehörigen Processes in die ComponentID der ReceiveTransitionCondition umgewandelt. Der Name des IntermediateCatchMessageEvents wird zum ComponentLabel der ReceiveTransitionCondition.

Außerdem wird mit Hilfe der ID des messageFlows des IntermediateCatchMessageEvents der zugehörige MessageExchange gefunden. Dieser wird dann zur ReceiveTransitionCondition hinzugefügt. Schlussendlich wird noch der sender des MessageExchanges zum messagSentFrom Attribut der ReceiveTransitionCondition.

Als Nächstes wird die ReceiveTransition erstellt. Hierfür wird der Name des messageFlows des IntermediateCatchMessageEvents und der Name des sourceParticipants des messageFlows des IntermediateCatchMessageEvents zum ComponentLabel der ReceiveTransition umgewandelt.

Wenn sich das IntermediateCatchMessageEvent nach einem EventBasedGateway befindet wird die ID, die incoming ID, die outgoing ID und die process ID des IntermediateCatchMessageEvents in die ComponentID der ReceiveTransition umgewandelt. Danach wird mit Hilfe der incoming ID des IntermediateCatchMessageEvents der sourceState gefunden und hinzugefügt. Wenn dies nicht der Fall ist, wird nur die outgoing ID und die process ID des IntermediateCatchMessageEvents in die ComponentID der ReceiveTransition umgewandelt. Danach wird mit Hilfe der ID des IntermediateCatchMessageEvents der sourceState gefunden und hinzugefügt.

In beiden Fällen wird als Nächstes mit Hilfe der outgoing ID des IntermediateCatchMessageEvents der targetState gefunden und hinzugefügt. Zum Schluss wird noch die zuvor erstellte ReceiveTransitionCondition hinzugefügt.

Nachdem die ReceiveTransition erstellt wurde, muss sie noch zur richtigen Action hinzugefügt werden. Je nachdem, ob sich vor dem IntermediateCatchMessageEvent ein EventBasedGateway befindet oder nicht, wird zum Suchen der Action entweder die ID oder die incoming ID des IntermediateCatchMessageEvents benutzt. Danach wird die gefundene Action aus der Liste gelöscht, damit es später keine Duplikate gibt. Anschließend wird die ReceiveTransition zur Action hinzugefügt und schlussendlich wird die so erweiterte Action wieder zur Liste hinzugefügt.

Diese Umwandlungen werden in Code-Snippet 4.23 gezeigt.

```
1  // imtermediateCatchMessageEvents umwandeln (Transition)
2  foreach (SimpleBPMN.IntermediateCatchMessageEvent icemOld in
       pOld.intermediateCatchMessageEvents)
3  {
4      // ReceiveTransitionCondition erstellen
5      PASS.ReceiveTransitionCondition rtcNew = new
           PASS.ReceiveTransitionCondition();
6      rtcNew.componentId = "SBD_" + pOld.id + "_ReceiveTransition_" +
           icemOld.outgoing + "_receiveTransitionCondition";
7      rtcNew.componentLabel = icemOld.name;
8      PASS.MessageExchange me = findMessageExchange(
           icemOld.messageFlow.id, newProcess.messageExchangeLists);
9      rtcNew.performedMessageExchange = me;
10     rtcNew.messagSentFrom = me.sender;
11
12     // ReceiveTransition erstellen
13     PASS.ReceiveTransition rtNew = new PASS.ReceiveTransition();
14     rtNew.componentLabel = "From: " +
           icemOld.messageFlow.sourceParticipant.name + " Msg: " +
           icemOld.messageFlow.name;
15     if (EventGatewayWasBefore(icemOld.incoming, pOld.sequenceFlows,
           pOld.eventBasedGateways))
```

```
16      {
17          rtNew.componentId = "SBD_" + pOld.id + "_ReceiveTransition_"
                + icemOld.incoming + "+" + icemOld.id + "+" +
                icemOld.outgoing;
18          string sourceId = FindBPMNIdOfSourceState(icemOld.incoming,
                pOld.sequenceFlows);
19          rtNew.sourceState = FindActionWithId(sourceId,
                sbNew.actions).state;
20      }
21      else
22      {
23          rtNew.componentId = "SBD_" + pOld.id + "_ReceiveTransition_"
                + icemOld.outgoing;
24          rtNew.sourceState = FindActionWithId(icemOld.id,
                sbNew.actions).state;
25      }
26      string targetId = FindBPMNIdOfTargetState(icemOld.outgoing,
            pOld.sequenceFlows);
27      rtNew.targetState = FindActionWithId(targetId,
            sbNew.actions).state;
28      rtNew.transitionCondition = rtcNew;
29
30      PASS.Action aNew;
31      if (EventGatewayWasBefore(icemOld.incoming, pOld.sequenceFlows,
            pOld.eventBasedGateways))
32      {
33          string sourceId = FindBPMNIdOfSourceState(icemOld.incoming,
                pOld.sequenceFlows);
34          aNew = FindActionWithId(sourceId, sbNew.actions);
35      }
36      else
37      {
38          aNew = FindActionWithId(icemOld.id, sbNew.actions);
39      }
40      sbNew.actions.Remove(aNew);
41      aNew.transitions.Add(rtNew);
42      sbNew.actions.Add(aNew);
43  }
```

Code-Snippet 4.23: C# Umwandlung der SimpleBPMN IntermediateCatch-MessageEvents (Transition)

4.2.10 Auslösende Zwischenereignisse

Um Nachrichten an andere Subjekte zu verschicken, gibt es in einem PASS Prozess ein oder mehrere SendStates und SendTransitions. Diese Elemente werden aus den `IntermediateThrowEvents` erzeugt.

Zuerst wird der `SendState` erstellt. Hierfür wird die `outgoing` ID des `IntermediateThrowEvents` und die ID des zugehörigen `Processes` in die `ComponentID` des `SendStates` umgewandelt. Der Name des `IntermediateThrowEvents` wird zum `ComponentLabel` des `SendStates`.

Als Nächstes wird die `SendTransitionCondition` erstellt. Hierfür wird die `outgoing` ID des `IntermediateThrowEvents` und die ID des zugehörigen `Processes` in die `ComponentID` der `SendTransitionCondition` umgewandelt. Der Name des `IntermediateThrowEvents` wird zum `ComponentLabel` der `SendTransitionCondition`. Außerdem wird mit Hilfe der ID des `messageFlows` des `IntermediateThrowEvents` der zugehörige `MessageExchange` gefunden. Dieser wird dann zur `SendTransitionCondition` hinzugefügt. Schlussendlich wird noch der `receiver` des `MessageExchanges` zum `messagSentTo` Attribut der `SendTransitionCondition`.

Danach wird die `SendTransition` erstellt. Hierfür wird die `outgoing` ID und die `process` ID des `IntermediateThrowEvents` in die `ComponentID` der `SendTransition` umgewandelt. Der Name des `messageFlows` des `IntermediateThrowEvents` und der Name des `targetParticipants` des `messageFlows` des `IntermediateThrowEvents` wird zum `ComponentLabel` der `SendTransition` umgewandelt. Als Nächstes wird mit Hilfe der `outgoing` ID des `IntermediateThrowEvents` der `targetState` gefunden und hinzugefügt. Danach wird der zuvor erstellte `SendState` als `sourceState` hinzugefügt. Am Ende wird noch die zuvor erstellte `SendTransitionCondition` hinzugefügt.

Zum Schluss wird die zugehörige `Action` erstellt. Hierfür wird die ID des `IntermediateThrowEvents` und die ID des zugehörigen `Processes` in die `ComponentID` und das `ComponentLabel` der `Action` umgewandelt. Anschließend wird der `SendState` und eine leere Liste von `Transitions` hinzugefügt. Abschließend wird die zuvor erstellte `SendTransition` zur Liste hinzugefügt.

Diese Umwandlungen werden in Code-Snippet 4.24 gezeigt.

```
1  // ausloesende Zwischenereignisse umwandeln
2  foreach (SimpleBPMN.IntermediateThrowEvent iteOld in
       pOld.intermediateThrowEvents)
3  {
4      // SendState erstellen
5      PASS.SendState ssNew = new PASS.SendState();
6
7      ssNew.componentId = "SBD_" + pOld.id + "_SendState_" + iteOld.id;
8      ssNew.componentLabel = iteOld.name;
9
10     // SendTransitionCondition erstellen
11     PASS.SendTransitionCondition stcNew = new
           PASS.SendTransitionCondition();
```

```
12
13      stcNew.componentId = "SBD_" + pOld.id + "_SendTransition_" +
            iteOld.outgoing + "_sendTransitionCondition";
14      stcNew.componentLabel = iteOld.name;
15
16      PASS.MessageExchange me =
            findMessageExchange(iteOld.messageFlow.id,
            newProcess.messageExchangeLists);
17
18      stcNew.performedMessageExchange = me;
19      stcNew.messagSentTo = me.receiver;
20
21      // SendTransition erstellen erstellen
22      PASS.SendTransition stNew = new PASS.SendTransition();
23
24      stNew.componentId = "SBD_" + pOld.id + "_SendTransition_" +
            iteOld.outgoing;
25      stNew.componentLabel = "To: " +
            iteOld.messageFlow.targetParticipant.name + " Msg: " +
            iteOld.messageFlow.name;
26
27      string targetId = FindBPMNIdOfTargetState(iteOld.outgoing,
            pOld.sequenceFlows);
28
29      stNew.sourceState = ssNew;
30      stNew.targetState = FindActionWithId(targetId,
            sbNew.actions).state;
31      stNew.transitionCondition = stcNew;
32
33      // Action erstellen
34      PASS.Action aNew = new PASS.Action();
35
36      aNew.bpmnId = iteOld.id;
37      aNew.componentId = "action_of_SBD_" + pOld.id + "_SendState_" +
            iteOld.id;
38      aNew.componentLabel = "action_of_SBD_" + pOld.id + "_SendState_"
            + iteOld.id;
39
40      aNew.state = ssNew;
41      aNew.transitions = new List<PASS.Transition>();
42      aNew.transitions.Add(stNew);
43
44      sbNew.actions.Add(aNew);
45  }
```

Code-Snippet 4.24: C# Umwandlung der SimpleBPMN IntermediateThrowEvents

4.2.11 Sequenzfluss

Zwischen allen States in einem PASS Prozess benötigt man Transitions, um die möglichen Zustandsübergänge darzustellen. Bisher wurden alle Transitions, bis auf die DoTransitions, erstellt. Die fehlenden DoTransitions werden aus den `SequenceFlows` erzeugt.

Hierfür wird zuerst bei jedem `SequenceFlow` festgestellt, ob der State, der mit der `sourceRef` ID des `SequenceFlows` gefunden wurde, ein `DoState` ist. Sollte dies der Fall sein, wird eine `DoTransition` erstellt.

Hierfür wird die ID und die `process` ID des `SequenceFlows` in die `ComponentID` der `DoTransition` umgewandelt. Der Name des `SequenceFlows` wird zum `ComponentLabel` der `DoTransition`. Als Nächstes wird mit Hilfe der `sourceRef` ID des `SequenceFlows` der `sourceState` gefunden und hinzugefügt. Zum Schluss wird mit Hilfe der `targetRef` ID des `SequenceFlows` der `targetState` gefunden und hinzugefügt.

Nachdem die `DoTransition` erstellt wurde, muss sie noch zur richtigen `Action` hinzugefügt werden. Zuerst muss diese mit Hilfe der `sourceRef` ID des `SequenceFlows` gefunden werden. Danach wird die gefundene `Action` aus der Liste gelöscht, damit es später keine Duplikate gibt. Anschließend wird die `DoTransition` zur `Action` hinzugefügt und schlussendlich wird die so erweiterte `Action` wieder zur Liste hinzugefügt.

Diese Umwandlungen werden in Code-Snippet 4.25 gezeigt.

```
1  // sequenceFlows umwandeln
2  foreach (SimpleBPMN.SequenceFlow sfOld in pOld.sequenceFlows)
3  {
4      if (isDoState(sfOld.sourceRef, sbNew.actions))
5      {
6          // DoTransition erstellen
7          PASS.DoTransition rtNew = new PASS.DoTransition();
8          rtNew.componentId = "SBD_" + pOld.id + "_DoTransition_" +
               sfOld.id;
9          rtNew.componentLabel = sfOld.name;
10         rtNew.sourceState = FindActionWithId(sfOld.sourceRef,
               sbNew.actions).state;
11         rtNew.targetState = FindActionWithId(sfOld.targetRef,
               sbNew.actions).state;
12
13         // Action finden
14         PASS.Action aNew = FindActionWithId(sfOld.sourceRef,
               sbNew.actions);
15         sbNew.actions.Remove(aNew);
16         aNew.transitions.Add(rtNew);
17         sbNew.actions.Add(aNew);
18     }
19 }
```

Code-Snippet 4.25: C# Umwandlung der SimpleBPMN SequenceFlows

4.3 Unterschiedliche Konzepte

Es gibt viele unterschiedliche Konzepte in BPMN und PASS. Neben den großen Unterschieden, wie der Subjektorientierung von PASS, gibt es auch noch kleinere Designunterschiede.

In dieser Arbeit stellte sich vor allem der Unterschied bei der Modellierung mit States oder Tasks heraus. Beim BPMN Standard wird mit Tasks modelliert. Das bedeutet, dass Aufgaben modelliert werden. Beim PASS Standard hingegen, wird mit States modelliert. Das bedeutet, dass Zustände modelliert werden.

Dadurch ist die Übersetzung von BPMN in PASS Modelle schwierig. Auch wenn die Nutzung von Tasks und States in den jeweiligen Standards sehr ähnlich funktioniert und es daher nahe liegt Tasks in States zu übersetzen, ist dies nicht ganz korrekt. Trotzdem wird bei dieser Arbeit genau diese Übersetzung durchgeführt.

Wie bereits in Kapitel 4.4 beschrieben, kann dies umgangen werden, indem die Tasks als States modelliert werden. Diese Methode ist zwar auch nicht ganz korrekt, aber besser als es nicht zu tun. Wenn man eine komplett richtige Modellierung haben möchte, müsste mit Zwischenereignissen modelliert werden. Diese dienen dann als States und dazwischen sind die Tasks. Abbildung 4.1 zeigt diese Modellierung.

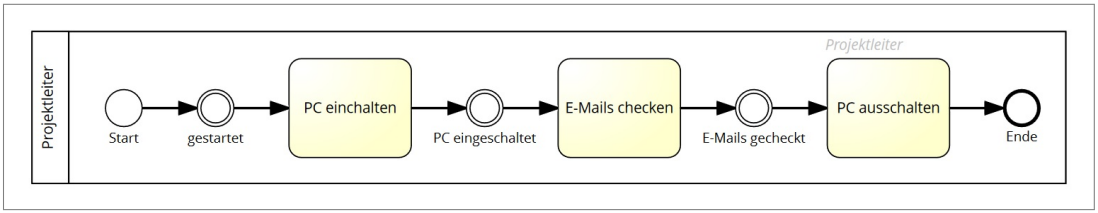

Abbildung 4.1: Subjektorientierter BPMN Prozess mit Zwischenereignissen

Damit kann man die BPMN Zwischenereignisse in PASS States umwandeln und die BPMN Tasks in PASS Transitions. Obwohl dies die korrekte Modellierung wäre, wurde für diese Arbeit entschieden, dass man dies nicht unterstützt. Zum einen liegt das daran, dass diese Modellierungsart nicht intuitiv und auch unnötig kompliziert für die Modellierer ist. Außerdem ist dies zwar mit einfachen Prozessen möglich, sobald mehr Elemente dazu modelliert werden, wird auch die Übersetzung schwieriger. Schlussendlich liegt dies aber daran, dass diese Modelierungsart erst entstanden ist, nachdem die praktische Umwandlung abgeschlossen wurde.

Bei einem nachfolgenden Projekt kann aber durchaus überlegt werden, ob die Übersetzungssoftware auf diese Prozesse angepasst wird.

5 Praktische Umsetzung

Eines der Ziele dieser Arbeit ist die praktische Umsetzung einer Übersetzungssoftware, die BPMN Modelle in PASS Modelle umwandelt. In diesem Kapitel geht es darum, zu beschreiben, wie dieser Teil umgesetzt wurde. Außerdem wird gezeigt, wie überprüft wurde, dass beim Übersetzen keine wichtigen Prozessinformationen verloren gegangen sind. Hierfür werden die Modelle auf Vollständigkeit geprüft. Danach werden die Modelle noch auf deren Richtigkeit überprüft. Am Ende wird noch gezeigt, welche Prozesse übersetzt werden können und welche nicht.

5.1 Übersetzungsprozess

Die Übersetzungssoftware wurde mit C# und Microsoft Visual Studio umgesetzt. Die vollständige Umsetzung befindet sich auf GitHub unter:

https://github.com/Urb4nOutl4w/MasterThesis

Um das Projekt nutzen zu können, muss zuerst folgende Ordnerstruktur angelegt werden:

- BPMN
  - Input
  - Clean
  - Reverse
- PASS
  - Output

Danach muss in der `Program.cs` Datei der Pfad zur Ordnerstruktur angegeben werden. Alle BPMN Dateien, die übersetzt werden sollen, müssen anschließend in den `BPMN/Input` Ordner gelegt werden.

Die Software exportiert beim Ausführen alle Dateien aus diesem Ordner die mit `.bpmn` enden. Anschließend werden die Dateien bereinigt. Alle `extensionElements` werden gelöscht und der `
` Ausdruck, welcher für einen Zeilenumbruch steht, durch eine Leertaste ersetzt. Auch der `xsi:type="tFormalExpression"` Ausdruck wird gelöscht, da dieser nicht gelesen werden kann. Danach werden die bereinigten Dateien wieder im XML Format in den `BPMN/Clean` Ordner exportiert.

Die Software beinhaltet einen Klassenbaum, der aus allen BPMN Elementen, die übersetzt werden können, besteht und gleich aufgebaut ist wie die XML Datei. Mit diesem Klassenbaum und einem `XML-Deserializer` wird aus der bereinigten Datei ein BPMN Objekt erzeugt. Anschließend wird dieses Objekt wie in Kapitel 4 beschrieben in ein PASS

Objekt umgewandelt. Anschließend wird das PASS Modell mit Hilfe der `alps.net.api` [16] Library in den `PASS/Output` Ordner exportiert.

Diese Datei ist das Ergebnis der praktischen Umsetzung. Es ist eine fertige PASS Datei, die anschließend ausgeführt werden kann.

5.2 Kontrolle auf Vollständigkeit

Um feststellen zu können, ob beim Übersetzen der Modelle wichtige Informationen verloren gegangen sind, wird eine automatische Kontrollfunktion benötigt. Hierfür wird das fertige PASS Modell wieder in ein BPMN Modell zurückübersetzt. Hierfür wurden alle Schritte, die in Kapitel 4 beschrieben worden sind, rückwärts ausgeführt und anschließend wurde das dadurch entstandene Modell mit einem `XML-Serializer` im `BPMN/Reverse` Ordner gespeichert. Anschließend kann das ursprüngliche Modell mit dem zurückübersetzten Modell verglichen werden.

5.2.1 Probleme bei der Zurückübersetzung

Bei der Zurückübersetzung sind zwei große Probleme aufgetreten:

Das erste Problem bestand darin, dass es nicht möglich war, die fertigen PASS Modelle aus dem `PASS/Output` Ordner zu importieren. Die verwendete `alps.net.api` [16] Library war zum Zeitpunkt der Fertigstellung dieser Arbeit nicht in der Lage, die Modelle ohne Fehler zu importieren.

Um dieses Problem zu lösen, wurde das PASS Modell, welches genutzt wurde um die fertige Datei zu exportieren, direkt wieder zurückübersetzt. Das bedeutet beim Testen der Modelle auf Vollständigkeit, wird der Schritt, bei dem das PASS Modell exportiert wird und anschließend wieder importiert wird, übersprungen.

Das zweite Problem bestand darin, dass beim Übersetzungsprozess einige Informationen des Originalmodells verloren gingen, da sie im PASS Modell nicht mehr gebraucht werden. Daher kann das zurückübersetzte Modell nicht einfach mit dem Originalmodell verglichen werden.

Um dieses Problem zu lösen, wurde eine eigene Vergleichsfunktion für die beiden Modelle geschrieben. Dabei werden nicht die gesamten Modelle verglichen, sondern nur ausgewählte Objekte und Attribute. Hierfür wurde eine eigene Liste mit Vergleichskriterien erstellt.

5.2.2 Vergleichskriterien

Tabelle 5.1 zeigt alle Vergleichskriterien, die zwischen zwei Modellen berücksichtigt werden, um festzustellen ob die Modelle identisch sind. Die linke Spalte zeigt ein BPMN Element und die rechte Spalte zeigt alle Attribute dieses Elementes, die gleich sein müssen. Dabei sind fast alle Elemente ineinander verschachtelt. So muss bei den

`Definitions` etwa die `Collaboration` gleich sein und bei dieser wiederum die Liste der `Participants` und darin wiederum die `ID`, der `Name` und die `ProcessRef`.

BPMN Component	**Elemente die gleich sein müssen**
Definitions	• Collaboration • Liste der Processes
Collaboration	• Liste der Participants • Liste der MessageFlows
Participant	• Id • Name • ProcessRef
MessageFlow	• Id • Name • SourceRef • TargetRef
Process	• Id • Name • LaneSet • StartEvent • Liste der Tasks • Liste der ExclusiveGateways • Liste der IntermediateThrowEvents • Liste der EndEvents • Liste der SequenceFlows • Liste der IntermediateCatchEvents • Liste der EventBasedGateways
LaneSet	• Lane
Lane	• Liste der FlowNodeRefs
StartEvent	• Id • Name • Outgoing
Task	• Id • Name • Outgoing • Liste der Incomings
ExclusiveGateway	• Id • Name • Liste der Incomings • Liste der Outgoings
IntermediateThrowEvent	• Id • Name • Outgoing • Liste der Incomings
EndEvent	• Id • Name • Liste der Incomings

SequenceFlow	• Id • SourceRef • TargetRef
IntermediateCatchEvent	• Id • Name • Incoming • Outgoing • TimeDuration
TimeDuration	• Text
EventBasedGateway	• Id • Liste der Incomings • Liste der Outgoings

Tabelle 5.1: Vergleichskriterien bei der Kontrolle der Modelle

Alle Modelle die übersetzt werden, werden automatisch zurückübersetzt und danach auf Vollständigkeit überprüft. Dadurch wurden bei der Umsetzung kleinere Fehler entdeckt und die Überstzungssoftware weiter verbessert.

Wenn die Software ausgeführt wird, öffnet sich ein Terminal und darin werden alle Dateinamen der zu übersetzenden Modelle aufgelistet. Nach dem jeweiligen Namen erscheint dann entweder `true`, wenn die Überprüfung vollständig war, oder `false`, wenn sie gescheitert ist.

5.3 Kontrolle auf Richtigkeit

Nachdem die Modelle übersetzt wurden, müssen sie noch auf Richtigkeit kontrolliert werden. Hierfür wurde die Arbeit von Herrn Zeisler [1] verwendet. Mit dieser Arbeit können PASS Modelle ausgeführt werden. Die Modelle wurden damit ausgeführt, wobei keine Fehler auftraten. Damit wurde bewiesen, dass die fertigen Modelle korrekte PASS Modelle sind.

5.4 Mächtigkeit der übersetzbaren Modelle

Am Ende der praktischen Umsetzung wird noch geklärt, welche Modelle übersetzt werden können und welche nicht. Da unendlich viele Modelle erstellt werden können, ist schwer messbar wie viele Modelle mit der Software dieser Arbeit übersetzbar sind. Darum wird hierbei auf die Service Interaction Patterns zurückgegriffen. Dies ist eine Sammlung von Problemstellungen, die mit Hilfe von Prozessmodellen dargestellt werden können. Sie wurden in der Arbeit [17] definiert und dienen dazu, unterschiedliche Modellierungsmethoden miteinander zu vergleichen.

Hierfür werden alle Service Interaction Patterns in BPMN modelliert und es wird getestet, welche von ihnen mit der Software übersetzt werden können. Es wird auch geklärt

ob es Probleme bei den Patterns gibt und ob man sie ausführen kann. Als Inspiration für die BPMN Modelle dient die Arbeit von Herrn Raß [18]. Zum Teil werden sie sehr ähnlich modelliert, teilweise werden ein paar Details verändert und in manchen Fällen werden die BPMN Modelle von Grund auf neu modelliert.

Neben den BPMN Modellen werden auch die entsprechenden, übersetzten PASS Modelle abgebildet. Da es zum Zeitpunkt der Verfassung dieser Arbeit noch keine Möglichkeit gibt, PASS OWL Modelle einzulesen und graphisch darzustellen, mussten auch diese Modelle neu gezeichnet werden. Um den Aufwand diese Vorgangs zu verringern wurden hierfür die PASS Modelle von Herrn Zeisler [1] hergenommen, die ebenfalls die Service Interaction Patterns abbilden, und so ummodelliert, dass sie den übersetzten Modellen entsprechen.

In jedem der folgenden Kapitel wird zuerst ein Zitat gezeigt, welches das Pattern beschreibt. Darunter wird das zugehörige BPMN Modell gezeigt und es wird beschrieben, ob es übersetzbar ist. Danach wird das übersetzte PASS Modell abgebildet. Hierbei wird auch beschrieben ob das Modell ausführbar ist.

5.4.1 Single-transmission bilateral interaction patterns

Da das `Send/Receive` Pattern sowohl das `Send` als auch das `Receive` Pattern abbildet, werden die ersten beiden Patterns nicht modelliert und getestet.

5.4.1.1 Send/Receive

> “ A party X engages in two causally related interactions: in the first interaction X sends a message to another party Y (the request), while in the second one X receives a message from Y (the response). ” [17]

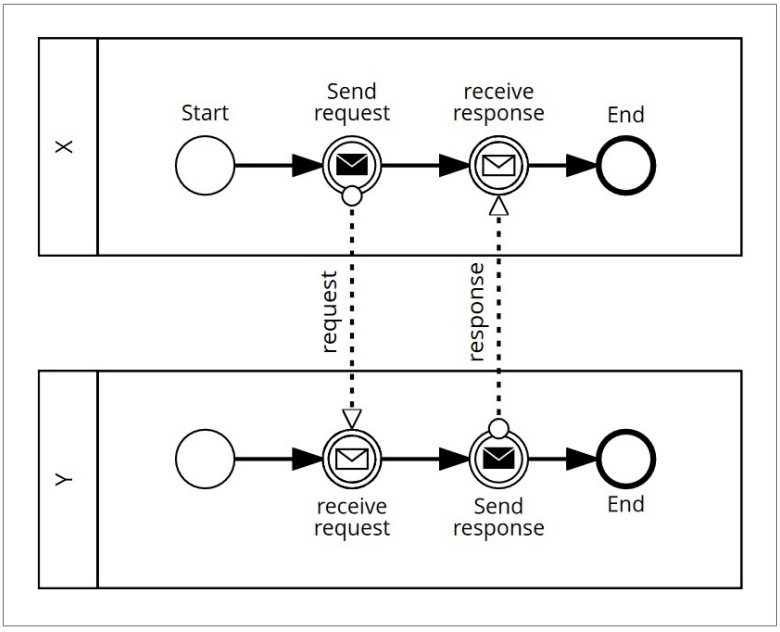

Abbildung 5.1: BPMN Send/Receive Pattern

Das Pattern aus Abbildung 5.1 kann übersetzt werden.

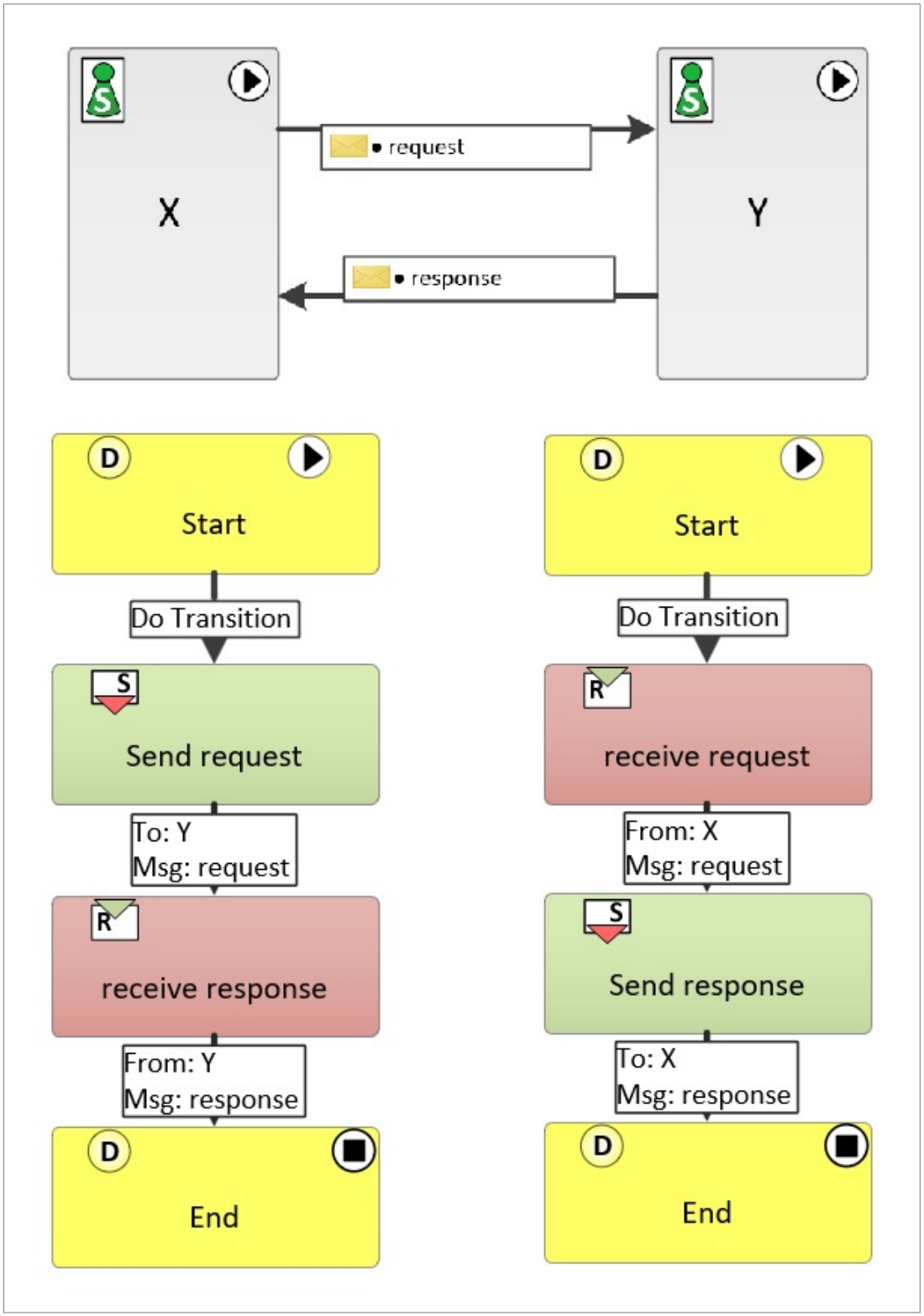

Abbildung 5.2: PASS Send/Receive Pattern (modifiziert von [1])

Das Modell aus Abbildung 5.2 kann ohne Probleme ausgeführt werden.

5.4.2 Single-transmission multilateral interaction patterns

5.4.2.1 Racing incoming messages

> " A party expects to receive one among a set of messages. These messages may be structurally different (i.e. different types) and may come from different categories of partners. The way a message is processed depends on its type and/or the category of partner from which it comes. " [17]

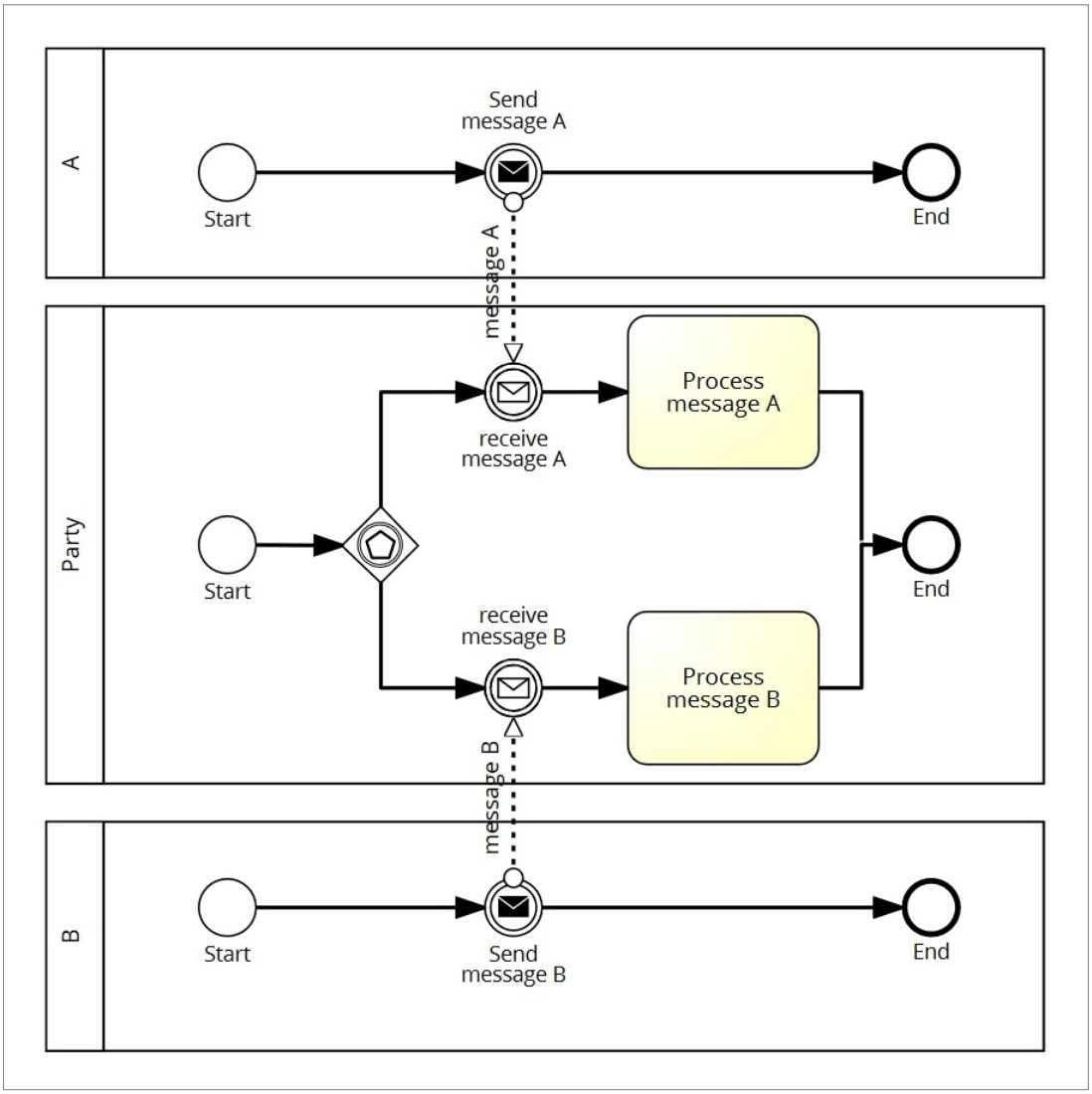

Abbildung 5.3: BPMN Racing incoming messages Pattern

Das Pattern aus Abbildung 5.3 kann übersetzt werden.

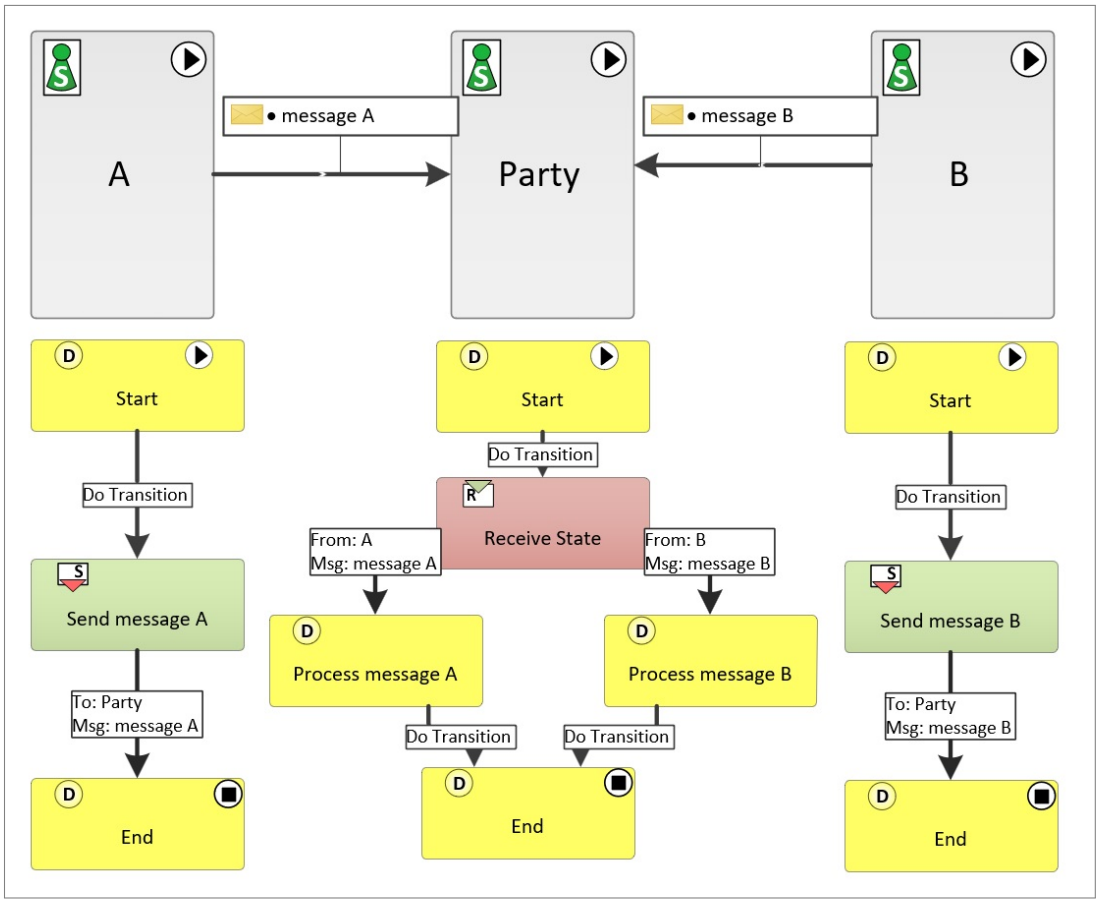

Abbildung 5.4: PASS Racing incoming messages Pattern (modifiziert von [1])

Das Modell aus Abbildung 5.4 kann ohne Probleme ausgeführt werden.

5.4.2.2 One-to-many send/receive

Da das `One-to-many send/receive` Pattern sowohl das `One-to-many send` als auch das `One-from-many receive` Pattern abbildet, werden diese beiden Patterns nicht modelliert und getestet.

> “ A party sends a request to several other parties, which may all be identical or logically related. Responses are expected within a given timeframe. However, some responses may not arrive within the timeframe and some parties may even not respond at all. The interaction may complete successfully or not depending on the set of responses gathered. ” [17]

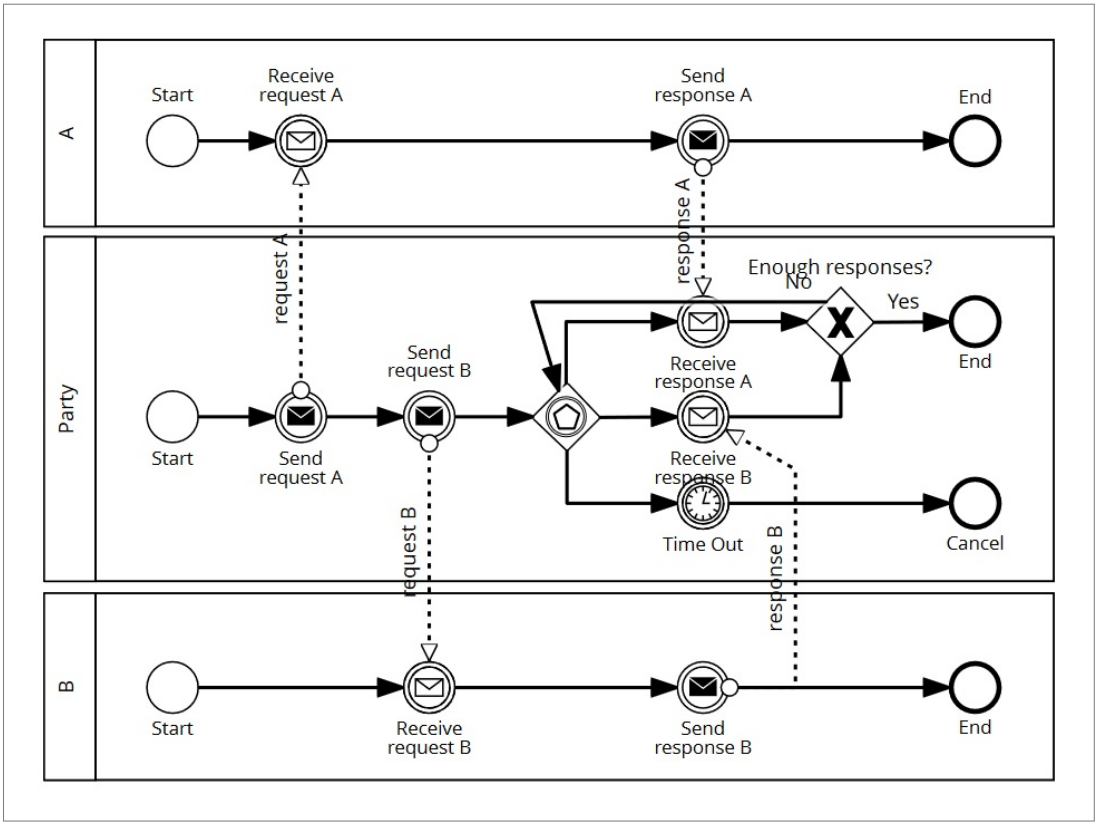

Abbildung 5.5: BPMN One-to-many send/receive Pattern

Das Pattern aus Abbildung 5.5 kann übersetzt werden.

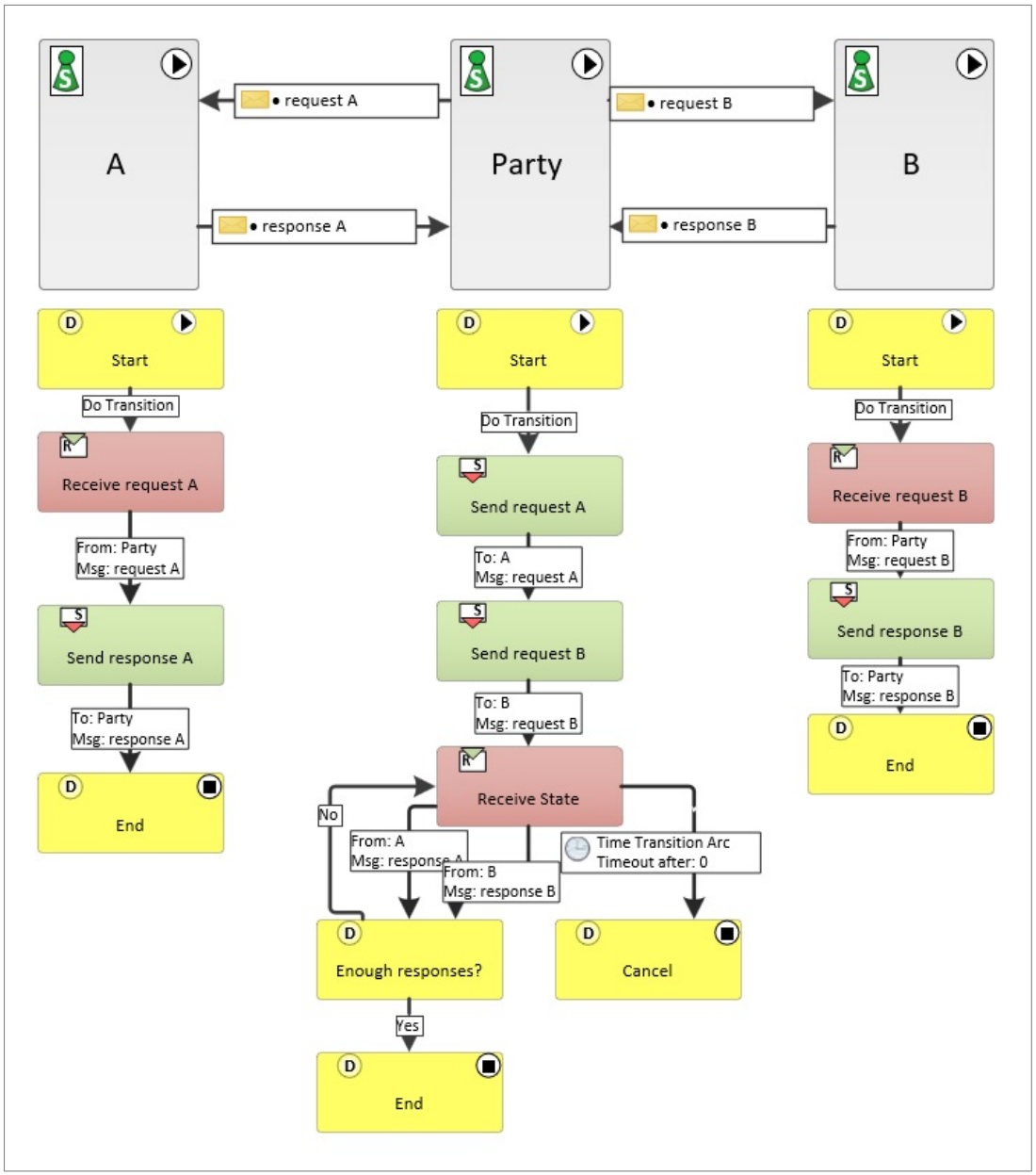

Abbildung 5.6: PASS One-to-many send/receive Pattern (modifiziert von [1])

Das Modell aus Abbildung 5.6 kann ohne Probleme ausgeführt werden.

5.4.3 Multi-transmission interaction patterns

5.4.3.1 Multi-responses

> “ A party X sends a request to another party Y. Subsequently, X receives any number of responses from Y until no further responses are required. The trigger of no further responses can arise from a temporal condition or message content, and can arise from either X or Y’s side. Responses are no longer expected from Y after one or a combination of the following events: (i) X sends a notification to stop; (ii) a relative or absolute deadline indicated by X; (iii) an interval of inactivity during which X does not receive any response from Y; (iv) a message from Y indicating to X that no further responses will follow. From this point on, no further messages from Y will be accepted by X. ” [17]

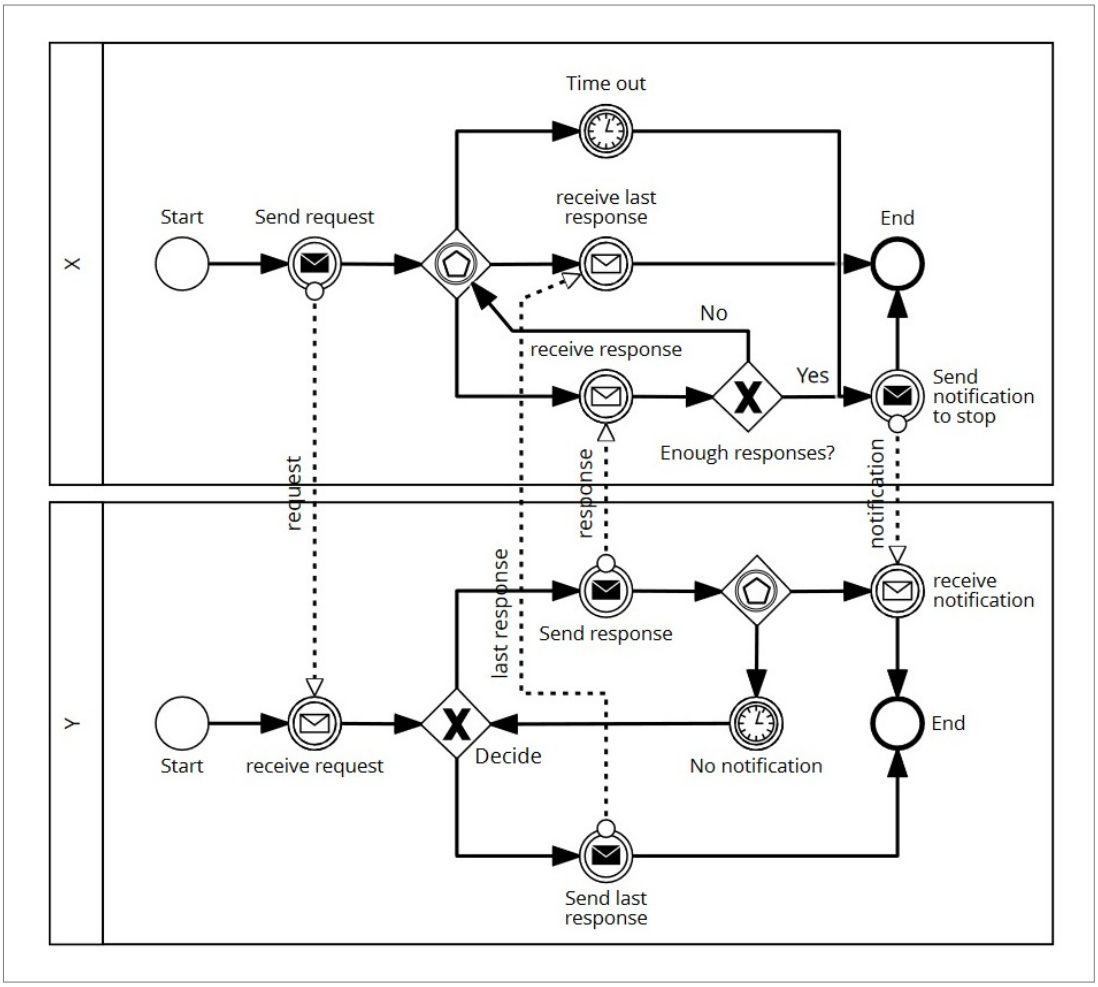

Abbildung 5.7: BPMN Multi-responses Pattern

Das Pattern aus Abbildung 5.7 kann übersetzt werden.

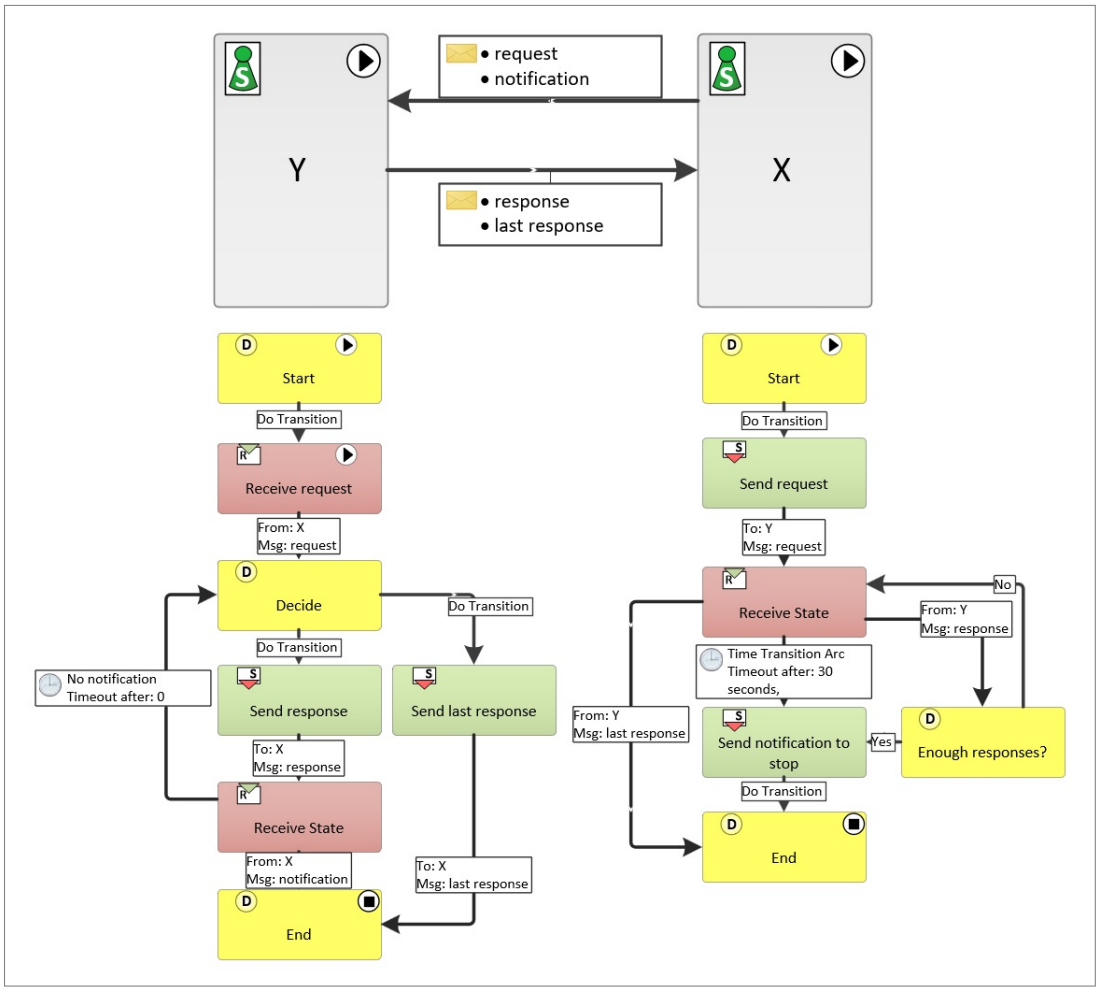

Abbildung 5.8: PASS Multi-responses Pattern (modifiziert von [1])

Das Modell aus Abbildung 5.8 kann ohne Probleme ausgeführt werden.

5.4.3.2 Contingent requests

> “ A party X makes a request to another party Y. If X does not receive a response within a certain timeframe, X alternatively sends a request to another party Z, and so on. ” [17]

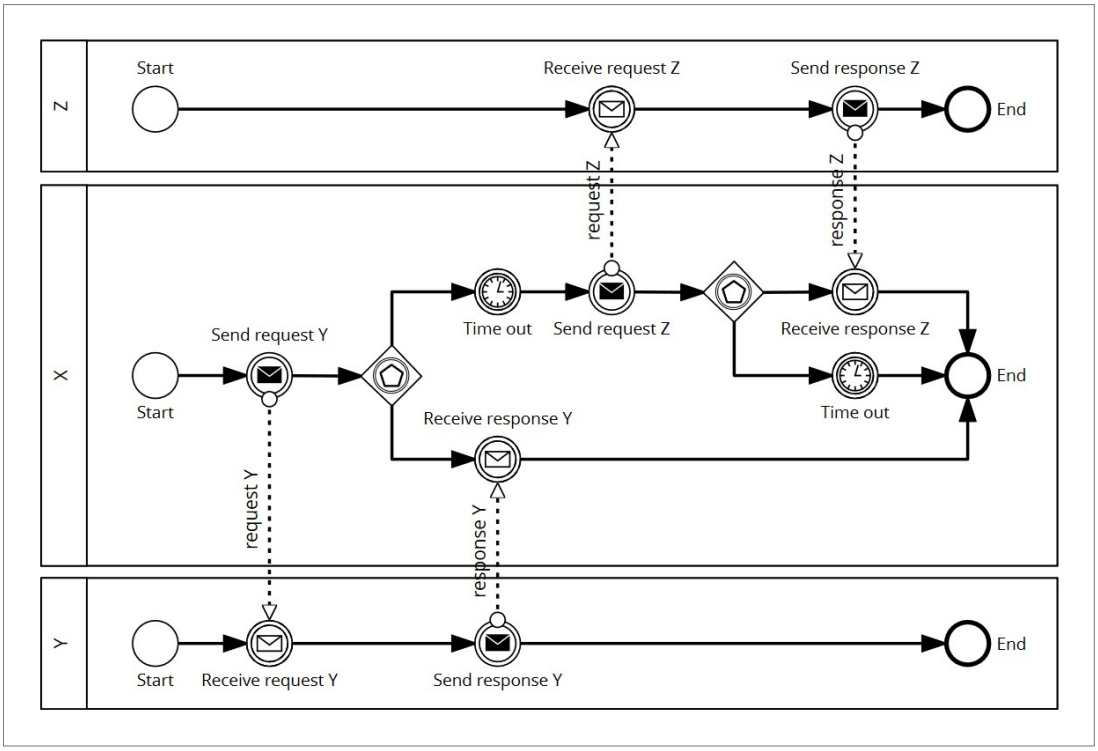

Abbildung 5.9: BPMN Contingent requests Pattern

Das Pattern aus Abbildung 5.9 kann übersetzt werden.

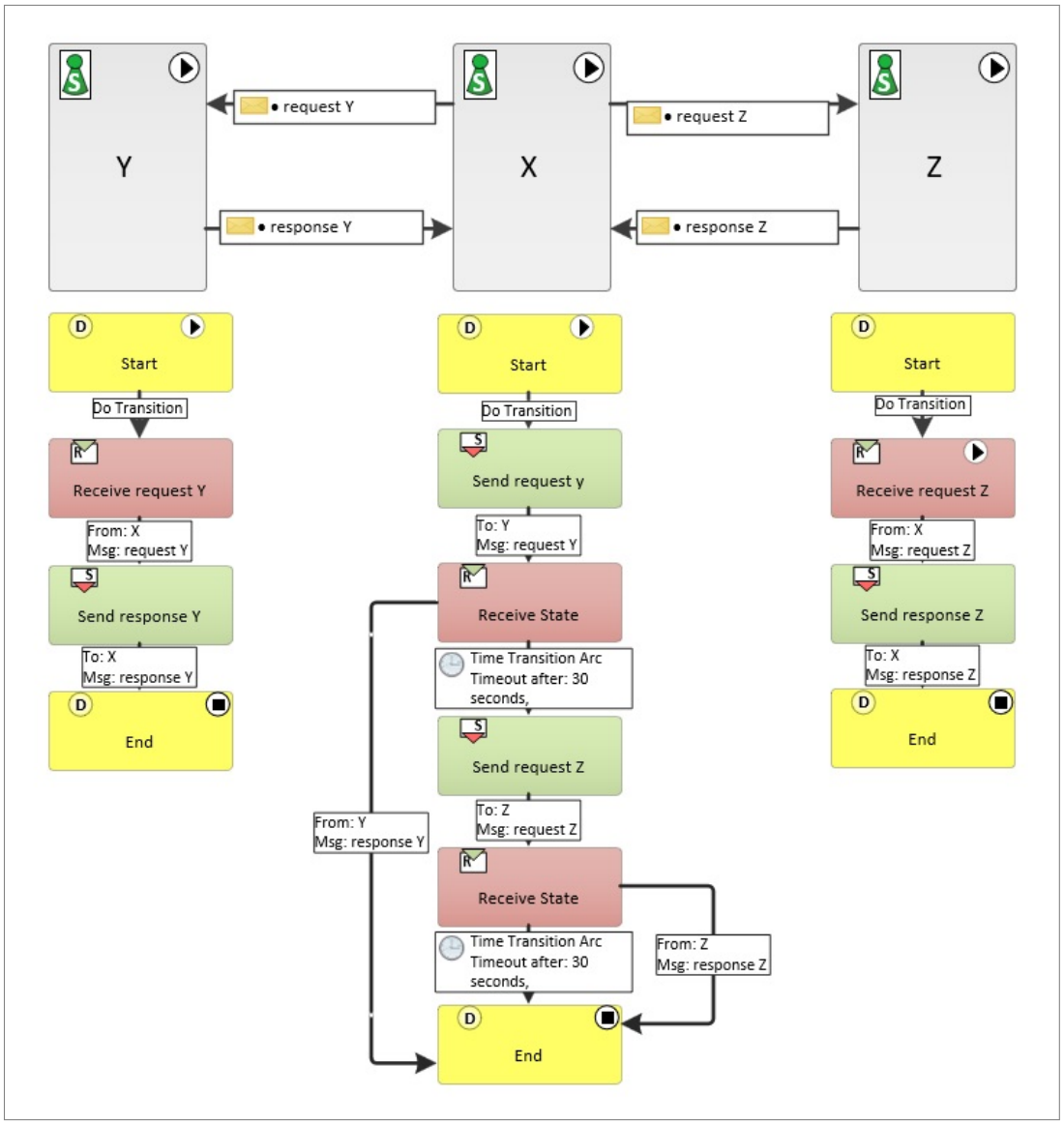

Abbildung 5.10: PASS Contingent requests Pattern (modifiziert von [1])

Das Modell aus Abbildung 5.10 kann ausgeführt werden. Jedoch kann bei der Ausführung ein Problem auftreten: Alle drei Subjekte werden direkt gestartet, aber es kann vorkommen, dass Z nie das Ende erreicht. Dies tritt auf, wenn Y rechtzeitig die *response Y* schickt, dann wir der Prozess von X beendet und Z erhält nie den *request Z*.

5.4.3.3 Atomic multicast notification

> “ A party sends notifications to several parties such that a certain number of parties are required to accept the notification within a certain time-frame. For example, all parties or just one party are required to accept the notification. In general, the constraint for successful notification applies over a range between a minimum and maximum number. ” [17]

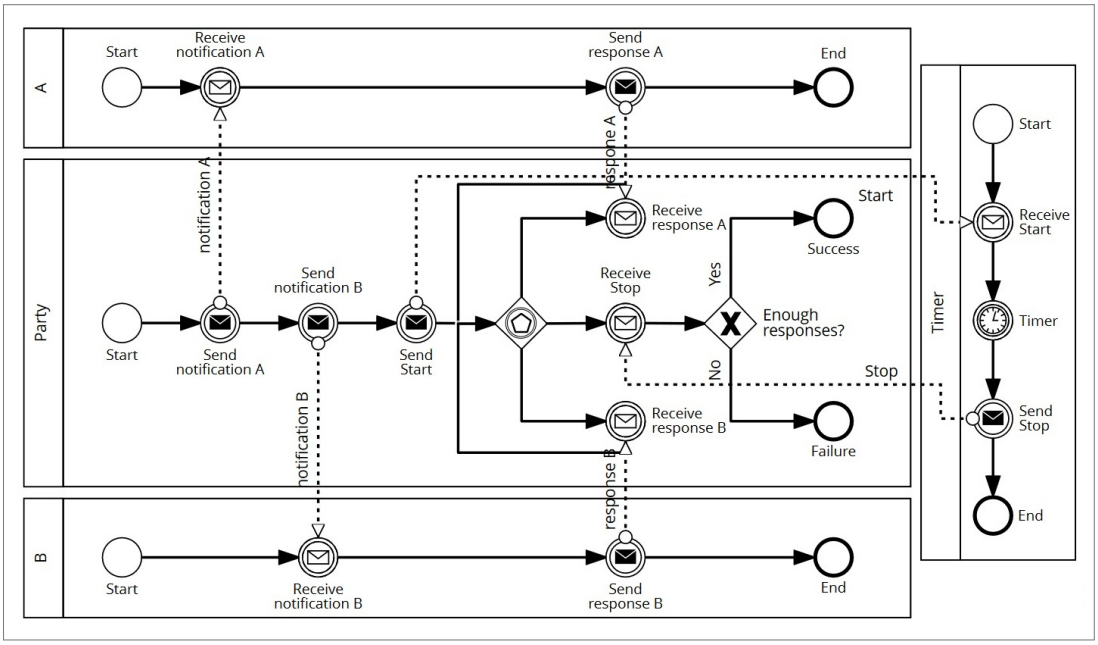

Abbildung 5.11: BPMN Atomic multicast notification Pattern

Das Pattern aus Abbildung 5.11 kann übersetzt werden.

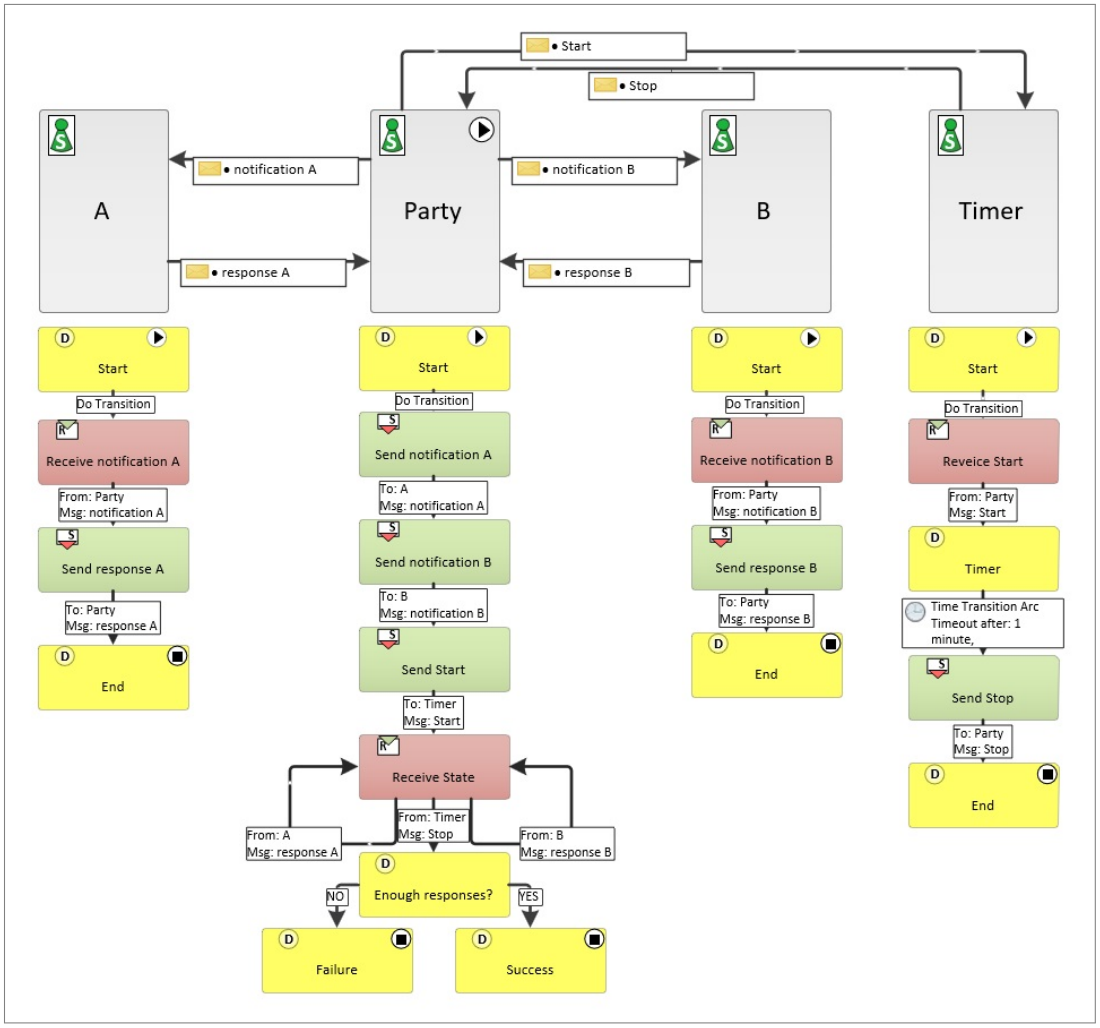

Abbildung 5.12: PASS Atomic multicast notification Pattern (modifiziert von [1])

Das Modell aus Abbildung 5.12 kann ohne Probleme ausgeführt werden.

5.4.4 Routing patterns

5.4.4.1 Request with referral

> “ Party A sends a request to party B indicating that any follow-up response should be sent to a number of other parties (P1, P2, ..., Pn) depending on the evaluation of certain conditions. While faults are sent by default to these parties, they could alternatively be sent to another nominated party (which may be party A). ” [17]

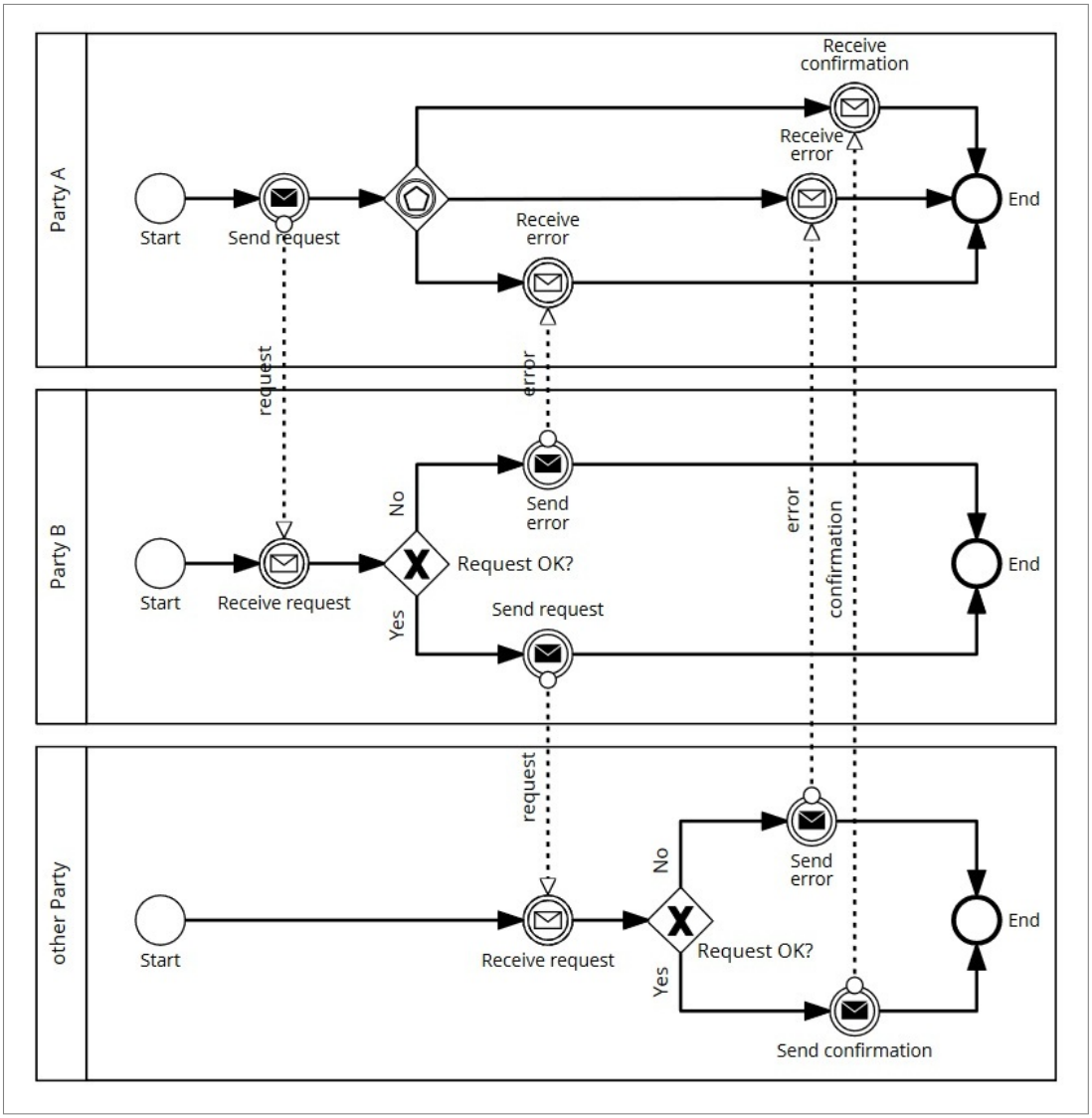

Abbildung 5.13: BPMN Request with referral Pattern

Das Pattern aus Abbildung 5.13 kann übersetzt werden.

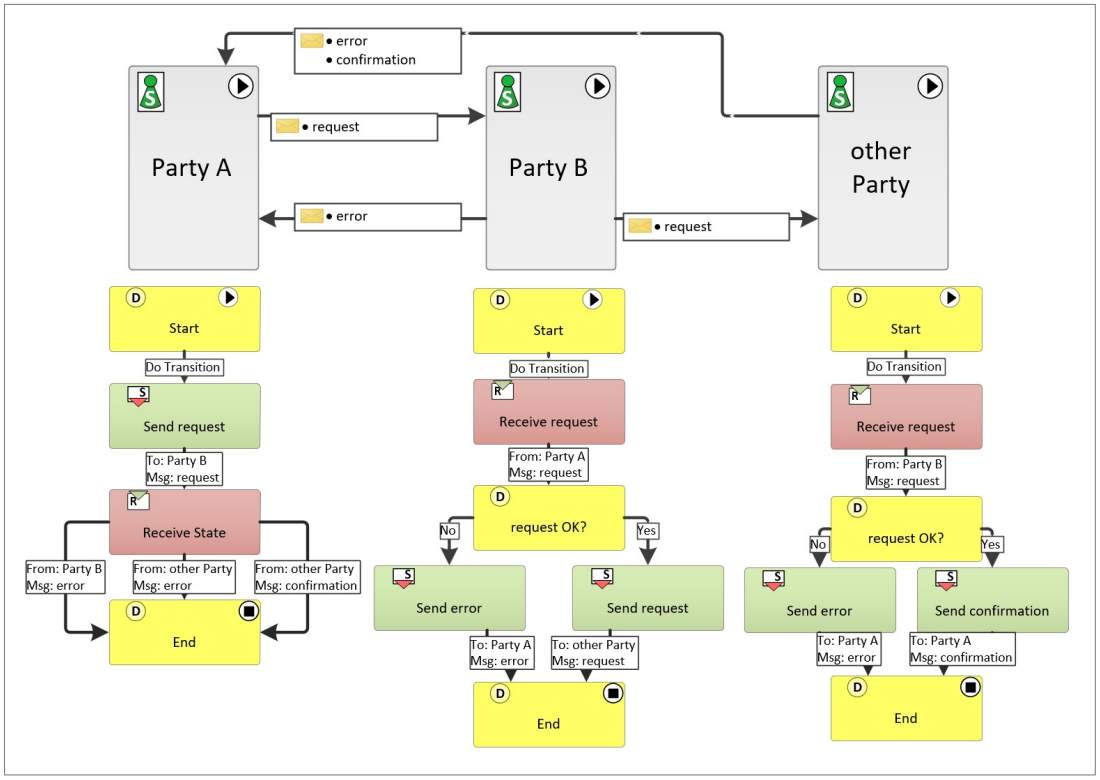

Abbildung 5.14: PASS Request with referral Pattern (modifiziert von [1])

Das Modell aus Abbildung 5.14 kann ausgeführt werden. Jedoch kann bei der Ausführung ein Problem auftreten: Alle drei Subjekte werden direkt gestartet, aber es kann vorkommen, dass die `other Party` nie das Ende erreicht. Dies tritt auf, wenn `Party B` einen Fehler im *request* findet und damit diesen nie an die `other Party` weitergibt.

5.4.4.2 Relayed request

> “ Party A makes a request to party B which delegates the request to other parties (P1, ..., Pn). Parties P1, ..., Pn then continue interactions with party A while party B observes a ”view” of the interactions including faults. The interacting parties are aware of this ”view” (as part of the condition to interact). ” [17]

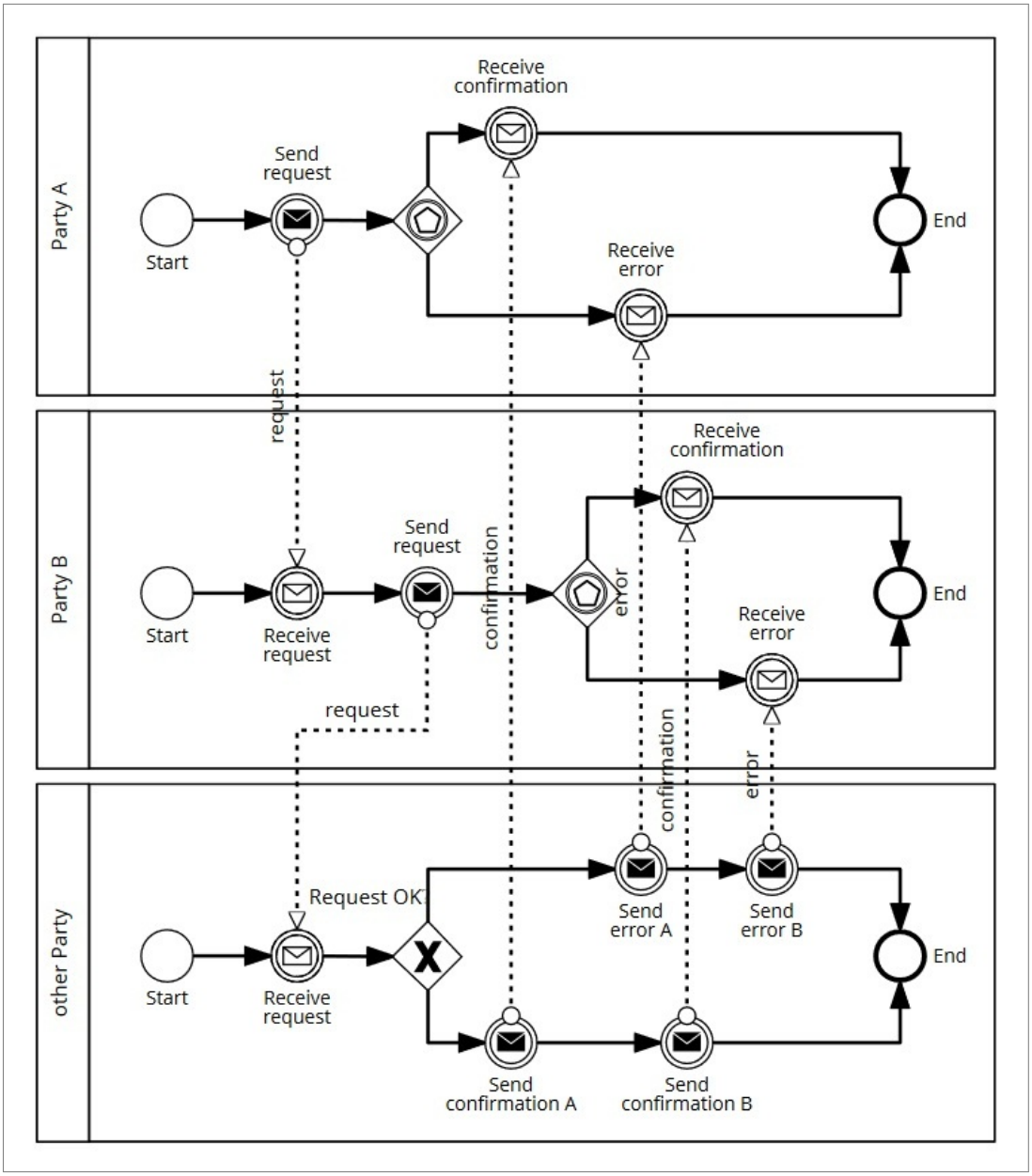

Abbildung 5.15: BPMN Relayed request Pattern

Das Pattern aus Abbildung 5.15 kann übersetzt werden.

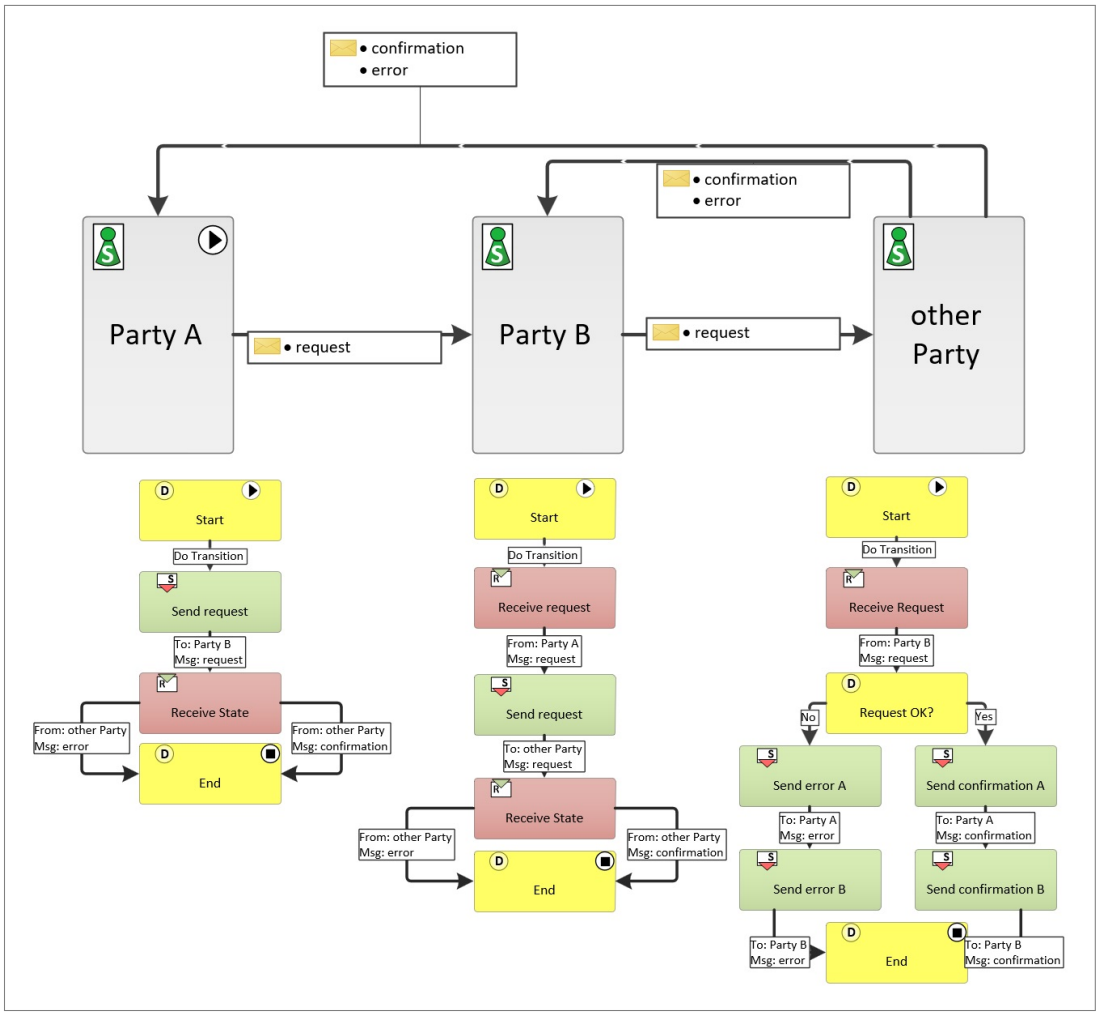

Abbildung 5.16: PASS Relayed request Pattern (modifiziert von [1])

Das Modell aus Abbildung 5.16 kann ohne Probleme ausgeführt werden.

5.4.4.3 Dynamic routing

> “ A request is required to be routed to several parties based on a routing condition. The routing order is flexible and more than one party can be activated to receive a request. When the parties that were issued the request have completed, the next set of parties are passed the request. Routing can be subject to dynamic conditions based on data contained in the original request or obtained in one of the ”intermediate steps”. ” [17]

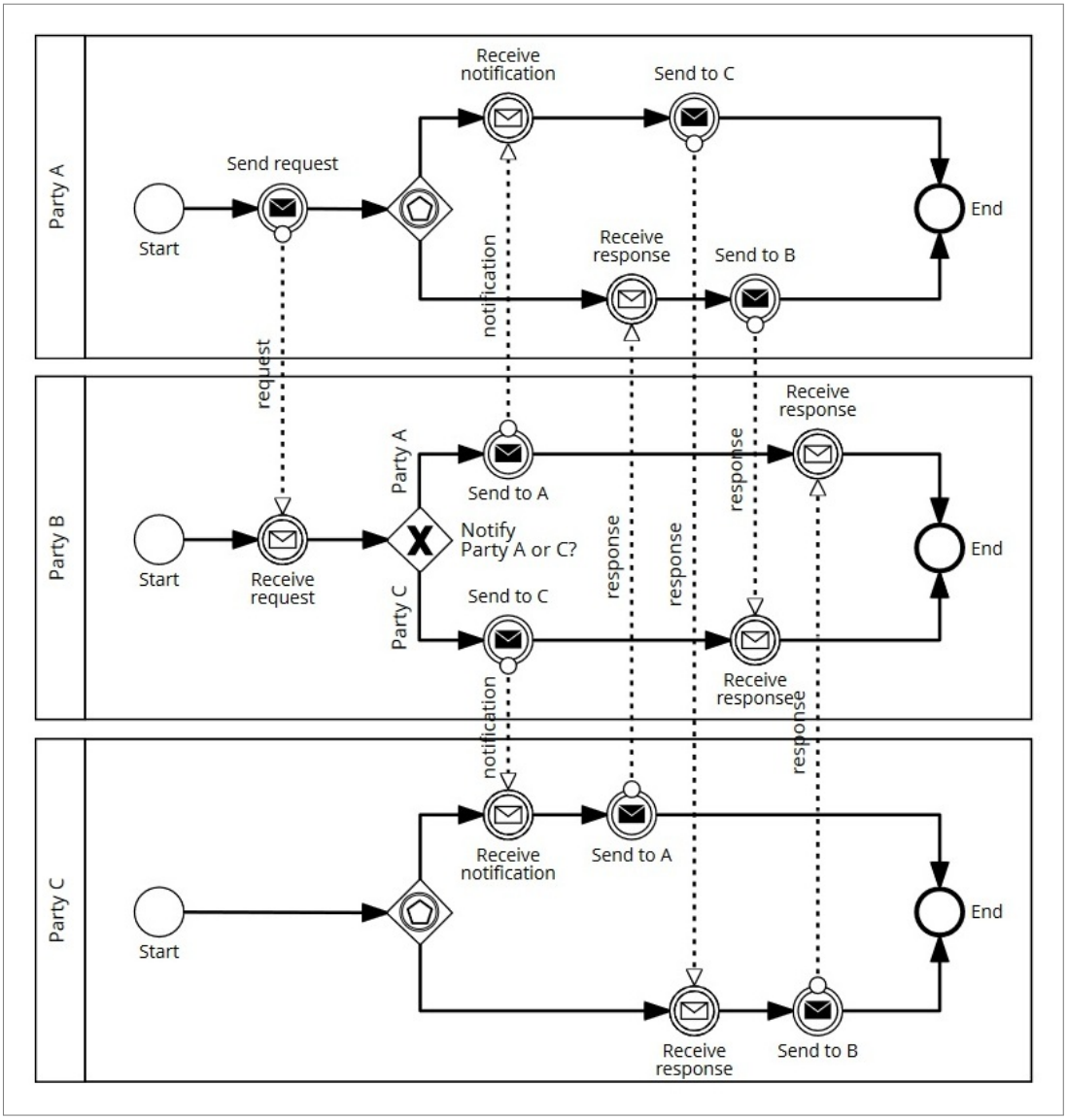

Abbildung 5.17: BPMN Dynamic routing Pattern

Das Pattern aus Abbildung 5.17 kann übersetzt werden.

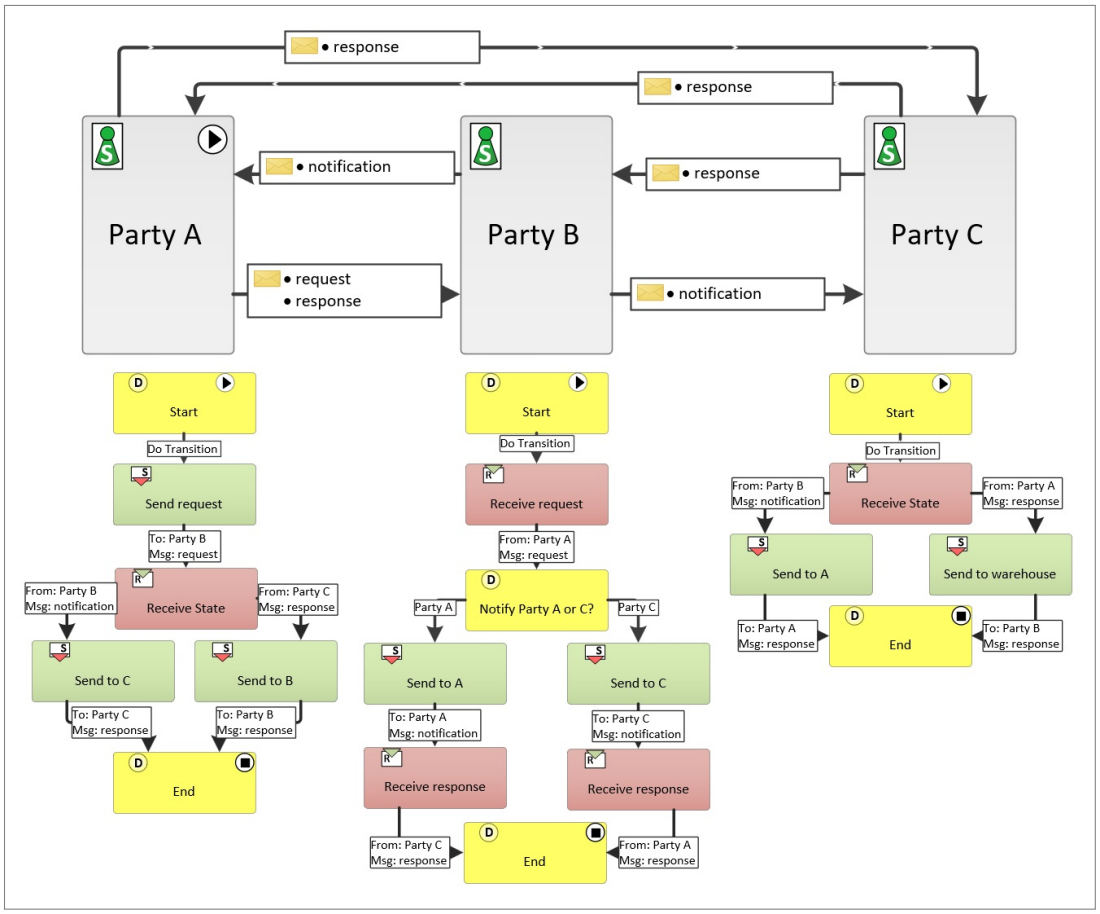

Abbildung 5.18: PASS Dynamic routing Pattern (modifiziert von [1])

Das Modell aus Abbildung 5.18 kann ohne Probleme ausgeführt werden.

5.4.5 Zusammenfassung

Um die Mächtigkeit der übersetzbaren Modelle zusammenzufassen wurde Tabelle 5.2 erstellt. Darin sieht man welche Service Interaction Patterns übersetzbar und ausführbar sind.

Service Interaction Pattern	**Übersetzbar**	**Ausführbar**
Send Receive	Ja	Ja
Racing incoming messages	Ja	Ja
One-to-many send/receive	Ja	Ja
Multi-responses	Ja	Ja
Contingent requests	Ja	Teilweise
Atomic multicast notification	Ja	Ja
Request with referral	Ja	Teilweise
Relayed request	Ja	Ja
Dynamic routing	Ja	Ja

Tabelle 5.2: Zusammenfassung der Service Interaction Patterns

Die Tabelle zeigt, dass alle Service Interaction Patterns übersetzbar sind. Dies liegt unter anderem daran, dass die Modelle entsprechend dem Modellierungsleitfaden erstellt wurden.

Jedoch sind nur sieben der neun Patterns ohne Probleme ausführbar. Dies liegt daran, dass es in den betroffenen Modellen Pfade gibt, die nicht immer abgeschlossen werden können. In manchen Fällen wird das Modell ohne Probleme ausgeführt, in anderen Fällen jedoch läuft das Modell in eine Sackgasse und kann nicht beendet werden.

In beiden Fällen kommt dieses Problem dadurch zustande, dass in übersetzten Modellen alle Subjekte gleichzeitig gestartet werden. In einigen Modellen kann es jedoch vorkommen, dass nicht immer alle Subjekte für die Ausführung benötigt werden. Daher werden die Prozesse dieser Subjekte nicht richtig abgeschlossen. Dies ist ein Implementierungsproblem, welches in einem zukünftigen Projekt behoben werden kann.

5.5 Nicht übersetzbare Elemente

In diesem Kapitel wird erklärt welche Elemente nicht übersetzbar sind und die Ursachen dafür. Es wird gezeigt ob es sich dabei um konzeptionelle Probleme handelt, die aufgrund der unterschiedlichen Konzepte von PASS und BPMN auftreten, oder ob es sich um Implemetierungsprobleme handelt, die in der umgesetzten Software enthalten sind, jedoch in zukünftigen Projekten verbessert werden können.

5.5.1 Konzeptionelle Probleme

Das größte und, soweit bekannt, auch das einzige konzeptionelle Problem bei der Übersetzung von BPMN in PASS sind Aufgaben, die parallel ausgeführt werden. Bei einem BPMN Modell gibt es ein `Parallel Gateway`, welches den Prozess in mehrere Pfade aufteilen kann, die gleichzeitig ausgeführt werden. Dies ist bei PASS Modellen nicht möglich. Hier muss sich ein Subjekt immer in genau einem State befinden.

Dieses Problem kann man beim Modellieren auf mehrere Arten umgehen. Aufgaben die parallel ausgeführt werden sollen, können sequenziell ausgeführt werden. In vielen Fällen werden die Aufgaben die parallel modelliert werden, auch in der Praxis sequentiell ausgeführt. Eine weitere Möglichkeit um Aufgaben parallel auszuführen ist das Erstellen eines Hilfssubjekts. Zwei Subjekte können zeitgleich in zwei verschiedenen States sein. Vor allem wenn man das Ablaufen eines Timers modellieren will, bietet sich diese Möglichkeit an. Das Subjekt und das Hilfssubjekt können dann Nachrichten austauschen, um den Prozess wie gewünscht zu steuern.

Da dies ein konzeptionelles Problem ist, kann es auch nicht in weiteren Arbeiten behoben werden.

5.5.2 Implementierungsprobleme

5.5.2.1 MultiSubjects

In der Übersetzungssoftware werden keine `MultiSubjects` unterstützt.

In BPMN Modellen kann man Pools (`Participants`) als Mehrfachinstanz modellieren. Ursprünglich war geplant, dass solche *Participants* zu `MultiSubjects` übersetzt werden. Bei der Ausführung der übersetzten Modelle stellte sich jedoch heraus, dass es nicht ausreicht, ein Subjekt in PASS Modellen zu einem `MultiSubject` umzuwandeln. Auch der gesamte Nachrichtenfluss des Subjekts muss geändert werden. Da zu diesem Zeitpunkt die Übersetzung des Nachrichtenflusses bereits umgesetzt war und die Neukonzeptionierung und Änderung der Übersetzung sehr aufwändig gewesen wäre, wurde entschieden, `MultiSubjects` nicht zu unterstützen.

Dieses Problem kann in weiteren Arbeiten behoben werden, indem der Nachrichtenfluss anders übersetzt wird.

5.5.2.2 Hinterlegte Logik

In der Übersetzungssoftware wird keine hinterlegte Logik unterstützt.

In BPMN Modellen kann man bei `ExclusiveGateways` eine Logik hinterlegen. So kann zum Beispiel ein Wert auf eine Bedingung überprüft werden und somit einer von zwei Wegen beschritten werden. Dies ist vor allem für die automatische Ausführung von BPMN Modellen wichtig. Bei der Umsetzung der Übersetzungssoftware wurde dieses Feature nicht berücksichtigt. Das bedeutet, dass diese Informationen zwar keinen Fehler auslösen, aber auch nicht richtig übersetzt werden, sodass sie bei der Ausführung des PASS Modells nicht automatisch mitausgeführt werden.

Dieses Problem kann in weiteren Arbeiten behoben werden, indem ein äquivalentes Feature in PASS gefunden wird und danach die Übersetzungslogik von `ExclusiveGateways` dementsprechend angepasst wird.

5.5.2.3 Zusammenführende Gateways

In der Übersetzungssoftware werden keine zusammenführenden Gateways unterstützt.

In BPMN Modellen werden Gateways dafür benutzt, um das Modell in mehrere Pfade aufzuteilen. Man kann die Gateways jedoch auch dafür nutzen, um mehrere Pfade wieder zu einem zusammenzuführen. Dies ist optional und kann beim Modellieren von `ExclusiveGateways` und `EventBasedGateways` weggelassen werden. Bei der Umsetzung der Übersetzungssoftware wurden nur die Gateways berücksichtigt, die den Prozess in mehrere Pfade aufteilen. Das bedeutet, dass Fehler beim Übersetzen auftauchen, wenn man zusammenführende Gateways modelliert.

Dieses Problem kann in weiteren Arbeiten behoben werden, indem beim Übersetzungsprozess auf die zusammenführenden Gateways eingegangen wird und diese entweder weggelassen oder richtig übersetzt werden.

5.5.2.4 Nachrichten Start Ereignisse

In der Übersetzungssoftware werden keine `MessageStartEvents` unterstützt.

In BPMN Modellen werden `MessageStartEvents` dafür benutzt, um Prozesse beim Erhalt einer Nachricht zu starten. Man kann dies jedoch auch durch ein `StartEvent` und ein `IntermediateCatchEvent` modellieren. Bei der Umsetzung der Übersetzungssoftware wurden deshalb `MessageStartEvents` nicht berücksichtigt. Bei der Übersetzung gibt es hierbei auch keine Probleme, jedoch kann es dadurch bei der Ausführung der PASS Modelle zu einem Fehler kommen. Es werden durch das `StartEvent` alle Subjekte gleichzeitig gestartet und in manchen Fällen kann es vorkommen, dass der Prozess eines Subjects nicht starten hätte sollen und damit auch nicht beendet werden kann. Darum sind auch nicht alle übersetzten Service Interaction Patterns ohne Probleme ausführbar.

Dieses Problem kann in weiteren Arbeiten behoben werden, indem beim Übersetzungsprozess auch `MessageStartEvents` erkannt und richtig übersetzt werden.

6 Conclusio

Diese Arbeit zeigt, wie die beiden Modellierungsstandards BPMN und PASS aufgebaut sind. Es wurden deren Konzepte, Unterschiede und Gemeinsamkeiten hervorgehoben. Das Ziel war es, herauszuarbeiten welche Konzepte und Elemente in beiden Standards gleich sind und wie BPMN Modelle in PASS Modelle umgewandelt werden können.

Zuerst wurde der PASS Standard analysiert und es wurde gezeigt, wie dieser aufgebaut ist. Danach wurden die wichtigsten Elemente sowohl semantisch als auch syntaktisch beschrieben. Dabei wurde auch gezeigt, warum der Standard als Ontologie aufgebaut ist.

Im nächsten Schritt wurde der BPMN Standard analysiert. Hier wurden allerdings nicht alle Elemente beschrieben, sondern nur jene, die ein Pendant im PASS Standard haben oder die mit automatisierten Änderungen eines erhalten können. Diese Elemente sind dadurch übersetzbar und wurden auch sowohl semantisch als auch syntaktisch beschrieben. Neben der Aufzählung und Beschreibung der übersetzbaren Elemente wurde noch ein Modellierungsleitfaden geschrieben, der bei jedem Element zeigt, worauf beim Modellieren geachtet werden muss, um den Prozess später in PASS umwandeln zu können.

Der nächste Teil dieser Arbeit beschäftigt sich damit, wie die BPMN Elemente automatisiert umgewandelt werden können, um PASS Elemente zu generieren. Diese Umwandlung geschieht in zwei Schritten.

Dies diente als Basis für den praktischen Teil dieser Arbeit. Dieser bestand daraus, eine Übersetzungssoftware zu schreiben, die automatisiert BPMN Modelle in PASS Modelle umwandeln kann. Es wurde genau beschrieben, wie die Software funktioniert und auch der gesamte Source Code ist einsehbar. Danach wurde die Software erweitert, um die Modelle wieder zurückumzuwandeln. Somit konnte man das ursprüngliche BPMN Modell mit dem zurückübersetzten vergleichen. Dadurch wurde gezeigt, dass beim Übersetzungsvorgang keine Elemente oder wichtigen Informationen verloren gingen.

Um festzustellen, wie viele BPMN Modelle übersetzbar sind, wurden neun Service Interaktion Patterns modelliert und übersetzt. Es wurde gezeigt, dass alle Patterns dadurch, dass sie anhand des Modellierungsleitfadens erstellt wurden, übersetzbar sind. Außerdem ließen sich sieben der neun übersetzten Modelle ohne Probleme ausführen. Die anderen beiden waren auch ausführbar, jedoch traten in manchen Fällen Probleme auf. Dies lag daran, dass bei diesen übersetzten Modellen manchmal ein Subjekt nicht das Ende des Prozesses erreichte. Dieser Fehler wäre jedoch in einer zukünftigen Arbeit behebbar.

Diese Arbeit veranschaulicht, dass BPMN Modelle in PASS Modelle übersetzt werden können. Das bedeutet aber nicht, dass jedes BPMN Modell übersetzbar ist. Dafür muss es den Regeln der Elementeinschränkung und des Modellierungsleitfadens, welche in dieser Arbeit definiert wurden, entsprechen. Bestehende Modelle, die nicht diesen Regeln entsprechen, können aber oft ummodelliert werden, sodass auch diese übersetzbar sind.

In der Arbeit von Frau Reiter [2], die ein ähnliches Ziel verfolgte, wurden die BPMN Modelle erst in eine BPMN Ontologie übersetzt und danach erst in die PASS Ontologie. In dieser Arbeit wird der Zwischenschritt übersprungen und die BPMN Modelle werden direkt in PASS Modelle übersetzt. Dies wurde zu Beginn der Arbeit beschlossen. Das Ziel hierbei war es die Komplexität zu reduzieren und Fehler, die bei der Übersetzung in eine Ontologie auftreten können, zu vermeiden. Nach Abschluss der praktischen Arbeit wurde festgestellt, dass diese Vorgangsweise gut funktioniert hat. Im Nachhinein betrachtet, würde die selbe Vorgangsweise erneut verwendet werden. Das Einzige, dass bei einer Überarbeitung der Arbeit geändert werden sollte, ist, dass die fehlenden Elemente, die in Kapitel 5.5.2 erwähnt wurden, umgesetzt werden sollten. Vor allem das `MessageStartEvent` sollte implementiert werden, sodass alle Modelle ohne Probleme ausführbar sind.

Mit Hilfe dieser Arbeit und durch die Übersetzungssoftware können ProzessmodelliererInnen, die bisher mit BPMN gearbeitet haben, die Vorteile des PASS Standards nutzen. Sie müssen sich nicht erst in eine neue Technologie einlesen, sondern können bekanntes Wissen nutzen und mit Hilfe des Modellierungsleitfadens PASS Prozesse mit BPMN modellieren.

Literaturverzeichnis

[1] G. Zeisler, *Automatic code generation from Business Process Models*. Graz: FH Joanneum, 2022.

[2] M. Reiter, "Ontology based approach for the modelling and execution of business processes," 2018, book Title: Ontology based approach for the modelling and execution of business processes.

[3] A. Fleischmann, "What Is S-BPM?" in *S-BPM ONE – Setting the Stage for Subject-Oriented Business Process Management*, ser. Communications in Computer and Information Science, H. Buchwald, A. Fleischmann, D. Seese, and C. Stary, Eds. Berlin, Heidelberg: Springer, 2010, pp. 85–106.

[4] A. Fleischmann, W. Schmidt, C. Stary, S. Obermeier, and E. Börger, Eds., *Subjektorientiertes Prozessmanagement: Mitarbeiter einbinden, Motivation und Prozessakzeptanz steigern ; [einfach und intuitiv umzusetzen]*. München: Hanser, 2011.

[5] A. Fleischmann, S. Raß, and R. Singer, *S-BPM Illustrated: A Storybook about Business Process Modeling and Execution*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. [Online]. Available: http://link.springer.com/10.1007/978-3-642-36904-9

[6] M. Reiter, "reiterma13/BPMN-to-S-BPM-Ontology: Take a BPMN file as input and generates a S-BPM OWL file." 2018. [Online]. Available: https://github.com/reiterma13/BPMN-to-S-BPM-Ontology

[7] A. Hödl, "hoedlale16/BPMN20OntologyTester: Development project for bachlor theses at university of applied sciencies FH JOANNEUM. A tool to test an BPMN2.0 Ontology," 2019. [Online]. Available: https://github.com/hoedlale16/BPMN20OntologyTester

[8] S. White, "Introduction to BPMN," 2004. [Online]. Available: https://www.studocu.com/en-us/document/miami-university/principles-of-marketing/introduction-to-bpmn-1/14405349

[9] R. Singer, "An Ontological Analysis of Business Process Modeling and Execution," *arXiv:1905.00499 [cs]*, Apr. 2019, arXiv: 1905.00499. [Online]. Available: http://arxiv.org/abs/1905.00499

[10] B. Silver, *BPMN method and style: with BPMN implementer's guide*, 2nd ed. Aptos, Calif: Cody-Cassidy Press, 2011.

[11] S. Sneed, "Mapping Possibilities of S-BPM and BPMN 2.0," in *S-BPM ONE - Education and Industrial Developments*, ser. Communications in Computer and Information Science, S. Oppl and A. Fleischmann, Eds. Berlin, Heidelberg: Springer, 2012, pp. 91–105.

[12] OMG, “Business Process Model and Notation (BPMN), Version 2.0,” 2013.

[13] M. Elstermann, E. Börger, S. Borgert, A. Fleischmann, R. Gniza, H. Kindermann, F. Krenn, T. Schaller, W. Schmidt, R. Singer, C. Stary, F. Strecker, A. Wolski, and C. Zebold, “I2PM/PASS-Standard-Book-Tex-Project: A Tex-Only Excerpt from the Standard-Document Intended For Shared Editing via the Overleaf-Platform,” 2020. [Online]. Available: https://github.com/I2PM/PASS-Standard-Book-Tex-Project

[14] M. Elstermann, “Proposal for Using Semantic Technologies as a Means to Store and Exchange Subject-Oriented Process Models,” in *Proceedings of the 9th Conference on Subject-oriented Business Process Management*, ser. S-BPM ONE ’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1–9. [Online]. Available: https://doi.org/10.1145/3040565.3040573

[15] M. Elstermann and F. Krenn, “The Semantic Exchange Standard for Subject-Oriented Process Models,” in *Proceedings of the 10th International Conference on Subject-Oriented Business Process Management*, ser. S-BPM One ’18. New York, NY, USA: Association for Computing Machinery, Apr. 2018, pp. 1–8. [Online]. Available: https://doi.org/10.1145/3178248.3178257

[16] L. Gnad and M. Elstermann, “I2PM/alps.net.api,” 2022. [Online]. Available: https://github.com/I2PM/alps.net.api

[17] A. Barros, M. Dumas, and A. H. M. ter Hofstede, “Service Interaction Patterns,” in *Business Process Management*, ser. Lecture Notes in Computer Science, W. M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, Eds. Berlin, Heidelberg: Springer, 2005, pp. 302–318.

[18] S. Raß, *Requirements for and capabilities of cloud based BPMS*. Graz: FH Joanneum, 2013.