Entwicklung eines graphischen Modellierungs-Tools zur Spezifikation von Simulationsmodellen

Vorstudie und Machbarkeitsanalyse

Studienarbeit

Abteilung Informatik

OST – Ostschweizer Fachhochschule

Herbstsemester 2020

Autoren: Fabienne König, Daniel Steudler

Betreuer: Andreas Rinkel

18. Dezember 2020

Abstract

Ausgangslage

Um bestehende oder neue Geschäftsprozesse zu analysieren, validieren und optimieren, wird häufig die Prozesssimulation genutzt. Hierbei wird das reale Modell zuerst modelliert und in einem zweiten Schritt simuliert. Derzeit wird das Modellierung nicht erschöpfend durch angepasste Werkzeuge unterstützt. Für die Verifikation ist jedoch ein formales Modell der Spezifikation erforderlich.

Ziel

Es soll analysiert und evaluiert werden, ob bereits ein geeignetes Modellierungstool existiert, das man in einem weiteren Schritt um die benötigten Erweiterungen und eine Validierung ergänzen kann. Alternativ müsste geprüft werden, wie die Implementation eines eigenen Tools am besten zu realisieren ist. Wichtig ist hierbei, dass sich das Tool hauptsächlich an Leute richten soll, die weder grosse Erfahrungen in der Simulation, noch in der Informatik haben.

Ergebnis

In einen ersten Schritt wurden die Elemente der grafischen Notationssprache SimBPMN, einer Sprache die auf der Business Process Model and Notification (BPMN) Sprache basiert, auf ihre Formalisierbarkeit und die Formalisierungsbedingungen geprüft. Diese wurde wo nötig erweitert und konkretisiert. Damit wurde gezeigt, dass sich die wichtigsten Elemente der SimBPMN formalisieren lassen. Anschliessend wurde die Machbarkeit der Implementation dieser erweiterten Simulationssprache durch die Umsetzung eines Prototypen aufgezeigt. Schlussendlich wurde gezeigt, dass sich das BPMN-JS Framework gut als Grundlage für die Implementation eines SimBPMN Tools eignet.

Management Summary

Kontext und Problemstellung

Um bestehende oder neue Geschäftsprozesse nachvollziehen zu können, wird häufig auf die Prozesssimulation gesetzt. Hierbei wird das reale Modell zuerst modelliert und in einem zweiten Schritt simuliert. Derzeit werden diese beiden Schritte voneinander getrennt durchgeführt. Die Modellierung wird durch einen Domänenexperten durchgeführt, die Simulation hingegen durch einen Simulationsexperten. Eine formale Prüfung des Systems findet erst in der Simulation statt. Auf dem Markt existieren Tools für Simulationsexperten und Tools für Domänenexperten. Es gibt jedoch kein Tool, dass den Domänenexperten bei der Überprüfung seines Modells als Vorbereitung auf die Simulation unterstützt.

Die ASIM Forschungsgruppe arbeitet mit SimBPMN an einer Simulationserweiterung der bereits vorhandenen und weitverbreiteten Sprache BPMN. Mit BPMN lassen sich in der Simulationsmodellierung derzeit nur die Token-Flows der Prozesslogik modellieren. Die Elemente für das Modellieren des Entity-Flows sind noch nicht ausreichend gegeben. Ein SimBPMN Tool kann den Domänenexperten bereits früh in Modellierung unterstützen, in dem gewisse Teile des Modells bereits dort formal geprüft werden können. Es hat sich in dieser Arbeit gezeigt, dass die von BPMN und SimBPMN definierten Elemente erweitert werden müssen, um eine formale Prüfung möglich zu machen. Diese Elemente wurden erweitert und konkretisiert. Anhand eines Beispiels zum Duck eines Fotobuches wurde aufgezeigt, welche Erweiterungen die SimBPMN zur formalen Prüfung noch benötigt. Abschliessend wurde geprüft, inwiefern sich BPMN-JS eignen könnte, um als Grundlage eines SimBPMN Tools zu dienen.

Ausblick

Es wurde gezeigt, dass sich die die von Camunda entwickelte opensource Library BPMN-JS gut für das Implementieren eines eigenen SimBPMN Tools eignen könnte. Gegenüber der Erweiterung von bereits etablierten Produkten hat dies den Vorteil, dass ein auf SimBPMN zugeschnittenes Userinterface entwickelt werden kann. Bestehende Tools besitzen zum Teil eine Plugin Architektur. Diese Tools fokussieren sich jedoch meistens

zu stark auf ihre Features, so dass eine Integration von SimBPMN sich dort schlechte integrieren lässt. Der Zustand und die Archtitektur spricht für die Library.

Konkret würde eine SimBPMN Applikation aus einer Kern Applikation bestehen welche Grundfunktionalitäten bereitstellt. Diese Applikation wird dann mit BPMN-JS Plugins angereichert welche es erlauben BPMN-JS nebst mit BPMN auch mit SimBPMN umzugehen. Die Pluginarchitekur stellt sicher das jede BPMN Applikation welche auf BPMN-JS aufbaut, auch mit SimBPMN Funktionalität ausgerüstet werden kann.

OST
Ostschweizer
Fachhochschule

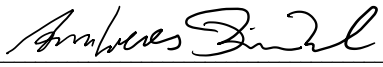
Aufgabenstellung

Im Rahmen der Studienarbeit ist eine Vorstudie zur Entwicklung eines Tools zur Unterstützung der Spezifikation von Simulationsmodellen zu erarbeiten.

Basierend auf der bereits erstellten Spezifikation der GI/ASIM Arbeitsgruppe "Formale Spezifikation in der Simulation" soll untersucht werden, welche Erweiterungen erforderlich sind. Ferner ist ein Umsetzungskonzept zu erarbeiten. Hierzu müssen geeignete Plattformen/Tools gesucht und untersucht werden.

Daniel Steudler: 

Fabienne König: 

Andreas Rinkel: 

Änderungsverlauf

Datum	Version	Änderung	Autor
2020-09-18	1	Initiale Version des Dokumentes	Daniel Steudler
2020-10-23	2	Lösungskonzept aus Google Drive ins LaTeX übernommen	Daniel Steudler, Fabienne König
2020-11-05	3	Kapitel Problembeschrieb geschrieben	Daniel Steudler
2020-11-16	4	Fotobuch Beispiel hinzugefügt	Daniel Steudler, Fabienne König
2020-11-19	4	Dokument überarbeitet	Daniel Steudler, Fabienne König
2020-11-29	5	Kapitel Umsetzung geschrieben	Daniel Steudler, Fabienne König
2020-12-03	6	Kapitel Einleitung geupdated	Fabienne König
2020-12-10	7	Abstract hinzugefügt	Daniel Steudler, Fabienne König
2020-12-13	8	Bilder hinzugefügt	Fabienne König
2020-12-17	9	Dokument finalisiert	Daniel Steudler, Fabienne König

Inhaltsverzeichnis

1. Einleitung

Die Prozessmodellierung ist ein wichtiges Werkzeug, um beispielsweise Geschäftsabläufe eines Unternehmens zu verstehen. Die Modellierung dient als Grundlage zur Spezifikation und ist ein wichtiger Schritt zur Erstellung des Modells und für die spätere Verifikation der Simulation. Während für die Simulation bereits viele nützliche Tools existieren, gibt es aktuell nur unzureichend brauchbare Tools für die unerlässliche vorangehende Modellierung der System Architektur.

Als weitverbreitete Sprache für Geschäftsprozesse wird weitläufig die Business Process Model and Notation (BPMN) eingesetzt. Bei der BPMN handelt es sich um eine graphische Spezifikationssprache, die neben Geschäftsprozessen auch zur Beschreibung von Workflows und Arbeitsprozessen angewendet wird.[1] Sie reicht als Modellierungssprache für Simulationen nicht aus. Für die Spezifizierung von Systemarchitekturen von Simulationen wurde Business Process Model and Notation for Simulations (SimBPMN) geschaffen. SimBPMN ist eine Erweiterung der BPMN und enthält ergänzende Elemente und Funktionalitäten. Die SimBPMN befindet sich zur Zeit noch im Draft-Status.

Diese Studienarbeit geht auf den Draft vom SimBPMN zurück und ergänzt ihn um die im Kapitel 2 zum Lösungskonzept aufgeführten Punkten.

1.1. Ausgangslage

Der Markt für Tools im Bereich der Simulation kann in zwei Lager unterteilt werden. Einerseits in Tools für die Simulation, die den Input formal prüfen und Erfahrung im Bereich der Simulation voraussetzen. Andererseits Tools, die sich auf die BPMN Modellierung spezialisiert haben und meistens gar keine formale Prüfung vornehmen können.

Um die Brücke zwischen dem ersten Schritt, der Modellierung eines Prozesses, und dem zweiten Schritt, der Simulation, zu schlagen, arbeitet die Arbeitsgemeinschaft Simulation (ASIM Fachgruppe) an der Entwicklung der SimBPMN – einer Erweiterung der BPMN.

In Abbildung 1.1 betrifft es die Schritte 1 bis 4. Sie wird im Modul „System Modeling and Simulation“ verwendet um von Anfang an einen Überblick über die Abläufe eines Simulationsprojekts zu schaffen.

Abbildung 1.1.: Die Schritte eines Simulationsprojekts[2]

Das Ziel der ASIM Fachgruppe ist es, die SimBPMN bis zu einem gewissen Grad formal prüfbar zu machen, um somit Experten in der Modellierung ihrer Prozesse zu unterstützen.

1.2. Problembeschreibung

> Für den praktischen Gebrauch wird also eine Sprache gesucht, die einerseits einfach und intuitiv zu verstehen ist und andererseits ohne konzeptionellen Bruch, um weiter, wie die oben erwähnten Kalküle, ergänzt werden kann. Dies kann z.B. notwendig sein, wenn gewisse Aspekte des Modells für die spätere Implementierung formal präzisiert werden müssen.[3]

Das Ziel dieser Arbeit ist es herauszufinden, ob es möglich ist ein Tool zu schreiben oder zu erweitern, das die SimBPMN Syntax unterstützt und somit das vom User modellierte System verifizieren kann. Bei der Verifikation wird die Übereinstimmung zwischen Modellierung und tatsächlichem System überprüft. Das Tool soll den User hierbei unterstützen, indem es ihm Feedback zur Verbesserung des Modells liefert. Da sich das Tool vor allem an Domainexperten richten soll, muss es simpel genug sein, um von nicht-Informatikern bedient werden zu können. Durch die Einbeziehung des Verifikati-

onsprozesses können in fernerer Zukunft auch statische Analysen durchgeführt werden, bei denen beispielsweise die Utilisation berechnet wird.

1.3. Lösungskonzept

Das Lösungskonzept dieser Studienarbeit besteht aus zwei Teilen. Einerseits aus der Definition einer Simulationssprache als Erweiterung der SimBPMN und andererseits aus der Umsetzung der Sprache mittels eines simplen Prototyps, der die Machbarkeit der zuvor definierten Formalisierungen aufzeigen soll.

2. Definition einer Simulationssprache

Als Vorlage für die Definition der Simulationssprache dient die SimBPMN. Bei der SimBPMN handelt es sich um eine Simulationserweiterung der bereits etablierten BPMN Syntax. Diese besteht aus vielen unterschiedlichen Elementen, die aneinandergereiht ein System modellieren. Das Ziel von SimBPMN ist es, die BPMN Syntax um geeignete Elemente zu erweitern und wo nötig zu formalisieren. Hiermit soll der Entity-Flow verifiziert und zu einem späteren Zeitpunkt auch simuliert werden können. Zusätzlich könnte dadurch auch das Durchführen von statischen Analysen vereinfacht werden.

2.1. Erweiterung der SimBPMN

In dieser Studienarbeit werden die für die SimBPMN notwendigen Elemente formal beschrieben. Hierfür wird auf bereits bestehende Elemente der BPMN zurückgegriffen, die für die Erweiterung zur SimBPMN konkretisiert werden.

SimBPMN liegt als Draft in der Version 1.5.1 vor. Diese Arbeit orientiert sich an diesem Dokument.

2.1.1. Begrifflichkeiten

Für eine eindeutige Sprache wird in dieser Arbeit Wert darauf gelegt, alle Begrifflichkeiten in der englischen Sprache zu belassen. Zudem wird strikt darauf geachtet, dass keine halb deutschen/halb englischen zusammengesetzte Wörter, wie beispielsweise „Entity-Fluss“, verwendet werden.

Nachfolgend werden Begriffe und Elemente aus der SimBPMN aufgegriffen und wenn nötig um Formalisierungen ergänzt, die es erlauben die Elemente formal zu prüfen.

2.1.2. System

Ein System wird bei[4] als Teilmenge der Realität verstanden. In dieser Arbeit wird unter einem System das komplette zu modellierende System verstanden.

Das System besteht aus einer Anzahl Komponenten, die wiederum (Teil-)Systeme sein können. Zu diesen Komponenten werden unter anderem die Architektur und die Prozesslogik gezählt, die im Verlauf der Arbeit genauer beschrieben werden. Das System stellt damit die oberste detailärmste Abstraktionsebene des Modells dar. Es dient als Grundlage, um gewünschte Teilsysteme in tieferen, detaillierteren Abstraktionsebenen näher zu beschreiben.

Generell wird bei Systemen zwischen geschlossenen und offenen Systemen unterschieden. Während es bei einem geschlossenen System keine Verbindung nach aussen gibt, hat ein offenes System immer mindestens eine Verbindung nach aussen.[5, 4]

Dadurch ist das System grob betrachtet im einfachsten Fall ein Task, bei dem Objekte hinein gehen und transformiert wieder herauskommen. Durch diese möglichen Ein- und Ausgänge sind die Systemgrenzen automatisch gegeben.

2.1.3. Ereignisorientierte Simulation

Wenn in dieser Arbeit von Simulation gesprochen wird, ist damit die „Ereignisorientierte Simulation“ gemeint.

Bei der ereignisorientierten Simulation (discrete event simulation) handelt es sich um ein diskretes Simulationsmodell. Hierbei werden die Komponenten des Realsystems im Simulationsmodell als Entity bezeichnet. Diese Entitys werden durch einen Zustand und durch eine Anzahl von Methoden definiert. Als anschauliches Beispiel kann man sich hier eine Verkehrssimulation vorstellen. Dort zählen die Fahrzeuge, die Ampeln, aber auch die Strassen zu den Entitys. Durch die Methoden wird einen Zustandswechsel ermöglicht.[4]

In der Simulation spielt die Zeit eine grosse Rolle. Der Zeitpunkt für einen Zustandswechsel wird durch die Eigenschaften der jeweiligen Methoden bestimmt. Hierbei kann es sich entweder um die reale Zeit oder um eine fiktive Modellzeit handeln.

Die im diskreten Simulationsmodell verwendeten diskreten Zeitmodelle sind durch Warteschlangen und Wahrscheinlichkeitsverteilungen darstellbar. Als diskretes Ereignis kann beispielsweise der Zufluss von Material für einen Verarbeitungsschritt in einer Produktionslinie bezeichnet werden.[4] Ein kontinuierliches Simulationsmodell könnte nicht mit solchen standardisierten Elemente abgebildet werden. Dort werden stetige Prozesse mittels Differentialgleichungen dargestellt.[6]

2.1.4. Token- und Entity-Flow

In der Simulation wird innerhalb eines Systems zwischen dem Entity-Flow (der Architektur) und dem Token-Flow (der Prozesslogik) unterschieden.

Die Unterschiede werden in den folgenden Kapiteln genauer erläutert.

Um die Unterschiede zwischen den beiden Darstellungen hervorzuheben, wird ein Farbcode (Color Code) verwendet (siehe Abbildung 2.1), der in Kapitel 2.1.15 genauer beschrieben wird.

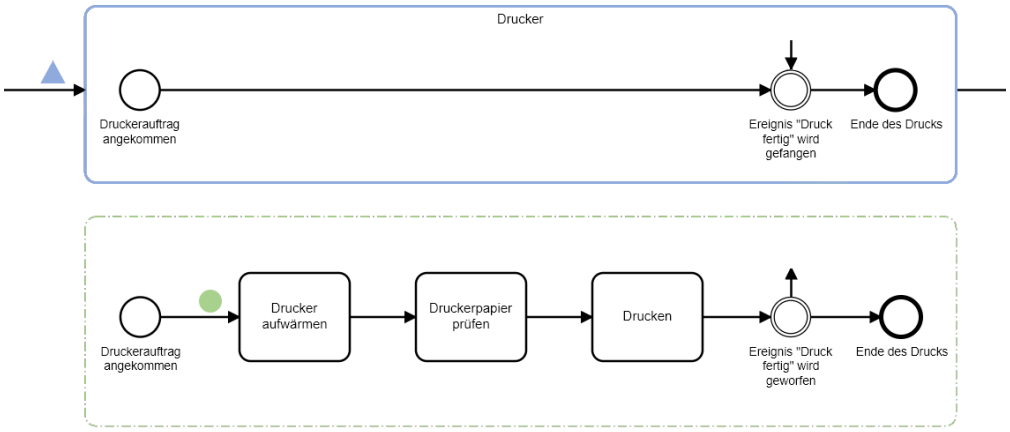


Abbildung 2.1.: Der Unterschied zwischen dem Entity-Flow (oben in blau) mit einem Dreieck als Entity und dem Token-Flow (unten in grün) mit dem runden Token

Für den Entity-Flow werden die Erweiterungen in der SimBPMN Syntax benötigt, die in dieser Studienarbeit erarbeitet werden.

Entity-Flow

Da der Entity-Flow in der BPMN nicht ausreichend beschrieben werden kann, ist er der Kern dieser Arbeit. Der Entity-Flow stellt die Architektur und damit die Systembeschreibung dar. In der Architektur werden alle Elemente beschrieben, die bei der Behandlung eines Problems auf der höchsten Abstraktionsebene erkannt werden können. Im Entity-Flow werden die unterschiedlichen Flow-Elemente passend aneinandergereiht, damit das Entity anschliessend physisch durch das System fliessen kann. Zu diesen Flow-Elementen

zählen die Tasks, die Gateways und die Events, die im Verlauf dieser Arbeit noch genauer beschrieben werden.

Im Entity-Flow werden immer wieder Punkte erreicht, bei denen eine Beschreibung in einem tieferen Abstraktionslevel erforderlich ist. Somit kann durch Subprocesses ein Task mit mehreren Schritten, detaillierter beschrieben werden.

Die unterste Abstraktionsschicht wird nicht mehr als Entity-Flow, sondern als Token-Flow in der Prozesslogik dargestellt.

Token-Flow

Die Prozesslogik beschreibt einen klar strukturierten und festgeschriebenen Ablauf innerhalb eines Prozesses. Die in der Prozesslogik verwendeten Token haben zwar Ähnlichkeiten mit den Entitys aus der Architektur, weichen allerdings in einigen Punkten vom Entity-Flow ab. Im Gegensatz zum Entity sind Token nicht physisch vorhanden, sie fungieren hauptsächlich als Steuerungstoken. Damit stellen sie beispielsweise die Kapazität der einzelnen Prozesse dar und synchronisieren sich während des Token-Flows mit den durch den Prozess fliessenden Entitys.

Für den Token-Flow reicht die BPMN Syntax aus. Daher wird in dieser Arbeit nicht weiter spezifisch auf den Token-Flow eingegangen.

2.1.5. Entity

Entitys gehören zu den diskreten Elementen einer diskreten Ereignissimulation. Das Entity wird in einem Entity Generator erzeugt und durchquert anschliessend ein Netzwerk aus Tasks und Gateways, bevor es mittels eines EndEvents das System verlässt.

Simulationsbedingung

Ein Entity...

- ...besitzt mindestens eine eindeutige ID.
- ...enthält alle Parameter, die für die Synchronisationsbedingung des Gateways benötigt werden.
- ...kann weitere optionale Parameter (Zustände) besitzen.

Zur Darstellung eines Entitys wird ein ausgefülltes Dreieck vorgeschlagen, das kleiner ist als der Start und das Ende. Dieser Vorschlag wird in Abbildung 2.2 dargestellt.

Abbildung 2.2.: Blaues ausgefülltes Dreieck als Vorschlag zur Verwendung als Entity.

Charakteristik der Entitys

Im Entity-Flow wird unterschieden, ob es sich bei einem Entity um etwas physisches oder um etwas nicht-physisches handelt. Hierbei ist auch eine Zwischenform möglich: Abhängig von der Entity Charakteristik sind im System unterschiedliche Pfade möglich. Richtungsweisend sind hier die Gateways, die im Verlauf der Arbeit noch genauer beleuchtet werden.

Physical Object Ein Physical Object besteht aus Atomen. Dadurch kann es (zumindest zum aktuellen Zeitpunkt) nicht geklont werden. Eine Zerstörung eines Pysical Objects ist möglich. Zur Veranschaulichung kann als Beispiel ein Auto hergezogen werden.

besteht aus	Atomen
lässt sich klonen	nein
lässt sich zerstören	ja
Beispiel	Auto

Abstract Object Ein Abstract Object besteht im Gegensatz zum Physical Object nur aus Informationen und ist nicht physisch vorhanden. Dadurch kann es geklont, bzw. kopiert werden. Auch eine Zerstörung eines Abstract Objects kann durchgeführt werden. Unter Abstract Objects werden beispielsweise (Video)-Files gezählt.

besteht aus	Informationen
lässt sich klonen	ja
lässt sich zerstören	ja
Beispiel	(Video-)File

Composite Object Ein Composite Object besteht aus einem Träger aus Atomen, der Informationen beherbergt. Die Information und der Träger können entweder als zusammengehöriges Entity oder getrennt voneinander betrachtet werden. Das Composite Object kann reproduziert werden. Hierbei muss die Reproduktion nicht in jedem Falle identisch mit dem Ausgangsobjekt sein.

Ein anschauliches Beispiel hierfür wäre ein Brief. Dieser kann eingescannt und anschliessend entweder digital abgelegt oder als Kopie ausgedruckt werden. Durch diesen Vorgang wird das analoge Original jedoch nicht invalidiert. In gewissen Situationen ist es schwierig zu entscheiden, ob es sich beim vorliegenden Entity um ein Composite Object handelt oder nicht.

besteht aus	Träger aus Atomen und Informationen
lässt sich klonen	ja
lässt sich zerstören	ja
Beispiel	Brief

2.1.6. Entity Generator

Der Entity Generator erzeugt Entitys, die sich nach ihrer Erzeugung durch das vorgegebene System bewegen. In einem System können mehrere unterschiedliche Entity Generatoren existieren, die jeweils unterschiedliche Entitys erzeugen. Häufig wird das Erzeugen von Entitys zeitlich gesteuert, beispielsweise über eine statistische Verteilung.

Simulationsbedingung

- Es muss mindestens ein Entity Generator vorhanden sein.
- Der Eingangsstrom des Entity Generators muss beschrieben werden. Hierbei muss eine Ankunftsrate oder eine Interarrival Zeit mit deren Verteilungen angegeben werden.

2.1.7. Flow

Der Flow definiert den Pfad durch das System, den das Entity nehmen kann. Er kann weitere Parameter, wie beispielsweise eine Zeitangabe, enthalten welche, die Dauer von einem Element zum nächsten angibt. Die drei Flow-Elemente Tasks, Gateways und Events werden mittels des Flows verbunden.

2.1.8. Task

Der Task ist ein zentrales Flow-Element der Modellierung. Er stellt den einfachsten Fall des übergeordneten Prozesses dar. Die Hauptaufgabe eines Prozesses ist es, Entitys zu verarbeiten. Dies geschieht mittels eines Akteurs, der eine Verhaltensvorschrift ausführt. Mit der Ausführung dieser Verhaltensvorschrift wird beispielsweise eine Transformation eines Entitys erreicht.

Ein Task stellt den simpelsten Fall eines Prozesses dar und definiert sich dahingehend, dass er jeweils nur einen Ein- und einen Ausgang besitzt. Somit muss und kann er nicht detaillierter in einem weiteren Subprocess konkretisiert werden. Einen solchen Zustand bezeichnet man als terminal. In Abbildung 2.3 wird ein Minimalbeispiel eines Tasks dargestellt.

Ein default Task besteht mindestens aus folgenden Elementen:

- Einem Akteur, der die Verhaltensvorschrift ausführt.
- Einer Warteschlange (die im einfachsten Fall eine unendliche Kapazität besitzt).
- Eine zeitliche Verzögerung bei der Transformation.

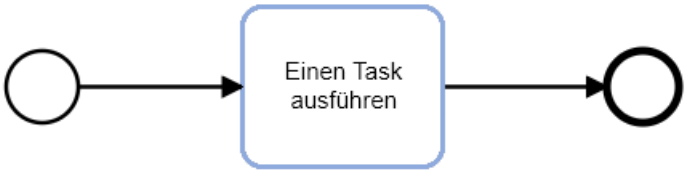


Abbildung 2.3.: Minimalbeispiel eines Tasks

Typische Tasks in einer Simulation:

- Source: In einer Source wird beispielsweise ein neues Entity generiert. Es ist ein Eintrittspunkt für die in das System eintretenden Entitys.
- Transformation: Bei einer Transformation wird das Entity innerhalb des Tasks verändert. Somit unterscheidet sich das eintretende, vom austretenden Entity. Bei Produktionstrassen ist es nicht unüblich, dass ein Entity mehrmals transformiert wird. Das wird am Beispiel in Abschnitt 2.2 verdeutlicht.
- Drain: In einem Drain werden Entitys am Ende des Flows zerstört bzw. sie verlassen das System.

Die soeben beschriebenen typischen Tasks einer Simulation sind in Abbildung 2.4 zusammenfassend dargestellt.

Abbildung 2.4.: Die typischen Tasks: Source, Transformation und Drain

Simulationsbedingung

- Jeder Prozess muss einen klar definierten Start- und Endpunkt haben.
- Jeder Task besitzt genau einen Ein- und einen Ausgang.
- Jeder Task muss terminal sein.
- Jeder Task muss ein $\diamond$ am Anfang haben. Damit wird verdeutlicht, dass die oberen beiden Punkte sichergestellt sind.
- Jeder Task benötigt Ressourcen:
  - mindestens einen Akteur, der die Verhaltensvorschrift ausführt und
  - Material, das während des Prozesses benötigt wird.
- Die Kapazität legt fest, wie oft die Verhaltensbeschreibung gleichzeitig ausgeführt werden kann.
- Es muss eine Warteschlangen-Strategie festgelegt werden: z.B. FIFO, LIFO, random, FIRO, prioritätsgesteuert, ...

2.1.9. Subprocess/Subtask

Ein Subprocess, auch *Nested Process* genannt, erlaubt es Tasks detaillierter zu beschreiben.

In einer Systembeschreibung ist es selten erwünscht, den ganzen Flow im Detail zu beschreiben. Mit Subprocessen könnnen daher die fehlenden ergänzende Details im Flow beschrieben werden.

Sobald ein Prozess mehrere Ein- und Ausgänge besitzt, handelt es sich hierbei zwingend um einen Subprocess und ist nicht terminal. Die detaillierte Ausführung dieses Teilprozesses wird in einem separaten Diagramm modelliert und zum ursprünglichen Task verlinkt.

Simulationsbedingungen

- Jeder Subprocess/Nested Process besteht aus weiteren Prozessen, Tasks oder Subprocessen und deren Beziehungen zueinander.
- Jeder Subprocess hat klar definierte Eingänge und Ausgänge. Bemerkung: Im Gegensatz zur Task, die nur einen Eingang hat.

2.1.10. Gateways

Gateways sind neben den Tasks das Zweite von drei Elementen des Flows. Sie teilen und vereinen den Pfad des Flows anhand von vorgegebenen Kriterien. In dieser Arbeit werden nur die simplen Gateways XOR, OR, AND und das Event-based-Gateway behandelt. Das Complex-Gateway wird nicht genauer beschrieben, da es sich hierbei um eine Art Joker handelt. Um ein Complex-Gateway validieren zu können, müsste jede mögliche Kombination überprüft werden. Dies steht in keinem Verhältnis zu dessen Nutzen.

Verwendung von Gateways:

- Der Name des jeweiligen Gateways bestimmt die mögliche Anzahl der folgenden Pfade
- Entscheidungskriterium für die Abzweigung
  - Prozent (z.B. 30%:70%)
  - Vorbedingung
  - Default Pfad, falls keine Bedingung definiert wird
  - Basierend auf einer Eigenschaft (Rechnungen werden gescannt, Werbung wird entsorgt)
  - Basierend auf einem Entity Typ

Gateway Modus

Gateways haben zwei verschiedene Modi – Teilen und Vereinigen/Zusammenführen. Für beide Modi gelten jeweils pro Gate unterschiedliche Regeln. Nachfolgend werden die Regeln näher erläutert.

Synchronisationsbedingung

Wird ein Entity innerhalb eines Tasks oder mittels eines Gateways aufgeteilt, und soll später wieder zu einem vereinigt werden, so wird für diese Vereinigung eine Synchronisationsbedingung benötigt. Diese Synchronisationsbedingung beschreibt, wie sich welche aufgesplitteten Teile wieder zusammenfügen lassen. Als Bedingung für die Synchronisation eignen sich beispielsweise ID's. Ohne eine Synchronisationsbedingung werden die Entitys willkürlich zusammengesetzt. So sollte ein Hamburger erwartungsgemäss nicht mit einem Auto kombiniert werden können. Beschreibt der Entity-Flow jedoch einen Drive-Thru einer Schnellimbisskette, dann ist dies schon wahrscheinlicher. Die Synchronisationsbedingung beschreibt dann, wie die beiden Entitys zusammengesetzt werden dürfen.

Exclusive OR Gateway

Exclusive OR Gateways werden bei Entscheidungen verwendet, bei denen es nur einen möglichen Pfad zum Weiterkommen gibt.

Beispiel: Auf der Autobahn kann man entweder die Ausfahrt nach Zürich nehmen, oder weiter Richtung Bern fahren. Beides gleichzeitig geht nicht. Siehe Abbildung 2.5.

Zeichen

Beschreibung
Das Exclusive OR Gateway (XOR) funktioniert wie eine Zugweiche.

Teilen Bedingung
Identisch zum Token-Flow. Es verhält sich in beiden Situationen wie eine Weiche.

Zusammenführen Bedingung
Kein Synchronisationsvorgang

Warnungen
Keine

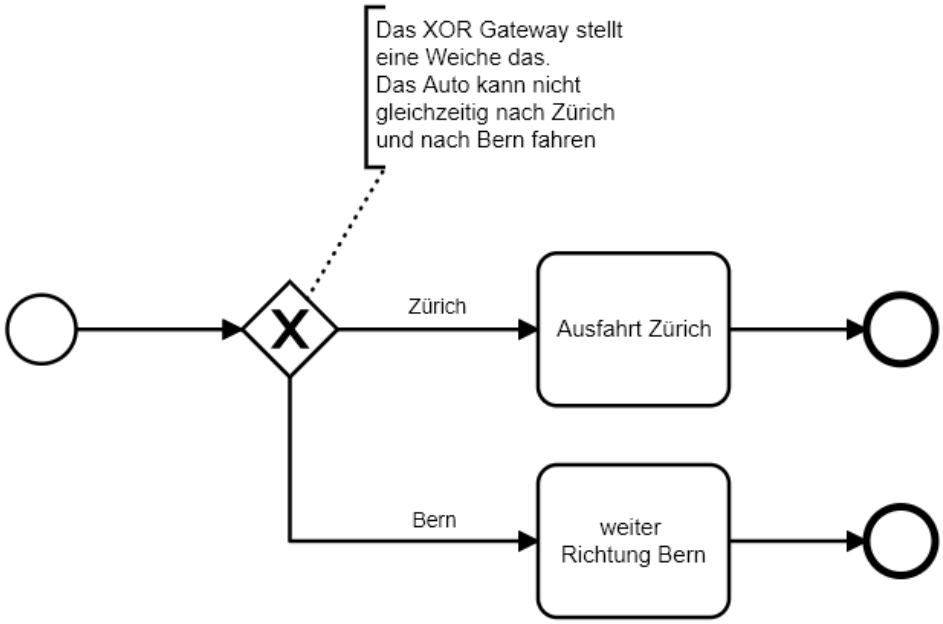


Abbildung 2.5.: Einfaches Beispiel eines einfachen XOR Gateways

Simulationsbedingungen Das Entity geht in eine vordefinierte Richtung, die durch eine Bedingung festgelegt wird.

Inclusive OR Gateway

Das Inclusive OR Gateway erlaubt es, Varianten zu kombinieren.

Beispiel: Im Restaurant – Ich kann Burger mit Pommes bestellen, aber auch nur Pommes oder nur einen Burger. Siehe Abbildung 2.6.

Zeichen

Beschreibung

Das Inclusive OR funktioniert wie ein Menü im Restaurant. Die Ausgänge können beliebig miteinander kombiniert werden. Beim Kombinieren eines Gerichtes wird dieses erst dann serviert, sobald der Teller mit allen bestellten Gerichten fertig

angerichtet ist. Gemessen an obenstehenden Beispiel tritt der Zustand ein, sobald der Burger und die Pommes auf dem Teller liegen

Teilen Bedingung

Abstraktes Objekt oder Composite

Zusammenführen Bedingung

Objekte des Inputs müssen derselben Klasse angehören und können unter Umständen die gleiche ID besitzen (Synchronisationsbedingung muss festgelegt werden)

Warnungen

Funktioniert nur mit Token oder Composite als Input (Synchronisationsbedingung muss festgelegt werden)

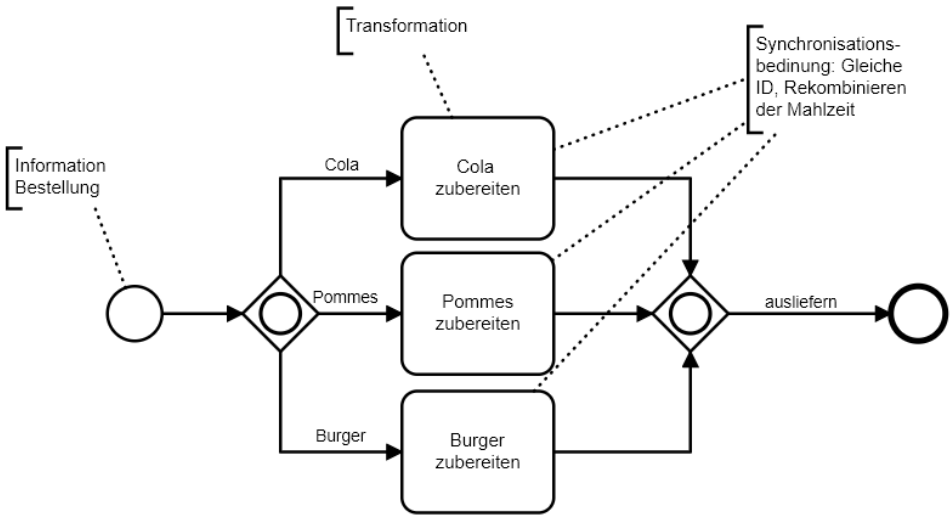


Abbildung 2.6.: Einfaches Beispiel eines einfachen OR Gateways

Das Beispiel von obige zeigt, dass es sich bei der Bestellung von oben um eine Information handeln muss. Das Resultat dieser Bestellung wird zu einem oder mehreren Objekten (den Bestandteilen der zubereiteten Bestellung) transformiert. Diese einzelnen Bestandteile müssen vor dem Servieren mit der Synchronisationsbedingung synchronisiert werden. Damit wird sichergestellt, dass die richtigen Bestandteile kombiniert werden.

Simulationsbedingungen

- Das Entity kann in eine oder mehrere Richtungen gehen, und sich somit aufteilen.

- Das Entity muss kopierbar sein, da es gleichzeitig in mehrere Richtungen gehen kann.
- Ist eine spätere Zusammenführung erforderlich, so muss eine Synchronisationsbedingung definiert sein.

AND Gateway/Parallel Gateway

Bei einem AND Gateway gehen die Entitys bei Eingang in alle Richtungen. Dadurch werden sie vermehrt. Bei der Vereinigung muss von allen Seiten ein Entity ankommen, damit es wieder zusammengeführt werden kann.

Beispiel: Ein Brief kann sowohl eingescannt, als auch abgelegt werden. Siehe Abbildung 2.5.

Zeichen

Beschreibung

Das AND Gateway funktioniert beim Teilen wie ein Gerät das Entitys klont. Beim Zusammenführen verhält es sich wie das Inclusive OR Gateway. Es unterscheidet sich allerdings dahingehend, dass es beim AND Gateway nur weitergeht, wenn auf allen Eingängen etwas angekommen ist.

Teilen Bedingung

Abstraktes Objekt oder Composite

Zusammenführen Bedingung

Objekte des Inputs müssen derselben Klasse angehören und können unter Umständen die gleiche ID besitzen (Synchronisationsbedingung muss festgelegt werden) Unter normalen Umständen können wir einen Burger nicht mit einem Auto kombinieren.

Warnungen

Synchronisationsbedingung muss definiert werden

Simulationsbedingungen

- Das Entity muss in alle Richtungen gehen und sich damit aufteilen.
- Das Entity muss kopierbar sein, da es gleichzeitig in alle Richtungen gehen muss.

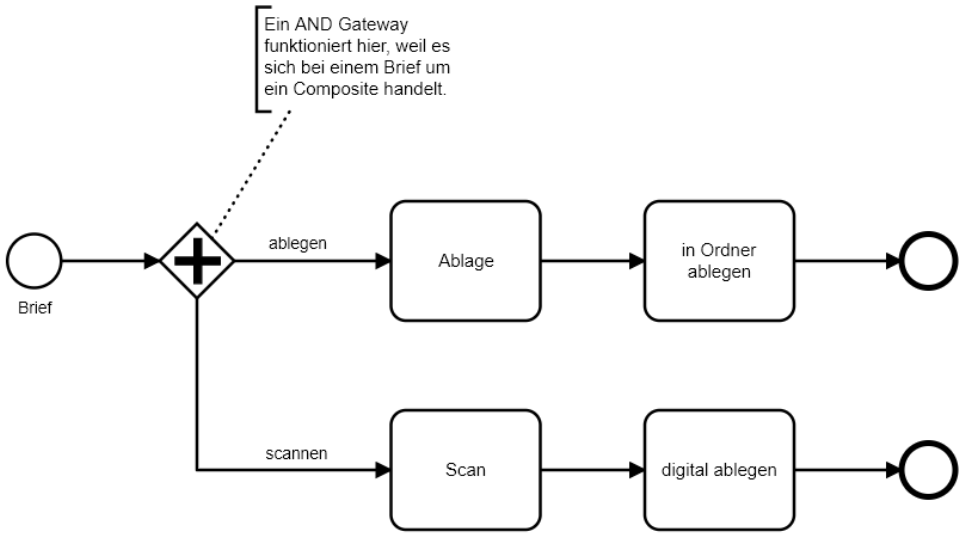


Abbildung 2.7.: Einfaches Beispiel mit einem AND Gateway

- Ist eine spätere Zusammenführung erforderlich, so muss eine Synchronisationsbedingung definiert sein.

Complex Gateways

Bei Complex Gateways handelt es sich um Joker Gateways. Sie lassen sich nicht so einfach formal prüfen, da hierzu alle Möglichkeiten formalisiert werden müssten. Daher wird auf diese Gateways in dieser Arbeit nicht weiter eingegangen. Complex Gateways können in der Umsetzung durch Freitext definiert werden.

2.1.11. Zustandsdiagramm

Mit einem Zustandsdiagramm können alle möglichen Wege von unterschiedlichen Gateway-Entscheidungen auf einen Blick dargestellt werden. Es zeigt die einzelnen Zustände und die dazugehörigen Übergangsbedingungen an. Beim Zustandsdiagramm handelt es sich um eine Ergänzung zum Simulationsdiagramm. Abbildung 2.8 zeigt, wie der Entity-Flow mit Gateways als Zustandsdiagramm dargestellt werden kann. Es ist vor allem bei Diagrammen mit vielen Gateways und somit vielen unterschiedlichen Zuständen hilfreich.

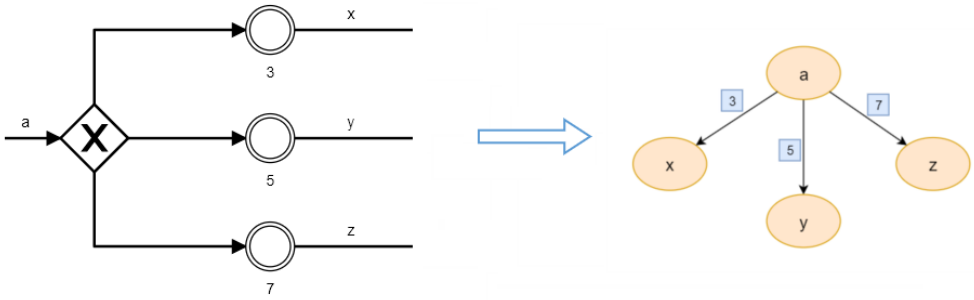


Abbildung 2.8.: Links: Entity-Flow mit Gateway, rechts: dazugehöriges Zustandsdiagramm

2.1.12. Events

Neben den Tasks und den Gateways zählen auch die Events zu den Elementen des Flows, die somit komplett sind. Mit Events können Ereignisse modelliert werden, die während des Durchlaufens eines Entitys durch den Flow auftreten können. Dies kann beispielsweise das Ablaufen eines Timers sein, oder das Eintreffen einer Nachricht. Bei den Events wird zwischen Catching und Throwing Events unterschieden.

Throwing Events

Unter einem Throwing Event wird das Auslösen eines Events verstanden. Die Events können an beliebigen Stellen im Entity-Flow geworfen werden. Man unterscheidet zwischen generischen System-Events, die automatisch geworfen werden und individuell modellierten Event-Elementen. Damit ein Event Auswirkungen auf das System hat, muss es abgefangen werden. Falls ein Event nicht abgefangen wird, so hat es keine Auswirkungen und verfällt.

Ein Throwing Event wird, wie in Abbildung 2.9 ersichtlich, mit einem vom Event weg gerichteten Pfeil dargestellt.

Catching Events

Catching Events fangen die geworfenen Events. Ein Event kann auch von mehreren unterschiedlichen Elementen gleichzeitig abgefangen werden. Wird ein Event von niemandem abgefangen, so verfällt das Event. Es gibt keine Warteschlange oder ähnliches für Events. Das Event ist atomar und existiert nur zum Zeitpunkt der Erzeugung. Catching

Abbildung 2.9.: Darstellung eines Throwing Events

Events können auch an Prozessen „attached“ werden. Hierfür werden sie am Rand eines Prozesses angebracht. Bei sogenannten *Attached Events* muss unterschieden werden, ob das Event *interrupting* ist und den laufenden Process unterbricht, oder ob es sich um eine *non-interrupting* Event handelt, bei dem beide Paths gleichwertig weiterlaufen und zu Ende ausgeführt werden.

Ein Catching Event wird mit einem auf das Event gerichteten Pfeil dargestellt (siehe Abbildung 2.10.

Abbildung 2.10.: Darstellung eines Catching Events

Arten von Events

In der BPMN werden unterschiedlichste Arten von spezifischen Events mit eigenen Zeichen angeboten. Dazu gehören beispielsweise Timer oder Messages. Jedoch können alle Arten auch mit dem untypisierten Event Typ parametrisiert und somit modelliert werden.

Zudem wird unter *Start-*, *Intermediate-* und *End-Events* unterschieden. Sie bestimmen, an welchem Bereich des Entity-Flows ein Event geworfen oder abgefangen wird.

Eine Übersicht der unterschiedlichen Arten ist in Abbildung 2.11 dargestellt.

Abbildung 2.11.: Eventtypen und Zeitpunkte der Events

Simulationsbedingungen

- Events (Ereignisse) steuern den Ablauf der Simulation über der Zeit.
- Events werden zu einem Zeitpunkt geworfen oder abgefangen.
- Ein geworfenes Event kann von jedem abgefangen werden, der auf dieses Event wartet.
- Ein Event existiert nur zu diesem Zeitpunkt (es wird nicht gespeichert).
- Events werden von den Prozessen und Entitys geworfen und gegebenenfalls abgefangen. Beispiel: Catching Events leiten das Token erst weiter, wenn das entsprechende Event aufgetreten ist → praktisch für event-based Gateways.

2.1.13. Ressourcen und Material

In der Simulation wird zwischen Ressourcen und Material unterschieden. Unter Material werden häufig Objekte verstanden, die innerhalb eines Tasks verbraucht werden. Beispielsweise Wasser. Material wird auch als „enabler“ bezeichnet, da die Tasks ohne vorhandenes Material in einen Wartezustand kommen. Ressourcen hingegen können häufiger und auch von weiteren Tasks seriell verwendet werden. Hierzu zählt man häufig

Mitarbeiter und Werkzeuge. Ressourcen bezeichnet man auch als „doer“ oder „maker“, da sie innerhalb des Tasks häufig als Akteure die Verhaltensvorschrift ausführen. Beispiele für Ressourcen und Materialien sind in Abbildung 2.12 anhand eines Druckers ersichtlich. Die Übergänge zwischen Ressourcen und Material sind manchmal schwammig und können von Modell zu Modell anders ausgelegt werden. Zusätzlich ist es in manchen Fällen auch sinnvoll, eine Ressource als Entity darzustellen. Es muss sichergestellt werden, dass jedes Material und jede Ressource, die in einen Task hineinkommen auch wieder heraus gehen.

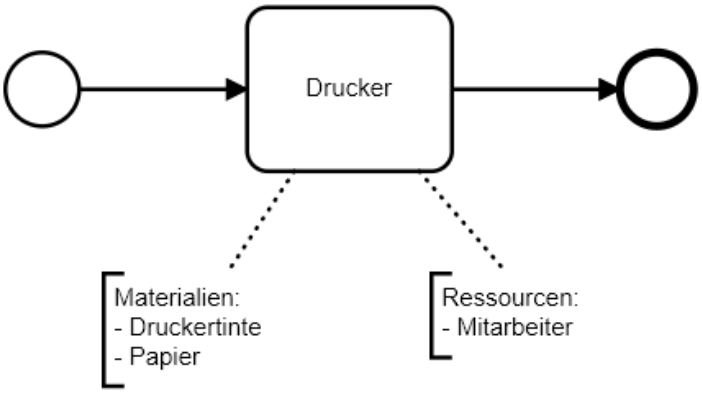


Abbildung 2.12.: Unterschied von Materialien und Ressourcen am Beispiel eines Druckers

Auf eine Unterscheidung zwischen Ressource und Material wird derzeit noch verzichtet, um keine unnötige Komplexität hinzuzufügen.

2.1.14. Warteschlangen

Kommen an einem Task mehr Entitys an als die Kapazität vertragen kann, so landen sie in einer Warteschlange. Diese kann sowohl am Anfang als auch am Ende eines Tasks stehen.

Im einfachsten Fall müssen keine Entitys vor einem Task warten und werden direkt ausgeführt. Für den Moment gehen wir daher davon aus, dass die Warteschlange unendlich ist.

2.1.15. Color Code

Um dem Benutzer des Tools auf den ersten Blick klar zu machen ob er sich im Entity- oder im Token-Flow befindet, wird das Tool in zwei Hauptfabcodes für den Entity- und den Token-Flow dargestellt. Der Entity-Flow in der höchsten Abstraktionsebene wird blau dargestellt und der Token-Flow in grün. Um diese Unterschiede weiter zu verdeutlichen, wird vor der Implementation des Tools versucht, eine zusätzliche visuelle Unterstützung für den Benutzer zu finden.

2.1.16. Probleme mit dem Farbcode (Color Code)

Die im SimBPMN Draft vorgeschlagene Farbcodierung der Elemente ist im Zusammenhang mit Farbenblindheit problematisch. In Abbildung 2.13 ist zu sehen, dass die Farben der im Draft vorgeschlagenen Elemente für jemanden mit Deuteranopia (Rot-Grün-Sehschwäche) nicht zu unterscheiden sind.

Deshalb müsste der Unterschied, den der Farbcode hinsichtlich der Accessibility Informationen darstellen soll, immer mindestens doppelt codiert sein.[7]

Ein Beispiel, das als Inspiration dienen könnte, ist ProcessMaker. Wie in Abbildung 2.14 zu sehen ist, verwendet ProcessMaker sowohl Farbe als auch Form, um Informationen zu codieren.

Aus Zeitgründen wurde dies nicht weiter vertieft.

2.1.17. Ebenen in SimBPMN

Wie bereits Anfangs erwähnt, behandelt diese Arbeit das System mit seiner Architektur und seiner Prozesslogik. Ein System wird häufig in vielen Diagrammen unterschiedlich detailliert dargestellt. Diese Diagramme stellen unterschiedliche Abstraktionsebenen dar. Für das Tool ist es wünschenswert, dass die Diagramme der Abstraktionsebenen untereinander verlinkt werden können. Ansonsten wird es sehr mühsam die zusammengehörigen Diagramme zu finden.

Jede Ebene stellt einen Detaillierungsgrad dar. Hierbei gilt: Je tiefer die Ebene, desto höher der Detaillierungsgrad. Auf der tiefsten Abstraktionsebene befindet sich der Token-Flow.

Das C4 Diagramm [1] diente hier als Inspiration.

[1] `https://c4model.com/`

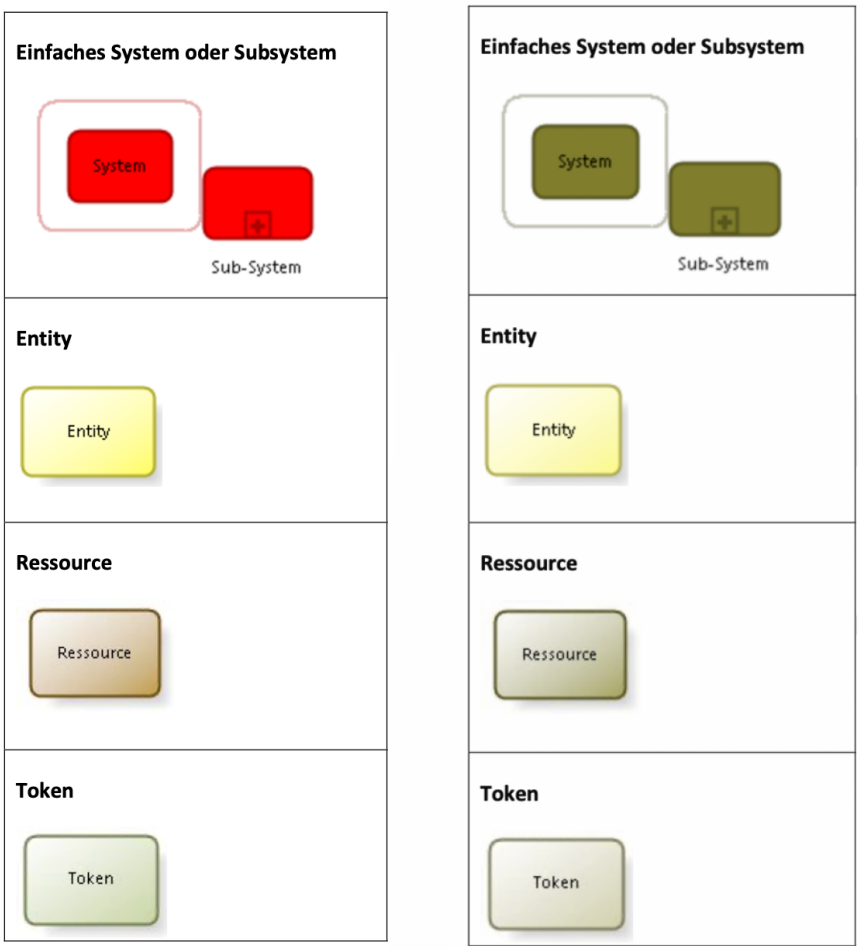


Abbildung 2.13.: Links: Der Farbcode des SimBPMN Drafts, rechts: Der Farbcode wie ihn ein Farbenblinder sehen würde.

2.2. Fotobuchproduktion als Beispiel

Um die Verwendung von SimBPMN beispielhaft zu zeigen, wird in diesem Kapitel der Ablauf der Herstellung eines Fotobuchs zur Veranschaulichung mit SimBPMN modelliert.

In Worten beschriebenes System:

> Gegeben ist folgendes Wort-Modell
> Eine kleine Produktionsfirma fertigt auf Bestellung Fotobücher an. Zur Zeit

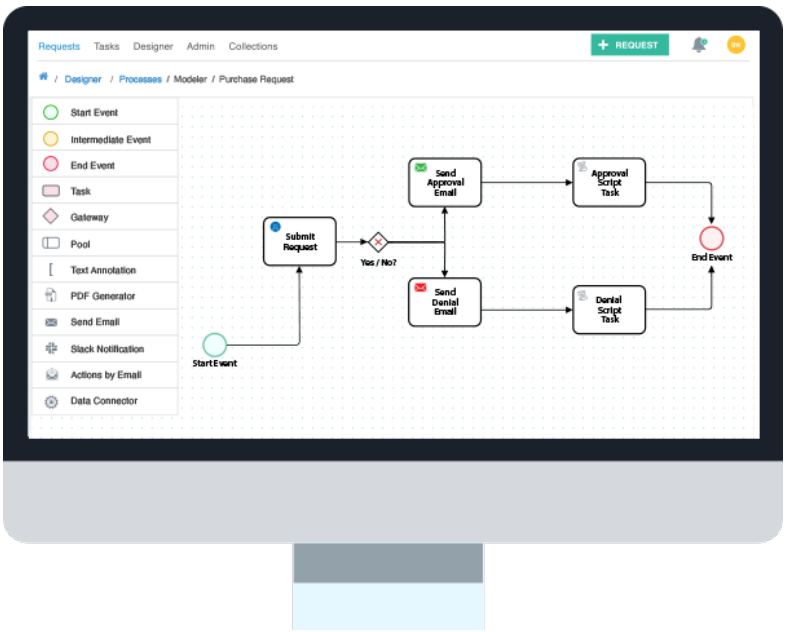


Abbildung 2.14.: ProcessMaker als Vorbild für die Doppelcodierung

gehen pro Tag im Mittel 500 Aufträge ein.

Ein Druckauftrag, der heute auf drei gleichen Maschinen bearbeitet wird, dauert im Mittel 7 Minuten. Auf Grund der vom Prozess nach oben und unten begrenzten Seitenzahlen, liegt die Dauer eines Auftrags zwischen mindestens 3 und maximal 15 Minuten.

Anschliessend gehen die Fotos entweder zum aufwendigen Binden oder zum einfachen, schnelleren Kleben.

Danach werden die Fotobücher für den Versand fertig gemacht, was im Mittel nur eine Minute dauert.

Die durchschnittliche Zeit zur Bearbeitung eines Jobs vom Auftragseingang bis zur Fertigstellung zum Versand dauert ca. 4.5 Stunden. Die Firma arbeitet im drei-schicht Betrieb.

Problem: Wie gross muss der Lagerplatz (Anzahl Jobs) vor jeder Arbeitsstation sein?

Beim Lesen des Modells kann die Bearbeitung eines Auftrags auf zwei Arten interpretiert werden. Bei der ersten Variante wird ein komplettes Fotobuch auf einem einzigen Drucker gedruckt. Bei der zweiten Variante hingegen wird das Fotobuch auf alle vorhandenen

Drucker aufgeteilt und parallel gedruckt. Dadurch verringert sich bei der zweite Variante die Durchlaufzeit eines Fotobuchs durch das System. Im Gegensatz zur ersten Variante wird aber auch die Komplexität erhöht.

Unabhängig der Variante kann die Architektur bzw. das System auf folgende Komponenten eingegrenzt werden.

Als Systemkomponenten können folgende Komponenten identifiziert werden, sie werden in Abbildung 2.15 dargestellt:

- Drucken
- Kleben/Binden
- Versenden

Als Entity wird in diesem Fall der Fotobuchauftrag identifiziert, der beim Durchlaufen des Systems in ein physisches Fotobuch transformiert wird.

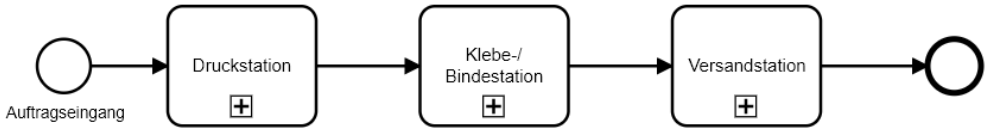


Abbildung 2.15.: Systemarchitektur des Fotobuchs

2.2.1. Variante 1

In der Variante 1 wird beim Erstellen des Druckauftrags der Fotos ein einzelner Drucker ausgewählt. Auf diesem Drucker werden alle Fotos für ein Fotobuch gedruckt. Die Bedingungen zur Wahl eines spezifischen Druckers müssen zuvor festgelegt werden. Das kann beispielsweise aufgrund der Warteschlange oder aufgrund von vordefinierten Druckkontingenten, Druckergeschwindigkeit, etc. sein. Das Diagramm von Variante 1 ist in Abbildung 2.16 dargestellt.

2.2.2. Variante 2

In der Variante 2 wird der Auftrag bereits beim Aufbereiten in einzelne Fotos aufgeteilt. Ein Complex Gateway teilt dann die Fotos aufgrund von zuvor definierten Bedingungen auf die einzelnen Drucker auf. Beispielweise als Sequenz, Bild 1 auf Drucker 1, Bild 2 auf Drucker 2, Bild 3 auf Drucker 3 und Bild 4 wieder auf Drucker 1. Oder das erste Drittel der Bilder wird auf dem ersten Drucker, das zweite Drittel auf dem zweiten und

das dritte Drittel auf dem dritten Drucker gedruckt. Variante 2 des Fotobuchs wird in Abbildung 2.17 gezeigt.

Diese müssen dann jeweils nach dem Druck wieder einsortiert werden.

Die zweite Variante verwendet eine Syntax mit Gateways vor den Tasks. Die Idee dahinter ist es, dem Nutzer zu zeigen, dass ein einfacher Task nur einen Ein- und Ausgang hat. Als Alternative dazu kann dem User durch Linting (mithilfe von bpmn-lint[2]) eine visuelle Hilfe geboten werden. Ihm wird hier ein Hinweis angezeigt, sobald ein Task mehr als einen Ein- oder Ausgang besitzt. Die Hinweise durch bpmn-lint sind in der Abbildung 3.7 zu unserem Prototypen sichtbar.

[2] `https://github.com/bpmn-io/bpmnlint`

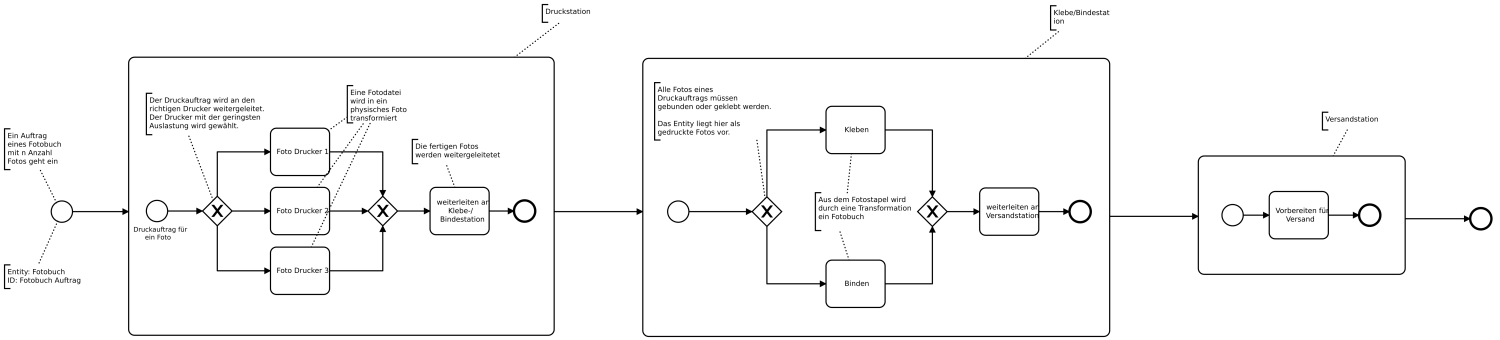


Abbildung 2.16.: Variante 1 der Fotobuchproduktionsstrasse

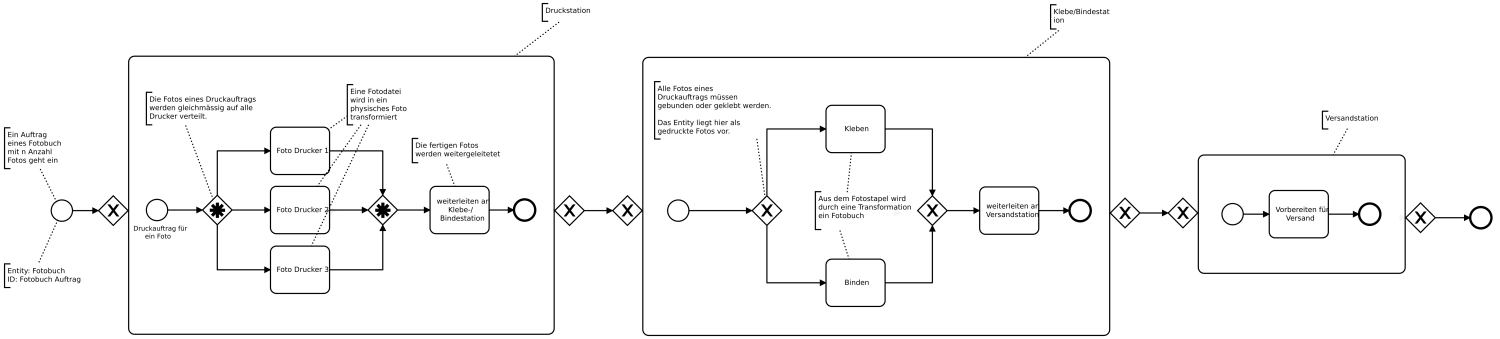


Abbildung 2.17.: Variante 2 der Fotobuchproduktionsstrasse

3. Umsetzung der Sprache

Um sich ein besseres Bild von bereits bestehenden Tools zu machen, wurden zu Beginn der Arbeit folgende Tools angeschaut. Das Ziel war es, ein Tool zu finden, das für die Umsetzung der Spracherweiterung in Frage kommt.

In einer ersten Phase wird das zu erarbeitende Tool nur ein Zeichnungstool sein.

- Camunda
- Bonita Studio Community Edition
- ProcessMaker
- BPMN-JS

Bei Bonita Studio und Camunda handelt es sich um Java Applikationen. Camunda wird als Webapplikation installiert und benötigt einen Tomcat-Webserver. Bonita Studio hingegen ist eine klassische Desktopapplikation, die jedoch zusätzlich ein Webportal bereitstellt. Camunda legt den Fokus auf das *Process Workflow Managment*, also auf das Kreieren und Verwalten von Prozessen. Bonita Studio konzentriert sich mehr auf die Entwicklung von Business Applikationen. Als Grundlage dieser Applikationen liegt jeweils ein BPMN Prozess zu Grunde.

Beide Applikationen sind bereits sehr ausgereift und haben sich mit ihren Features auf den BPMN Process-Flow spezialisiert. Eine Erweiterung um SimBPMN Fähigkeiten scheint hier wenig sinnvoll. Durch die höhere Komplexität wird bei Bonita wohl zudem, sowohl für künftige User als auch als Entwickler, eine längere Einarbeitungszeit notwendig zu sein. Die Applikationen haben ihre Nische und die Erweiterung um SimBPMN Funktionialitäten scheint durch den derzeitigen Fokus der Applikationen eher aufwendig und wenig zielführend. Es ist unklar wie hier eine technische Erweiterung aussehen könnte. Obwohl sich Bonita selbst als: „[. . .]open-source and extensible platform for business process automation and optimization[. . .]“ sieht.

ProzessMaker macht bezüglich des Interfaces durchaus einen guten Eindruck. Es ist jedoch nicht Open Source und kann für diese Arbeit deshalb leider nicht berücksichtigt werden.

BPMN-JS ist die Open Source Library von Camunda, die von Camunda genutzt wird um Diagramme anzuzeigen. Ein wichtiger Einsatzzweck davon ist es, für Camunda BPMN

Diagramme auf der eigenen Webseite anzuzeigen. Für den Anwendungszweck einer eigenen SimBPMN Anwendung ist das demnach ein guter Ausgangspunkt.

Mit BPMN-JS ergibt sich die Möglichkeit eine eigene Applikation zu bauen. Diese kann individuell nach eigenen Bedürfnissen mit Funktionalitäten ausgestattet werden. Weitere Funktionalitäten können mit Plugins nachgereicht werden.

Der grösste Vorteil einer BPMN-JS basierten App ist, dass die BPMN-JS Library alleinstehend nur ein Zeichenprogramm für valide BPMN 2.0 Diagramme ist.

Derzeit gibt es bereits sehr ausgereifte Plugins, die einen Teil der für die folgende Bachelorarbeit erwünschten Funktionalität besitzen. Darauf vorbereitend wurden diese vorhandenen Plugins in einem späteren Kapitel zum POC getestet und eine Demo Applikation mit angepasster Funktionalität gebaut.

BPMN-JS stellt ein Repository[1] mit Beispielen zur Verfügung, das aufzeigt was mit BPMN-JS alles möglich ist. Darin enthalten sind die bestehenden Plugins auf denen aufgebaut werden kann.

Kurz vor Ende dieser Arbeit wurde eine Closed Source Webapp namens Duckflow[2] veröffentlicht. Duckflow sieht sich als leichtgewichtige Zeichenapplikation, die auf den Open Source Komponenten von BPMN-JS basiert. Es verfolgt somit ein ähnliches Ziel wie diese Arbeit, jedoch für BPMN 2.0.

3.1. BPMN-JS Architektur

3.1.1. Libraries

BPMN-JS ist eine Javascript Frontend Library/Framework, die von Camunda[3] in ihren kommerziell vertriebenen Produkten verwendet wird. Die Library wird durch Camunda und der Community aktiv gepflegt und erweitert.

Die wichtigsten Komponenten werden hier kurz beschrieben. Eine umfassendere Beschreibung gibt es im Anhang D.1

BPMN-JS ist im Grunde nur eine Javascript Library/Framework, die auf einer Webseite eingebunden werden kann um BPMN Diagramme anzuzeigen. Ein Minimalbeispiel mit BPMN-JS kann vollständig im Browser des Benutzers laufen. Mit diesem Minimalbeispiel ist es möglich, BPMN 2.0 Diagramme zu erstellen und als XML und SVG zu

[1] `https://github.com/bpmn-io/bpmn-js-examples`
[2] `https://duckflow.app`
[3] `https://camunda.com/de/`

exportieren. Durch die Implementierung als reine Frontend Library spricht nichts dagegen, gewünschte Features serverseitig zu implementieren und diese dann zum Beispiel mit AJAX auszutauschen.

BPMN-JS selbst baut auf zwei Libraries auf – diagram-js und bpmn-moddle.

Diagramm-js

Diagramm-js kennt Formen und Verbindungen. Es erlaubt die Interaktion mit dem Diagramm und bietet eine Infrastruktur für Plugins. Die Plugins funktionieren als simple Module, welche via Dependency Injection geladen werden können.

Bpmn-moddle

Bpmn-moddle kennt den BPMN 2.0 Standard und erlaubt es BPMN 2.0 konforme XML Dokumente zu lesen und zu schreiben. Es wandelt XML in einen Javascript Object Tree um.

Ausserdem kennt es das BPMN Meta Model und dient zur Schemavalidierung.

3.1.2. Bpmnlint

Bereits zu Beginn der Studienarbeit sind wir ein Beispielprojekt auf bpmnlint[4] gestossen. Der Linter gibt dem Benutzer Feedback zu den unterschiedlichsten Fehlern, die er in der Flow-Architektur machen kann.

```
> bpmnlint invoice.bpmn

/Projects/process-application/resources/invoice.bpmn
Flow_1    error    Sequence flow is missing condition  conditional-flows
Process   error    Process is missing end event        end-event-required
Task_13   warning  Element is missing label/name       label-required
Event_12  warning  Element is missing label/name       label-required
Event_27  warning  Element is missing label/name       label-required
Process   error    Process is missing start event      start-event-required

x 6 problems (6 errors, 0 warnings)
```

Listing 3.1: Beispiel Output von bpmlint

[4] https://github.com/bpmn-io/bpmnlint

Für SimBPMN müssen noch Regeln definiert werden. Eine Schwierigkeit könnte darin bestehen, die selben Symbole zu verwendet wie bei BPMN 2.0, da der Linter dann nicht weiss, nach welchen Regeln er den Flow prüfen soll.

3.1.3. XML als Austauschformat

BPMN-JS setzt auf das .bpmn Format als Austauschformat. Beim .bpmn Format handelt es sich um ein XML, das mit dem BPMN 2.0 Standard verabschiedet wurde. Der Vorteil von XML als Austauschformat ist, dass es relativ flexibel und gut erweiterbar ist.

Im Unterschied zu anderen Diagrammformaten wie beispielsweise SeqDiag oder PlantUML hat XML den Nachteil, dass es nur bedingt menschenlesbar ist. Während beispielsweise UML Diagramme im PlantUML Format ohne Probleme ähnlich wie Soucecode von Hand geschrieben werden können, ist bei XML meistens nicht praktikabel.

```
skinparam monochrome true
skinparam ranksep 20
skinparam dpi 150
skinparam arrowThickness 0.7
skinparam packageTitleAlignment left
skinparam usecaseBorderThickness 0.4
skinparam defaultFontSize 12
skinparam rectangleBorderThickness 1

rectangle "Main" {
    (main.view)
    (singleton)
}
rectangle "Base" {
    (base.component)
    (component)
    (model)
}
rectangle "<b>main.ts</b>" as main_ts

(component) ..> (base.component)
main_ts ==> (main.view)
(main.view) --> (component)
(main.view) ...> (singleton)
(singleton) ---> (model)
```

Listing 3.2: Austauschformat im PlantUML Format

Dem gegenüber steht ein simples BPMN Beispiel, von dem der Sourcecode von Abbildung 2.7 im Anhang unter D.2 abgelegt wurde. Auf den ersten Blick ist nicht zu erkennen, was sich hinter dem Code verbirgt.

3.2. Was kann BPMN-JS schon

Um zu sehen was BPMN-JS bereits kann, wurden die Beispiele aus dem bereits erwähnten Repository[5] evaluiert und partiell in produktiver Umgebung getestet. Hier wurde nach Beispielen gesucht, die sich für die in dieser Arbeit gefundenen fehlenden Funktionalitäten der SimBPMN eignen würden.

3.2.1. Verlinken von Dokumenten

Das in einer Folgearbeit zu implementierenden Tool soll alle zusammengehörigen Diagramme und Dokumente verlinkbar machen, damit sie einfach gefunden werden. Für diesen Zweck müssen unterschiedliche externe Dokumente an Elemente angehängt werden können. Bereits beim POC wurde die Implementation eines Properties-Panel ausprobiert. Da man dort viele unterschiedliche Custom Properties definieren kann, könnte sich das Properties-Panel generell als ein Feature mit einen grossen Nutzen für das Tool herausstellen. Das Properties-Panel mit einem Custom Property ist in Abbildung 3.1 zu sehen. Neben den anderen Properties der Elemente könnte somit ein Link zur Dokumentation im Panel gesetzt werden.

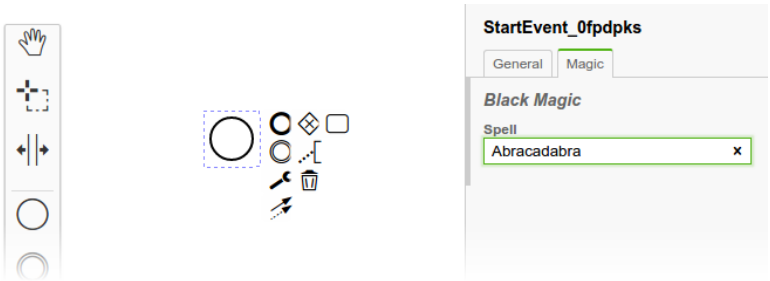


Abbildung 3.1.: Beispiel eines Feldes im Properties-Panel[8]

Alternativ kann für das Hinzufügen der Links auch auf Commenting oder Overlays zurückgegriffen werden. Beim Commenting lässt sich pro Element ein Textfenster öffnen, in das der Link gespeichert werden kann (siehe Abbildung 3.2). Overlays werden bei „Hinweise anzeigen“ genauer beschrieben.

[5] https://github.com/bpmn-io/bpmn-js-examples

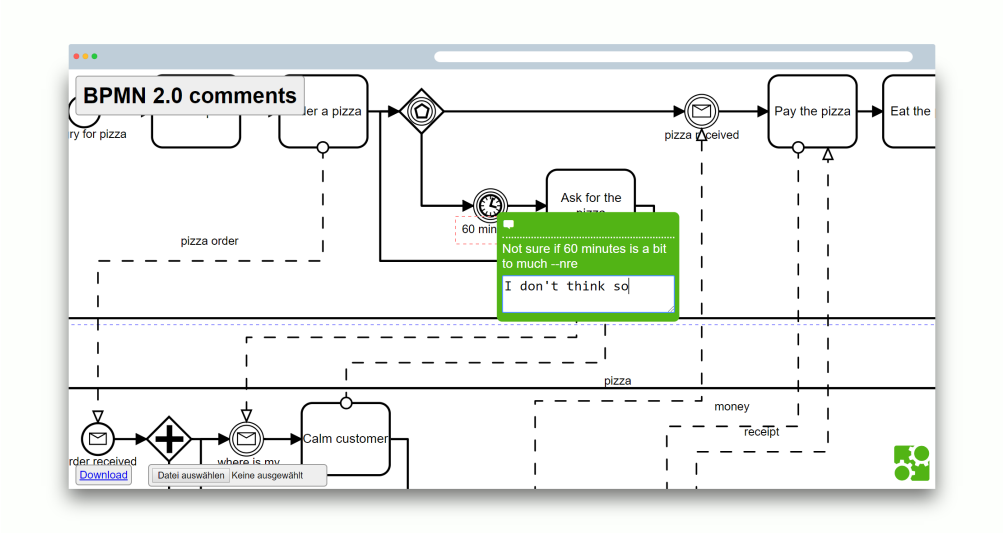


Abbildung 3.2.: Beispiel eines Kommentars[9]

3.2.2. Hinweise anzeigen

Das Hauptziel des Tool ist es, dem Benutzer dabei zu helfen sein Modell zu optimieren. Hierfür ist das Anzeigen von hilfreichen Hinweisen unabdingbar. Dies könnte sich gut mit den bereits erwähnten Overlays umsetzen lassen. Overlays können ganz individuell, beispielsweise bei Interaktion mit einem Element, konfiguriert und dargestellt werden. Beispiele von Overlays sind in Abbildung 3.3 dargestellt.

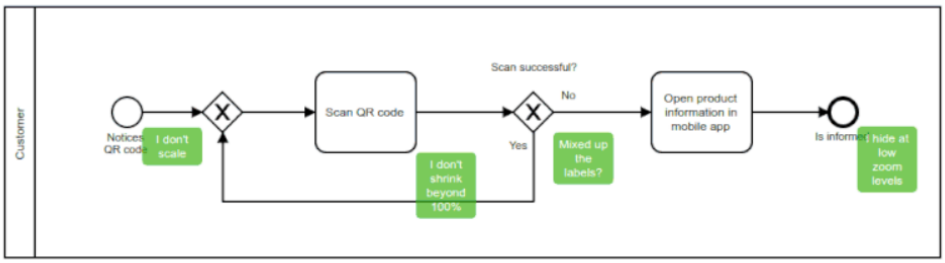


Abbildung 3.3.: Beispiel Hinweises als Overlay[10]

3.2.3. Color Code

Bei BPMN-JS gibt es mehrere Ansätze, die für die Umsetzung des Color Codes hilfreich sein können. Im Beispiel Projekt wird hierbei auf folgende vier Implementationsmöglichkeiten eingegangen:

- Overlay
- BPMN 2.0 Extension
- Marker + CSS Styling
- Custom Render

Das Ergebnis mit Farben könnte ungefähr so aussehen, wie in Abbildung 3.4 gezeigt.

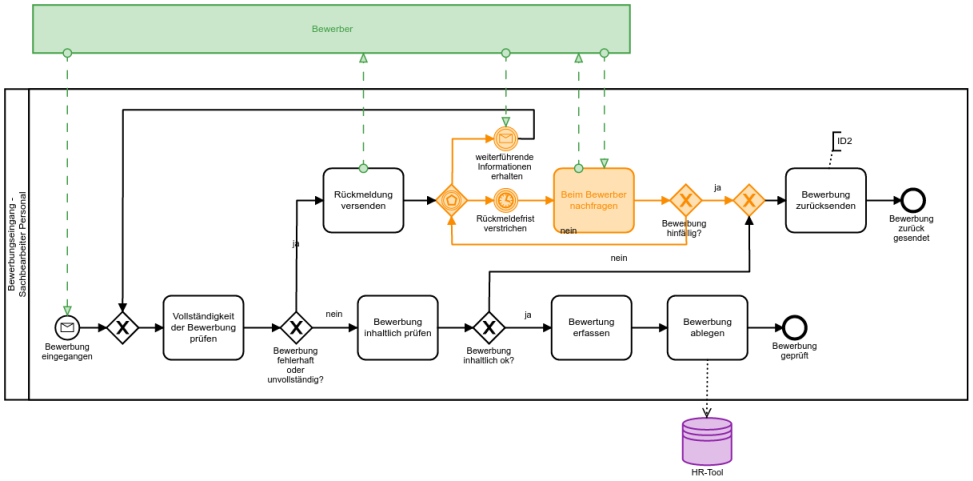


Abbildung 3.4.: Beispiel einer möglichen Umsetzung des Color Codes[11]

3.2.4. BPMN Icons

BPMN-JS setzt bei der Verwendung von Symbolen auf eine eigene Schriftart. Sämtlich Icons (siehe Abbildung 3.5) im GUI werden mit dieser Font umgesetzt. Die Schrift[6] ist Open Source und die Symbole liegen im SVG Format vor. Für SimBPMN müsste die Font geforkt werden, damit das Set an Symbolen um gegebenenfalls benötigte SimBPMN Symbole erweitert werden kann.

[6] `https://github.com/bpmn-io/bpmn-font`

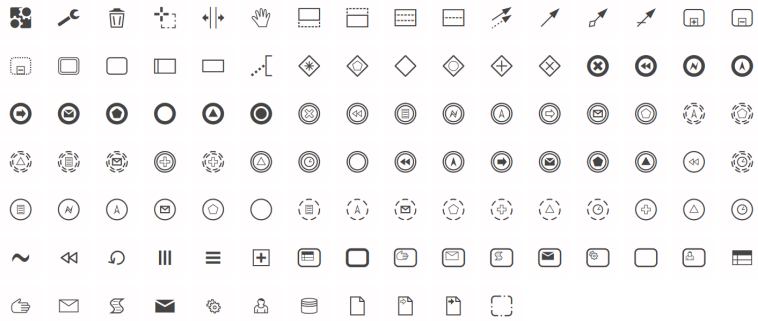


Abbildung 3.5.: Derzeit verfügbare Zeichen in der bpmn-font[12]

3.3. Was kann BPMN-JS noch nicht

Beim Modellieren der Beispielprojekte war die Webseite `https://demo.bpmn.io/` eine sehr gute Orientierung. Diesee implementieren ein minimales BPMN-JS, das auf den zweiten Blick noch ein paar Wünsche offen lassen. Diese Quality of Life Verbesserungen liegen in der Verantwortung der App und nicht von BPMN-JS selbst. Die Verbesserungen könnten aber zum Teil durch bereits existierende, aber hier nicht geladene, Plugins gelöst werden.

3.3.1. Notwendige Erweiterungen

Damit eine SimBPMN Applikation auf BPMN-JS basieren kann, müssen sowohl das bpmn-moddle, als auch das GUI erweitert werden. Camunda, die Entwickler hinter BPMN-JS, haben hierfür eine moddle Erweiterung geschrieben, mit der Custom Properties eingeführt werden können. Für die weiterführende Arbeit wird daran ein Beispiel[7] genommen.

Aufgrund des Aufbaus der Library gibt es kein spezifiziertes Pluginformat. Die Libray bietet Dependency Injection für tiefergreifende Plugins.

> Under the hood, diagram-js employs dependency injection (DI) to wire and discover diagram components. This mechanism is built on top of didi.[13]

Weitere oberflächliche Plugins dieser Art können direkt in der eigenen Applikation implementiert werden.

Das bereits erwähnte Github Projekt bpmn-js-examples[8] hat verschiedene Beispiele für

[7] `https://github.com/camunda/camunda-bpmn-moddle`
[8] `https://github.com/bpmn-io/bpmn-js-examples`

Plugins, die verschieden tiefer in die Materie einsteigen.

1. Starter
2. Basic
3. Intermediate
4. Advanced

Starter versteht sich als erster Kontakt mit einem vorpaketierten BPMN-JS. Die Basic Beispiele gehen von *Bundlings*, über die Funktionalität von *Interaktionen* und *Overlays*, bis hin zur Möglichkeit des *Einfärbens der Diagramme*. Im Intermediate Teil werden *Custom Properties*, *Properties Panel*, *Theming*, *modeling-api commenting* und Internationalisierung eingeführt. Die Advanced Beispiele handeln zuletzt von den Erweiterungen des *Models* und des *GUIs*. Komplexe Änderungen benötigen dementsprechend mehr Einarbeitungszeit in die Library.

3.3.2. File-Name

Aktuell heissen alle exportierten Dateien immer diagram.bpmn oder diagram.svg. Dies ist ziemlich unpraktisch, da innerhalb kürzester Zeit viele verschiedene Diagramme oder Versionen von Diagrammen angelegt werden. Die Möglichkeit, dem Projekt einen Namen zu geben würde dem Tool einen grossen Mehrwert bezüglich Usability bringen.

3.3.3. Refresh

Ein weiterer störender Faktor ist das Neuladen der Webseite. Wird die Seite neu geladen, ist der gesamte Fortschritt des modellierten Diagramms weg. Gewisse Beispielprojekte speichern jedoch den Zustand bereits in der Browser Session.

3.3.4. Speichern

Für das Speichern von angehängten Dokumenten muss geklärt werden, wie dies am besten portabel gemacht werden kann. Falls das Tool als Applikation auf dem Rechner des Benutzers läuft, müsste ein Export eines Projekts alle Dateien enthalten. Eine andere Variante wäre es, das Tool mittels einer Client Server Applikation aufzubauen. Hier würden auf dem Server die Dokumente abspeichert werden. Ein Export-File könnte dann entweder alle Dokumente enthalten, oder auch nur auf den absoluten Pfad verweisen.

Zur Veranschaulichung: Angenommen unser Tool wird lokal ausgeführt. Alice hängt ein Dokument an ein Element im Diagramm an. Dieses verweist auf den lokalen Systempfad von Alice. Beispielsweise *C:\Alice\Dokumente\Lorem.pdf*. Alice exportiert nun das .bpmn File damit es Bob bearbeiten kann. Der Pfad im .bpmn File verweist aber immer noch auf das File auf Alice's Computer. Bob kann daher nicht darauf zugreifen.

Ein SimBPMN Dateiformat könnte ein ZIP sein, das mehrere .bpmn Dateien mit SimBPMN Syntax beinhaltet. Des Weiteren könnte es eine Datei mit Metainformationen enthalten, welche die verwendete Version und weitere Informationen beinhaltet. Die Anhänge müssten dann auch in der Datei enthalten sein.

3.4. Pipeline zur Generierung von Bildern aus .bpmn Files

BPMN-JS kann Dateien als XML basierte .bpmn Dateien exportieren. Da .bpmn Files wie bereits erwähnt schlecht menschenlesbar sind und es durch das Lesen der XML Datei kaum erkennbar ist, welches Diagramm dahinter steht, wurde nach einer Möglichkeit gesucht auf einfachste Weise aus den .bpmn Dateien PNG Bilder zu erstellen. Um dieses Problem zu beheben wurde eine CI/CD Pipeline gebaut, welche die automatische Umwandlung von vorhandenen .bpmn Dateien zu PNG Dateien vornimmt.

Der Prozess, bevor es die CI/CD Pipeline gab:

1. BPMN File öffnen oder neu erstellen
2. BPMN File herunterladen (diagram.bpmn)
3. SVG File herunterladen
4. BPMN File umbenennen und ablegen
5. SVG in ein PNG konvertieren
6. PNG einbinden im Dokument (falls noch nicht geschehen)

Punkte 1. bis 5. müssen bei jeder Änderung durchgeführt werden. Die Diagramme werden beim Schliessen des Tabs nicht automatisch gespeichert, daher müssen die .bpmn Files häufig heruntergeladen werden. Da die Diagramme nicht automatisch gespeichert werden wenn der Tab geschlossen wird, mussten die .bpmn Files häufig herunter geladen werden. Ansonsten ist mit einer unfreiwilligen Löschung der bereits erstellten Diagramme zu rechnen. Dies führt zu dem Nachteil, dass die Bilder erst nach Beendigung der Arbeit eingefügt werden können. Der Aufwand dazu war ansonsten zu gross.

Für die Konvertierung der unterschiedlichen Formate wurde das Tool bpmn-js-cmd[9] verwendet. Die Applikation nutzt BPMN-JS und Puppeteere[10]. Hierbei handelt es sich um eine high-level API um den Browser Chrome oder Chromium zu steuern. Darüber hinaus ist es damit möglich einen Screenshot des Diagramms anzufertigen. Um ein Bild eines Diagramms zu erstellen, wird im ersten Schritt ein Browser, in diesem Fall Google Chrome oder Chromium, geöffnet. Anschliessend wird eine Instanz der bpmn-js basierenden Applikation geladen, worin dann das Diagramm geöffnet wird um davon ein Screenshot zu machen.

Um bpmn-js-cmd mit der Gitlab-CI verwenden zu können, muss es in einem Docker Container lauffähig gemacht werden. Dieser Dockerc Container wiederum muss auf dem Integration Server zum laufen gebracht werden. Dies ist insofern eine Herausforderung, da auf dem Integration Server kein Chrome installiert ist und dieser somit mit in den Container mit eingebunden werden muss. Um die Lauffähigkeit zu ermöglichen, musste zuerst ein Fork[11] erstellt werden, der die Ausführung des Google Chromes ohne Sandbox erlaubt. Der Fork wurde Upstream als Patch eingereicht und vom Owner des Projekts akzeptiert. In Abbildung 3.3 ist das entsprechede Dockerfile, auf dem der Container gebaut werden kann, zu sehen.

Der Fork fügt ein Argument bei Puppeteer hinzu, womit die Ausführung des Chromes im Docker ohne Sandbox erlaubt wird. Dies ist unter normalen Umständen nicht erwünscht. Ohne dieses Argument könnte das Tool nicht im Docker Container ausgeführt werden.

In Abbildung 3.4 ist der Continuous Integration (CI) Job zu sehen, der die Bilder automatisch generiert.

```
FROM buildkite/puppeteer AS dependencies
RUN apt-get update; apt-get install -y git
ENV PATH="${PATH}:/node_modules/.bin"
ENV CHROMIUM_PATH="/usr/bin/google-chrome"
ENV PUPPETEER_SKIP_CHROMIUM_DOWNLOAD="true"

RUN npm install --save -g github:gtudan/bpmn-js-cmd

FROM dependencies
USER 1001
WORKDIR /app
ENTRYPOINT ["bpmn-js"]
```

Listing 3.3: Dockerfile für bpmn-js-cmd

```
image: alpine:latest
```

[9] https://github.com/gtudan/bpmn-js-cmd
[10] Pupeteer https://github.com/puppeteer/puppeteer
[11] https://github.com/indero/bpmn-js-cmd/tree/docker_container

```
variables:
DOCKER_DRIVER: overlay2

stages:
- convert

convert-to-images:
stage: convert
image:
name: $CI_REGISTRY/simulationstool/bpmn-diagramme/docker:latest
entrypoint: [""]
before_script:
- mkdir output
script:
- time find . -name \*.bpmn -type f -print0 | xargs -0 -n1 -P8 bpmn-js
- time find . -name \*.bpmn -type f -print0 | xargs -0 -n1 -P8 bpmn-js -t
    png
after_script:
- find \( -name "*.svg" -o -name "*.png" -type f \) -print0 | xargs -0 -n1 -
    P8 -I '{}' mv '{}' output/
artifacts:
paths:
- output
expire_in: 1 month
```

Listing 3.4: gitlab-ci.yml für bpmn-js-cmd

Das ist ein Beispiel für die Anpassungsfähigkeit von BPMN-JS. Bei einer Erweiterung für SimBPMN muss auch dieses Tool entsprechend erweitert werden. Es besteht die Möglichkeit die Erweiterung als Plugin zu implementieren, das einfach eingebunden werden kann.

3.5. Erste Erfahrungen mit BPMN-JS

Um zu verifizieren, dass die gewünschten Features mit BPMN-JS möglich sind, wurde ein kleiner Prototyp gebaut. Dieser Prototyp baut auf dem Beispiel Projekt bpmnlint-playground[12] auf. Das bpmnlint-playground Beispielprojekt basiert auf bpmn-js-bpmnlit. Bpmn-js-bpmlint wiederum integriert bpmlint in bpmn-js. Bpmn-js-bpmnlint bietet bereits eine integrierte Syntaxprüfung für BPMN 2.0. Dies ist für die Verwendung mit SimBPMN sehr vielversprechend.

Konkret waren folgende Fragen zu klären:

[12] `https://github.com/bpmn-io/bpmnlint-playground`

- Gibt es eine einfache Möglichkeit, einen Hilfstext zu einem Element darzustellen? Mit dem Hilfstext soll beschrieben werden, welche Funktionalität das jeweilige Element besitzt.
- Gibt es eine Möglichkeit, Dokumente an Elemente anzufügen?

Für Hinweise zu Objekten und Custom Attributes wurde das Properties-Panel bpmn-js-properties-panel[13] ausprobiert (siehe Abbildung 3.6). Damit handelt es sich um einen möglichen Kandidaten, um Features für die oben gestellten Fragen zu implementieren.

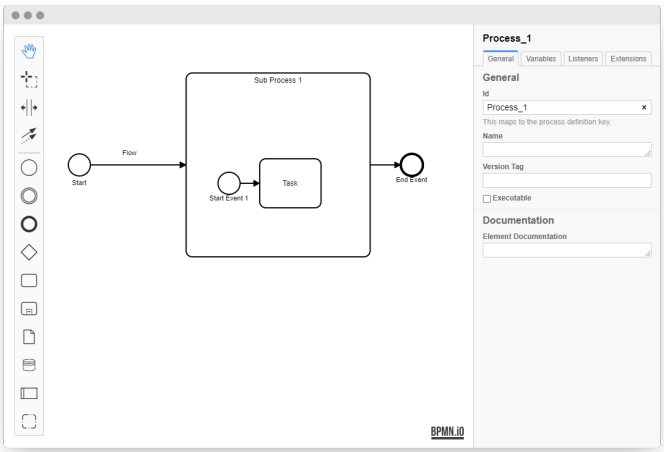


Abbildung 3.6.: Beispiel von BPMN-JS mit dem Properties Panel

3.6. Erkenntnisse zum Prototyp

Das Implementieren eigener Properties stellte sich als komplizierter heraus als Anfangs gedacht. Die einfachste Möglichkeit scheint zu sein, BPMN-JS mit einem oder mehreren Plugins zu erweitern. Camunda[14], die Entwickler hinter BPMN-JS und dem kommerziellen Produkt Camunda verwenden für ihre eigenen Erweiterungen ein eigenes „Model“. Hierzu stellen sie Anleitungen zur Verfügung die zeigen, wie solche Erweiterungen selbst implementiert werden können[15]. Die Erstellung eines eigenen Models übersteigt den Scope des Prototypen dieser Arbeit, da hierfür eine längere Einarbeitungszeit mit dem Framework nötig wäre.

[13] `https://github.com/bpmn-io/bpmn-js-properties-panel`
[14] `https://github.com/camunda/camunda-bpmn-moddle`
[15] `https://github.com/bpmn-io/bpmn-js-example-model-extension`

Wir gehen hier von einem lohnenswerten Einsatz aus, denn die Grundlagen, die das Framework bietet, entsprechen den erwarteten Software Engineering Vorstellungen.

3.7. Ergebnis

Der Prototyp und auch das Tool um automatisch Bilder zu generieren haben gezeigt wie gut der Umgang mit der Library ist. Hier wäre es wünschenswert gewesen sich noch genauer mit der Library auseinander zu setzen, was aber aus Zeitgründen nicht mehr möglich war.

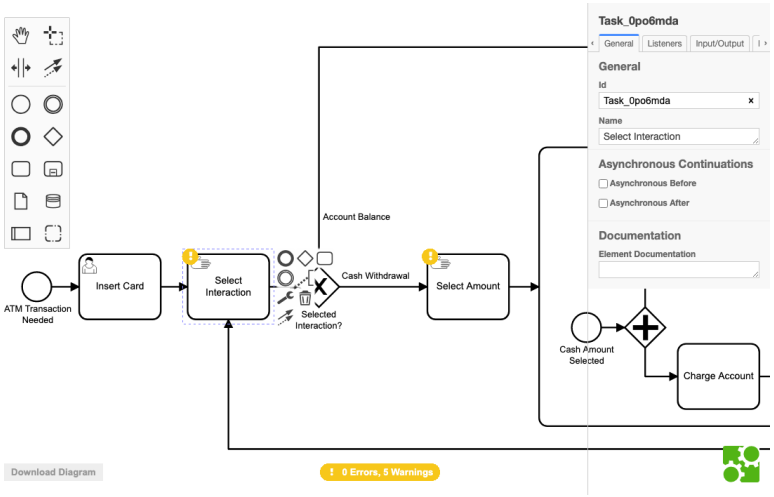


Abbildung 3.7.: Prototyp mit aktiviertem BPMN Lint und Propteries Panel

4. Ausblick

In dieser Arbeit wurde gezeigt dass es möglich ist, ein auf SimBPMN basierendes Tool zur Unterstützung beim Modellieren zu schreiben.

Die auf BPMN-JS basierenden Tools, die in dieser Studienarbeit analysiert wurden, geben einen Ausblick auf die mögliche Implementation in der Bachelorarbeit. In dieser wird eine eine auf BPMN-JS aufbauende SimBPMN Applikation implementiert.

Bei der Reihenfolge der unten aufgeführten Prototypen handelt es sich um ein Best Case Szenario. Der Fokus liegt jedoch auf dem SimBPMN Plugin oder der Plugin Sammlung.

4.1. Quality of Life

Damit das eigentliche Entwickeln so unkompliziert wie möglich ablaufen kann, wird zuerst ein Prototyp erstellt. Hier wäre der Wechsel von einer Webapp zu Electron oder zu self-hosted in Erwägung zu ziehen. Zu weiteren unabdingbaren Funktionalitäten zählen:

- Das automatische Speichern der Dokumente
- Die Möglichkeite die Projekte und Diagramme zu Versionieren
- Das Erstellen von Projekten bestehend aus mehreren Diagrammen
- Eine Halb-Automatische Namensgebung der Dateien beim Export der Diagramme

Durch diesen Schritt fällt es leichter, sich mit der BPMN-JS Architektur vertraut zu machen.

4.2. Vorhandene Plugins Integrieren

Es wurden verschiedene Plugins evaluiert, die für sich alleinstehend dem GUI schon relativ viele Features hinzufügen. Diese müssten in einer SimBPMN Applikation zusammengefasst werden.

Denkbar sind hier eine Electron App mit Singlepage Applikation, oder als Alternative dazu eine Webapp, die auf einem Server läuft. Die Architektur muss auf jeden Fall so gestaltet werden, dass selektive Tasks an einen Server ausgelagert werden können.

Auch das Verlinken von Diagrammen, damit verschiedene Detailierungsgrade abgebildet werden können, muss realisiert werden. Sowie das Zusammenfassen von Diagrammen zu einem Projektfile um die Portabilität zu gewährleisten. Evtl. Anfügen von Attachments. Als Austausch Format bei Projekten mit Attachments könnte zum Beispiel ein Zip fungieren, das alle Dokumente und Diagramme beherbergt.

Dieser Schritt dient dazu sich mit der BPMN-JS Architektur vertrauter zu machen. Ausserdem soll in diesem Schritt erarbeitet werden, auf welche Weise die Features in den Plugins implementiert werden.

4.3. SimBPMN Plugin

Bei den Plugins handelt es sich um den eigentliche Kern des Ganzen. Sie erweitern das BPMN Model um die benötigten SimBPMN Eigenschaften. Zudem erlauben sie die Gestaltung und Implementierung von eignen Symbolen. Auch eige Linting Regeln lassen dich durch Plugins verwirklichen.

4.4. SimBPMN Tools

Die Grafiken werden im besten Fall mit dem zu erstellenden Tool angefertigt. Daher ist es wichtig, dass dieser Schritt des Erstellen des Prototypen simultan durchgeführt wird.

Beispielsweise ein Tool, dass aus SimBPMN Dateien Bilder erzeugt und gleichzeitig alle relevanten Plugins installiert.

Anhang

Glossar

terminal Ein Terminalsymbol (auch Terminalzeichen oder kurz Terminal genannt) einer formalen Grammatik ist ein Symbol, das einzeln nicht weiter durch eine Produktionsregel ersetzt werden kann. 12–14

Akronyme

ASIM Fachgruppe Arbeitsgemeinschaft Simulation. 3, 4

BPMN Business Process Model and Notation. 3, 6, 8, 9, 21, 30–33, 41

CI Continuous Integration. 40

SimBPMN Business Process Model and Notation for Simulations. 3–6, 8, 24, 25, 30, 31, 33, 34, 36, 37, 39, 41, 44, 45

Anhang A.

Literaturverzeichnis

[1] "Was ist BPMN? - Wissen kompakt - t2informatik," https://t2informatik.de/wissen-kompakt/bpmn/.

[2] A. Rinkel, "Modellbildung und Simulation - 01. Einführung und Übersicht."

[3] A. Rinkel and L. Hollenstein, "Einsatz formaler Methoden in der Simulation," Nov. 2018.

[4] B. Page and W. Kreutzer, "Simulating discrete event systems with UML and JAVA," *Environmental Science and Pollution Research*, vol. 13, no. 6, pp. 441–441, Oct. 2006.

[5] Z. A. Badran, "Eine Methode zur Simulation von Geschäftsprozessen unter dem Aspekt der Energieeffizienz," Master's thesis, Universität Zürich.

[6] "Prof. Dr. Josef L. Staud," http://www.staud.info/index.htm.

[7] W. W. A. Initiative (WAI), "Designing for Web Accessibility – Tips for Getting Started," https://www.w3.org/WAI/tips/designing/, Dec. 2020.

[8] "Bpmn-io/bpmn-js-examples/tree/master/properties-panel-extension," https://github.com/bpmn-io/bpmn-js-examples.

[9] "Bpmn-io/bpmn-js-examplestree/master/commenting," https://github.com/bpmn-io/bpmn-js-examples.

[10] "Bpmn-io/bpmn-js-examplestree/master/overlays," https://github.com/bpmn-io/bpmn-js-examples.

[11] "Colors are Here | bpmn.io," https://bpmn.io/blog/posts/2016-colors-bpmn-js.html.

[12] "Bpmn-io/bpmn-font," bpmn.io, Dec. 2020.

[13] "Walkthrough | bpmn.io," https://bpmn.io/toolkit/bpmn-js/walkthrough/.

Anhang B.

Abbildungsverzeichnis

Anhang C.

Listings

Anhang D.

Dokumente

D.1. BPMN-JS Walkthrough

bpmn-js

Toolkit Download Examples Walkthrough

Getting familiar with bpmn-js, one step at a time.

This document is a work in progress. ***Help us to improve it****.*

A Quick Introduction

bpmn-js is a BPMN 2.0 rendering toolkit and web modeler. It is written in JavaScript, embeds BPMN 2.0 diagrams into modern browsers and requires no server backend. That makes it easy to embed it into any web application.

The library is built in a way that it can be both a viewer and web modeler. Use the **viewer** to embed BPMN 2.0 into your applications and **enrich it with your data**. Use the **modeler** to create BPMN 2.0 diagrams inside your application.

This walkthrough will give you an introduction on how to use the library as well as some insights into its internals, i.e. the components that contribute to its highly modular and extensible structure.

Contents

Using the Library

There are two approaches to use **bpmn-js** in your application. An all-in-one pre-packaged version of the library allows you to quickly add BPMN to any website. The **npm** version is more complicated to set-up but gives you access to individual library components and allows for easier extensibility.

This section gives you an overview of both approaches. We start with an introduction on how to embed the pre-packaged version of the BPMN viewer into a website. Following that, we show how to bundle bpmn-js with your application to create a web-based BPMN editor.

Embed the Pre-Packaged Viewer

The **pre-packaged version** of bpmn-js allows you to embed BPMN to your website with a simple script include.

Add a container element for the rendered diagram to your website and include the library into the page.

```
<!-- BPMN diagram container -->
<div id="canvas"></div>

<!-- replace CDN url with local bpmn-js path -->
<script src="https://unpkg.com/bpmn-js/dist/bpmn-viewer.development.js"></
```

The included script makes the viewer available via the `BpmnJS` variable. We may access it via JavaScript as shown below.

```
<script>
  // the diagram you are going to display
  var bpmnXML;

  // BpmnJS is the BPMN viewer instance
  var viewer = new BpmnJS({ container: '#canvas' });

  // import a BPMN 2.0 diagram
  viewer.importXML(bpmnXML, function(err) {
    if (err) {
      // import failed :-(
```

```
    } else {
      // we did well!

      var canvas = viewer.get('canvas');
      canvas.zoom('fit-viewport');
    }
  });
</script>
```

The snippet uses the **`Viewer#importXML`** API to display a pre-loaded BPMN 2.0 diagram. Importing a diagram is asynchronous and, once finished, the viewer notifies us via a callback about the results.

After import, we may access various diagram services via **`Viewer#get`**. In the snippet above, we interact with the **`Canvas`** to fit the diagram to the currently available viewport size.

Often times it is more practical to load the BPMN 2.0 diagram dynamically via AJAX. This can be accomplished using plain JavaScript (as seen below) or via utility libraries such as **jQuery**, which provide more convenient APIs.

```
<script>
  var xhr = new XMLHttpRequest();

  xhr.onreadystatechange = function() {
    if (xhr.readyState === 4) {
      viewer.importXML(xhr.response, function(err) {
        // ...
      });
    }
  };

  xhr.open('GET', 'path-to-diagram.bpmn', true);
  xhr.send(null);
</script>
```

Check out the **pre-packaged example** as well as our **starter examples** to learn more.

Roll Your Own Modeler

Use bpmn-js via **npm** if you would like to build customizations around the library. This approach has various advantages such as access to individual library components. It also gives us more control over what to package as part of the viewer / modeler. However, it requires us to bundle bpmn-js with our application using an ES module aware bundler such as **Webpack**.

> If you are new to the world of JavaScript bundling follow along our **bundling example**.

In the remainder of this section we loosely follow the **modeler example** to create a web-based BPMN editor.

Include the Library

First install bpmn-js via **npm**:

```
npm install bpmn-js
```

Then access the BPMN modeler via an ES `import`:

```
import Modeler from 'bpmn-js/lib/Modeler';

// create a modeler
var modeler = new Modeler({ container: '#canvas' });

// import diagram
modeler.importXML(bpmnXML, function(err) {
  // ...
});
```

Again, this assumes you provide an element with the id `canvas` as part of your HTML for the modeler to render into.

Add Stylesheets

When embedding the modeler into a webpage, include the **diagram-js** stylesheet as well as the **BPMN icon font** with it. Both are shipped with the bpmn-js distribution under the **`dist/assets` folder**.

```
<link rel="stylesheet" href="bpmn-js/dist/assets/diagram-js.css" />
<link rel="stylesheet" href="bpmn-js/dist/assets/bpmn-font/css/bpmn.css" />
```

Adding the stylesheets ensures diagram elements receive proper styling as well as context pad and palette entries show BPMN symbols.

Bundle for the Browser

bpmn-js and its dependencies distribute **ES modules**. Use an ES module aware bundler to pack bpmn-js along with your application. Learn more by following along with the **bundling example**.

Hook into Life-Cycle Events

Events allow you to hook into the life-cycle of the modeler as well as diagram interaction. The following snippet shows how changed elements and modeling operations in general can be captured.

```
modeler.on('commandStack.changed', function() {
  // user modeled something or
  // performed an undo/redo operation
});

modeler.on('element.changed', function(event) {
  var element = event.element;

  // the element was changed by the user
});
```

Use `Viewer#on` to register for events or the `EventBus` inside extension modules. Stop listening for events using the `Viewer#off` method. Check out the **interaction example** to see listening for events in action.

Extend the Modeler

You may use the `additionalModules` option to extend the `Viewer` and `Modeler` on creation. This allows you to pass custom *modules* that amend or replace existing functionality.

```
import OriginModule from 'diagram-js-origin';

// create a modeler
var modeler = new Modeler({
  container: '#canvas',
  additionalModules: [
    OriginModule,
    require('./custom-rules'),
    require('./custom-context-pad')
  ]
});
```

A *module* (cf. **Module System section**) is a unit that defines one or more named *services*. These services configure bpmn-js or provide additional functionality, i.e. by hooking into the diagram life-cycle.

Some modules, such as **diagram-js-origin** or **diagram-js-minimap**, provide generic user interface additions. Built-in bpmn-js modules, such as **bpmn rules** or **modeling**, provide highly BPMN-specific functionality.

One common way to extend the BPMN modeler is to add **custom modeling rules**. In doing so, you can limit or extend the modeling operations allowed by the user.

Other examples for extensions are:

- **Adding custom elements**
- **Custom palette / context pad**
- **Custom shape rendering**

Check out the **bpmn-js-examples project** for many more toolkit extension show cases.

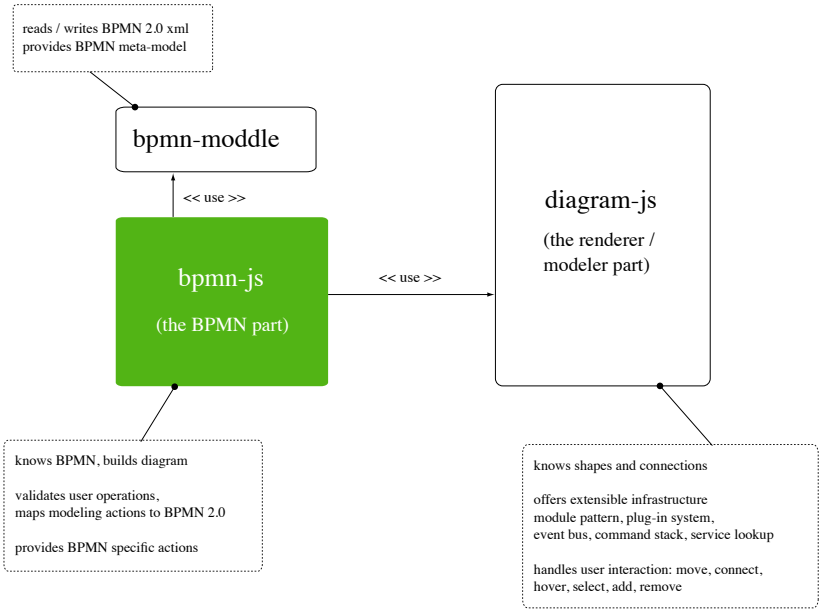
Build a Custom Distribution

If you would like to create your own pre-packaged version of your custom modeler or viewer, refer to the **custom-bundle** example. This could make sense if you carried out heavy customizations that you would like to ship to your users in simple way.

Understanding bpmn-js Internals

This section explores some **bpmn-js** internals.

As depicted in the architecture diagram below, bpmn-js builds on top of two important libraries: **diagram-js** and **bpmn-moddle**.



bpmn-js Architecture: Parts and Responsibilities

We use **diagram-js** to draw shapes and connections. It provides us with ways to interact with these graphical elements as well as additional tools such as overlays that help users to build powerful BPMN viewers. For advanced use cases such as modeling it contributes the context pad, palette and facilities like redo/undo.

bpmn-moddle knows about the BPMN 2.0 meta-model defined in the **BPMN 2.0 standard**. It allows us to read and write BPMN 2.0 schema-compliant XML documents and access BPMN-related information behind shapes and connections drawn on the diagram.

On top of these two libraries, bpmn-js defines the BPMN specifics such as look and feel, modeling rules and tooling (i.e. palette). We will go into detail about the individual components in the following paragraphs.

Diagram Interaction / Modeling (diagram-js)

diagram-js is a toolbox for displaying and modifying diagrams on the web. It allows us to render visual elements and build interactive experiences on top of them.

It provides us with a very simple module system for building features and dependency injection for service discovery. This system also provides a number of core services that implement the diagram essentials.

Additionally, diagram-js defines a data model for graphical elements and their relationships.

Module System

Under the hood, **diagram-js** employs dependency injection (DI) to wire and discover diagram components. This mechanism is built on top of **didi**.

When talking about *modules* in the context of diagram-js, we refer to units that provide named services along with their implementation. A *service* in that sense is a function or instance that may consume other services to do stuff in the context of the diagram.

The following shows a service that **hooks into life-cycle events**. It does so by registering an event via the `eventBus`, another well-known service:

```
function MyLoggingPlugin(eventBus) {
  eventBus.on('element.changed', function(event) {
    console.log('element ', event.element, ' changed');
  });
}

// ensure the dependency names are still available after minification
MyLoggingPlugin.$inject = [ 'eventBus' ];
```

We must publish the service under a unique name using a module definition:

```
import CoreModule from 'diagram-js/lib/core';

// export as module
export default {
  __depends__: [ CoreModule ], // {2}
  __init__: [ 'myLoggingPlugin' ], // {3}
  myLoggingPlugin: [ 'type', MyLoggingPlugin ] // {1}
};
```

The definition tells the DI infrastructure that the service is called `myLoggingPlugin {1}`, that it depends on the diagram-js core module `{2}` and that the service should be initialized upon diagram creation `{3}`. For more details have a look at the **didi documentation**.

We may now bootstrap diagram-js, passing our custom module:

```
import MyLoggingModule from 'path-to-my-logging-module';

var diagram = new Diagram({
  modules: [
    MyLoggingModule
  ]
});
```

To plug in the module into **bpmn-js**, you would use the `additionalModules` option as shown in the **Extend the Modeler section**.

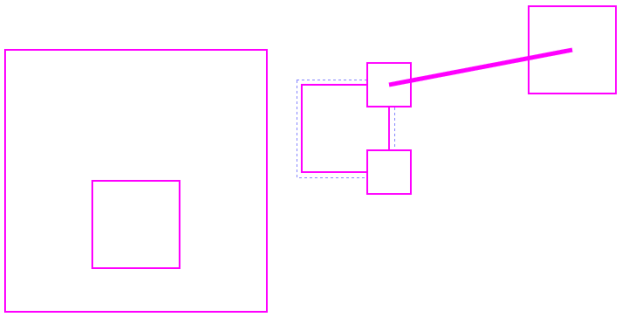
Core Services

The **diagram-js core** is built around a number of essential services:

- **`Canvas`** - provides APIs for adding and removing graphical elements; deals with element life cycle and provides APIs to zoom and scroll.
- **`EventBus`** - the library's global communication channel with a *fire and forget* policy. Interested parties can subscribe to various events and act upon them once they are emitted. The event bus helps us to decouple concerns and to modularize functionality so that new features can hook up easily with existing behavior.
- **`ElementFactory`** - a factory for creating shapes and connections according to diagram-js' internal data model.
- **`ElementRegistry`** - knows all elements added to the diagram and provides APIs to retrieve the elements and their graphical representation by *id*.
- **`GraphicsFactory`** - responsible for creating graphical representations of shapes and connections. The actual look and feel is defined by renderers, i.e. the **`DefaultRenderer`** inside the **draw module**.

Data Model

Under the hood, diagram-js implements a simple data model consisting of shapes and connections.



Data Model Essentials: Shapes and Connections

A *shape* has a parent, a list of children as well as a list of incoming and outgoing *connections*.

A *connection* has a parent as well as a source and target, pointing to a *shape*.

The **`ElementRegistry`** is responsible for creating shapes and connections **according to that model**. During modeling, element relationships will be updated according to user operations by the **`Modeling` service**.

Auxiliary Services (i.e. the Toolbox)

Aside from the data model and its core services, diagram-js provides a rich toolbox of additional helpers.

- **`CommandStack`** - responsible for redo and undo during modeling.
- **`ContextPad`** - provides contextual actions around an element.
- **`Overlays`** - provides APIs for attaching additional information to diagram elements.
- **`Modeling`** - provides APIs for updating elements on the canvas (moving, deleting)
- **`Palette`**
- ...

Let's move on to the BPMN magic that is happening behind the scenes.

BPMN Meta-Model (bpmn-moddle)

bpmn-moddle encapsulates the BPMN 2.0 meta-model and provides us with facilities to read and write BPMN 2.0 XML documents. On import, it parses the XML document into a JavaScript object tree. That tree is edited and validated during modeling and then exported back to BPMN 2.0 XML once the user wishes to save the diagram. Because bpmn-moddle encapsulates knowledge about BPMN, we are able to validate during import and modeling. Based on the results, we can constrain certain modeling actions and output helpful error messages and warnings to the user.

Much like **bpmn-js**, the foundations of **bpmn-moddle** are built on top of two libraries:

- **moddle** which offers a concise way to define **meta-models** in JavaScript
- **moddle-xml** which reads and writes XML documents based on **moddle**

In essence **bpmn-moddle** adds the BPMN spec as a meta-model and offers a simple interface for BPMN schema validation. From the library perspective it provides the following API:

- `fromXML` - create a BPMN tree from a given XML string
- `toXML` - write a BPMN object tree to BPMN 2.0 XML

The BPMN meta-model is essential for bpmn-js, as it allows us to validate BPMN 2.0 documents we consume, provide proper modeling rules and export valid BPMN documents that *all* compliant BPMN modelers can understand.

Plugging Things Together (bpmn-js)

We learned **bpmn-js** is built on top of **diagram-js** and **bpmn-moddle**. It ties both together and adds the BPMN look and feel. This includes a BPMN palette, BPMN context pad as well as BPMN 2.0 specific rules. In this section, we'll be explaining how that works in different phases of modeling.

When we import a BPMN 2.0 document, it is parsed from XML into an object tree by **bpmn-moddle**. bpmn-js renders all visible elements of that tree, i.e. it creates the respective shapes and connections on the canvas. Thereby it ties both the BPMN elements and the graphical elements together. This results in a structure, as shown below, for a start event shape.

```
{
  id: 'StartEvent_1',
  x: 100,
  y: 100,
  width: 50,
```

```
    height: 50,
    businessObject: {
      $attrs: Object
      $parent: {
        $attrs: Object
        $parent: ModdleElement
        $type: 'bpmn:Process'
        flowElements: Array[1]
        id: 'Process_1'
        isExecutable: false
      }
      $type: 'bpmn:StartEvent'
      id: 'StartEvent_1'
    }
  }
```

You may access the underlying BPMN type from each graphical element via the `businessObject` property.

bpmn-js also knows how each BPMN element looks like thanks to the **`BpmnRenderer`**. By plugging into the render cycle, you may also define custom representations of individual BPMN elements.

We can start modeling once the importing is done. We use rules to allow or disallow certain modeling operations. These rules are defined by **`BpmnRules`**. We base these rules on the BPMN 2.0 standard as defined by the **OMG**. However as mentioned earlier, others may also hook up with the rule evaluation to contribute different behavior.

The **modeling module** bundles BPMN 2.0 related modeling functionality. It adds BPMN 2.0 specific modeling behaviors and is responsible for updating the BPMN 2.0 document tree with every modeling operation carried out by the user (cf. **`BpmnUpdater`**). Check it out to get a deeper insight into rules, behaviors and the BPMN update cycle.

When looking at bpmn-js purely from the library perspective, it's worth mentioning it can be used in three variants:

- **`Viewer`** to display diagrams
- **`NavigatedViewer`** to display and navigate BPMN diagrams
- **`Modeler`** to model BPMN diagrams

The only difference between the versions is that they bundle a different set of functionality. The **`NavigatedViewer`** adds modules for navigating the canvas and the

`Modeler` adds a whole lot of functionality for creating, editing and interacting with elements on the canvas.

Going Further

In the first part of this walkthrough, we focused on using bpmn-js as a BPMN viewer as well as a modeler. This should have given you a good understanding of the toolkit from the library perspective.

In the second part, we focused on bpmn-js internals. We presented diagram-js as well as bpmn-moddle, the two foundations bpmn-js is built upon and gave you an overview of how bpmn-js plugs all of these together.

There exists a number of additional resources that allow you to progress further:

- **Examples** - numerous examples that showcase how to embed and extend bpmn-js.
- Source Code (**bpmn-js**, **diagram-js**) - mostly well documented; should give you great insights into the library's internals.
- **Forum** - a good place to get help for using and extending bpmn-js.

Was there anything that we could have explained better / you got stuck with? **Propose an improvement** to this document or tell us about it in our **forums**.

D.2. AND Simple Gateway Beispiel

Beispiel des BPMN 2.0 XML Format. Abbildung 2.7 als Code.

```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn:definitions xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL"
    xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
    xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
    xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
    id="Definitions_154422e" targetNamespace="http://bpmn.io/schema/bpmn"
    exporter="bpmn-js (https://demo.bpmn.io)" exporterVersion="7.3.0">
  <bpmn:process id="Process_00zdgaf">
    <bpmn:startEvent id="Event_05f276v" name="Brief">
      <bpmn:outgoing>Flow_0cyrkyj</bpmn:outgoing>
    </bpmn:startEvent>
    <bpmn:sequenceFlow id="Flow_0cyrkyj" sourceRef="Event_05f276v"
        targetRef="Gateway_10xrnm1"/>
    <bpmn:task id="Activity_19r34lg" name="Ablage">
      <bpmn:incoming>Flow_11l21b4</bpmn:incoming>
      <bpmn:outgoing>Flow_0fh64rf</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="Flow_11l21b4" name="Ablegen"
        sourceRef="Gateway_10xrnm1" targetRef="Activity_19r34lg"/>
    <bpmn:task id="Activity_0js25p0" name="Scan">
      <bpmn:incoming>Flow_1rmbved</bpmn:incoming>
      <bpmn:outgoing>Flow_1sl1o0l</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="Flow_1rmbved" name="Scanen"
        sourceRef="Gateway_10xrnm1" targetRef="Activity_0js25p0"/>
    <bpmn:task id="Activity_0syazcl" name="Digital Ablegen">
      <bpmn:incoming>Flow_1sl1o0l</bpmn:incoming>
      <bpmn:outgoing>Flow_0dnb54k</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="Flow_1sl1o0l" sourceRef="Activity_0js25p0"
        targetRef="Activity_0syazcl"/>
    <bpmn:endEvent id="Event_149a09z">
      <bpmn:incoming>Flow_0dnb54k</bpmn:incoming>
    </bpmn:endEvent>
    <bpmn:sequenceFlow id="Flow_0dnb54k" sourceRef="Activity_0syazcl"
        targetRef="Event_149a09z"/>
    <bpmn:task id="Activity_0xn91r1" name="In Ordner ablegen">
      <bpmn:incoming>Flow_0fh64rf</bpmn:incoming>
      <bpmn:outgoing>Flow_1m1el3b</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="Flow_0fh64rf" sourceRef="Activity_19r34lg"
        targetRef="Activity_0xn91r1"/>
    <bpmn:endEvent id="Event_0nfkfxy">
      <bpmn:incoming>Flow_1m1el3b</bpmn:incoming>
    </bpmn:endEvent>
```

```
    <bpmn:sequenceFlow id="Flow_1m1el3b" sourceRef="Activity_0xn91r1"
        targetRef="Event_0nfkfxy"/>
    <bpmn:parallelGateway id="Gateway_10xrnm1">
      <bpmn:incoming>Flow_0cyrkyj</bpmn:incoming>
      <bpmn:outgoing>Flow_11l21b4</bpmn:outgoing>
      <bpmn:outgoing>Flow_1rmbved</bpmn:outgoing>
    </bpmn:parallelGateway>
    <bpmn:textAnnotation id="TextAnnotation_1d5ctml">
      <bpmn:text>Ein AND Gateway funktioniert hier weil es sich einem Brief
          um ein Composite handelt.</bpmn:text>
    </bpmn:textAnnotation>
    <bpmn:association id="Association_0i85cki" sourceRef="Gateway_10xrnm1"
        targetRef="TextAnnotation_1d5ctml"/>
  </bpmn:process>
  <bpmndi:BPMNDiagram id="BPMNDiagram_1">
    <bpmndi:BPMNPlane id="BPMNPlane_1" bpmnElement="Process_00zdgaf">
      <bpmndi:BPMNShape id="TextAnnotation_1d5ctml_di"
          bpmnElement="TextAnnotation_1d5ctml">
        <dc:Bounds x="320" y="80" width="99.99629300118623"
            height="111.50652431791221"/>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="Flow_0dnb54k_di" bpmnElement="Flow_0dnb54k">
        <di:waypoint x="760" y="450"/>
        <di:waypoint x="872" y="450"/>
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="Flow_1sl1o0l_di" bpmnElement="Flow_1sl1o0l">
        <di:waypoint x="550" y="450"/>
        <di:waypoint x="660" y="450"/>
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="Flow_1rmbved_di" bpmnElement="Flow_1rmbved">
        <di:waypoint x="280" y="305"/>
        <di:waypoint x="280" y="450"/>
        <di:waypoint x="450" y="450"/>
        <bpmndi:BPMNLabel>
          <dc:Bounds x="345" y="433" width="38" height="14"/>
        </bpmndi:BPMNLabel>
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="Flow_11l21b4_di" bpmnElement="Flow_11l21b4">
        <di:waypoint x="305" y="280"/>
        <di:waypoint x="450" y="280"/>
        <bpmndi:BPMNLabel>
          <dc:Bounds x="343" y="262" width="41" height="14"/>
        </bpmndi:BPMNLabel>
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="Flow_0cyrkyj_di" bpmnElement="Flow_0cyrkyj">
        <di:waypoint x="189" y="280"/>
        <di:waypoint x="255" y="280"/>
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="Flow_0fh64rf_di" bpmnElement="Flow_0fh64rf">
        <di:waypoint x="550" y="280"/>
```

```
        <di:waypoint x="660" y="280"/>
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="Flow_1m1el3b_di" bpmnElement="Flow_1m1el3b">
        <di:waypoint x="760" y="280"/>
        <di:waypoint x="872" y="280"/>
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNShape id="Activity_19r34lg_di"
          bpmnElement="Activity_19r34lg">
        <dc:Bounds x="450" y="240" width="100" height="80"/>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="Activity_0js25p0_di"
          bpmnElement="Activity_0js25p0">
        <dc:Bounds x="450" y="410" width="100" height="80"/>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="Activity_0syazcl_di"
          bpmnElement="Activity_0syazcl">
        <dc:Bounds x="660" y="410" width="100" height="80"/>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="Event_149a09z_di" bpmnElement="Event_149a09z">
        <dc:Bounds x="872" y="432" width="36" height="36"/>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="Activity_0xn91r1_di"
          bpmnElement="Activity_0xn91r1">
        <dc:Bounds x="660" y="240" width="100" height="80"/>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="Event_0nfkfxy_di" bpmnElement="Event_0nfkfxy">
        <dc:Bounds x="872" y="262" width="36" height="36"/>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="Event_05f276v_di" bpmnElement="Event_05f276v">
        <dc:Bounds x="153" y="262" width="36" height="36"/>
        <bpmndi:BPMNLabel>
          <dc:Bounds x="160" y="305" width="23" height="14"/>
        </bpmndi:BPMNLabel>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="Gateway_1kobh5h_di"
          bpmnElement="Gateway_10xrnm1">
        <dc:Bounds x="255" y="255" width="50" height="50"/>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="Association_0i85cki_di"
          bpmnElement="Association_0i85cki">
        <di:waypoint x="288" y="263"/>
        <di:waypoint x="323" y="192"/>
      </bpmndi:BPMNEdge>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
</bpmn:definitions>
```