FH JOANNEUM - University of Applied Sciences

Die Komplexität von Geschäftsprozessen

Masterarbeit

zur Erlangung des akademischen Grades eines/einer

„Diplomingenieurs/Diplomingenieurin für technisch-wissenschaftliche Berufe“

eingereicht am Master-Studiengang Informationsmanagement

Verfasser:

Stefan Heider, BSc

Betreuer:

FH-Prof. Mag. Dr. Robert Singer

Graz, 2022

FH JOANNEUM
University of Applied Sciences

Ehrenwörtliche Erklärung:

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

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

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

Inhaltsverzeichnis

Abbildungsverzeichnis iii

Abkürzungsverzeichnis v

1 Einleitung 1

2 Bestimmung der Komplexität von Geschäftsprozessen 2
2.1 Informationsgehalt von Geschäftsprozessen . . . 2
2.2 Berechnung der Unsicherheit von Blöcken . . . 4
2.2.1 Notation . . . 4
2.2.2 Sequenzfluss . . . 4
2.2.3 Paralleles Gateway . . . 5
2.2.4 Exklusives Gateway . . . 6
2.2.5 Inklusives Gateway . . . 11
2.2.6 Schleife . . . 14
2.2.7 Berechnung von gesamten Prozessen . . . 16
2.2.8 Formelübersicht . . . 19
2.3 Verzweigungstypen . . . 19
2.3.1 Ereignisbasierte Gateways . . . 19
2.3.2 Komplexe Gateways . . . 20

3 Automatisierung 22
3.1 Webapplikation . . . 22
3.1.1 Installation . . . 22
3.1.2 Verwendung . . . 23
3.1.3 Funktionsweise . . . 25
3.1.4 Demonstration . . . 37
3.2 Herausforderungen der Automatisierung . . . 42
3.2.1 Besondere Muster . . . 43
3.2.2 Schlussfolgerung . . . 47
3.3 Berechnung der Unsicherheit von einzelnen Elementen . . . 48
3.3.1 Notation . . . 49
3.3.2 Exklusive Verzweigungen . . . 49
3.3.3 Parallele Verzweigungen . . . 54
3.3.4 Inklusive Verzweigungen . . . 56
3.3.5 Schleifen . . . 58
3.3.6 Berechnung von Prozessen . . . 60

4 Fazit 65

5 Anhang 67

Literatur 88

Abbildungsverzeichnis

2.1 Beispielprozess . . . 3
2.2 Sequenzfluss . . . 5
2.3 Paralleles Gateway . . . 6
2.4 Exklusives Gateway . . . 7
2.5 Berechnung eines exklusiven Blocks . . . 8
2.6 Berechnung eines exklusiven Blocks 2 . . . 9
2.7 Berechnung eines exklusiven Blocks 3 . . . 10
2.8 Exklusiver Block Extremfall . . . 11
2.9 Inklusives Gateway . . . 13
2.10 Inklusives Gateway mit Dummy Prozessblöcken . . . 13
2.11 Inklusives Gateway mit Tasks . . . 14
2.12 Schleife . . . 15
2.13 Beispiel Prozess - Erste Iteration . . . 17
2.14 Beispiel Prozess - Zweite Iteration . . . 17
2.15 Beispiel Prozess - Dritte Iteration . . . 18
2.16 Beispiel Prozess - Vierte Iteration . . . 18
2.17 Ereignisbasiertes Gateway . . . 20
2.18 Komplexes Gateway . . . 21

3.1 Anwendung als ZIP Datei . . . 23
3.2 Webapplikation Startseite . . . 24
3.3 ID des Startelements . . . 24
3.4 Ausführungswahrscheinlichkeiten festlegen . . . 25
3.5 Webapplikation Prozess geladen . . . 25
3.6 Beispiel: Instanzen . . . 27
3.7 Beispiel: Gateway-Paare erste Iteration . . . 29
3.8 Beispiel: Gateway-Paare zweite Iteration . . . 30
3.9 Beispiel: Gateway-Paare dritte Iteration . . . 30
3.10 Beispiel: Pfade einer Schleife . . . 31
3.11 Beispiel Prozess 1 . . . 39
3.12 Berechnung mittels Webapplikation . . . 40
3.13 Geschäftsprozess 2 . . . 40
3.14 Geschäftsprozess 3 . . . 41
3.15 Geschäftsprozess 4 . . . 42
3.16 XOR-Block: An einem Punkt zusammengeführt . . . 43
3.17 XOR-Block: Nicht zusammengeführt . . . 43
3.18 XOR-Block: Teilweise zusammengeführt . . . 44
3.19 XOR-2 aufgelöst . . . 44
3.20 XOR-Block: Teilweise zusammengeführt alternativ . . . 45
3.21 XOR-Block: Teilweise zusammengeführt alternativ XOR-2 aufgelöst . . . 45
3.22 XOR-Block: Teilweise zusammengeführt Extremfall . . . 46
3.23 XOR-Block: Verschachtelt . . . 46
3.24 XOR-Block: Verschachtelt und nicht Zusammengeführt . . . 47
3.25 XOR-Block: Überkreuzungen . . . 47

3.26 Beispiel: Berechnung von einzelnen Elementen 50
3.27 Beispiel: erster Schritt 52
3.28 Beispiel: zweiter Schritt 53
3.29 Beispiel: dritter Schritt 54
3.30 Beispiel: Berechnung von einzelnen parallelen Elementen 55
3.31 Beispiel: Berechnung von einzelnen Inklusiven Elementen 56
3.32 Schleife 58
3.33 Schleife 61
3.34 Beispiel: Berechnung von Elementen 64

Abkürzungsverzeichnis

BPMN	Business Process Model and Notation
BPEL	Business Process Execution Language
XML	Extensible Markup Language
npm	node package manager
URL	Uniform Resource Locator

Kurzfassung

Ziel dieser Arbeit ist es zu untersuchen, wie die Komplexität von Geschäftsprozessen berechnet werden kann. Des Weiteren wird untersucht, ob die Berechnung der Komplexität automatisiert werden kann. Eine Methode zu definieren, mit welcher die Komplexität von Geschäftsprozessen berechnet werden kann, ermöglicht die Untersuchung von Zusammenhängen zwischen der Komplexität eines Prozesses und anderen relevanten Kenngrößen. Somit können diese Fragestellungen für Unternehmen von Interesse sein.

Als Maß für die Komplexität wird in dieser Arbeit der Informationsgehalt von Shannon verwendet. Ein erhöhter Informationsgehalt führt zu einer erhöhten Unsicherheit und somit zu einer erschwerten Prognose über das Resultat einer Entscheidung innerhalb eines Geschäftsprozesses. In dieser Arbeit werden Formeln zur Berechnung der Unsicherheit von Geschäftsprozessen untersucht. Außerdem wird mithilfe einer praktischen Umsetzung in Form einer entwickelten Webapplikation untersucht, ob die Berechnung der Unsicherheit von Geschäftsprozessen automatisiert werden kann.

Die Methode zur Berechnung der Unsicherheit von Geschäftsprozessen, welche in der praktischen Umsetzung eingesetzt wurde, sieht die Zerlegung der Prozesse in Blöcke vor. In einem iterativen Prozess wird die Unsicherheit von Blöcken berechnet, bis schlussendlich die Unsicherheit des gesamten Prozesses berechnet werden kann. Die praktische Umsetzung dieser Methode zeigt, dass eine automatisierte Berechnung der Unsicherheit von Blöcken innerhalb von modellierten Geschäftsprozessen in der Praxis nur eingeschränkt möglich ist. Aus diesem Grund wird eine Methode zur Berechnung der Unsicherheit von Geschäftsprozessen vorgestellt, in welcher die Unsicherheit von einzelnen Elementen anstatt der Unsicherheit von Blöcken berechnet wird.

Der Begriff ”Komplexität” wurde noch nicht in ausreichender Form definiert, um darüber zu entscheiden, ob der Informationsgehalt eines Systems ein geeignetes Maß für dessen Komplexität darstellt. Dennoch kann die Reduktion der Unsicherheit eines Prozesses Nutzen für Unternehmen erzeugen. Die Entwicklung einer ausgereiften und für die Praxis geeigneten Anwendung zur Berechnung der Unsicherheit von Prozessen ermöglicht Untersuchungen von Zusammenhängen zwischen der Unsicherheit von Prozessen und anderen relevanten Kenngrößen. Diese Arbeit zeigt, dass die Entwicklung einer solchen Anwendung grundsätzlich möglich ist.

Abstract

The aim of this work is to investigate how the complexity of business processes can be calculated. Furthermore, it is investigated whether the calculation of complexity can be automated. If a method for the calculation of the complexity of a business process is defined, this enables further research about possible correlations between the complexity of a process and other significant parameters. Thus, these questions can be of interest to companies.

Shannon's concept of entropy is used as a measure of complexity in this paper. If the entropy of a process is increased, this leads to an increased uncertainty and to an increased difficulty for the prediction of the outcome of decisions within a business process. In this thesis, formulas for calculating the uncertainty of business processes are examined. In addition, it is investigated whether the calculation of the uncertainty of business processes can be automated with the help of an application.

The method for calculating the uncertainty of business processes, which was used for developing the application, focuses on calculating the uncertainty of blocks within the business processes. The uncertainty of blocks is calculated in an iterative manner until finally the uncertainty of the entire process can be calculated. The practical implementation of this method shows that an automated calculation of the uncertainty of blocks within business processes is only possible to a limited extent. For this reason, a method for calculating the uncertainty of business processes is presented in which the uncertainty of individual elements is calculated.

The term "complexity" has not yet been defined in sufficient form to decide whether the entropy of a system is an appropriate measure of its complexity. Nevertheless, reducing the uncertainty of a process can generate benefits for companies. The development of an application for calculating the uncertainty of processes enables investigations of the relationships between the uncertainty of processes and other relevant parameters. This work shows that the development of such an application is possible.

1 Einleitung

Geschäftsprozessmanagement ist für viele Unternehmen ein wesentlicher Grundstein für eine effiziente Gestaltung der internen Prozesse. Gängige Kenngrößen zur Bestimmung von „guten“ Geschäftsprozessen sind unter anderem die Durchlaufzeit, die Kosten, die Flexibilität sowie die Effizienz eines Geschäftsprozesses. Ein Faktor, welcher einen erheblichen Einfluss auf diese Kenngrößen nehmen kann, ist die Komplexität des Geschäftsprozesses. Auch wenn es noch keine allgemein anerkannte Methode zur Ermittlung der Komplexität von Geschäftsprozessen gibt, so kann angenommen werden, dass eine erhöhte Anzahl an möglichen Instanzen innerhalb des Prozesses dessen Komplexität steigert. Ebenfalls ist es naheliegend, dass eine erhöhte Komplexität von Systemen deren Informationsgehalt steigert. Eine bereits etablierte Methode zur Bestimmung des Informationsgehalts von Systemen ist das Modell der Entropie nach Shannon [1].

Ein Ziel dieser Arbeit ist es zu untersuchen, ob das Modell der Entropie, welches von Shannon im Jahr 1948 veröffentlicht wurde, dafür geeignet ist, die Komplexität eines Geschäftsprozesses als quantitative Größe darzustellen. Ein erster Ansatz dieser Idee findet sich bereits in einer Arbeit aus dem Jahr 2011, in welcher ein Algorithmus zur Berechnung der Komplexität von Geschäftsprozessen vorgeschlagen wurde [2]. Diese Arbeit fokussierte sich dabei auf die Bewertung von einfachen logischen Bausteinen und Konzepten wie etwa UND-Gateway, ODER-Gateway sowie Schleifen innerhalb des Modells.

Ziel dieser Arbeit ist es nun, auf den Erkenntnissen der Arbeit von Jung, Chin und Cardoso aufzubauen und eine Methode zur Bewertung der restlichen Elemente zu entwickeln. Dabei liegt der Fokus auf den beschreibenden Elementen (Descriptive Process Modeling Conformance Subclass) des Business Process Model and Notation (BPMN) 2.0 Standards. Der entstehende Algorithmus zur Berechnung der Komplexität von Geschäftsprozessen soll anschließend in Form eines ausführbaren Codes festgehalten werden, sodass die Berechnung automatisiert stattfinden kann. Hierfür sollen die XML Dateien von modellierten Geschäftsprozessen herangezogen werden, um den Zugriff auf die benötigten Informationen über den zu betrachtenden Geschäftsprozess im Code zu ermöglichen. Die XML Dateien von modellierten Geschäftsprozessen entsprechen dabei dem BPMN 2.0 Standard und sind somit vereinheitlicht und können problemlos in diversen Anwendungen wie etwa Camunda aus modellierten Geschäftsprozessen generiert werden.

Der Output dieser Arbeit ist eine Software, welche die Komplexität von Geschäftsprozessen berechnen soll. Dies bietet die Möglichkeit, unterschiedliche Geschäftsprozesse zu vergleichen und die Komplexität von Geschäftsprozessen gezielt anzupassen. Dabei soll jedoch gesagt sein, dass eine hohe Komplexität an sich keine negative Eigenschaft sein muss. Eine hohe Komplexität kann durchaus durch einen wohlüberlegten Aufbau des Geschäftsprozesses sowie durch die benötigte Tiefe verursacht werden. Eine zu geringe Komplexität kann hingegen zu einem Mangel an notwendigen Informationen und damit zu einer erschwerten Durchführung des Geschäftsprozesses führen. Somit ist die Komplexität von Geschäftsprozessen lediglich zum Vergleich mit anderen Geschäftsprozessen geeignet und muss stets unter Berücksichtigung diverser Faktoren betrachtet werden.

2 Bestimmung der Komplexität von Geschäftsprozessen

Um eine eindeutige Aussage zur Komplexität von Geschäftsprozessen treffen zu können, muss zunächst eine klare Definition des Begriffs ”Komplexität” sowie eine allgemein anerkannte Methode zur Ermittlung der Komplexität vorliegen. Bislang wurde jedoch noch keine eindeutige und allgemein akzeptierte Definition des Begriffs ”Komplexität” vorgewiesen. Es gibt jedoch bereits unzählige Versuche, den Begriff ”Komplexität” zu definieren. Melanie Mitchell definiert ein komplexes System wie folgt [3, p. 13]:

> ”A system in which large networks of components with no central control and simple rules of operation give rise to complex collective behaviour, sophisticated information processing, and adaption via learning or evolution.”

Des Weiteren ergänzt Mitchell, dass man zwischen adaptiven komplexen Systemen und nicht-adaptiven komplexen Systemen unterscheiden kann [3, p. 13]. Wenn man diese Definition mit dem Aufbau von Geschäftsprozessmodellen vergleicht, so kann man feststellen, dass mehrere Parallelen sowie auch einzelne Widersprüche aufzufinden sind. Geschäftsprozesse können als Netzwerk von Aktivitäten, Gateways sowie diversen weiteren Kategorien von Elementen verstanden werden. Jedes Element in einem Geschäftsprozessmodell hat klar definierte Eigenschaften und führt eine bestimmte Operation aus. Anders als in Mitchells Definition von komplexen Systemen gibt es jedoch in Geschäftsprozessen zumindest eine implizite zentrale Kontrolle.

Auch wenn bereits Ansätze für die Definition von Komplexität sowie von komplexen Systemen vorhanden sind, so stellt sich dennoch weiterhin die Frage, wie man das Ausmaß an Komplexität eines Systems ermitteln kann. Diese Arbeit setzt sich damit auseinander, inwiefern sich die Methode der Berechnung des Informationsgehaltes von Systemen, welche von Claude Elwood Shannon im Jahr 1948 veröffentlicht wurde, dafür eignet, die Komplexität eines Geschäftsprozessmodells zu ermitteln [1].

Dabei wird angenommen, dass der Informationsgehalt eines Geschäftsprozessmodells in direktem Zusammenhang mit dessen Komplexität steht. Der Informationsgehalt des Geschäftsprozesses sagt dabei aus, inwiefern sich Vorhersagen lässt, wie der Geschäftsprozess abgewickelt wird. Ein einfacher Geschäftsprozess, in welchem lediglich eine Reihe von Tasks ohne jegliche Verzweigungen oder Bedingungen abgearbeitet werden, entspricht dem Informationsgehalt null und hat somit eine niedrige Komplexität. Umso mehr Verzweigungen und Bedingungen in einem Geschäftsprozessmodell zu finden sind, desto schwieriger ist es vorherzusagen, wie der Geschäftsprozess abgearbeitet wird. Somit erhöhen Verzweigungen und Bedingungen den Informationsgehalt eines Geschäftsprozessmodells. Diese Idee wurde bereits in einer Arbeit auf Basis von einigen grundlegenden Elementen, welche in Geschäftsprozessmodellen zu finden sind, veröffentlicht [2].

2.1 Informationsgehalt von Geschäftsprozessen

Die Unsicherheit von Information bzw. der Informationsgehalt einer bestimmten Information berechnet sich nach Shannons Entropie wie folgt [1]:

$$H(X) = -K \sum_{i=1}^{n} P(x_i) \log_2 P(x_i) \tag{2.1}$$

Dabei stellt X eine diskrete Variable dar, welche die Werte x_1, x_2, ..., x_n mit den Wahrscheinlichkeiten $P(x_1)$, $P(x_2)$, ..., $P(x_n)$ annehmen kann. Dabei gilt: $1 \leq i \leq n$, $P(x_i) \geq 0$, $\sum P(x_i) = 1$, $K > 0$. Die Konstante K dient lediglich als Maßstab und kann angepasst werden. Soll der Informationsgehalt in Bit dargestellt werden, so gilt: $K = 1$.

Ein Geschäftsprozess wurde dann erfolgreich abgewickelt, wenn er in einem der im Modell definierten Szenarien durchlaufen und abschließend terminiert wurde. Dabei kann derselbe Geschäftsprozess, sofern das Design des Geschäftsprozesses dies ermöglicht, auf unterschiedliche Weise durchlaufen werden. Die Methode zur Berechnung des Informationsgehaltes von Geschäftsprozessen, welche von Jung et al. [2] vorgestellt wurde, fokussiert sich auf die Unsicherheit darüber, welche der möglichen Instanzen des Geschäftsprozesses durchlaufen wird.

Die Methode kann anhand eines simplen Geschäftsprozesses demonstriert werden. Angenommen, ein Geschäftsprozess besteht aus einem Task t_0, welcher über ein exklusives Gateway mit den Tasks t_1, t_2 und t_3 verbunden ist (siehe Abbildung 2.1). Nachdem Task t_0 abgeschlossen wurde, wird anschließend einer der Tasks t_1, t_2 oder t_3 ausgeführt. Somit ergeben sich drei Szenarien, wie dieser Geschäftsprozess durchlaufen werden kann. Die Wahrscheinlichkeiten für jedes der drei Szenarien werden in diesem Beispiel wie folgt angenommen: $P(t_0 \rightarrow t_1) = 1/5$, $P(t_0 \rightarrow t_2) = 3/10$ und $P(t_0 \rightarrow t_3) = 1/2$. In diesem Geschäftsprozess B, berechnet sich die Unsicherheit $U(B)$ nach Shannons Entropie wie folgt [2]:

$$\begin{aligned} U(B) &= -\sum_{i=1}^{3} P(t_0 \rightarrow t_i) \log_2 P(t_0 \rightarrow t_i) \\ &= -\left(\frac{1}{5} \log_2 \frac{1}{5} + \frac{3}{10} \log_2 \frac{3}{10} + \frac{1}{2} \log_2 \frac{1}{2}\right) = 1.48 \end{aligned} \tag{2.2}$$

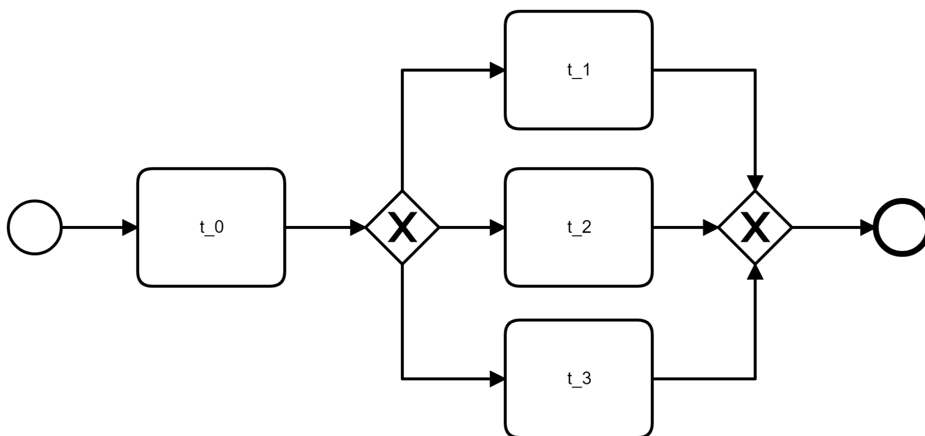

Abbildung 2.1: Beispielprozess

Die geringstmögliche Unsicherheit ($U(B) = 0$) ergibt sich dann, wenn die Eintrittswahrscheinlichkeit P eines Szenarios dem Wert 1 entspricht, während die Eintrittswahrscheinlichkeit der anderen beiden Szenarien 0 entspricht. In besagtem Fall kann das ausgeführte Szenario ohne

Fehlerquote vorhergesagt werden, wodurch die Unsicherheit im Geschäftsprozess entfällt. Die Unsicherheit erreicht dann ihren Maximalwert, wenn jedes der möglichen Szenarien über dieselbe Eintrittswahrscheinlichkeit verfügt [2].

2.2 Berechnung der Unsicherheit von Blöcken

Nun gilt es Formeln für die Berechnung der Unsicherheit von den einzelnen beschreibenden Elementen des BPMN 2.0 Standards zu definieren. Jung et al. [2] haben in ihrer Arbeit bereits Formeln für Sequenzflüsse, parallele Gateways, exklusive Gateways, inklusive Gateways sowie für Schleifen innerhalb von Prozessmodellen definiert. Das Ziel ist es, durch die Definition von Formeln zur Berechnung der Unsicherheit von Blöcken, die Entwicklung eines Algorithmus zu ermöglichen, welcher es vermag die Unsicherheit von vollständigen Geschäftsprozessen zu ermitteln.

2.2.1 Notation

Um Formeln zur Berechnung der Unsicherheit von Geschäftsprozessmodellen definieren zu können, ist es notwendig, eine Notation für bestimmte Ausdrücke zu definieren. Jung et al. [2] definieren in ihrer Arbeit folgende Ausdrücke:

- B: Das zu betrachtende Prozessmodell
- N: Die Anzahl an Prozessblöcken, welche Teil des Prozessmodells B sind
- M: Die Anzahl an möglichen Ausführungsszenarien für die N Prozessblöcke in Prozessmodell B
- BS_k und $P(BS_k)$: Das k-te Ausführungsszenario für die N Prozessblöcke im Prozessmodell B sowie die zugehörige Wahrscheinlichkeit der Ausführung
- B_g: Der g-te Prozessblock in Prozessmodell B; $1 \leq g \leq N$
- $P(B_g)$: Die Wahrscheinlichkeit, dass der g-te Prozessblock B_g von allen möglichen Ausführungsszenarien M ausgeführt wird, um das Ziel des Prozessmodells B zu erreichen; $1 \leq q \leq N$
- $S_{g,i}$ und $s_{g,i}$: Das i-te Ausführungsszenario von Prozessblock B_g sowie die zugehörige Wahrscheinlichkeit der Ausführung; $1 \leq i \leq V_g$
- $R_{g,h}$ und $r_{g,h}$: Der Übergang zwischen zwei sequenziellen Prozessblöcken B_g und B_h sowie die zugehörige Wahrscheinlichkeit des Übergangs
- N_g: Die Anzahl an Sub-Prozessblöcken, welche Teil des Prozessblocks B_g sind
- $B_{g,j}$: Der j-te Sub-Prozessblock in Prozessblock B_g; $1 \leq j \leq N_g$

2.2.2 Sequenzfluss

Ein Sequenzblock besteht aus einer Reihe von Prozessblöcken, welche in Form einer Sequenz ausgeführt werden. In Abbildung 2.2 wird ein solcher Sequenzfluss dargestellt. Der Sequenzblock B_{SEQ} beinhaltet die Prozessblöcke B_g ($1 \leq g \leq N$). Die Formel zur Berechnung der Unsicherheit von Sequenzflüssen lautet [2]:

$$U(B_{SEQ}) = \sum_{g=1}^{N} U(B_g) \tag{2.3}$$

Ihre Formel beweisen Jung et al. [2] mithilfe eines Beispielprozesses, wie er in Abbildung 2.2 dargestellt wird. Der Beispielprozess B_{SEQ} beinhaltet zwei Prozessblöcke B_1 und B_2 ($N = 2$). Wenn die Prozessblöcke B_1 und B_2 sequenziell und selbstständig in den jeweiligen Ausführungsszenarien $S_{1,i}$ und $S_{2,j}$ ($1 \leq i \leq V_1$ und $1 \leq j \leq V_2$) mit den entsprechenden Übergängen $R_{0,1}$ und $R_{1,2}$ ausgeführt werden, so ergibt sich nach Shannons Entropie die Unsicherheit $\log_2(r_{0,1}s_{1,i}r_{1,2}s_{2,j})^{-1}$. Dabei gilt: $r_{0,1} = r_{1,2} = 1$ sowie $\sum_{i=1}^{V_1} s_{1,i} = \sum_{j=1}^{V_2} s_{2,j} = 1$. Somit wird die Unsicherheit des Prozessblocks B_{SEQ} nach Jung et al. [2] wie folgt berechnet:

$$
\begin{aligned}
U(B_{SEQ}) &= \sum_{i=1}^{V_1} \sum_{j=1}^{V_2} P(R_{0,1})P(S_{1,j})P(R_{1,2})P(S_{2,j}) \\
&\quad \times \log_2 \left[P(R_{0,1})P(S_{1,j})P(R_{1,2})P(S_{2,j})\right]^{-1} \\
&= -\left[r_{0,1}r_{1,2} \log_2(r_{0,1}r_{1,2})\right] \\
&\quad + \left[r_{0,1}r_{1,2} \sum_{i=1}^{V_1} s_{1,i} \log_2(s_{1,i})^{-1} + r_{0,1}r_{1,2} \sum_{j=1}^{V_2} s_{2,j} \log_2(s_{2,j})^{-1}\right] \\
&= -\sum_{k=1}^{1} P(BS_k) \log_2 P(BS_k) + \sum_{g=1}^{2} P(B_g)U(B_g) \\
&= \sum_{g=1}^{2} U(B_g)
\end{aligned}
\tag{2.4}
$$

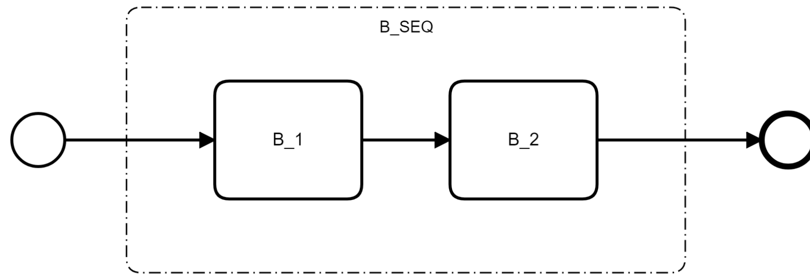

Abbildung 2.2: Sequenzfluss

2.2.3 Paralleles Gateway

Wird ein paralleles Gateway verwendet, um den Prozessfluss zu teilen, so werden alle Ausgangszweige des Gateways gleichzeitig aktiviert. Bei der Zusammenführung durch ein paralleles Gateway wird gewartet, bis alle eingehenden Zweige bereit sind, bevor der ausgehende Zweig aktiviert wird. Da in einem parallelen Block (siehe Abbildung 2.3) stets alle Prozessblöcke ausgeführt werden, erhöht die Integration von parallelen Gateways nicht die Unsicherheit des Prozessmodells. Der parallele Block B_{AND} beinhaltet dabei die Prozessblöcke B_g ($1 \leq g \leq N$). Die Formel zur Berechnung der Unsicherheit von parallelen Blöcken gleicht somit der des Sequenzflusses und lautet [2]:

$$
U(B_{AND}) = \sum_{g=1}^{N} U(B_g) \tag{2.5}
$$

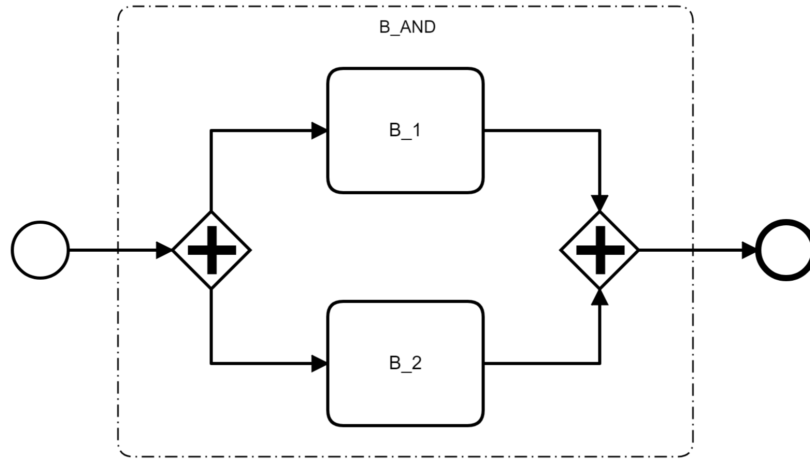

Abbildung 2.3: Paralleles Gateway

2.2.4 Exklusives Gateway

Bei der Verwendung eines exklusiven Gateways werden zwei oder mehrere ausgehende Zweige mit dem Gateway verbunden, wovon jedoch lediglich ein Zweig aktiviert wird. Der XOR-Block (B_{XOR}), welcher durch die Einbindung eines exklusiven Gateways entsteht, wird in Abbildung 2.4 dargestellt. Der XOR-Block beinhaltet die Prozessblöcke B_g ($1 \leq g \leq N$). Die durch XOR-Blöcke entstehende Unsicherheit wird nach Jung et al. [2] wie folgt berechnet:

$$U(B_{XOR}) = -\sum_{g=1}^{N} r_{0,g} \log_2(r_{0,g}) - \sum_{g=1}^{N} r_{0,g} \times \sum_{i=1}^{V_g} s_{g,i} \log_2(s_{g,i}) \tag{2.6}$$

Der Beweis der Formel erfolgte nach Jung et al. [2] anhand des in Abbildung 2.4 dargestellten Prozesses. Wenn Prozessblock B_1 des Beispielprozesses B_{XOR} im Ausführungsszenario $S_{1,i}$ ($1 \leq i \leq V_1$) mit dem Übergang $R_{0,1}$ ausgeführt wird, so ergibt sich nach Shannons Entropie die Unsicherheit $\log_2(r_{0,1}s_1)^{-1}$. Dabei gilt: $\sum_{g=1}^{N} r_{0,g} = 1$ sowie $\sum_{i=1}^{V_1} s_{1,i} = 1$. Somit wird die Unsicherheit des Prozessblocks B_{XOR} nach Jung et al. [2] wie folgt berechnet:

$$\begin{aligned} U(B_{XOR}) &= \sum_{g=1}^{N} \sum_{i=1}^{V_1} P(R_{0,g}) P(S_{g,i}) \times \log_2 \left[P(R_{0,g}) P(S_{g,i})\right]^{-1} \\ &= -\sum_{g=1}^{N} r_{0,g} \log_2(r_{0,g}) + \sum_{g=1}^{N} r_{0,g} r_{0,g} \sum_{i=1}^{V_g} s_{g,i} \log_2(s_{g,i})^{-1} \\ &= -\sum_{k=1}^{N} P(BS_k) \log_2 P(BS_k) + \sum_{g=1}^{N} P(B_G) U(B_G) \end{aligned} \tag{2.7}$$

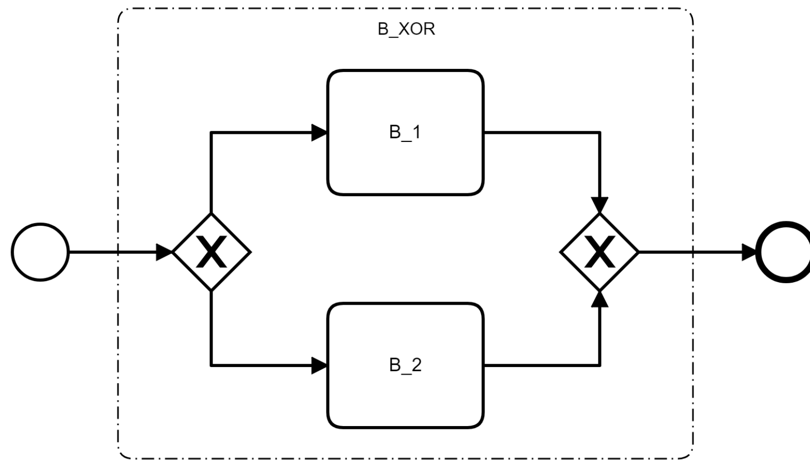

Abbildung 2.4: Exklusives Gateway

Die Formel für die Berechnung der Unsicherheit von exklusiven Blöcken besteht aus zwei Teilen. Um dies zu verdeutlichen, wird die Formel hier nochmals mit einer zusätzlichen Klammer angeschrieben:

$$U(B_{XOR}) = -\sum_{g=1}^{N} r_{0,g} \log_2(r_{0,g}) - \sum_{g=1}^{N} \left[r_{0,g} \times \sum_{i=1}^{V_g} s_{g,i} \log_2(s_{g,i}) \right] \tag{2.8}$$

Der erste Teil der Formel ($-\sum_{g=1}^{N} r_{0,g} \log_2(r_{0,g})$) dient zur Berechnung der Unsicherheit, welche vom exklusiven Gateway ausgeht. Der zweite Teil der Formel ($-\sum_{g=1}^{N} \left[r_{0,g} \times \sum_{i=1}^{V_g} s_{g,i} \log_2(s_{g,i}) \right]$) dient dazu, die Unsicherheit der Blöcke B_g (in Abbildung 2.4 sind dies die Blöcke B_1 und B_2) zu berücksichtigen. Die Blöcke B_g können dabei ebenfalls logische Blöcke sein. Die Anwendung der Formel wird im Folgenden anhand eines Beispiels (siehe Abbildung 2.5) demonstriert. Für die Berechnung dieses Beispiels werden folgende Werte für die jeweiligen Übergangswahrscheinlichkeiten angenommen: $r_{0,1} = 0.7$, $r_{0,2} = 0.3$, $s_{1,1} = 0.6$, $s_{1,2} = 0.4$, $s_{2,1} = 0.8$, $s_{2,2} = 0.2$

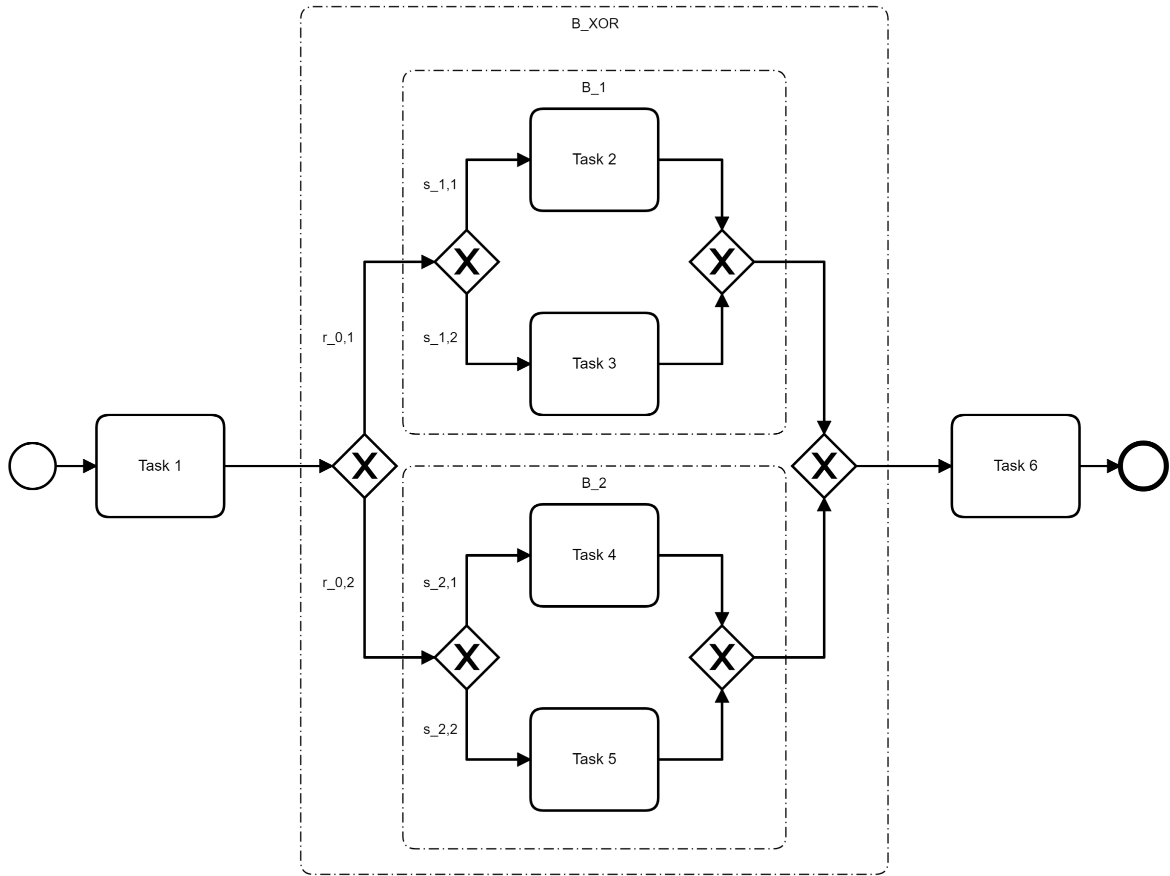

Abbildung 2.5: Berechnung eines exklusiven Blocks

Durch Anwendung der Formel berechnet sich die Unsicherheit des Blocks B_{XOR} aus Abbildung 2.5 wie folgt:

$$\begin{aligned} U(B_{XOR}) &= -\sum_{g=1}^{N} r_{0,g} \log_2(r_{0,g}) - \sum_{g=1}^{N} \left[r_{0,g} \times \sum_{i=1}^{V_g} s_{g,i} \log_2(s_{g,i}) \right] \\ &= -\left[0.7 \log_2(0.7) + 0.3 \log_2(0.3)\right] \\ &\quad - \{0.7 \times [0.6 \log_2(0.6) + 0.4 \log_2(0.4)] + 0.3 \times [0.8 \log_2(0.8) + 0.2 \log_2(0.2)]\} \\ &= -[-0.36 - 0.52] - \{0.7\,[-0.44 - 0.53] + 0.3\,[-0.26 - 0.46]\} \\ &= 0.88 - \{-0.68 - 0.22\} \\ &= 1.78 \end{aligned} \tag{2.9}$$

Befinden sich einfache Tasks innerhalb des exklusiven Blocks, so kann die Formel vereinfacht werden. Dies kann anhand eines kurzen Beispiels, wie in Abbildung 2.6 dargestellt, demonstriert werden. Um den Unterschied zwischen den Übergangswahrscheinlichkeiten $r_{0,1}$ und $s_{1,1}$ beziehungsweise $r_{0,2}$ und $s_{2,1}$ auch visuell zu verdeutlichen, wurden im modellierten Geschäftsprozess Dummy-Gateways vor den Tasks platziert. Da diese Dummy-Gateways jeweils nur über einen ausgehenden Sequenzfluss verfügen, haben sie keinerlei Einfluss auf die Unsicherheit des Prozesses. Für die Berechnung gilt: $s_{1,1} = s_{2,1} = 1$. Somit wird die Unsicherheit des Blocks B_{XOR} wie folgt berechnet:

$$\begin{aligned} U(B_{XOR}) &= -\sum_{g=1}^{N} r_{0,g} \log_2(r_{0,g}) - \sum_{g=1}^{N} \left[r_{0,g} \times \sum_{i=1}^{V_g} s_{g,i} \log_2(s_{g,i}) \right] \\ &= -\sum_{g=1}^{N} r_{0,g} \log_2(r_{0,g}) - \{ r_{0,1} \times [1 \log_2(1)] + r_{0,2} \times [1 \log_2(1)] \} \\ &= -\sum_{g=1}^{N} r_{0,g} \log_2(r_{0,g}) - \{ r_{0,1} \times 0 + r_{0,2} \times 0 \} \\ &= -\sum_{g=1}^{N} r_{0,g} \log_2(r_{0,g}) \end{aligned} \tag{2.10}$$

Somit entfällt der zweite Term der ursprünglichen Formel, sollten sich lediglich Tasks innerhalb des exklusiven Blocks befinden. Die vereinfachte Formel für diesen Fall lautet daher:

$$U(B_{XOR}) = -\sum_{g=1}^{N} r_{0,g} \log_2(r_{0,g}) \tag{2.11}$$

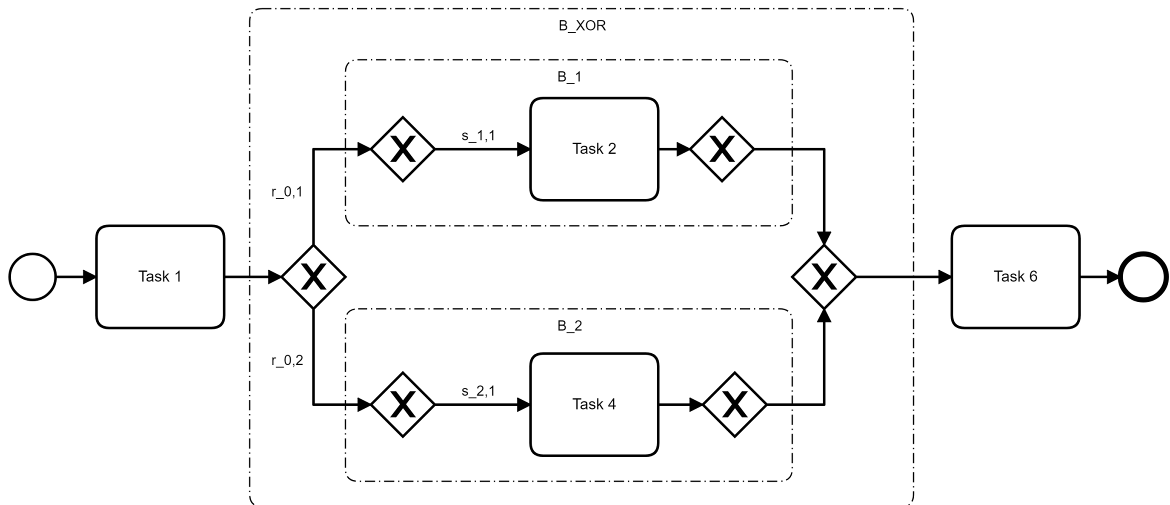

Abbildung 2.6: Berechnung eines exklusiven Blocks 2

Der Term $-\sum_{g=1}^{N} \left[r_{0,g} \times \sum_{i=1}^{V_g} s_{g,i} \log_2(s_{g,i}) \right]$ aus der Formel für die Berechnung der Unsicherheit von exklusiven Blöcken (siehe Formel 2.8) bildet die Summe der Unsicherheiten der Blöcke B_g, wobei die Werte für die Unsicherheit der jeweiligen Blöcke mit der Wahrscheinlichkeit, mit welcher die jeweiligen Blöcke ausgeführt werden, multipliziert werden. Der Term $\sum_{i=1}^{V_g} s_{g,i} \log_2(s_{g,i})$ gleicht der allgemeinen Formel zur Berechnung der Unsicherheit, welche von Shannon veröffentlicht wurde ($H(X) = -K \sum_{i=1}^{n} P(x_i) \log_2 P(x_i)$). Da sich innerhalb von exklusiven Blöcken jedoch auch andere logische Blöcke befinden können, muss der Term $\sum_{i=1}^{V_g} s_{g,i} \log_2(s_{g,i})$ dementsprechend erweitert werden. Angenommen, es sollten sich inklusive Blöcke innerhalb des exklusiven Blocks befinden, wie in Abbildung 2.7 dargestellt, so lautet die konkrete Formel für die Berechnung der Unsicherheit wie folgt:

$$U(B_{XOR}) = -\sum_{g=1}^{N} r_{0,g} \log_2(r_{0,g}) - \sum_{g=1}^{N} \left\{ r_{0,g} \times \sum_{i=1}^{V_g} [s_{g,i} \log_2(s_{g,i}) + (1 - s_{g,i}) \log_2(1 - s_{g,i})] \right\} \tag{2.12}$$

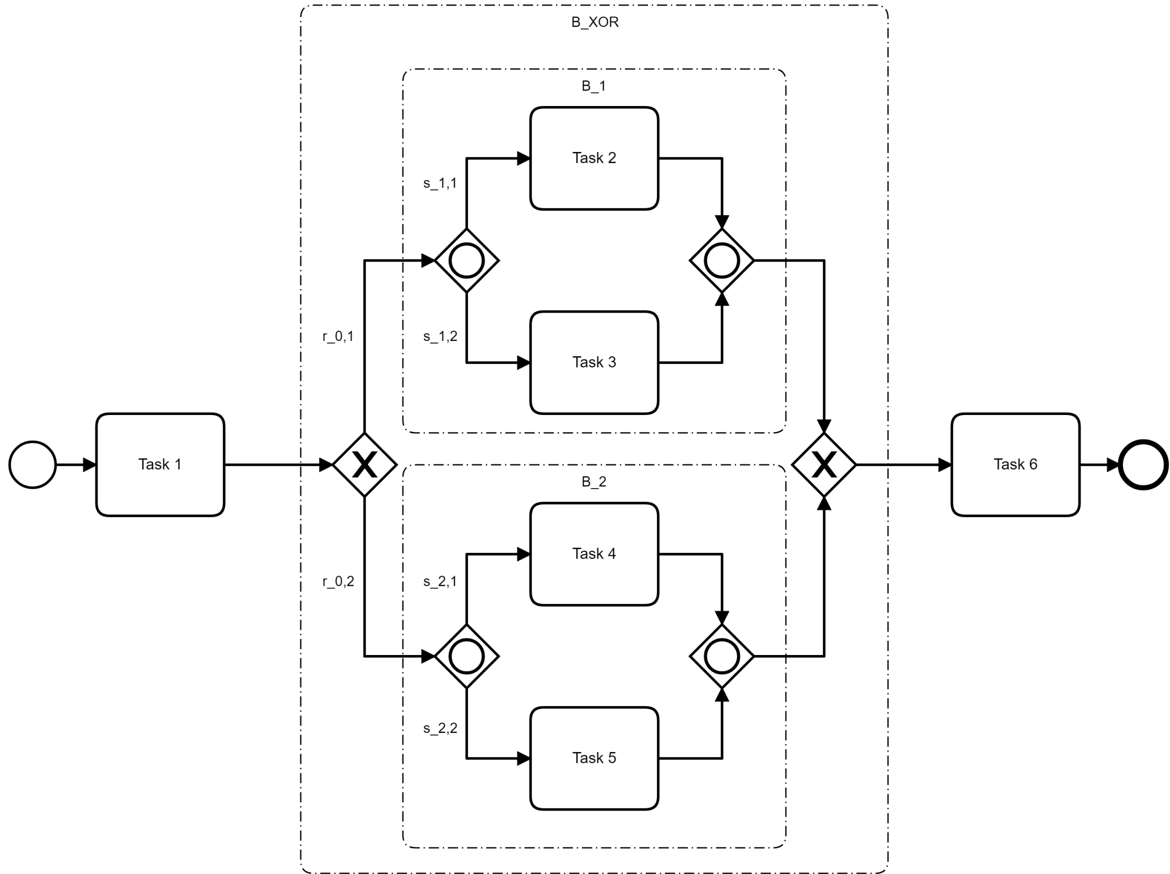

Abbildung 2.7: Berechnung eines exklusiven Blocks 3

Um die Formel zur Berechnung von exklusiven Blöcken einfacher darzustellen, kann sie alternativ auch wie folgt definiert werden:

$$U(B_{XOR}) = -\sum_{g=1}^{N} r_{0,g} \log_2(r_{0,g}) + \sum_{g=1}^{N} r_{0,g} \times U(B_g) \tag{2.13}$$

Den Wert für die Unsicherheit von B_g ($U(B_g)$) mit der zugehörigen Ausführungswahrscheinlichkeit $r_{r,g}$ zu multiplizieren ist dabei ein essenzieller Teil der Formel, welcher auch in der alternativen Schreibweise beibehalten werden muss. Die Variable $r_{0,g}$ aus dem Term $-\sum_{g=1}^{N} \left[r_{0,g} \times \sum_{i=1}^{V_g} s_{g,i} \log_2(s_{g,i}) \right]$ (siehe Formel 2.8) ist nicht das Äquivalent zur Variablen K aus $H(X) = -K \sum_{i=1}^{n} P(x_i) \log_2 P(x_i)$ (siehe Formel 2.1). K aus Formel 2.1 dient als Maßstab, welcher für die Berechnung der Unsicherheit verwendet werden soll. Dabei kann für K ein beliebiger Wert gewählt werden. $r_{0,g}$ aus $-\sum_{g=1}^{N} \left[r_{0,g} \times \sum_{i=1}^{V_g} s_{g,i} \log_2(s_{g,i}) \right]$ dient dazu, die Unsicherheit des zugehörigen Blocks B_g zu gewichten, bevor sie zur bisherigen Unsicherheit des Blocks B_{XOR} addiert wird. Die Variable $r_{0,g}$ kann nur Werte zwischen 0 und

1 annehmen und kann auch nicht beliebig gewählt werden, sondern ist vom zu berechnenden Geschäftsprozess abhängig. Die Relevanz der Gewichtung der Unsicherheit von B_g wird deutlich, wenn ein Extremfall betrachtet wird. Als Grundlage für diesen Extremfall dient der Prozess aus Abbildung 2.8. Für die Übergangswahrscheinlichkeiten werden folgende Werte angenommen: $r_{0,1} = 1$, $r_{0,2} = 0$. Es kann bereits vor der Berechnung erkannt werden, dass die Unsicherheit des Blocks B_{XOR} der Unsicherheit des Blocks B_1 entsprechen muss. Wird ein ausgehender Sequenzfluss mit hundertprozentiger Wahrscheinlichkeit ausgeführt, so kann die Teilung der Sequenzflüsse am exklusiven Gateway keine Unsicherheit erzeugen [1]. Des Weiteren kann der Block B_2 keine Unsicherheit erzeugen, da er mit einer Wahrscheinlichkeit von null Prozent ausgeführt wird. Die Berechnung der Unsicherheit von B_{XOR} sieht demnach wie folgt aus:

$$\begin{aligned} U(B_{XOR}) &= -\sum_{g=1}^{N} r_{0,g} \log_2(r_{0,g}) + \sum_{g=1}^{N} r_{0,g} \times U(B_g) \\ &= (1 \log_2 1 + 0 \log_2 0) + (1 \times U(B_1) + 0 \times U(B_2)) \\ &= U(B_1) \end{aligned} \tag{2.14}$$

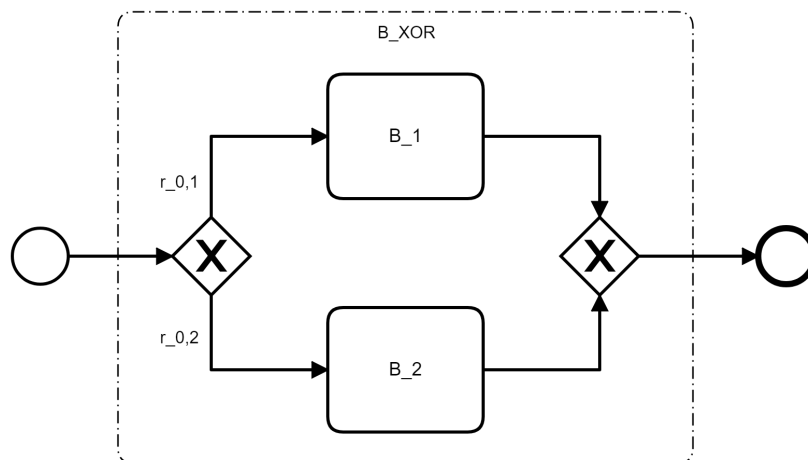

Abbildung 2.8: Exklusiver Block Extremfall

2.2.5 Inklusives Gateway

Ein inklusives Gateway stellt eine Verzweigung dar, in welcher der Sequenzfluss mehreren ausgehenden Zweigen (zwei oder mehr) auf Basis von unabhängigen Bedingungen folgen kann. Somit kann ein inklusives Gateway als eine Gruppierung von unabhängigen binären Bedingungen (Ja/Nein) betrachtet werden. Dabei ist es möglich, dass kein Ausgangszweig, alle Ausgangszweige sowie jede weitere mögliche Kombination von Ausgangszweigen abhängig von den jeweiligen Bedingungen ausgeführt werden [4, p. 38]. Ein beispielhafter OR-Block (B_{OR}), welcher durch die Verwendung eines inklusiven Gateways entsteht, kann in Abbildung 2.9 betrachtet werden. Für die Ableitung der Formel zur Berechnung der Unsicherheit, welche durch die Einbindung von inklusiven Gateways entsteht, sehen Jung et al. [2] vor, Dummy-Prozessblöcke in die Darstellung des OR-Blocks einzubinden (siehe Abbildung 2.10). In dieser Abbildung wird der OR-Block (B_{OR}) mithilfe von Prozessblöcken B_g ($1 \leq g \leq N$) sowie den jeweiligen Dummy-Prozessblöcken B_h ($N + 1 \leq h \leq 2N$) dargestellt. Die Dummy-Prozessblöcke werden verwendet um anzuzeigen, wenn die jeweilig zugehörigen

Prozessblöcke nicht ausgeführt werden. Dabei gilt: $U(B_h) = 0$ für $N + 1 \leq h \leq 2N$. Die durch OR-Blöcke entstehende Unsicherheit wird nach Jung et al. [2] wie folgt berechnet:

$$U(B_{OR}) = -\sum_{g=1}^{N} \left[r_{0,g} \log_2(r_{0,g}) + (1 - r_{0,g}) \log_2(1 - r_{0,g}) + r_{0,g} \sum_{i=1}^{V_g} s_{g,i} \log_2(s_{g,i}) \right] \tag{2.15}$$

Der Beweis der Formel erfolgte nach Jung et al. [2] anhand des in Abbildung 2.10 dargestellten Prozesses. Der g-te Block des zu betrachtenden Modells wird durch B_g^* dargestellt, welcher wiederum aus dem Prozessblock B_g, dem Dummy-Prozessblock B_{N+g} sowie den Übergängen $R_{0,g}$ und $R_{0,N+g}$ besteht. Dabei gilt: $s_{N+g,1} = 1$ sowie $\sum_{i=1}^{V_g} s_{g,i} = 1$. Die jeweiligen Prozessblöcke B_g^* können als XOR-Blöcke betrachtet werden, in welchen entweder der Prozessblock B_g oder der Dummy-Prozessblock B_{N+g} in den jeweiligen Ausführungsszenarien $S_{g,i}$ und $S_{N+g,j}$ ($1 \leq i \leq V_1$ und $j = 1$) mit den entsprechenden Übergängen $R_{0,g}$ und $R_{0,N+g}$ ausgeführt werden. Somit erfolgt die Berechnung der Unsicherheit von B_g^* wie bei XOR-Blöcken [2]:

$$\begin{aligned} U(B_g^*) &= -r_{0,g} \log_2(r_{0,g}) - (1 - r_{0,g}) \log_2(1 - r_{0,g}) - r_{0,g} \sum_{i=1}^{V_g} s_{g,i} \log_2(s_{g,i}) \\ &= -P(BS_g) \log_2 P(BS_g) - P(BS_{N+g}) \log_2 P(BS_{N+g}) - P(B_g)U(B_g) \end{aligned} \tag{2.16}$$

Hiermit kann die Unsicherheit des OR-Blocks B_{OR} wie bei einem parallelen Block, bestehend aus den einzelnen Blöcken B_g^*, berechnet werden [2]:

$$\begin{aligned} U(B_{OR}) &= \sum_{g=1}^{N} U(B_g^*) \\ &= -\sum_{g=1}^{N} \left[r_{0,g} \log_2(r_{0,g}) + (1 - r_{0,g}) \log_2(1 - r_{0,g}) + r_{0,g} \sum_{i=1}^{V_g} s_{g,i} \log_2(s_{g,i}) \right] \\ &= \sum_{k=1}^{2N} P(BS_k) \log_2 P(BS_k) + \sum_{g=1}^{2N} P(B_g)U(B_g) \end{aligned} \tag{2.17}$$

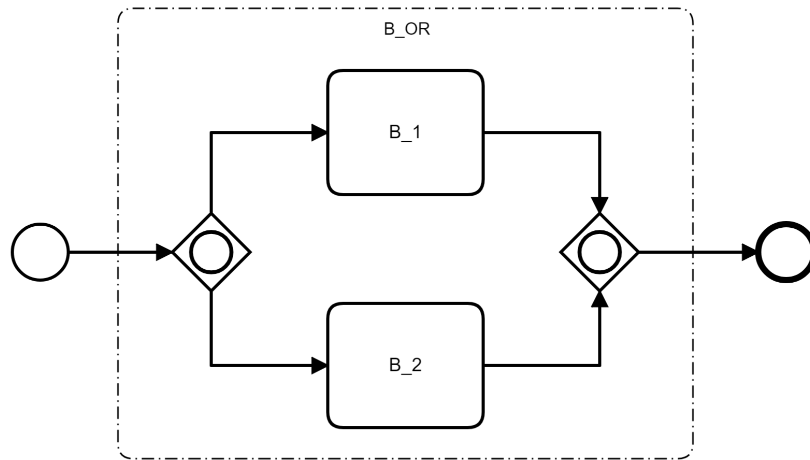

Abbildung 2.9: Inklusives Gateway

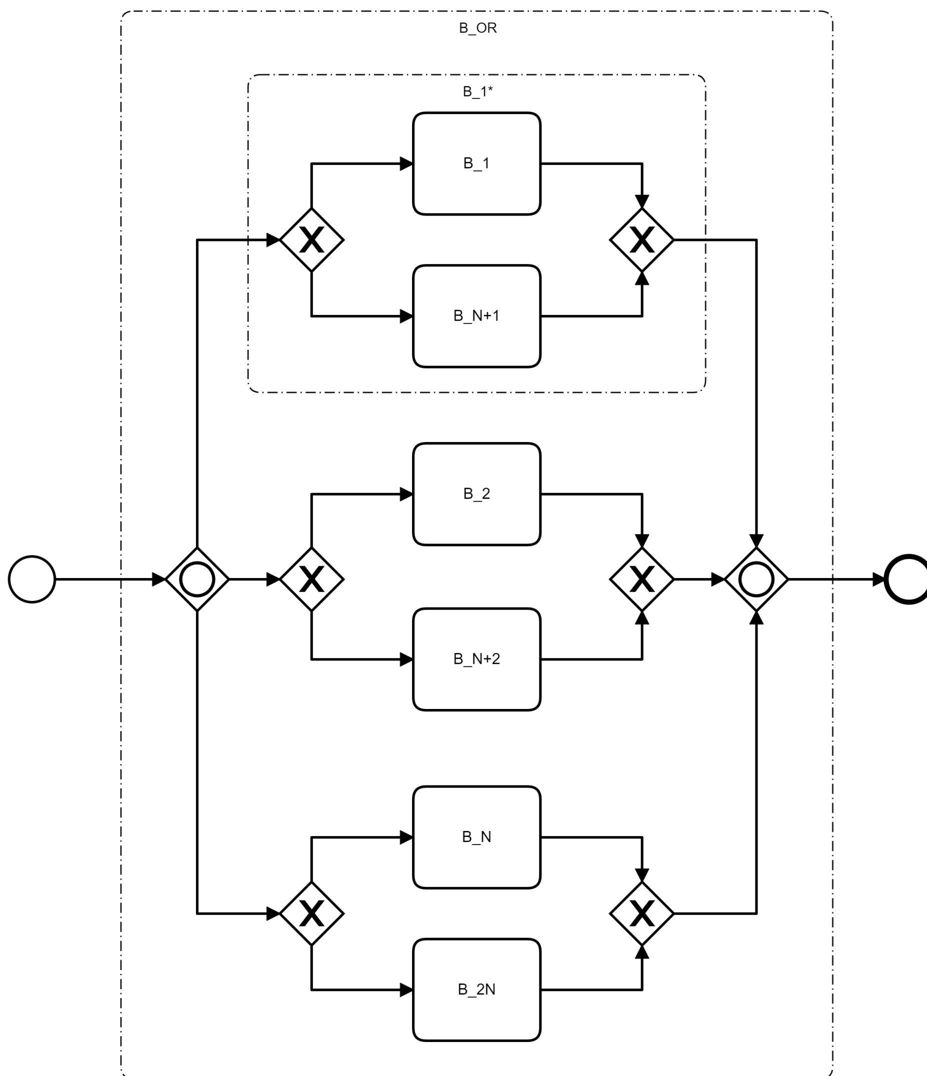

Abbildung 2.10: Inklusives Gateway mit Dummy Prozessblöcken

Für das inklusive Gateway gelten dieselben Überlegungen wie bei dem exklusiven Gateway (siehe Kapitel 2.2.4). Aus diesem Grund können auch dieselben Vereinfachungen für die Formel zur Berechnung der Unsicherheit von inklusiven Blöcken angewandt werden. Befinden

sich lediglich Tasks innerhalb des inklusiven Blocks, wie in Abbildung 2.11 dargestellt, so kann die ursprüngliche Formel zur Berechnung der Unsicherheit von inklusiven Blöcken (siehe Formel 2.15) wie folgt vereinfacht werden:

$$U(B_{OR}) = -\sum_{g=1}^{N} [r_{0,g} \log_2(r_{0,g}) + (1 - r_{0,g}) \log_2(1 - r_{0,g})] \tag{2.18}$$

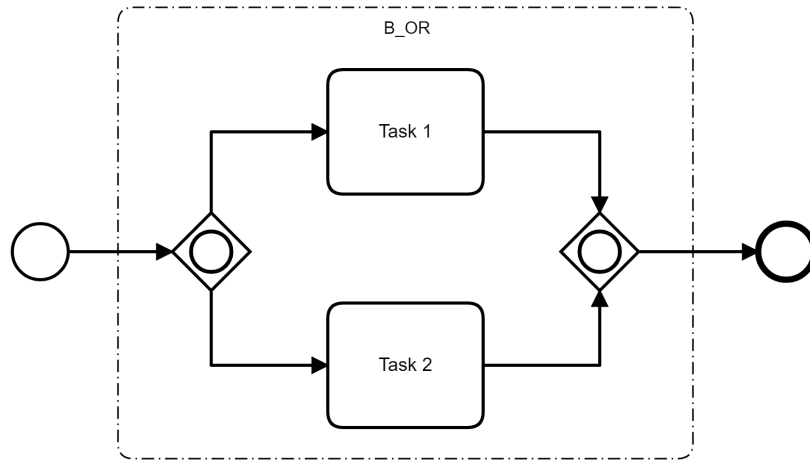

Abbildung 2.11: Inklusives Gateway mit Tasks

Generell kann die Formel zur Berechnung der Unsicherheit von inklusiven Blöcken alternativ auch wie folgt definiert werden:

$$U(B_{OR}) = -\sum_{g=1}^{N} [r_{0,g} \log_2(r_{0,g}) + (1 - r_{0,g}) \log_2(1 - r_{0,g})] + \sum_{g=1}^{N} r_{0,g} \times U(B_g) \tag{2.19}$$

2.2.6 Schleife

In einem Schleifen-Konstrukt wird eine Verzweigung rückgeführt, sodass bestimmte Elemente wiederholend ausgeführt werden können, bis eine bestimmte Bedingung erfüllt wird. Schleifen-Konstrukte sind nicht an ein bestimmtes Element gebunden, sondern können mithilfe verschiedener Elemente wie etwa exklusiven Gateways realisiert werden. In Abbildung 2.12 kann eine Schleife (B_{LOOP}) betrachtet werden, welche zwei Prozessblöcke B_g $(1 \leq g \leq 2)$ beinhaltet. Die durch diese Schleife entstehende Unsicherheit wird nach Jung et al. [2] wie folgt berechnet:

$$\begin{aligned} U(B_{LOOP}) = &- \left[1 - (r_{1,2})^L\right] \times \left[\frac{r_{1,2} \log_2(r_{1,2})}{r_{1,3}} + \log_2(r_{1,3})\right] \\ &+ \left[\frac{1 - (r_{1,2})^{L+1}}{r_{1,3}}\right] U(B_1) + \left[\frac{r_{1,2} - (r_{1,2})^{L+1}}{r_{1,3}}\right] U(B_2) \end{aligned} \tag{2.20}$$

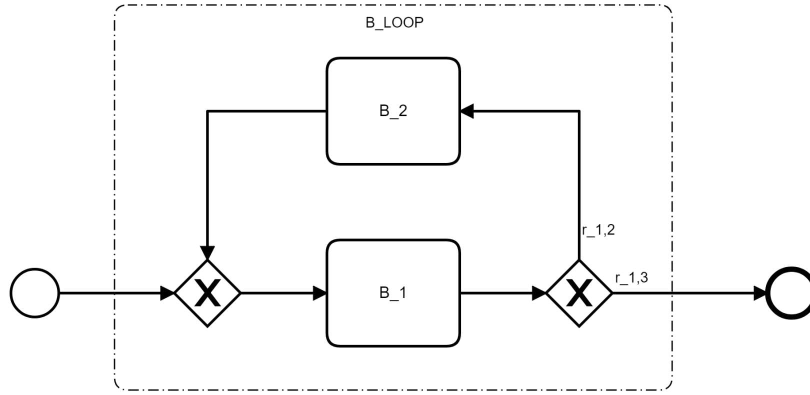

Abbildung 2.12: Schleife

Der Beweis der Formel erfolgt nach Jung et al. [2] anhand des in Abbildung 2.12 dargestellten Prozesses. Dabei gilt: $r_{0,1} = r_{2,1} = 1$ sowie $r_{1,2} + r_{1,3} = 1$. Alle auf dem Block-Level möglichen Ausführungsszenarien des B_{LOOP} Blocks werden in Tabelle 2.1 aufgeführt. Das k-te Ausführungsszenario auf dem Block-Level BS_k ist äquivalent zu einem Sequenz-Block, welcher $k+1$ Prozessblöcke B_1 und k Prozessblöcke B_2 beinhaltet. B_0 sowie B_3 sind dabei nicht in BS_k enthalten, da sie sich nicht in der Schleife befinden. Somit kann die Unsicherheit von BS_k nach Jung et al. [2] wie folgt berechnet werden:

$$U(BS_k) = (k+1) \times U(B_1) \times k \times U(B_2) \tag{2.21}$$

Der Schleifen-Block B_{LOOP} kann als XOR-Block mit L Zweigen, welche jeweils die Ausführungswahrscheinlichkeit aus Tabelle 2.1 aufweisen, betrachtet werden. Dabei gilt: $\sum_{k=0}^{L} P(BS_k) = r_{1,3} + r_{1,2}r_{1,3} + ... + (r_{1,2})^{L-1}r_{1,3} + (r_{1,2})^L = 1$ wenn $r_{1,3} = 1 - r_{1,2}$. Somit kann die Unsicherheit des Schleifen-Blocks B_{LOOP} wie folgt berechnet werden [2]:

$$U(B_{LOOP}) = \sum_{k=0}^{L} P(BS_k) \log_2 [P(BS_k)]^{-1} + \sum_{k=0}^{L} P(BS_k)U(BS_k) \tag{2.22}$$

Der erste Term der Formel für $U(B_{LOOP})$ wird wie folgt erweitert [2]:

$$\begin{aligned} &\sum_{k=0}^{L} P(BS_k) \log_2 [P(BS_k)]^{-1} \\ &= \sum_{k=0}^{L-1} \left\{ (r_{1,2})^k r_{1,3} \log_2 \left[(r_{1,2})^k r_{1,3} \right]^{-1} \right\} + (r_{1,2})^L \log_2 \left[(r_{1,2})^L \right]^{-1} \\ &= -\left[1 - (r_{1,2})^L\right] \times \left[\frac{r_{1,2} \log_2(r_{1,2})}{r_{1,3}} + \log_2(r_{1,3}) \right] \end{aligned} \tag{2.23}$$

Der zweite Term der Formel für $U(B_{LOOP})$ wird wie folgt erweitert [2]:

$$
\begin{aligned}
&\sum_{k=0}^{L} P(BS_k)U(BS_k) \\
&= \sum_{k=0}^{L-1} \left\{(r_{1,2})^k r_{1,3}\left[(k+1)U(B_1) + kU(B_2)\right]\right\} + (r_{1,2})^L\left[(L+1)U(B_1) + LU(B_2)\right] \\
&= \left[\frac{1-(r1,2)^{L+1}}{r_{1,3}}\right]U(B_1) + \left[\frac{r_{1,2}-(r_1,2)^{L+1}}{r_{1,3}}\right]U(B_2) \\
&= P(B_1)U(B_1) + P(B_2)U(B_2)
\end{aligned}
\tag{2.24}
$$

Somit kann die Unsicherheit eines Schleifen-Blocks B_{LOOP} mit zwei Prozessblöcken B_g $(1 \leq g \leq 2)$ mit folgender Formel dargestellt werden [2]:

$$
\begin{aligned}
U(B_{LOOP}) = &-\left[1-(r_{1,2})^L\right]\left[\frac{r_{1,2}\log_2(r_{1,2})}{r_{1,3}} + \log_2(r_{1,3})\right] \\
&+ \left[\frac{1-(r_{1,2})^{L+1}}{r_{1,3}}\right]U(B_1) + \left[\frac{r_{1,2}-(r_{1,2})^{L+1}}{r_{1,3}}U(B_2)\right] \\
&= -\sum_{k=0}^{L} P(BS_k)\log_2 P(BS_k) + \sum_{g=1}^{2} P(B_g)U(B_g)
\end{aligned}
\tag{2.25}
$$

Nummer	Block-Level Ausführungsszenarien	Wahrscheinlichkeit der Ausführungsszenarien
0	$B_0 - B_1 - B_3$	$P(BS_0) = r_{0,1}r_{1,3} = r_{1,3}$
1	$B_0 - B_1 - B_2 - B_1 - B_3$	$P(BS_1) = r_{0,1}r_{1,2}r_{2,1}r_{1,3}$
.	.	.
.	.	.
.	.	.
$L-1$	$B_0 - B_1 - B_2 - B_1 - \cdots - B_2 - B_1 - B_3$	$P(BS_{L-1}) = r_{0,1}r_{1,2}r_{2,1}\cdots r_{2,1}r_{1,3} = (r_{1,2})^{L-1}r_{1,3}$
L	$B_0 - B_1 - B_2 - B_1 - \cdots - B_2 - B_1 - B_2 - B_1 - B_3$	$P(BS_L) = r_{0,1}r_{1,2}r_{2,1}\cdots r_{2,1}r_{1,2}r_{2,1} = (r_{1,2})^L$

Tabelle 2.1: Ausführungsszenarien von B_{LOOP} [2]

2.2.7 Berechnung von gesamten Prozessen

Die Berechnung der Unsicherheit von gesamten Geschäftsprozessen ist ein iterativer Prozess. Im ersten Durchgang werden die größten logischen Blöcke, welche jedoch keine weiteren logischen Blöcke enthalten dürfen, innerhalb des Geschäftsprozesses identifiziert und berechnet. Im zweiten Durchgang werden die aus dem vorherigen Durchgang berechneten logischen Blöcke durch einen Task ersetzt, welcher den berechneten Wert für die Unsicherheit des logischen Blocks enthält. Dabei kommen die Formeln und Methoden aus Kapitel 2.2 zum Einsatz. Dieser Prozess wird wiederholt, bis im letzten Durchgang schlussendlich die Unsicherheit des gesamten Geschäftsprozesses berechnet wird. Den Prozess der Berechnung demonstrieren Jung et al. [2] anhand eines Beispiels, welches in Abbildung 2.13 betrachtet werden kann.

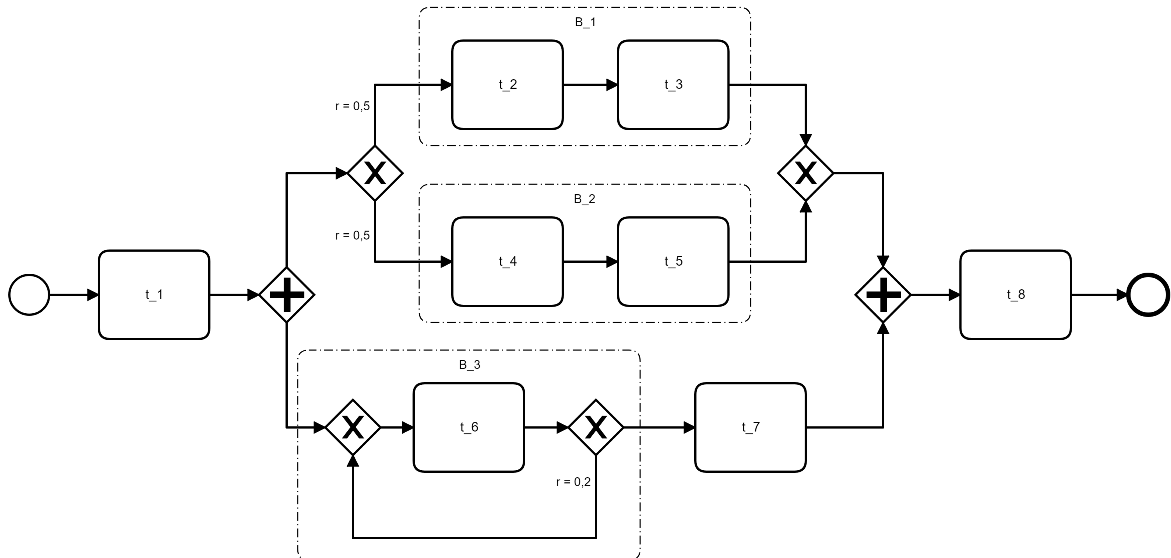

Abbildung 2.13: Beispiel Prozess - Erste Iteration

Im ersten Durchgang werden zwei Sequenzen (B_1 und B_2) sowie eine Schleife (B_3) identifiziert. Alle sich im Geschäftsprozess befindlichen Tasks besitzen die Unsicherheit $U(t_i) = 0$ für $1 \leq i \leq 8$. Für den Block B_3 wird angenommen, dass die Schleife über eine Rückführungswahrscheinlichkeit von $r = 0.2$ verfügt. Außerdem gibt es eine Limitierung für die Anzahl der Iterationen ($L = \infty$). Somit wird die Unsicherheit der einzelnen Blöcke wie folgt berechnet [2]:

$$U(B_1) = U(B_2) = U(t_2) + U(t_3) = U(t_4) + U(t_5) = 0 \tag{2.26}$$

$$U(B_3) = \lim_{L \to \infty} - \left[1 - (0.2)^L\right] \times \left[\frac{0.2 \log_2(0.2)}{0.8} + \log_2(0.8)\right] = 0.902 \tag{2.27}$$

Im zweiten Durchgang werden die berechneten Blöcke durch Tasks ersetzt, welche den berechneten Wert für die Unsicherheit enthalten. Der neue Prozess kann in Abbildung 2.14 betrachtet werden. Des Weiteren werden zwei logische Blöcke B_4 und B_5 identifiziert.

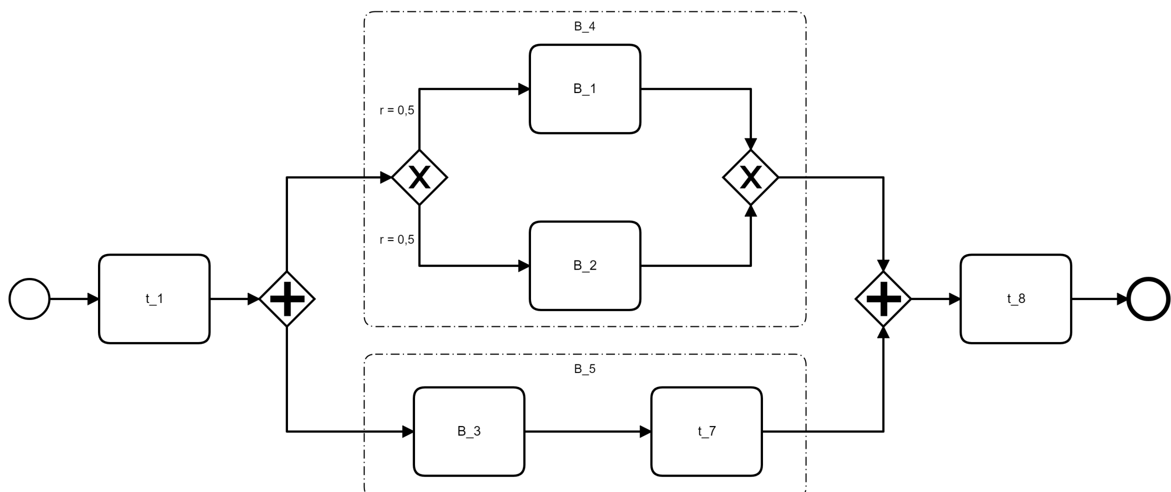

Abbildung 2.14: Beispiel Prozess - Zweite Iteration

Bei B_4 handelt es sich um einen XOR-Block mit zwei ausgehenden Sequenzflüssen. Dabei ist die Wahrscheinlichkeit des Übergangs der jeweiligen Sequenzflüsse $r = 0.5$. B_5 ist eine Sequenz zwischen B_3 und Task 7. Somit werden die Blöcke B_4 und B_5 wie folgt berechnet:

$$U(B_4) = -2 \times [0.5 \log_2(0.5)] = 1 \tag{2.28}$$

$$U(B_5) = 0.902 + 0 = 0.902 \tag{2.29}$$

In der dritten Iteration werden wieder die logischen Blöcke, welche in der vorherigen Iteration berechnet wurden, durch Tasks mit den jeweiligen Werten für die Unsicherheit ersetzt. Dieser Schritt wurde in Abbildung 2.15 dargestellt.

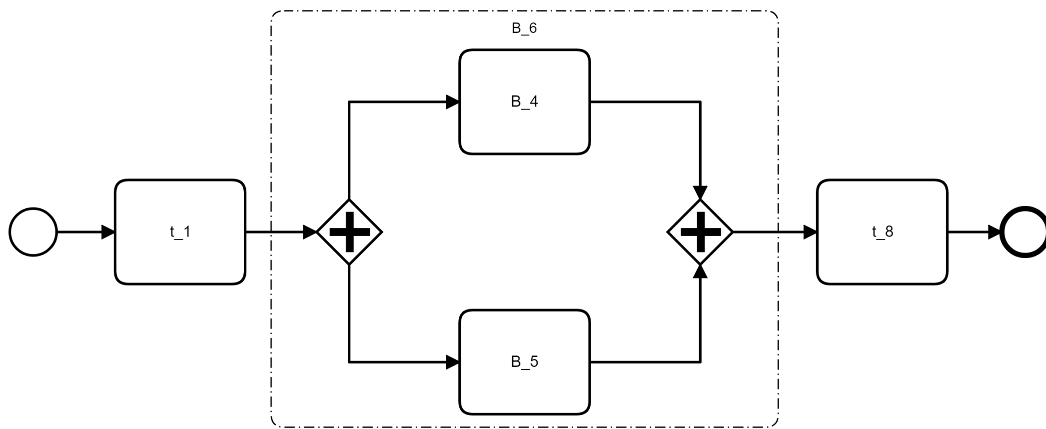

Abbildung 2.15: Beispiel Prozess - Dritte Iteration

In dieser Iteration wurde ein paralleler Block (B_6) identifiziert. Die Formel für die Berechnung von parallelen Blöcken gleicht der Formel für die Berechnung von Sequenzen. Somit wird die Unsicherheit von B_6 wie folgt berechnet:

$$U(B_6) = 1 + 0.902 = 1.902 \tag{2.30}$$

In der vierten und letzten Iteration verbleibt lediglich eine Sequenz aus drei Elementen (siehe Abbildung 2.16).

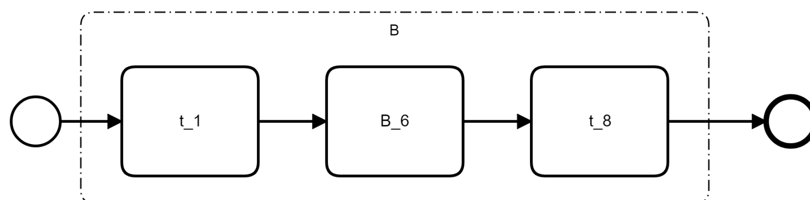

Abbildung 2.16: Beispiel Prozess - Vierte Iteration

Somit wird die Unsicherheit der verbleibenden Elemente, welche ebenfalls der Unsicherheit des gesamten ursprünglichen Geschäftsprozesses aus Abbildung 2.13 entspricht, wie folgt berechnet:

$$U(B) = U(t_1) + U(B_6) + U(t_8) = 1.902 \tag{2.31}$$

2.2.8 Formelübersicht

Hier werden lediglich die in Kapitel 2.2 vorgestellten Formeln noch einmal in einer übersichtlichen Form dargestellt.

Block	Formel
Sequenz	$U(B_{SEQ}) = \sum_{g=1}^{N} U(B_g)$
Parallel	$U(B_{AND}) = \sum_{g=1}^{N} U(B_g)$
Exklusiv	$U(B_{XOR}) = -\sum_{g=1}^{N} r_{0,g} \log_2(r_{0,g}) + \sum_{g=1}^{N} r_{0,g} \times U(B_g)$
Inklusiv	$U(B_{OR}) = -\sum_{g=1}^{N} [r_{0,g} \log_2(r_{0,g}) + (1-r_{0,g}) \log_2(1-r_{0,g})]$ $+ \sum_{g=1}^{N} r_{0,g} \times U(B_g)$
Schleife	$U(B_{LOOP}) = -\left[1-(r_{1,2})^L\right] \times \left[\frac{r_{1,2} \log_2(r_{1,2})}{r_{1,3}} + \log_2(r_{1,3})\right]$ $+ \left[\frac{1-(r_{1,2})^{L+1}}{r_{1,3}}\right] U(B_1) + \left[\frac{r_{1,2}-(r_{1,2})^{L+1}}{r_{1,3}}\right] U(B_2)$

Tabelle 2.2: Formelsammlung aus Kapitel 2.2

2.3 Verzweigungstypen

In Kapitel 2.2 wurden bereits verschiedene Verzweigungstypen vorgestellt, welche in modellierten Geschäftsprozessen häufig anzutreffen sind. Hierzu gehören:

- Parallele Verzweigungen: Unabhängig jeglicher Bedingungen wird jeder ausgehende Sequenzfluss ausgeführt.
- Exklusive Verzweigungen: Abhängig von einer Bedingung wird genau ein ausgehender Sequenzfluss ausgeführt.
- Inklusive Verzweigungen: Die Ausführung eines jeden ausgehenden Sequenzflusses hängt von einer individuellen Bedingung ab. Im Kontrast zu exklusiven Verzweigungen, ist die Ausführung der einzelnen ausgehenden Sequenzflüsse bei inklusiven Verzweigungen nicht von der Ausführung anderer Sequenzflüsse abhängig. Somit ist es unter anderem auch möglich, dass keine oder alle Sequenzflüsse ausgeführt werden [4, S. 292].

Jung et al. [2] veröffentlichten in ihrer Arbeit Formeln, mit welchen man die Unsicherheit basierend auf dem Informationsgehalt nach Shannon für eben jene Verzweigungstypen berechnen kann. Im Standard BPMN 2.0, welcher für die Modellierung von Geschäftsprozessen entwickelt wurde, ist für jeden dieser Verzweigungstypen ein bestimmtes Gateway vorhanden: das parallele Gateway, das exklusive Gateway sowie das inklusive Gateway [4]. Jedoch enthält der BPMN 2.0 Standard noch weitere Elemente, welche eine Verzweigung des Sequenzflusses hervorrufen. Diese lassen sich jedoch üblicherweise in einen der zuvor genannten Verzweigungstypen einordnen.

2.3.1 Ereignisbasierte Gateways

Ein ereignisbasiertes Gateway ist eine Verzweigung, in welcher die Entscheidung über den weiterführenden Sequenzfluss anhand von eintretenden Ereignissen getroffen wird. Dabei wird jener Sequenzfluss ausgewählt, dessen Ereignis als erstes eintritt. Die verbliebenen

Sequenzflüsse werden nicht ausgeführt [4, S. 297]. In Abbildung 2.17 kann ein beispielhafter Einsatz eines ereignisbasierenden Gateways betrachtet werden.

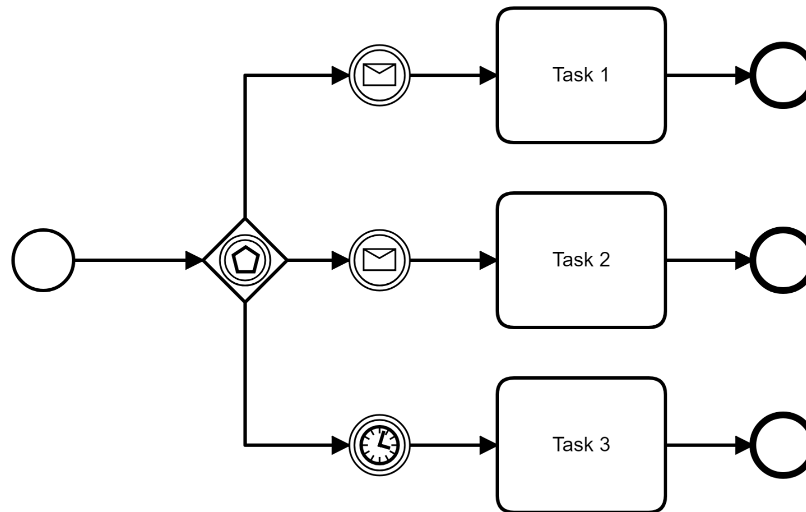

Abbildung 2.17: Ereignisbasiertes Gateway

Das ereignisbasierte Gateway ähnelt somit einem exklusiven Gateway mit dem Unterschied, dass die Entscheidung über den auszuführenden ausgehenden Sequenzfluss nicht über die Abfrage einer Bedingung erfolgt. Jedoch spielt es für die Berechnung der Unsicherheit nach Shannon keine Rolle, wie die Entscheidung zustande kommt. Entscheidend ist, dass lediglich ein weiterführender Sequenzfluss ausgeführt werden kann, wie es auch beim exklusiven Gateway der Fall ist. Die jeweiligen Sequenzflüsse können ebenfalls mit einer Ausführungswahrscheinlichkeit r versehen werden, wobei die Summe der einzelnen Wahrscheinlichkeiten dem Wert 1 (einhundert Prozent) entsprechen muss. Zumindest aus Sicht der Ermittlung der Unsicherheit von Geschäftsprozessen mithilfe der von Jung et al. [2] unterbreiteten Berechnungsmethode, besteht zwischen einem exklusiven Gateway und einem ereignisbasierten Gateway somit kein Unterschied. Somit kann die Formel aus Kapitel 2.2.4, welche für die Berechnung der Unsicherheit von exklusiven Blöcken vorgesehen ist, auch für die Berechnung von ereignisbasierten Blöcken verwendet werden.

2.3.2 Komplexe Gateways

Bei einem komplexen Gateway wird über die Ausführung eines jeden ausgehenden Zweigs einzeln entschieden. Dies erfolgt über bestimmte Bedingungen, welche den jeweiligen ausgehenden Zweigen zugeordnet sind. Somit ist es möglich, dass kein Zweig, ein Zweig, mehrere Zweige oder auch alle Zweige ausgeführt werden. Bei der Einbindung von komplexen Gateways sollten Prozesse so entworfen sein, dass immer zumindest ein ausgehender Zweig des komplexen Gateways ausgeführt wird [4, S. 295].

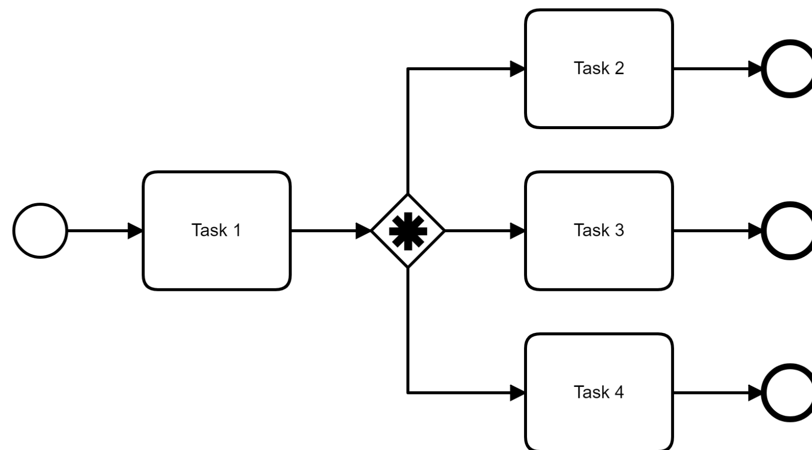

Abbildung 2.18: Komplexes Gateway

Hier lässt sich feststellen, dass das Verhalten des komplexen Gateways sehr stark einem inklusiven Gateway gleicht. Beide Elemente stellen einen Verzweigungspunkt mit beliebig vielen Verzweigungen in einem Prozess dar, in welchen über die Ausführung eines jeden ausgehenden Zweigs einzeln und unabhängig von anderen Zweigen bestimmt wird. Aus diesem Grund kann für die Berechnung der Unsicherheit, welche durch komplexe Gateways verursacht wird, dieselbe Formel wie bei inklusiven Gateways angewandt werden. Die Formel zur Berechnung von inklusiven Blöcken ist in Kapitel 2.2.5 zu finden.

3 Automatisierung

Ein Ziel dieser Arbeit ist es, eine Webapplikation zu entwickeln, welche die Komplexität von Geschäftsprozessmodellen berechnen kann. Als Grundlage der Berechnungsmethode dienen hierzu die Formeln und Methoden aus dem vorherigen Kapitel. Um Prozessmodelle in der Webapplikation einlesen zu können, müssen sie zunächst mit einer geeigneten Software modelliert werden. Dabei ist es wichtig, dass nach dem BPMN 2.0 Standard modelliert wird. Ein geeignetes Tool, welches für die Modellierung von Geschäftsprozessen verwendet werden kann, ist Camunda. Camunda wurde auch verwendet, um die Geschäftsprozesse, welche in dieser Arbeit abgebildet werden, zu modellieren. Modellierte Geschäftsprozesse können in Camunda als .bpmn Datei abgespeichert werden. Diese .bpmn Dateien werden benötigt, um Geschäftsprozesse in der entwickelten Webapplikation einlesen zu können.

3.1 Webapplikation

Als Basis für diese Webapplikation wurde ein bereits bestehendes Projekt verwendet. Es wurde vom Team von bpmn.io entwickelt. Bpmn.io ist ein Toolkit, welches unter anderem eine Bibliothek für JavaScript basierte Anwendungen bietet und von Camunda erhalten wird [5]. Das Projekt, welches als Basis verwendet wurde, kann auf dem folgenden GitHub Repository betrachtet werden:

https://github.com/bpmn-io/bpmn-js-examples/tree/master/modeler

Von diesem Projekt wurden hauptsächlich die Drag-and-Drop-Funktion mit welcher .bpmn Dateien im Code eingelesen werden können, sowie der Model Viewer, welcher eingelesene Geschäftsprozesse im Browser visualisieren kann, übernommen. Eine weitere Bibliothek, welche in die Webapplikation eingebunden wurde, ist die "bpmn-engine". Diese Bibliothek kann auf folgendem GitHub Repository gefunden werden:

https://github.com/paed01/bpmn-engine

Diese Bibliothek wurde eingesetzt, um Sequenzflüsse innerhalb von Geschäftsprozessen zu identifizieren.

3.1.1 Installation

Die Webapplikation wurde in JavaScript entwickelt und ist auf der Plattform GitHub unter folgendem Link einsehbar:

https://github.com/heiderst16/process-analyzer

Des Weiteren kann der Code auch im Anhang in dieser Arbeit betrachtet werden. Um die Webapplikation auf dem lokalen Gerät auszuführen, muss Node.js auf dem Gerät installiert sein. Node.js ist eine JavaScript basierte Laufzeitumgebung, welche für die Entwicklung von Webapplikationen ausgelegt ist [6]. Bei der Entwicklung der Anwendung wurde die Version 14.17.6 von Node.js verwendet. Der Packetmanager npm (node package manager), welcher für die Ausführung der Anwendung ebenfalls benötigt wird, ist üblicherweise in der Installation von Node.js inkludiert. Während der Entwicklung der Anwendung wurde npm Version 6.14.15 verwendet. Die Anwendung kann als ZIP Datei direkt vom GitHub

Repository heruntergeladen werden, wie es in Abbildung 3.1 dargestellt wird. Alternativ kann das Repository auch mit dem Kommando ”git clone https://github.com/heiderst16/process-analyzer.git” in der Kommandozeile in das gewünschte Verzeichnis heruntergeladen werden. Hierfür muss jedoch die Software Git am Gerät installiert sein.

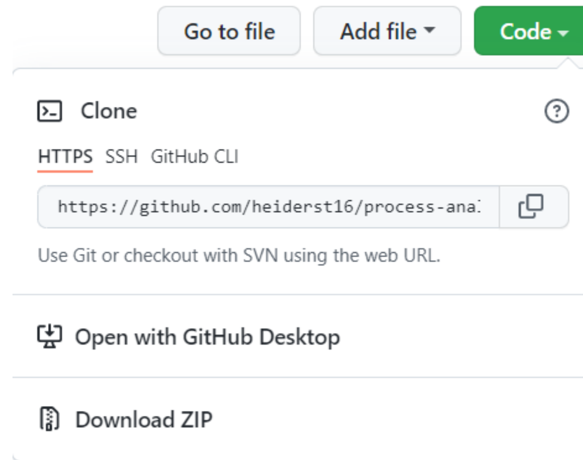

Abbildung 3.1: Anwendung als ZIP Datei

Befindet sich die Webapplikation auf dem Gerät und wurde Node.js sowie npm erfolgreich installiert, so kann die Webapplikation mithilfe von zwei Kommandos ausgeführt werden. Hierfür muss ein Kommandozeilenprogramm geöffnet, und in das Verzeichnis der Webapplikation navigiert werden. Anschließend führt man zunächst den Befehl ”npm install” aus. Dieser Befehl installiert alle Pakete, welche für die Ausführung der Webapplikation benötigt werden. Wurden alle Pakete erfolgreich installiert, so kann die Webapplikation mit dem Befehl ”npm start” lokal ausgeführt werden. Wurde die Webapplikation erfolgreich gestartet, so ist sie anschließend auf dem Port 8080 erreichbar. Sollte sich der Browser nicht automatisch öffnen, so kann die Webapplikation über die URL ”http://localhost:8080/” erreicht werden.

3.1.2 Verwendung

Das Interface der Webapplikation ist simpel aufgebaut. Der Aufbau der Webseite kann in Abbildung 3.2 betrachtet werden. Um einen modellierten Geschäftsprozess in die Webapplikation zu laden, muss eine .bpmn Datei mittels Drag and Drop in das Browserfenster gezogen werden.

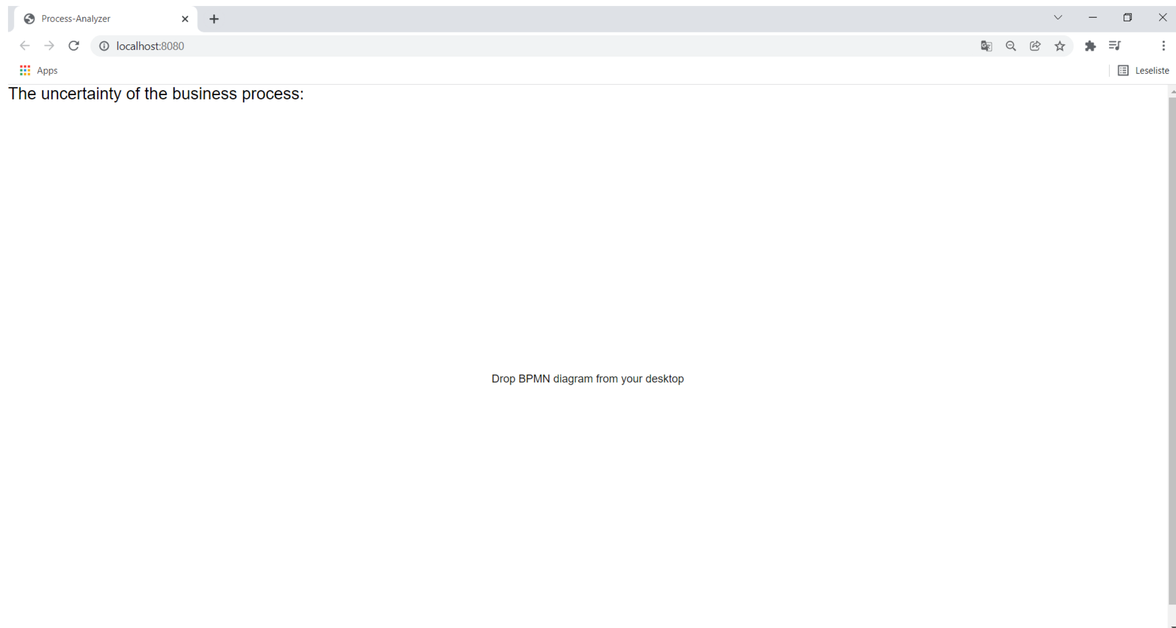

Abbildung 3.2: Webapplikation Startseite

An dieser Stelle ist erwähnenswert, dass die ID des Startelements des Geschäftsprozesses unbedingt als ”start” zu bezeichnen ist (siehe Abbildung 3.3). Sollte die ID des Startelements einen anderen Wert aufweisen, so kann die Unsicherheit des Prozesses nicht berechnet werden. Dies liegt daran, dass die Bibliothek ”bpmn-engine” den Startpunkt für die Strukturierung des Prozesses anhand der ID der Elemente ermittelt.

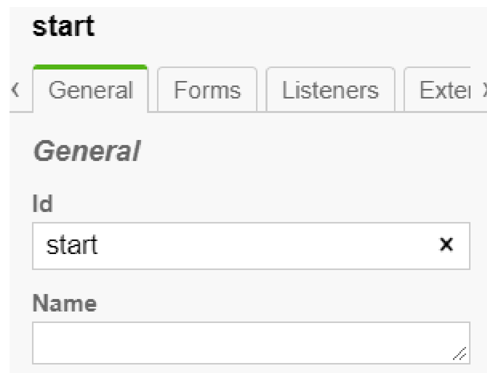

Abbildung 3.3: ID des Startelements

Die Ausführungswahrscheinlichkeit eines ausgehenden Zweiges eines Gateways wird festgelegt, indem die Ausführungswahrscheinlichkeit als Dezimalzahl zur ID des betreffenden Sequenzflusses hinzugefügt wird. Dabei gilt zu beachten, dass die Ausführungswahrscheinlichkeit vorne an die ID hinzugefügt und durch ”_” Zeichen abgegrenzt wird. Ein Beispiel hierfür wird in Abbildung 3.4 dargestellt.

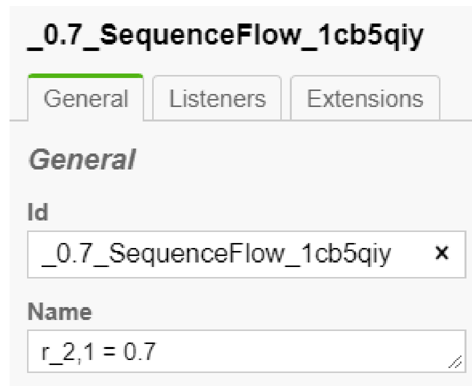

Abbildung 3.4: Ausführungswahrscheinlichkeiten festlegen

Wurde ein Prozess erfolgreich geladen, so wird der Prozess auf der Webseite visuell dargestellt. Des Weiteren wird der Wert für die Unsicherheit des Prozesses automatisch berechnet und ausgegeben. Dies kann in Abbildung 3.5 betrachtet werden.

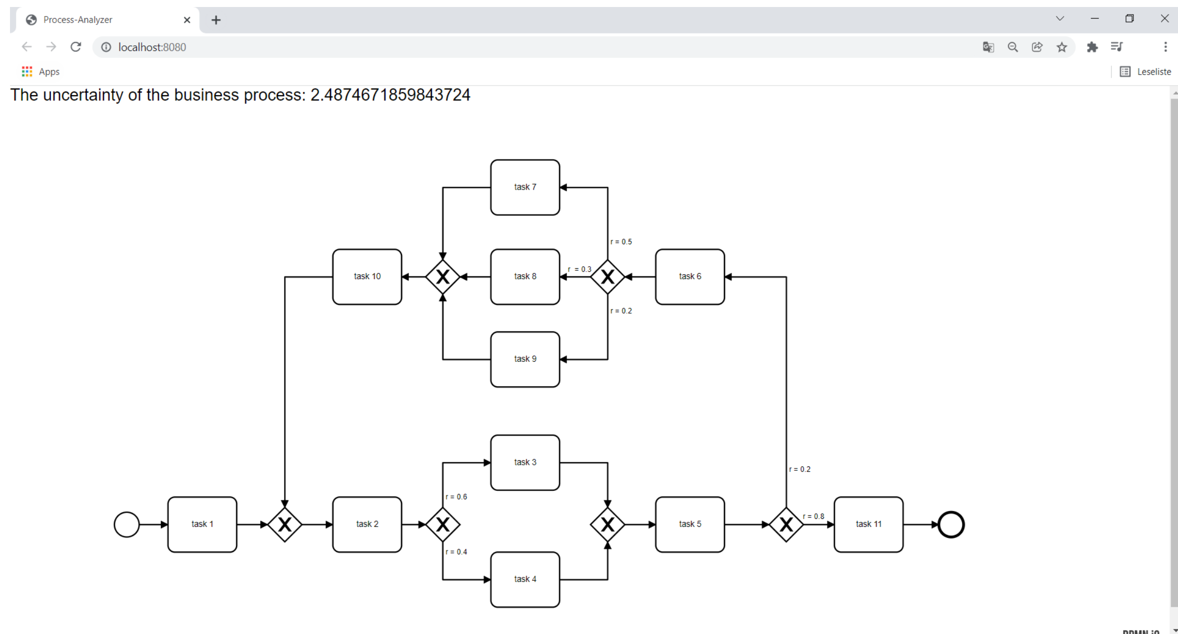

Abbildung 3.5: Webapplikation Prozess geladen

Der Algorithmus dieser Webapplikation fokussiert sich auf Geschäftsprozesse, welche mit den Formeln aus Kapitel 2.2 berechnet werden können. Dies bedeutet, dass nur Geschäftsprozesse berechnet werden können, welche problemlos in Blöcke unterteilbar sind. In Kapitel 3.2.1 werden verschiedene Muster beschrieben, welche die Einteilung des Prozesses in Blöcke erschweren.

3.1.3 Funktionsweise

In diesem Unterkapitel soll die Funktionsweise der Webapplikation beziehungsweise der Algorithmus zur Berechnung der Unsicherheit von Prozessen erläutert werden. Hierbei wird jedoch nicht auf den gesamten Code eingegangen. Stattdessen liegt der Fokus auf den wichtigsten Schritten der Vorgehensweise. Der gesamte Code kann im Anhang betrachtet werden. Die Funktionen, welche zum Einlesen der .bpmn Dateien benötigt werden, wurden

von einem GitHub Projekt übernommen. Dies gilt ebenfalls für die Visualisierung der Prozesse. Nähere Informationen hierzu befinden sich in der Einleitung dieses Hauptkapitels. Da es sich beim Einlesen von XML (Extensible Markup Language) basierten Dateien um einen trivialen Vorgang handelt, wird darauf nicht näher eingegangen. Erwähnenswert ist dabei lediglich, dass die Bibliotheken von bpmn-js hierfür eingesetzt wurden.

Sobald die .bpmn Datei eingelesen werden kann, stellt sich die Frage, wie die Struktur des Prozesses dargestellt werden soll. Die Herausforderung liegt hierbei darin, dass Elemente wie beispielsweise Gateways mit meheren Elementen verbunden sein können [4, S. 288]. Dabei können sie Verzweigungen verursachen, indem sie über mehrere ausgehende Verbindungen verfügen, oder aber auch Verzweigungen zusammenführen, indem sie mehrere eingehende Verbindungen aufweisen. Dies bedeuted, dass Elemente sowohl serielle als auch parallele Verhältnisse zu anderen Elementen aufweisen können. Die Informationen über die einzelnen Elemente eines Geschäftsprozesses, welche man beim Einlesen von .bpmn Dateien erhält, belaufen sich auf folgende:

- Die eindeutige ID des Elements
- Die Bezeichnung des Elements
- Der Typ des Elements (z.B. Task, exklusives Gateway etc.)
- Eingehende und ausgehende Sequenzflüsse (jeweils als eindeutige ID des jeweiligen Sequenzflusses)

Über die Sequenzflüsse des Geschäftsprozesses erhält man folgende Informationen:

- Die eindeutige ID des Sequenzflusses
- Ausgangspunkt sowie Endpunkt des Sequenzflusses (eindeutige ID der jeweiligen Elemente)

Mithilfe dieser Informationen ist es möglich, die Struktur des Geschäftsprozesses abzuleiten. Jedoch stellt sich dennoch die Frage, in welcher Form die Struktur des Geschäftsprozesses als Variable gespeichert werden soll, um die Anwendung eines Algorithmus auf die Variable einfach zu gestalten.

Durch den Einsatz der Bibliothek "bpmn-engine"[1] wurde diese Fragestellung im Zuge dieser Arbeit vom Entwickler dieser Bibliothek abgenommen. Die "bpmn-engine" liest die XML Datei, welche aus der .bpmn Datei extrahiert wurde, ein und liefert unter anderem Arrays, in welchen alle ausführbaren Pfade des Geschäftsprozesses gespeichert sind. Jeder dieser Pfade ist dabei in einem eigenen Array gespeichert. Wie bereits vorhin erwähnt wurde, können in Geschäftsprozessen sowohl serielle Verhältnisse als auch parallele Verhältnisse zwischen einzelnen Elementen bestehen. Die "bpmn-engine" stellt diese Verhältnisse mithilfe der unterschiedlichen Arrays dar. Innerhalb eines Arrays haben Elemente ein serielles Verhältnis zueinander. Sind Verzweigungen im Geschäftsprozess vorhanden, so wird dies durch das Vorliegen von mehreren Arrays dargestellt. Der Code, mittels welchem die "bpmn-engine" diese Instanzen aus der XML Datei extrahiert, lautet wie folgt:

```
53      const moddleContext = await (new BpmnModdle({
54        camunda: require('camunda-bpmn-moddle/resources/camunda.json'),
55      })).fromXML(xml);

66      const sourceContext = Serializer(moddleContext, TypeResolver(
          elements));
67      const engine = Engine({
68        sourceContext,
69      });
```

[1]Die Bibliothek steht als npm Paket zur Verfügung: https://www.npmjs.com/package/bpmn-engine

```
70
71      const [definition] = await engine.getDefinitions();
```

Liest man Geschäftsprozess aus Abbildung 3.6 ein, so erhält man von der ”bpmn-engine” vier Arrays, welche jeweils einen Pfad darstellen. Die vier Instanzen lauten wie folgt:

- $Start \rightarrow Task1 \rightarrow Ex1 \rightarrow Task2 \rightarrow Ink1 \rightarrow Task3 \rightarrow Ink2 \rightarrow Ex2 \rightarrow Task7 \rightarrow Ende$
- $Start \rightarrow Task1 \rightarrow Ex1 \rightarrow Task2 \rightarrow Ink1 \rightarrow Task4 \rightarrow Ink2 \rightarrow Ex2 \rightarrow Task7 \rightarrow Ende$
- $Start \rightarrow Task1 \rightarrow Ex1 \rightarrow Task5 \rightarrow Ex2 \rightarrow Task7 \rightarrow Ende$
- $Start \rightarrow Task1 \rightarrow Ex1 \rightarrow Task6 \rightarrow Ex2 \rightarrow Task7 \rightarrow Ende$

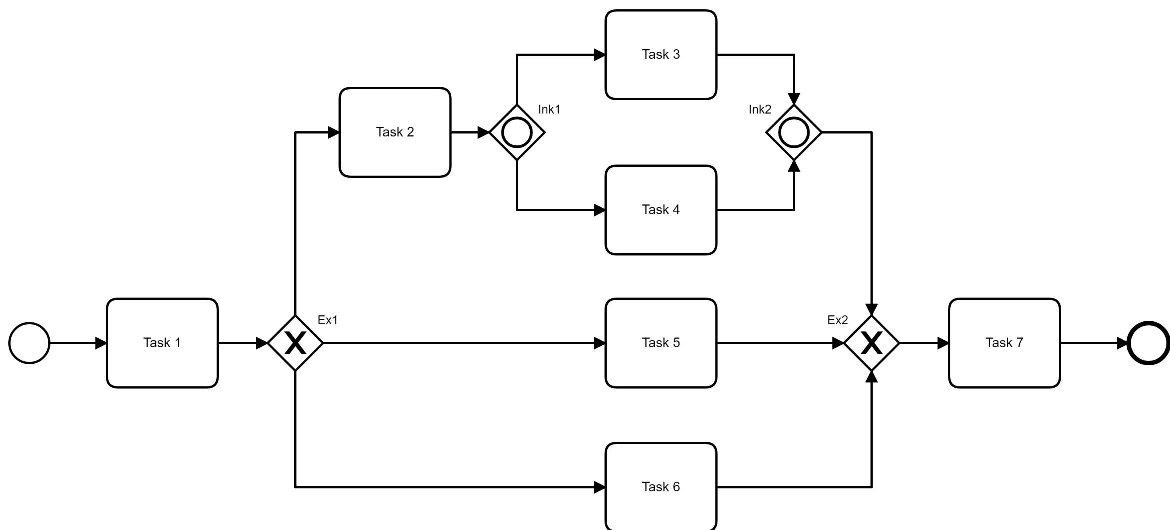

Abbildung 3.6: Beispiel: Instanzen

Die Darstellung der Struktur des Geschäftsprozesses mithilfe der möglichen Pfade bringt bei der Anwendung eines Algorithmus einen Nachteil mit sich. Wie im vorherigen Beispiel erkennbar ist, können sich Elemente in mehreren Pfaden befinden. Wenn also beispielsweise zusätzliche Informationen zu einem Element hinzugefügt werden sollen, dann muss sichergestellt werden, dass dieselben Informationen zu diesem Element in jedem Array hinzugefügt werden. Auch bei weiteren Operationen gilt diese Redundanz zu berücksichtigen. Da ein jedes Element über eine eindeutige ID verfügt, kann dieses Problem gelöst werden, indem über alle Arrays iteriert und nach der gewünschten ID gesucht wird. Diese Thematik führt jedoch dazu, dass an zahlreichen Stellen des Codes über alle Arrays iteriert werden muss, was sich negativ auf die Übersichtlichkeit und damit auch die Nachvollziehbarkeit des Codes auswirkt.

Zu Beginn werden die Arrays gefiltert. Dabei werden alle Elemente entfernt, bei welchen es sich nicht um Gateways handelt. Dies bedeutet jedoch auch, dass Verzweigungen nicht korrekt vom Algorithmus erkannt werden können, wenn sie von Tasks ausgehen. Aus diesem Grund ist bei der Modellierung von Geschäftsprozessen darauf zu achten, lediglich Gateways zur Darstellung von Verzweigungen zu verwenden, wenn diese Webapplikation zur Berechnung der Unsicherheit verwendet werden soll. Das Entfernen jeglicher Elemente, welche keine Gateways sind, ist für das Identifizieren von Blöcken hilfreich.

Der nächste Schritt befasst sich damit, die Elemente innerhalb der Arrays zu kategorisieren. Hierfür wird zunächst über die Arrays iteriert und ein jedes Gateway mit einem Attribut

versehen, welches die Information enthält, ob es sich bei dem Gateway um ein verzweigendes Gateway oder ein zusammenführendes Gateway handelt. Dies wird anhand der Anzahl der eingehenden und ausgehenden Sequenzflüsse beurteilt. Verfügt ein Gateway über einen eingehenden Sequenzfluss und mehrere ausgehende Sequenzflüsse, so handelt es sich bei diesem Gateway um ein verzweigendes Gateway. Verfügt ein Gateway hingegen über mehrere eingehende Sequenzflüsse und einen ausgehenden Sequenzfluss, so handelt es sich bei diesem Gateway um ein zusammenführendes Gateway. Der Fall, dass ein Gateway über mehrere eingehende und mehrere ausgehende Gateways verfügt, wurde in dieser Anwendung ausgeschlossen. Ein Geschäftsprozess, welcher über ein Gateway mit mehreren eingehenden und mehreren ausgehenden Sequenzflüssen verfügt, kann somit nicht korrekt mit dieser Anwendung berechnet werden. Zuzüglich zur Kategorisierung der Elemente in verzweigende und zusammenführende Gateways werden alle Elemente mit zusätzlichen Informationen versehen. Einige dieser zusätzlichen Attribute werden erst im späteren Verlauf befüllt. Die Attribute "calculated" und "removed" spiegeln den Status eines Elements wieder. Diese beiden Attribute werden im Laufe der Berechnung benötigt und entsprechend gesetzt. Der Code hierfür lautet wie folgt:

```
140     function convertElements(e) {
141       const fullElement = businessElements[e.id];
142       const incoming = fullElement.element.incoming.length;
143       const outgoing = fullElement.element.outgoing.length;
144       if (incoming == 1 && outgoing > 1) {
145         return ({
146           id: e.id, type: e.type, sm: "split", outgoing: outgoing,
147           uncertainty: 0, uncertaintyOfBranches: [], recordedBranches
              : [],
148           removed: false, calculated: false, counter: 0, paired:
              false, pairid: "",
149           isLoop: false, classification: ""
150         });
151       } else if (incoming > 1 && outgoing == 1) {
152         return ({
153           id: e.id, type: e.type, sm: "merge", incoming: incoming,
154           uncertainty: 0, uncertaintyOfBranches: [], recordedBranches
              : [],
155           removed: false, calculated: false, counter: 0, paired:
              false,
156           pairid: "", isLoop: false, classification: ""
157         });
158       }
159     };
```

Im nächsten Schritt werden die Gateway-Paare identifiziert. Um die Gateway-Paare zu identifizieren, wird über alle Pfade iteriert. In den einzelnen Arrays befinden sich lediglich Gateways, da die restlichen Elemente herausgefiltert wurden. Bei diesem Vorgang wird nach verzweigenden Gateways gesucht, auf welche ein zusammenführendes Gateway folgt. Wird ein solches Paar identifiziert, muss noch überprüft werden, ob sich in einem anderen Array Elemente zwischen den beiden Gateways befinden. Wenn sich in den anderen Arrays ebenfalls keine weiteren Elemente zwischen diesen beiden Gateways befinden, so handelt es sich hierbei um einen Block. Hierbei wird erneut angenommen, dass sich im Prozess keine Verzweigungen befinden, welche nur teilweise oder an verschiedenen Stellen zusammengeführt werden.

Wird ein Block erfolgreich identifiziert, so werden beide Gateways mit einer Paar-ID versehen, welche sich aus den beiden IDs der jeweiligen Gateways zusammensetzt. Dabei werden die IDs durch ein Zeichen getrennt, um sicherzustellen, dass es sich bei der zusammengesetzten ID ebenfalls um eine eindeutige ID handelt. Des Weiteren werden Gateways, welche bereits

einem Block zugeordnet werden konnten, mit einem booleschen Attribut "paired" markiert. Mithilfe dieses Attributs können diese Gateways bei der nächsten Iteration herausgefiltert werden, um nach weiteren Blöcken innerhalb des Prozesses zu suchen. Diese Vorgehensweise kann am besten anhand eines Beispiels demonstriert werden. Der Prozess, welcher als Beispiel dienen soll, ist in Abbildung 3.7 dargestellt.

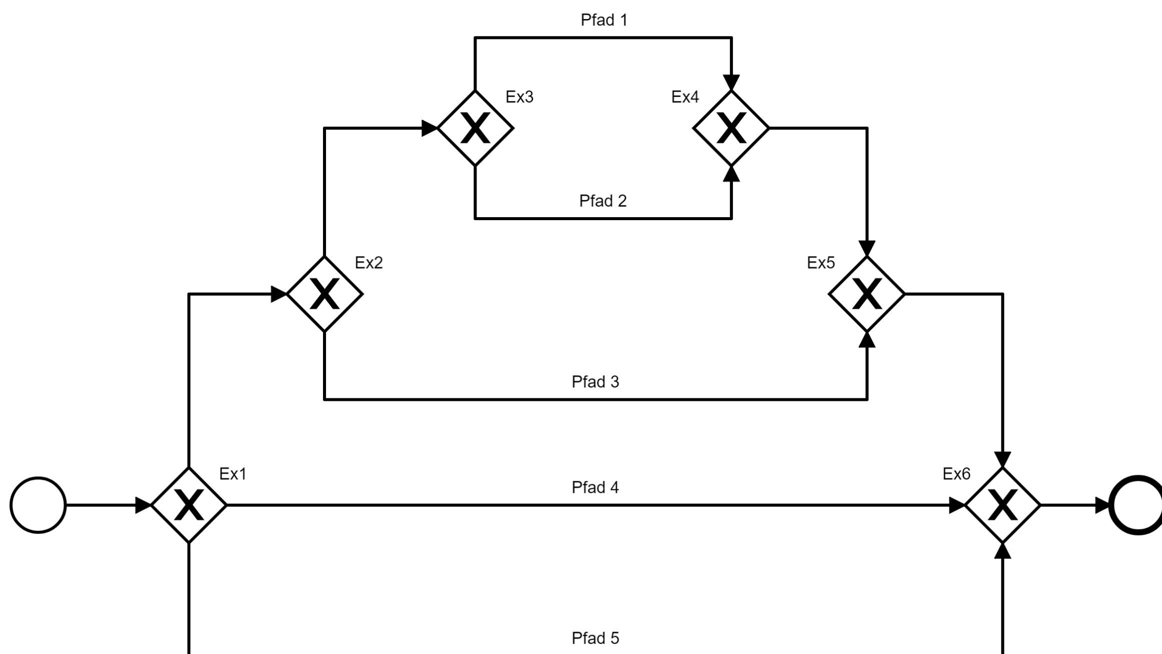

Abbildung 3.7: Beispiel: Gateway-Paare erste Iteration

In der ersten Iteration wird der Block Ex3-Ex4 identifiziert. Dies sind in der ersten Iteration die einzigen Gateways, auf welche die benötigten Bedingungen zutreffen. Auf ein verzweigendes Gateway muss ein zusammenführendes Gateway folgen. Außerdem dürfen sich keine anderen Elemente zwischen den beiden Gateways befinden. Diese Bedingung gilt für alle Instanzen. Die Pfade dieser Iteration lauten wie folgt:

- Pfad 1: $Ex1 \rightarrow Ex2 \rightarrow Ex3 \rightarrow Ex4 \rightarrow Ex5 \rightarrow Ex6$
- Pfad 2: $Ex1 \rightarrow Ex2 \rightarrow Ex3 \rightarrow Ex4 \rightarrow Ex5 \rightarrow Ex6$
- Pfad 3: $Ex1 \rightarrow Ex2 \rightarrow Ex5 \rightarrow Ex6$
- Pfad 4: $Ex1 \rightarrow Ex6$
- Pfad 5: $Ex1 \rightarrow Ex6$

In Pfad 4 und Pfad 5 folgt ein zusammenführendes Gateway (Ex6) auf ein verzweigendes Gateway (Ex1). Jedoch befinden sich Elemente zwischen Ex1 und Ex2 in den restlichen Instanzen. Aus diesem Grund wird dieser Block in der ersten Iteration nicht identifiziert. Nachdem der Block Ex3-Ex4 identifiziert wurde, werden die beiden Gateways mit einer Paar-ID versehen. In der nächsten Iteration werden alle Gateways herausgefiltert, welche bereits einem Block zugewiesen wurden. Der vereinfachte Prozess der zweiten Iteration wird in Abbildung 3.8 dargestellt.

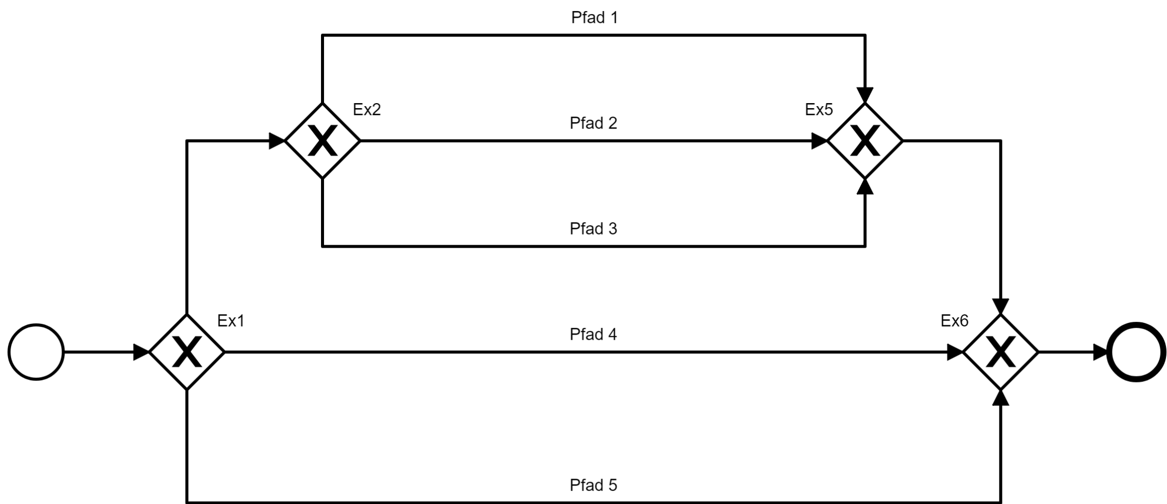

Abbildung 3.8: Beispiel: Gateway-Paare zweite Iteration

Die Pfade dieser Iteration lauten wie folgt:

- Pfad 1: $Ex1 \rightarrow Ex2 \rightarrow Ex5 \rightarrow Ex6$
- Pfad 2: $Ex1 \rightarrow Ex2 \rightarrow Ex5 \rightarrow Ex6$
- Pfad 3: $Ex1 \rightarrow Ex2 \rightarrow Ex5 \rightarrow Ex6$
- Pfad 4: $Ex1 \rightarrow Ex6$
- Pfad 5: $Ex1 \rightarrow Ex6$

In dieser Iteration wird der Block Ex2-Ex5 identifiziert. Daraufhin wird der Prozess erneut vereinfacht, indem die Gateways Ex2 und Ex5 herausgefiltert werden. Der vereinfachte Prozess der dritten Iteration ist in Abbildung 3.9 dargestellt.

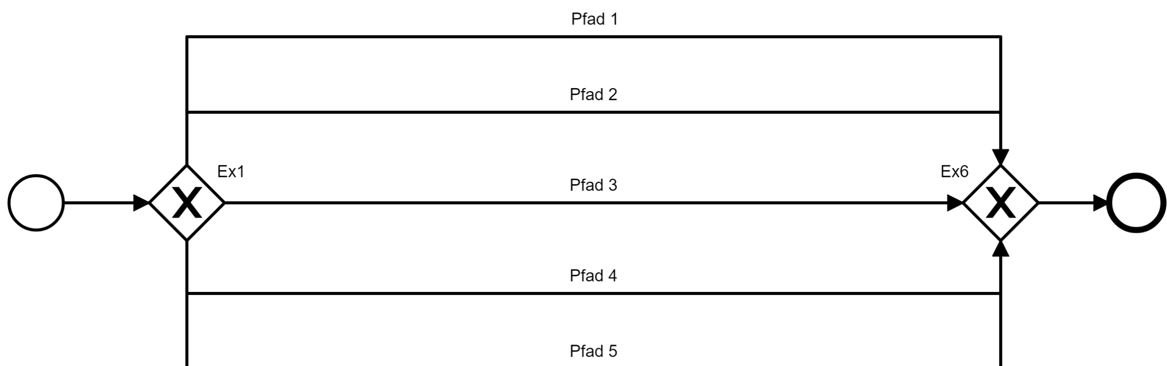

Abbildung 3.9: Beispiel: Gateway-Paare dritte Iteration

Die Pfade dieser Iteration lauten wie folgt:

- Pfad 1: $Ex1 \rightarrow Ex6$
- Pfad 2: $Ex1 \rightarrow Ex6$
- Pfad 3: $Ex1 \rightarrow Ex6$
- Pfad 4: $Ex1 \rightarrow Ex6$
- Pfad 5: $Ex1 \rightarrow Ex6$

In der dritten Iteration wird der Block Ex1-Ex6 identifiziert. Somit hat der Algorithmus die Blöcke Ex3-Ex4, Ex2-Ex5 und Ex1-Ex6 aus dem ursprünglichen Prozess des Beispiels identifiziert.

Befinden sich Schleifen innerhalb des Prozesses, so führt dieses Vorgehen jedoch zu einem unerwünschten Nebeneffekt. Üblicherweise befindet sich zu Beginn eines Blocks ein verzweigendes Gateway und am Ende des Blocks ein zusammenführendes Gateway. Schleifen-Blöcke verhalten sich jedoch entgegengesetzt zu diesem Schema. Bei einem Schleifen-Block befindet sich am Beginn des Blocks ein zusammenführendes Gateway und am Ende des Blocks ein verzweigendes Gateway. In Abbildung 3.10 kann eine Schleife betrachtet werden.

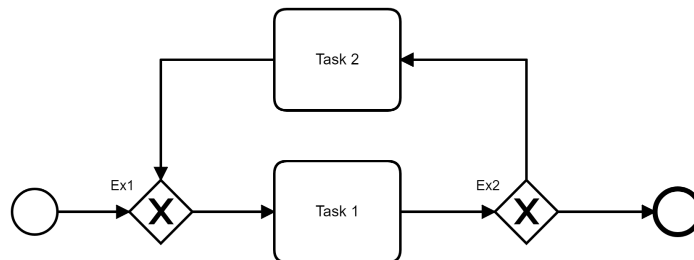

Abbildung 3.10: Beispiel: Pfade einer Schleife

Die "bpmn-engine" weist jedoch die Besonderheit auf, dass bei Abwicklung des rückführenden Pfades einer Schleife das zusammenführende Gateway als letztes Element des Pfades gespeichert wird. Dadurch ist das zusammenführende Gateway in diesem Pfad in zweimaliger Ausführung vorhanden. Die Pfade der Schleife aus Abbildung 3.10 lauten somit wie folgt:

- Pfad 1: $Ex1 \rightarrow Ex2$
- Pfad 2: $Ex1 \rightarrow Ex2 \rightarrow Ex1$

Bei genauerer Betrachtung fällt auf, dass in Pfad 2 ein Paar gebildet werden kann. Bei Ex2 handelt es sich um ein verzweigendes Gateway. In Pfad 2 folgt auf Ex2 das Gateway Ex1, welches wiederum ein zusammenführendes Gateway ist. Somit können auch Gateway-Paare, welche einen Schleifen-Block bilden, mit dem vorhin beschriebenen Algorithmus als Block identifiziert werden. Zu beachten gilt hierbei, dass zunächst falsche Paare gebildet werden können, wenn sich der Schleifen-Block innerhalb eines anderen Blocks befindet. Da sich Schleifen jedoch aufgrund der Duplikate innerhalb der Arrays identifizieren lassen, kann die Bildung der falschen Blöcke rückgängig gemacht werden.
Im Code wird der Algorithmus für die Identifizierung der Paare wie folgt abgebildet:

```
171     function pairElements(listsOfElements) {
172       var pairedList = listsOfElements;
173       for (let x = 0; x < listsOfElements.length; x++) {
174         var filteredUnpairedList = pairedList
175           .map(e => e.filter(m => m.paired != true));
176         if (filteredUnpairedList.length >= 0) {
177           for (let i = 0; i < filteredUnpairedList.length; i++) {
178             const listOfElements = filteredUnpairedList[i];
179             for (let j = 0; j < listOfElements.length; j++) {
180               if (checkIfSequence(filteredUnpairedList,
181                 listOfElements[j],
182                 listOfElements[j + 1])) {
183                 pairedList = setAttributes(pairedList,
184                   listOfElements[j],
185                   listOfElements[j + 1]);
186               }
187             }
188           }
189         }
190       }
191       const pairidArray = pairedList.map(m => m.map(e => e.pairid));
```

```
200        const duplicates = pairidArray
201          .map(e => e.filter((elem, index) => e.indexOf(elem) !== index
               ));
202        var duplicatesConverted = [];
203        for (let i = 0; i < duplicates.length; i++) {
204          duplicatesConverted = duplicatesConverted.concat(duplicates[i
               ]);
205        }
206        for (let i = 0; i < pairedList.length; i++) {
207          for (let j = 0; j < pairedList[i].length; j++) {
208            if (!duplicatesConverted.includes(pairedList[i][j].pairid))
                 {
209              pairedList[i][j].pairid = "";
210              pairedList[i][j].paired = false;
211            }
212          }
213        }
214        return (pairedList);
215      }
```

Nachdem die einzelnen Blöcke innerhalb des Prozesses identifiziert wurden, müssen die Blöcke klassifiziert werden. Dies erfolgt hauptsächlich über die Typen der jeweiligen Gateways. Jedoch muss an dieser Stelle beachtet werden, dass Schleifen nicht an einen bestimmten Gateway-Typen gebunden sind. Aus diesem Grund ist es notwendig, zunächst nach Schleifen innerhalb des Prozesses zu suchen. Die Identifizierung von Schleifen wird in dieser Applikation dadurch ermöglicht, dass die "bpmn-engine" in zumindest einem der ermittelten Pfade das zusammenführende Gateway zweimal speichert. Mithilfe einer Überprüfung von Duplikaten können Schleifen somit identifiziert werden. Wird eine Schleife identifiziert, so werden alle Elemente, welche dem Schleifen-Block zugehörig sind, mithilfe des Attributs "isLoop" markiert. Bei der Klassifizierung der Blöcke wird zunächst mit den Schleifen begonnen. Alle Blöcke, bei deren Gateways das "isLoop" Attribut gesetzt ist, werden als Schleife klassifiziert (im Code als "LOOP" definiert). Nachdem alle Schleifen identifiziert wurden, können die verbleibenden Blöcke anhand der Gateway-Typen klassifiziert werden. Besteht ein Block aus zwei exklusiven Gateways, so handelt es sich bei diesem Block um einen exklusiven Block (als "XOR" definiert im Code). Besteht ein Block aus inklusiven Gateways, so wird der Block als inklusiver Block ("OR") klassifiziert. Beinhaltet ein Block parallele Gateways, so wird er als paralleler Block ("AND") klassifiziert. Der Code, welcher für die Klassifizierung der Blöcke verantwortlich ist, lautet wie folgt:

```
281      function classifyElements(elemArray) {
282        const idArray = elemArray.map(m => m.map(e => e.id));
283        var toclassifyArray = elemArray;
284        const duplicates = idArray
285          .map(e => e.filter((elem, index) => e.indexOf(elem) !== index
               ));
286        var duplicatesConverted = [];
287        for (let i = 0; i < duplicates.length; i++) {
288          duplicatesConverted = duplicatesConverted.concat(duplicates[i
               ]);
289        }
290        for (let i = 0; i < toclassifyArray.length; i++) {
291          for (let j = 0; j < toclassifyArray[i].length; j++) {
292            if (duplicatesConverted.includes(toclassifyArray[i][j].id))
                 {
293              toclassifyArray[i][j].isLoop = true;
294              for (let x = 0; x < toclassifyArray.length; x++) {
295                for (let y = 0; y < toclassifyArray[x].length; y++) {
```

```
296                 if (toclassifyArray[x][y].pairid == toclassifyArray[i
                        ][j].pairid) {
297                   toclassifyArray[x][y].isLoop = true;
298                 }
299               }
300             }
301           }
302         }
303       }
304       for (let i = 0; i < toclassifyArray.length; i++) {
305         for (let j = 0; j < toclassifyArray[i].length; j++) {
306           const classifyElem = toclassifyArray[i][j];
307           if (classifyElem.isLoop) {
308             classifyElem.classification = "LOOP";
309           } else if (classifyElem.type.includes("Exclusive")) {
310             classifyElem.classification = "XOR";
311           } else if (classifyElem.type.includes("Parallel")) {
312             classifyElem.classification = "AND"
313           } else if (classifyElem.type.includes("Inclusive")) {
314             classifyElem.classification = "OR"
315           }
316         }
317       }
318       return toclassifyArray;
319     }
```

Im Anschluss an die Klassifizierung der Blöcke folgt die Berechnung der Unsicherheit. Hierbei handelt es sich um ein aufwendiges Unterfangen, da für die korrekte Berechnung der Unsicherheiten der einzelnen Blöcke verschiedene Faktoren zu berücksichtigen sind. Dies liegt vor allem an der Berechnungsmethode von Jung et al. [2], welche vorschreibt, Blöcke von innen heraus zu berechnen. Somit ist die Berechnung der Unsicherheit eines Geschäftsprozesses ein iterativer Prozess. Die Berechnung der Unsicherheit von einzelnen Blöcken darf nur dann stattfinden, wenn sich innerhalb des Blocks kein weiterer Block befindet. Befindet sich innerhalb des betrachteten Blocks ein weiterer Block, so muss zunächst der innere Block berechnet werden, bevor der äußere Block berechnet werden kann.

In der Webapplikation wird diese Thematik mit einem ”counter” Attribut gelöst. Ähnlich wie bei der Thematik mit der Ermittlung von Gateway-Paaren muss vor der Berechnung eines Blocks festgestellt werden, ob sich innerhalb dieses Blocks ein weiterer Block, welcher noch nicht berechnet wurde, befindet. An dieser Stelle ist das Attribut ”calculated” relevant. Wie der Name vermuten lässt, gibt dieses Attribut darüber Auskunft, ob ein Block bereits berechnet wurde. Mithilfe des Attributs ”calculated” können bereits berechnete Blöcke aus den Arrays gefiltert werden. Im Code werden diese gefilterten Arrays in einer neuen Variable (”filterCalculated”) gespeichert. In der neuen Variable befinden sich lediglich Blöcke, welche noch nicht berechnet wurden. Nun ist es möglich, über diese gefilterten Arrays zu iterieren, um zu überprüfen, ob sich innerhalb eines Blocks ein weiterer Block befindet, welcher noch nicht berechnet wurde. Befinden sich zwischen einem Block (dargestellt durch ein Gateway-Paar) in zumindest einem der Pfade Elemente, welche nicht beim Filtern entfernt wurden, so bedeutet dies, dass der Block noch nicht berechnet werden kann. In diesem Fall wird das ”counter” Attribut des Blocks erhöht. Im späteren Verlauf des Codes wird dieses ”counter” Attribut vor der tatsächlichen Berechnung eines Blocks überprüft. Nur wenn das ”counter” Attribut eines Blocks den Wert 0 besitzt, kann der Block berechnet werden. Nachdem ein Block berechnet wurde, müssen die ”counter” Attribute aller Blöcke zurückgesetzt werden. In der nächsten Iteration werden die ”counter” Attribute erneut ermittelt.

Im folgenden Code-Ausschnitt kann betrachtet werden, wie das ”counter” Attribut eines

Blocks bestimmt wird:

```
334            var filterCalculated = calculateArray
335              .map(e => e.filter(m => m.calculated == false));

354            for (let i = 0; i < filterCalculated.length; i++) {
355              for (let j = 0; j < filterCalculated[i].length; j++) {
356                if (filterCalculated[i][j + 1] != undefined &&
357                  filterCalculated[i][j].pairid != filterCalculated[i][
                       j + 1].pairid &&
358                  filterCalculated[i][j].sm == "split" &&
359                  filterCalculated[i][j].classification != "LOOP") {
360                  for (let x = 0; x < calculateArray.length; x++) {
361                    for (let y = 0; y < calculateArray[x].length; y++)
                         {
362                      if (filterCalculated[i][j].id == calculateArray[x
                           ][y].id) {
363                        calculateArray[x][y].counter += 1;
364                      }
365                    }
366                  }
```

Bei der Bestimmung des "counter" Attributs für Schleifen muss beachtet werden, dass Blöcke vor oder nach dem verzweigenden Gateway positioniert sein können. Aus diesem Grund wird für Schleifen ein leicht abgeänderter Algorithmus benötigt:

```
372                } else if (filterCalculated[i][j].sm == "split" &&
373                  filterCalculated[i][j].classification == "LOOP") {
374                  for (let x = 0; x < filterCalculated.length; x++) {
375                    var filterPairID = filterCalculated[x]
376                      .filter(m => m.pairid == calculateArray[i][j].
                           pairid);

384                    if (filterPairID.length >= 3) {
385                      for (let y = 0; y < filterCalculated[x].length; y
                           ++) {
386                        if (filterCalculated[x][y].id == calculateArray
                             [i][j].id &&
387                          (filterCalculated[x][y + 1].pairid !=
388                            filterCalculated[i][j].pairid ||
389                            filterCalculated[x][y - 1].pairid !=
390                            filterCalculated[i][j].pairid)) {
391                          for (let t = 0; t < filterCalculated.length;
                               t++) {
392                            for (let z = 0; z < filterCalculated[t].
                                 length; z++) {
393                              if (calculateArray[i][j].id ==
                                   calculateArray[t][z].id) {
394                                calculateArray[t][z].counter += 1;
395                              }
396                            }
397                          }
398                        }
399                      }
400                    }
```

Bei der Berechnung der Unsicherheit eines Blocks muss die berechnete Unsicherheit der inneren Blöcke eingebunden werden. Daher ist es notwendig, die berechnete Unsicherheit von inneren Blöcken im äußeren Block zu speichern. Im Code wird die berechnete Unsicherheit von inneren Blöcken im Attribut "uncertaintyOfBranches", welches durch ein Array dargestellt wird, gespeichert. Bevor die Unsicherheit von berechneten innen liegenden Blöcken in

das Array ”uncertaintyOfBranches” des äußeren Blocks hinzugefügt werden kann, muss das ”counter” Attribut abgefragt werden. Nur wenn alle inneren Blöcke bereits berechnet wurden, wird dieser Schritt durchgeführt. Des Weiteren werden weitere Bedingungen abgefragt:

```
414         for (let i = 0; i < calculateArray.length; i++) {
415           for (let j = 0; j < calculateArray[i].length; j++) {
416             if (calculateArray[i][j].counter == 0 &&
417               calculateArray[i][j + 1] != undefined &&
418               !calculateArray[i][j].recordedBranches
419                 .includes(calculateArray[i][j + 1].id) &&
420               calculateArray[i][j].sm == "split" &&
421               calculateArray[i][j].classification != "LOOP") {
```

Bevor die berechneten Unsicherheiten der inneren Blöcke im äußeren Block gespeichert werden, werden die Ausführungswahrscheinlichkeiten der jeweiligen Pfade ermittelt. Diese Ausführungswahrscheinlichkeiten sollten bereits bei der Modellierung des Geschäftsprozesses festgelegt werden. Ausführungswahrscheinlichkeiten von Pfaden werden festgelegt, indem die Ausführungswahrscheinlichkeit als Dezimalzahl zur ID des betreffenden Sequenzflusses hinzugefügt wird. Soll beispielsweise die Ausführungswahrscheinlichkeit 70 % für einen Pfad festgelegt werden, so muss ”_0.7_” vorne an die ID des betreffenden Sequenzflusses hinzugefügt werden. Lautet die ID des betreffenden Sequenzflusses etwa ”SequenceFlow_1cb5qiy”, so lautet die ID des Sequenzflusses ”_0.7_SequenceFlow_1cb5qiy” nach dem Hinzufügen der Ausführungswahrscheinlichkeit. Im Code wird die Ausführungswahrscheinlichkeit eines Pfades aus der ID des betreffenden Sequenzflusses extrahiert. Sollte keine gültige Zahl extrahiert werden, so wird ein Default Wert von 0.5 verwendet:

```
432               for (let k = 0; k < shakenStartsSequences[i].length;
                    k++) {
433                 if (shakenStartsSequences[i][k].id ==
                      calculateArray[i][j].id) {
434                   var id = shakenStartsSequences[i][k + 1].id;
435                   var probabilityArray = id.split('_');
436                   if (!isNaN(probabilityArray[1])) {
437                     var probability = Number(probabilityArray[1]);
438                   } else {
439                     var probability = 0.5;
440                   }
441                 }
442               }
```

Befindet sich in einem Pfad des betrachteten Blocks ein berechneter Block, so wird die Unsicherheit des berechneten Blocks als auch die zugehörige Ausführungswahrscheinlichkeit des Pfades zum ”uncertaintyOfBranches” Attribut des äußeren Blocks hinzugefügt:

```
451                     if (calculateArray[i][j].pairid !=
452                       calculateArray[i][j + 1].pairid) {
453                       if (!calculateArray[x][y].
                              uncertaintyOfBranches
454                         .map(e => e.id).includes(id)) {
455                         calculateArray[x][y].uncertaintyOfBranches
456                           .push({
457                             id: id,
458                             uncertainty: calculateArray[i][j + 1].
                                  uncertainty,
459                             probability: probability
460                           });
461                       }
```

Befindet sich in einem Pfad jedoch kein weiterer Block, so wird die Unsicherheit 0 als auch die zugehörige Ausführungswahrscheinlichkeit des Pfades zum "uncertaintyOfBranches" Attribut des äußeren Blocks hinzugefügt:

```
384                         } else if (calculateArray[i][j].pairid ==
385                           calculateArray[i][j + 1].pairid) {
386                           if (!calculateArray[x][y].
                                  uncertaintyOfBranches
387                             .map(e => e.id).includes(id)) {
388                             calculateArray[x][y].uncertaintyOfBranches
389                               .push({
390                                 id: id,
391                                 uncertainty: 0,
392                                 probability: probability
393                               });
394                           }
395                         }
```

Um diesen Vorgang korrekt für Schleifen abwickeln zu können, wird wieder ein angepasster Algorithmus benötigt. Dieser beruht jedoch auf demselben Prinzip.

Die Formeln zur Berechnung der Unsicherheit von Blöcken entsprechen den Formeln von Jung et al. [2] und wurden bereits in Kapitel 2.2 näher erläutert. Im Code werden Blöcke entsprechend ihrer Klassifizierung ("XOR", "OR", "LOOP", "AND") berechnet. Nachdem ein Block berechnet wurde, werden alle Elemente, welche für die nächsten Iterationen der Berechnung nicht mehr benötigt werden, mit dem Attribut "removed" markiert. Dies ermöglicht es, irrelevante Elemente zu filtern. Im Code wurden die verwendeten Formeln in Teile zerlegt. Dies ist lediglich ein stilistisches Mittel, um die Formeln übersichtlicher zu gestalten. Auf die Ergebnisse der Berechnung hat diese Zerlegung keine Auswirkung.

Bei der Berechnung von Schleifen muss angemerkt werden, dass diese unter der Annahme berechnet werden, dass kein Limit für die Anzahl der Wiederholungen vorhanden ist. Soll die Unsicherheit von Schleifen mit einem Limit für Wiederholungen berechnet werden, so muss das gewünschte Limit im Code direkt festgelegt werden. Wie die Formeln der Berechnung in diesem Fall angepasst werden müssen, ist im Code erläutert.

Im folgenden Code-Ausschnitt kann als Beispiel für die Berechnung der Unsicherheit eines Blocks betrachtet werden, wie die konkrete Berechnung von Schleifen erfolgt:

```
763                   } else if (calculateArray[i][j].classification == "
                         LOOP") {
764                     const loopProbability = findLoopProbability(
                            calculateArray[i][j]
765                       .uncertaintyOfBranches);

779                     const uncertainty1 = -1 * (1 /*- Math.pow(
                            loopProbability, 10)*/)
780                       * ((loopProbability * Math.log2(loopProbability))
                              /
781                         (1 - loopProbability) + Math.log2((1 -
                              loopProbability)));
782                     const uncertainty2 = ((1 /*- Math.pow(
                            loopProbability, 11)*/)
783                       / (1 - loopProbability)) *
784                       findUncertaintyOfBranch(calculateArray[i][j]
785                         .uncertaintyOfBranches, 1);
786                     const uncertainty3 =
787                       ((loopProbability /*- Math.pow(loopProbability,
                              11)*/)
788                         / (1 - loopProbability)) *
```

```
789                         findUncertaintyOfBranch(calculateArray[i][j]
790                           .uncertaintyOfBranches, 2);
791                       for (let g = 0; g < calculateArray.length; g++) {
792                         for (let h = 0; h < calculateArray[g].length; h
                              ++) {
793                           if (calculateArray[i][j].uncertaintyOfBranches
794                             .map(m => m.id).includes(calculateArray[g][h
                                ].id)) {

798                             calculateArray[g][h].removed = true;
799                           }
800                           if (calculateArray[i][j].pairid ==
801                             calculateArray[g][h].pairid) {

805                             calculateArray[g][h].uncertainty =
806                               uncertainty1 + uncertainty2 + uncertainty3;
807                           }
808                         }
809                       }
```

3.1.4 Demonstration

Um die Funktionalität der Applikation zu demonstrieren, wird die Applikation in diesem Kapitel anhand von einigen Beispielen vorgeführt. Die Geschäftsprozesse aus den Beispielen wurden im Zuge dieser Arbeit mit der Ambition entworfen, unterschiedliche Block-Typen sowie unterschiedliche Muster zu testen. Das erste Beispiel kann in Abbildung 3.11 betrachtet werden.

Wie bereits in der Abbildung zu erkennen ist, wird mit folgenden Werten gerechnet: $r_{2,1} = 0.7$, $r_{2,2} = 0.3$, $r_{4,1} = 0.4$, $r_{4,2} = 0.6$, $r_{7,1} = 0.8$, $r_{7,2} = 0.6$, $r_{9,1} = 0.2$, $r_{9,2} = 0.8$ Außerdem gilt für Schleifen: Kein Limit für die Anzahl der Wiederholungen ($\lim_{L\to\infty}$)

Um den Block LOOP1 berechnen zu können, müssen zunächst die inneren Blöcke berechnet werden:

$$\begin{aligned} U(XOR1) &= -\sum_{g=1}^{N} r_{0,g} \log_2 r_{0,g} + \sum_{g=1}^{N} r_{0,g} \times U(B_g) \\ &= -(0.7 \log_2 0.7 + 0.3 \log_2 0.3) + 0.7 \times 0 + 0.3 \times 0 \\ &= 0.8813 \end{aligned} \tag{3.1}$$

$$U(AND1) = \sum_{g=1}^{N} U(B_g) = 0 + 0 = 0 \tag{3.2}$$

Nun kann der Block LOOP1 berechnet werden:

$$
\begin{aligned}
U(LOOP1) = & -\left[1-(r_{4,1})^{L}\right] \times \left[\frac{r_{4,1}\log_2(r_{4,1})}{r_{4,2}} + \log_2(r_{4,2})\right] \\
& + \left[\frac{1-(r_{4,1})^{L+1}}{r_{4,2}}\right] U(B_1) + \left[\frac{r_{4,1}-(r_{4,1})^{L+1}}{r_{4,2}}\right] U(B_2) \\
& = \lim_{L\to\infty} -\left[1-0.4^{L}\right] \times \left[\frac{0.4\log_2(0.4)}{0.6} + \log_2(0.6)\right] \\
& + \left[\frac{1-0.4^{L+1}}{0.6}\right] \times 0.8813 + \left[\frac{0.4-0.4^{L+1}}{0.6}\right] \times 0 \\
& = 3.0871
\end{aligned} \tag{3.3}
$$

Um den Block INC1 berechnen zu können, müssen erneut zunächst die inneren Blöcke berechnet werden:

$$
U(AND2) = \sum_{g=1}^{N} U(B_g) = 0 + 0 = 0 \tag{3.4}
$$

$$
\begin{aligned}
U(LOOP1) = & -\left[1-(r_{9,1})^{L}\right] \times \left[\frac{r_{9,1}\log_2(r_{9,1})}{r_{9,2}} + \log_2(r_{9,2})\right] \\
& + \left[\frac{1-(r_{9,1})^{L+1}}{r_{9,2}}\right] U(B_1) + \left[\frac{r_{9,1}-(r_{9,1})^{L+1}}{r_{9,2}}\right] U(B_2) \\
& = \lim_{L\to\infty} -\left[1-0.2^{L}\right] \times \left[\frac{0.2\log_2(0.2)}{0.8} + \log_2(0.8)\right] \\
& + \left[\frac{1-0.2^{L+1}}{0.8}\right] \times 0 + \left[\frac{0.2-0.2^{L+1}}{0.8}\right] \times 0 \\
& = 0.9024
\end{aligned} \tag{3.5}
$$

Nun kann der Block INC1 berechnet werden:

$$
\begin{aligned}
U(INC1) = & -\sum_{g=1}^{N} \left[r_{0,g}\log_2(r_{0,g}) + (1-r_{0,g})\log_2(1-r_{0,g})\right] + \sum_{g=1}^{N} r_{0,g} \times U(B_g) \\
& = -(0.8\log_2 0.8 + 0.2\log_2 0.2 + 0.6\log_2 0.6 + 0.4\log_2 0.4) \\
& + 0.8 \times 0 + 0.6 \times 0.9024 \\
& = 2.2343
\end{aligned} \tag{3.6}
$$

Die Blöcke LOOP1 und INC1 sind seriell zueinander positioniert und müssen demnach addiert werden. Damit kommt die Berechnung der Unsicherheit des Prozesses auf folgendes Ergebnis:

$$
U = 3.0871 + 2.2343 = 5.3214 \tag{3.7}
$$

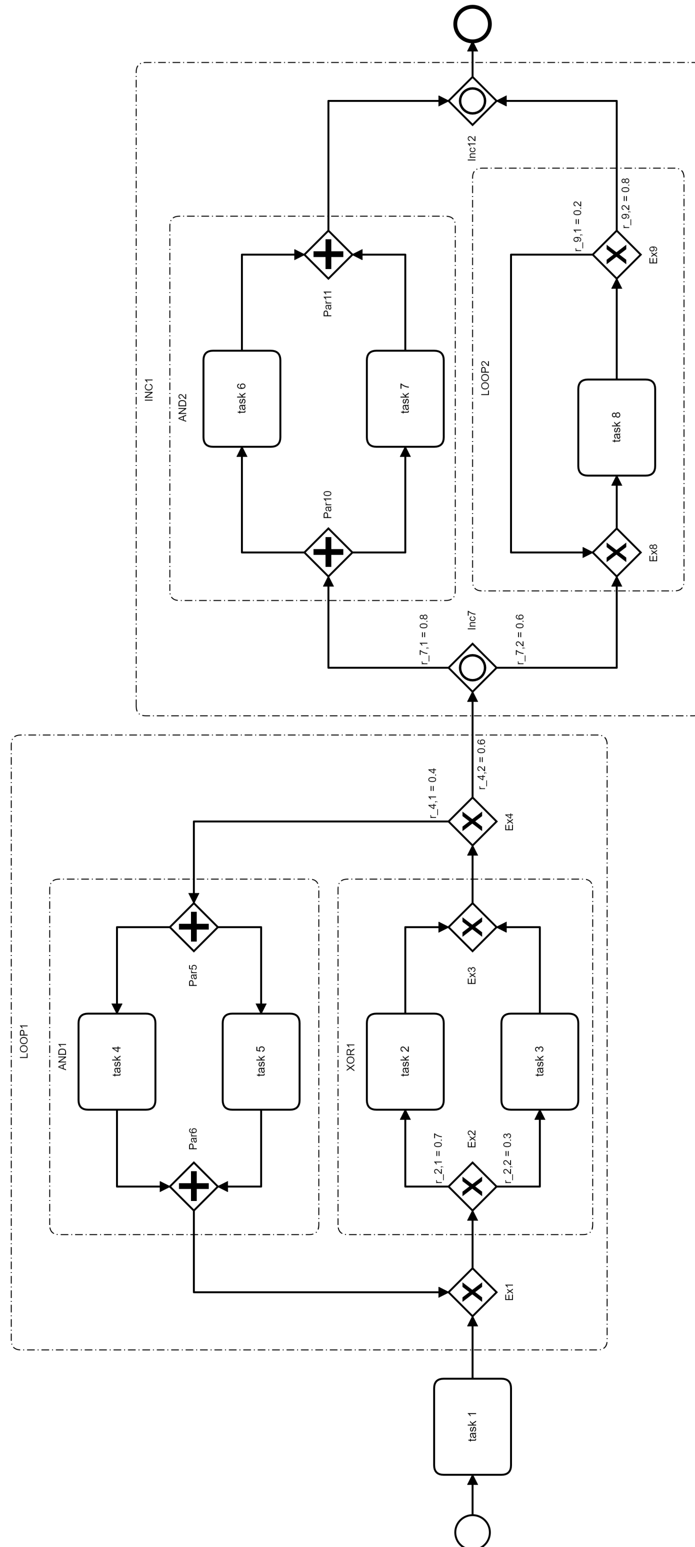

Abbildung 3.11: Beispiel Prozess 1

Wie in Abbildung 3.12 zu erkennen ist, gelangt die Webapplikation zum selben Ergebnis.

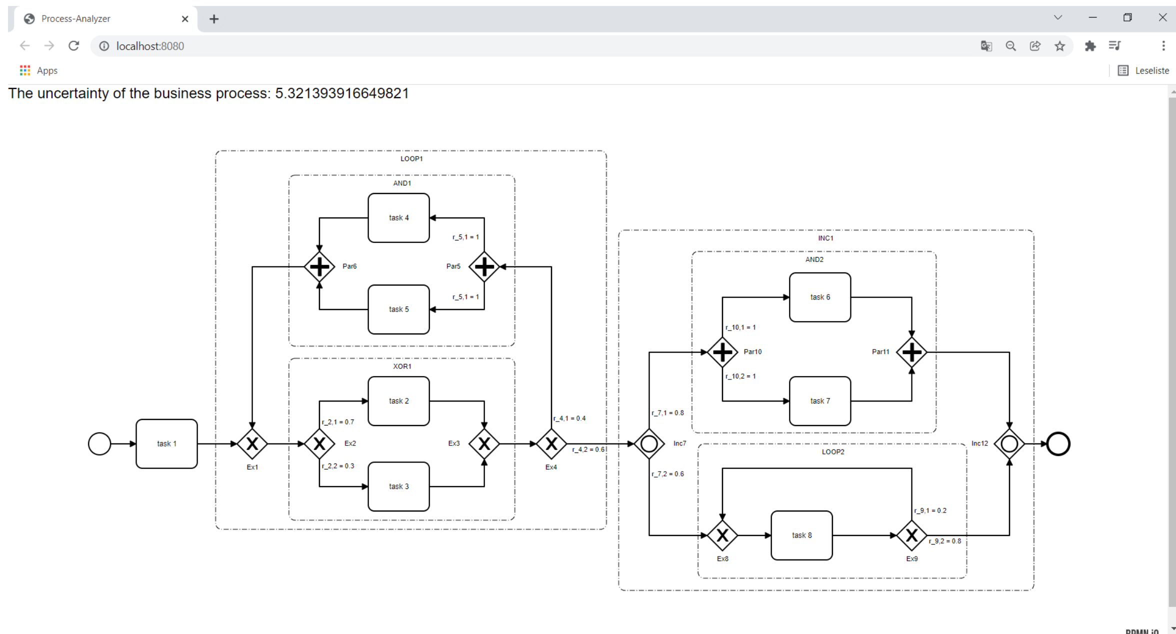

Abbildung 3.12: Berechnung mittels Webapplikation

In Abbildung 3.13 wird mittels der Webapplikation ein weiterer modellierter Geschäftsprozess korrekt berechnet.

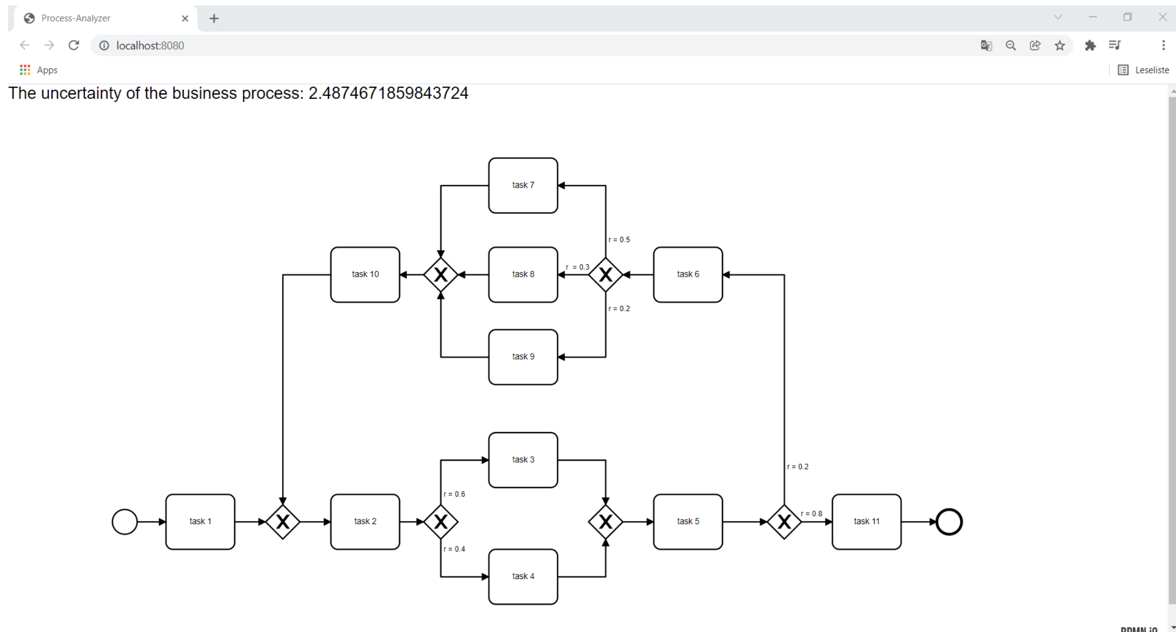

Abbildung 3.13: Geschäftsprozess 2

Im Zuge der Tests wurde festgestellt, dass die Webapplikation die Unsicherheit eines Blocks falsch berechnet, wenn sich innerhalb eines Pfades mehr als ein Block befindet. Dies kann anhand des Geschäftsprozesses demonstriert werden, welcher in Abbildung 3.14 betrachtet werden kann. Der Berechnungsfehler beruht darauf, dass der zweite innere Block nicht in die Berechnung des äußeren Blocks miteinbezogen wird. Anstatt den zweiten inneren Block als Teil des äußeren Blocks zu erkennen, erkennt der Algorithmus den zweiten inneren Block als seriellen Block zum äußeren Block.

Das korrekte Ergebnis des Prozesses wird wie folgt berechnet:

$$
\begin{aligned}
U = &-(0.2\log_2 0.2 + 0.6\log_2 0.6 + 0.2\log_2 0.2)\\
&-0.6\,(0.6\log_2 0.6 + 0.4\log_2 0.4)\\
&-0.6\,(0.7\log_2 0.7 + 0.3\log_2 0.3)\\
&= 2.4823
\end{aligned} \tag{3.8}
$$

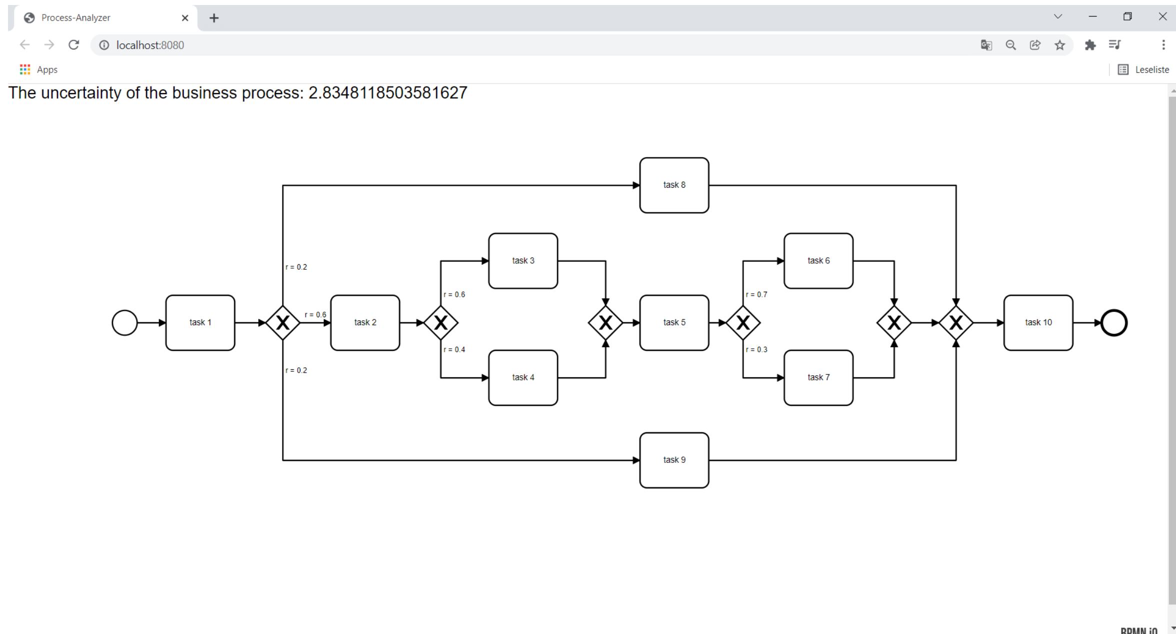

Abbildung 3.14: Geschäftsprozess 3

Im Hintergrund berechnet der Algorithmus aufgrund der falschen Zuordnung jedoch den Prozess, welcher in Abbildung 3.15 zu sehen ist. Somit lautet die Formel, welche von der Webapplikation angewendet wird wie folgt:

$$
\begin{aligned}
U = &-(0.2\log_2 0.2 + 0.6\log_2 0.6 + 0.2\log_2 0.2)\\
&-0.6\,(0.6\log_2 0.6 + 0.4\log_2 0.4)\\
&-(0.7\log_2 0.7 + 0.3\log_2 0.3)\\
&= 2.8348
\end{aligned} \tag{3.9}
$$

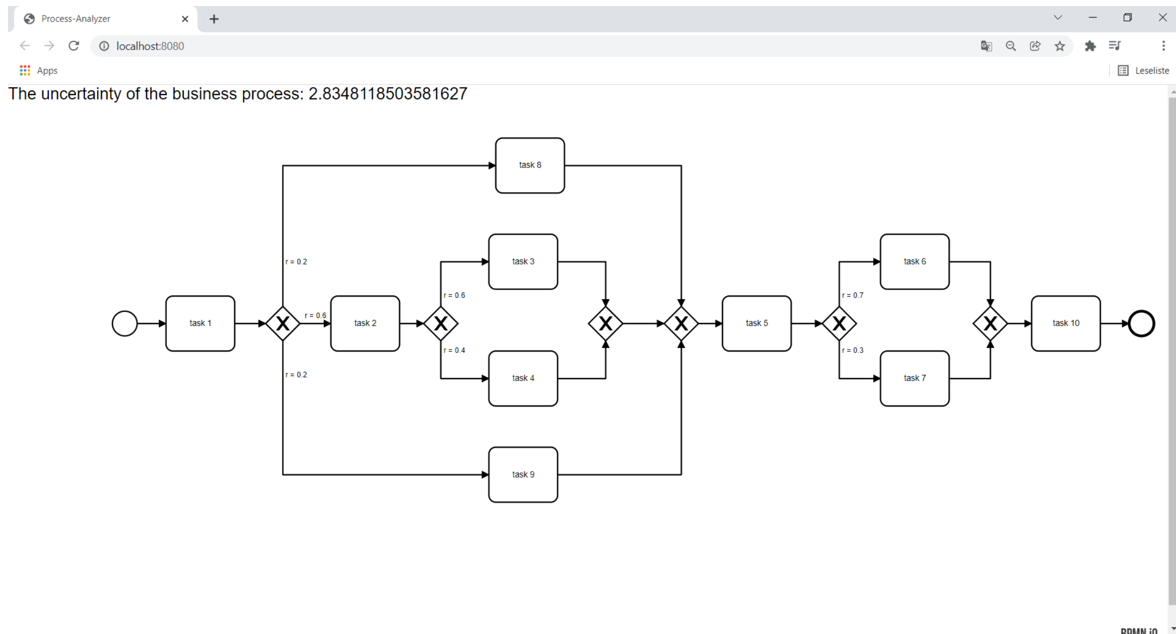

Abbildung 3.15: Geschäftsprozess 4

Dieses Fehlverhalten des Algorithmus kann korrigiert werden, indem der Algorithmus durch die Abänderung des Codes an den entsprechenden Stellen verbessert wird. Für eine Anwendung in der Praxis ist die Applikation somit noch nicht ausreichend ausgereift. Jedoch zeigt die Webapplikation, dass eine automatisierte Berechnung der Unsicherheit von Prozessen grundsätzlich möglich ist.

3.2 Herausforderungen der Automatisierung

Aufgrund der Auseinandersetzung mit dem Thema der automatisierten Berechnung der Unsicherheit von Geschäftsprozessen während der Entwicklung einer Webapplikation sind Überlegungen aufgetreten, welche für zukünftige Umsetzungen von Interesse sein könnten. In Kapitel 2.2 wurde das Thema der Berechnung der Unsicherheit von Geschäftsprozessen behandelt. Dabei müssen für verschiedene Elemente und Muster unterschiedliche Formeln zur Berechnung der Unsicherheit angewandt werden. Ein gemeinsamer Faktor dieser unterschiedlichen Formeln ist jedoch, dass Unsicherheit nur dann entstehen kann, wenn Verzweigungen im Prozess vorhanden sind. Elemente wie etwa das exklusive Gateway, können dabei mehr als zwei ausgehende Sequenzflüsse aufweisen. Bisher wurde im Falle dieser Verzweigungen angenommen, dass alle Zweige im selben Punkt wieder zusammengeführt werden, wie es beispielhaft in Abbildung 3.16 abgebildet ist.

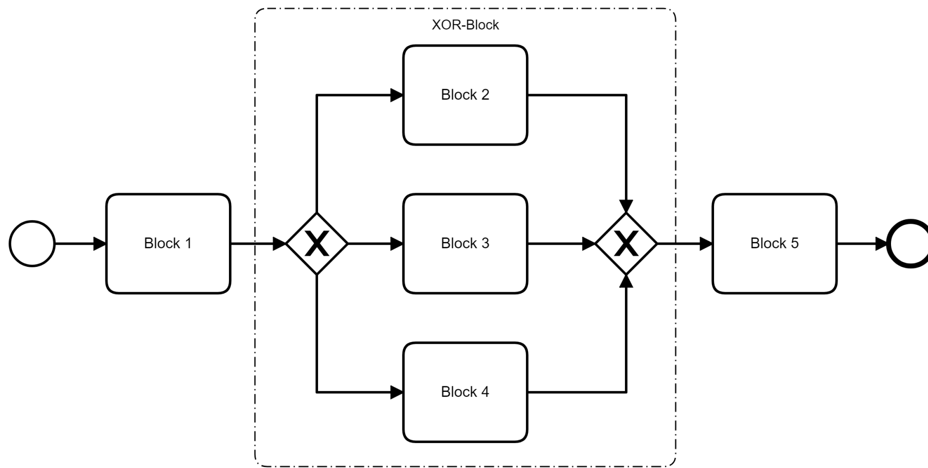

Abbildung 3.16: XOR-Block: An einem Punkt zusammengeführt

Es ist jedoch auch möglich, dass ausgehende Sequenzflüsse nicht im selben Punkt oder überhaupt nicht zusammengeführt werden. Dabei können Muster entstehen, welche nicht mit den bisher definierten Formeln und Methoden berechnet werden können.

3.2.1 Besondere Muster

Im Fall, dass die ausgehenden Sequenzflüsse eines exklusiven Gateways nicht zusammengeführt werden, wie in Abbildung 3.17 dargestellt, kann unter der Annahme, dass eine Erhöhung der Anzahl an möglichen Endpunkten des Geschäftsprozesses die Unsicherheit des Prozesses nicht erhöht, die Unsicherheit des XOR-Blocks dennoch nach der Methode von Jung et al. [2] (siehe Kapitel 2.2.4) berechnet werden. Dabei würde sich der XOR-Block jedoch über den gesamten restlichen Geschäftsprozess, ausgehend vom exklusiven Gateway, spannen.

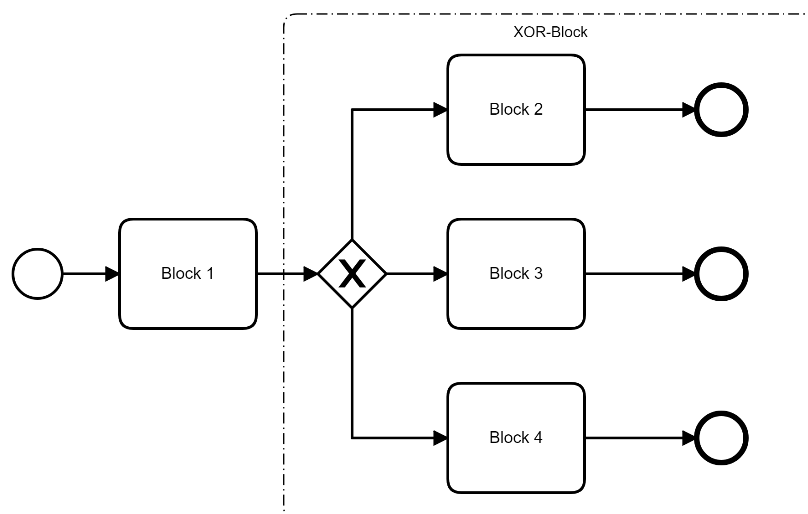

Abbildung 3.17: XOR-Block: Nicht zusammengeführt

Des Weiteren besteht die Möglichkeit, dass sich innerhalb eines exklusiven Blocks ein weiterer exklusiver Block befindet, welcher nur teilweise mit dem übergeordneten XOR-Block zusammengeführt wird. Ein Beispiel für ein solches Muster kann in Abbildung 3.18 betrachtet werden. Nach den Formeln und Methoden von Jung et al. [2], welche im vorherigen Kapitel vorgestellt wurden, kann dieses Muster nicht berechnet werden.

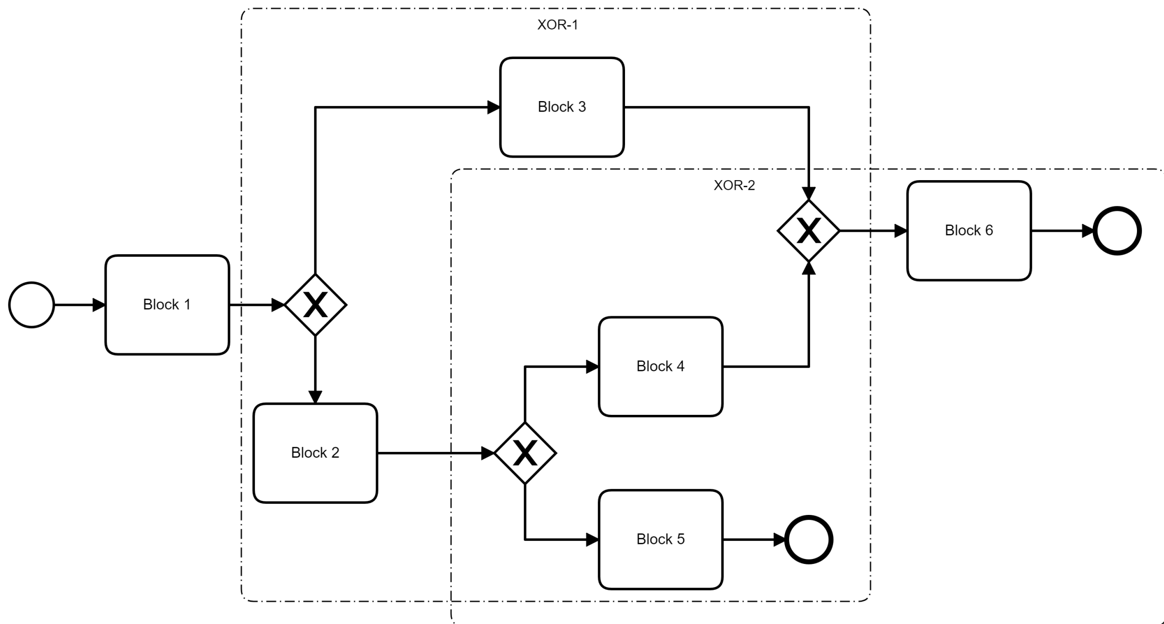

Abbildung 3.18: XOR-Block: Teilweise zusammengeführt

Die Berechnungsmethode von Jung et al. [2] sieht vor, dass die Unsicherheit von logischen Blöcken berechnet wird, um jene Blöcke anschließend durch einen einzelnen Block zu ersetzen, welcher die berechnete Unsicherheit enthält. Dies ist notwendig, um die Unsicherheit von verschachtelten logischen Blöcken zu berechnen. Wendet man diese Methode jedoch auf den Geschäftsprozess aus Abbildung 3.18 an, so müsste man zunächst den Block XOR-2 auflösen, da dieser sich innerhalb des Blocks XOR-1 befindet. Geht man nach diesem Schema vor, so entsteht ein Prozess, wie er in Abbildung 3.19 dargestellt ist. Hierbei fällt auf, dass Task 6 aus Abbildung 3.18 bereits mit dem XOR-2 Block mit aufgelöst wurde. Task 6 hat im ursprünglichen Geschäftsprozess jedoch auch ein serielles Verhältnis zum Block XOR-1, welches durch die Berechnung des Blocks XOR-2 zerstört wurde.

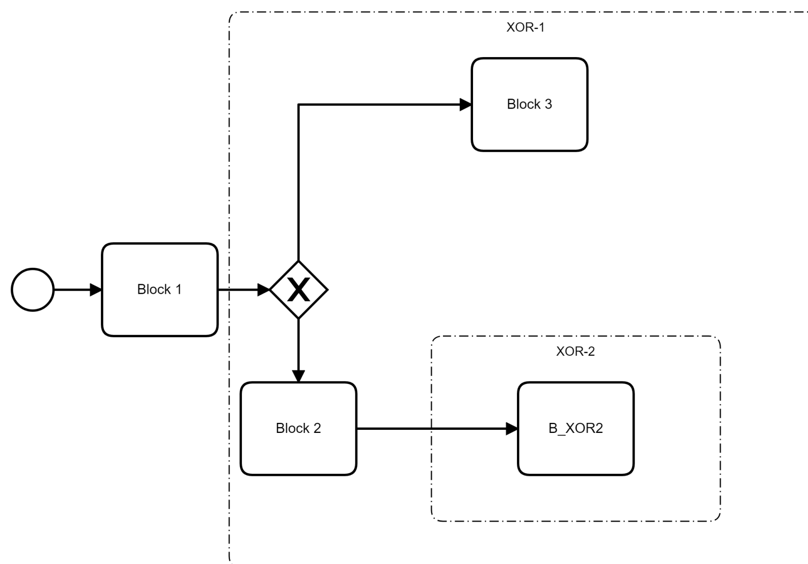

Abbildung 3.19: XOR-2 aufgelöst

Geht man anders vor und definiert die zu berechnenden Blöcke wie in 3.20 dargestellt, so stößt man auf eine andere Problematik. Das Problem wird ersichtlich, wenn man den Prozess

nach dem Schema von Jung et al. [2], welches in Kapitel 2.2 vorgestellt wurde, auflöst.

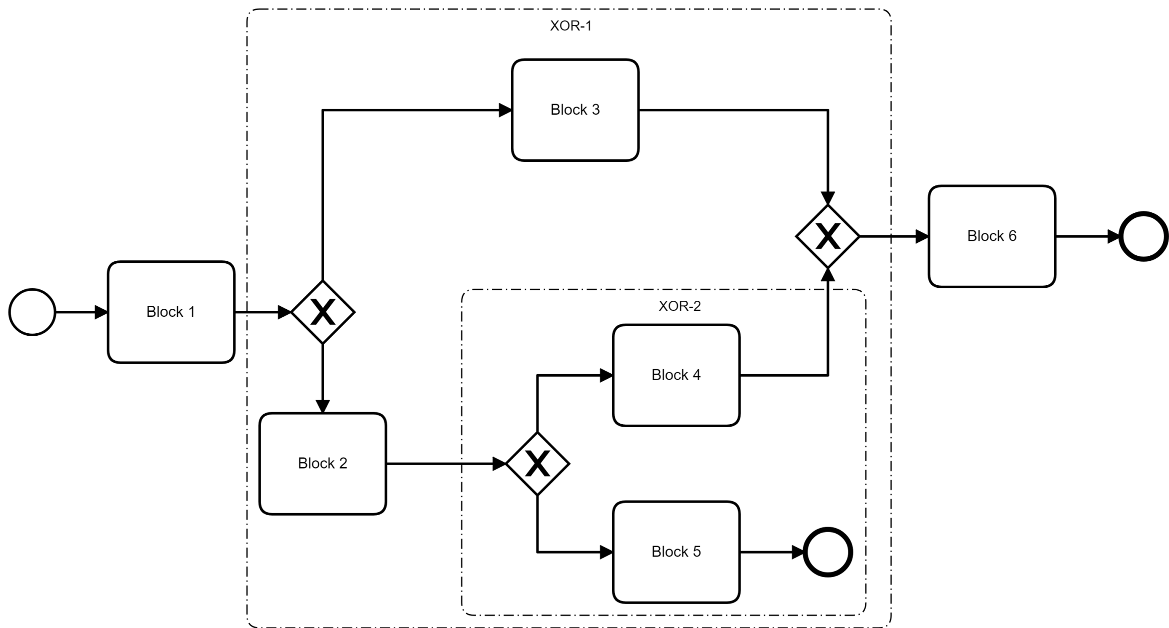

Abbildung 3.20: XOR-Block: Teilweise zusammengeführt alternativ

Hierfür wird zunächst der Block XOR-2 berechnet, woraufhin der Prozess wie in Abbildung 3.21 dargestellt vereinfacht wird.

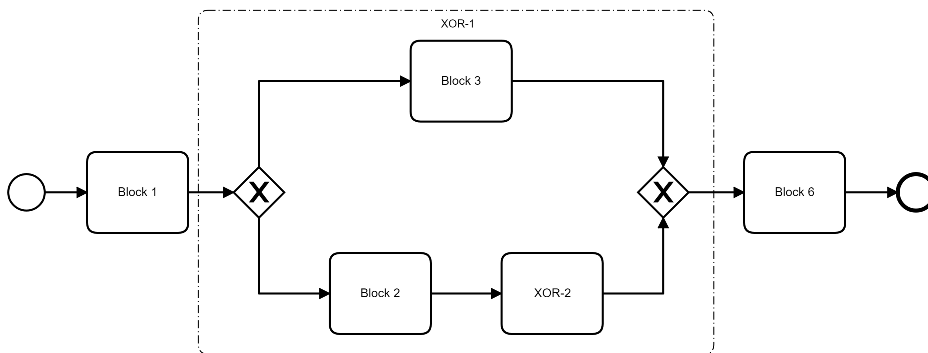

Abbildung 3.21: XOR-Block: Teilweise zusammengeführt alternativ XOR-2 aufgelöst

Hierbei fällt auf, dass der Block XOR-1 ein serielles Verhältnis zu Block 6 aufweist. Serielle Blöcke werden nach der Methodik von Jung et al. [2] mit folgender Formel berechnet: $U(B_{SEQ}) = \sum_{g=1}^{N} U(B_g)$. Somit wird die Unsicherheit von Block 6 bei dieser Vorgehensweise unabhängig von den Ausführungswahrscheinlichkeiten der jeweiligen Verzweigungen der vorangehenden exklusiven Gateways mit vollem Umfang zur Unsicherheit des gesamten Prozesses addiert. Wie bereits in Kapitel 2.2.4 angemerkt wurde, ist es jedoch ein fundamentaler Teil der Berechnungsmethode von Jung et al., dass die Unsicherheit von Blöcken zunächst mit deren Ausführungswahrscheinlichkeit multipliziert wird, bevor sie zur gesamt Unsicherheit des Prozesses addiert wird. Wenn man nun erneut den Prozess aus Abbildung 3.20 betrachtet, so fällt auf, dass die Möglichkeit besteht, dass Block 6 nicht ausgeführt wird. Man kann dies sogar auf einen Extremfall ausweiten, welcher in Abbildung 3.22 dargestellt ist. In diesem Extremfall wird Block 6 mit einer Wahrscheinlichkeit von null Prozent ausgeführt. Aufgrund dieser Tatsache erweist es sich nicht als sinnvoll, die Unsicherheit von Block 6 unabhängig

von den Ausführungswahrscheinlichkeiten der jeweiligen Instanzen zur Gesamtunsicherheit zu addieren. Somit kann die Unsicherheit des Geschäftsprozesses aus Abbildung 3.18 nicht korrekt nach den Formeln aus Kapitel 2.2 berechnet werden. Eine Berechnung der Unsicherheit dieses Geschäftsprozesses nach der Methode von Jung et al. [2] ist nur dann möglich, wenn eine neue Formel, welche speziell für diesen Fall entwickelt wurde, zur Anwendung kommt.

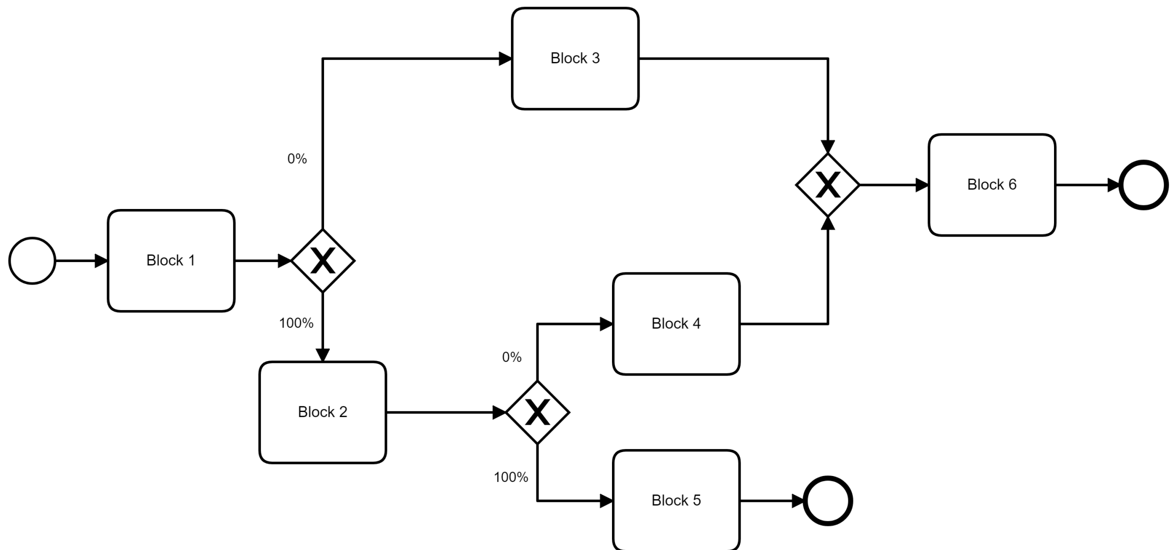

Abbildung 3.22: XOR-Block: Teilweise zusammengeführt Extremfall

Ein weiterer Sonderfall entsteht, wenn mehr als zwei Verzweigungen vorhanden sind, welche jedoch nicht im selben Punkt zusammengeführt werden. Ein Beispiel hierfür kann in Abbildung 3.23 betrachtet werden.

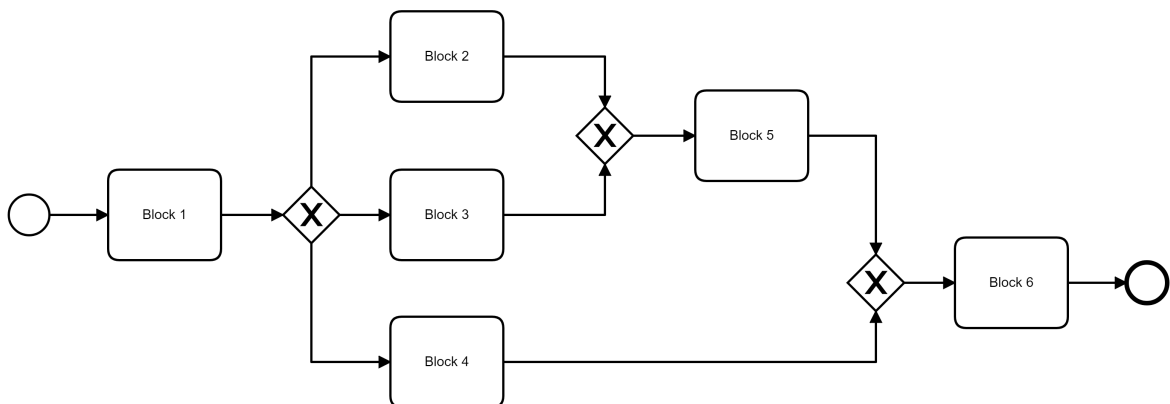

Abbildung 3.23: XOR-Block: Verschachtelt

Des Weiteren ist es auch möglich, dass nur ein Teil der Verzweigungen zusammengeführt wird, wie es in Abbildung 3.24 dargestellt ist.

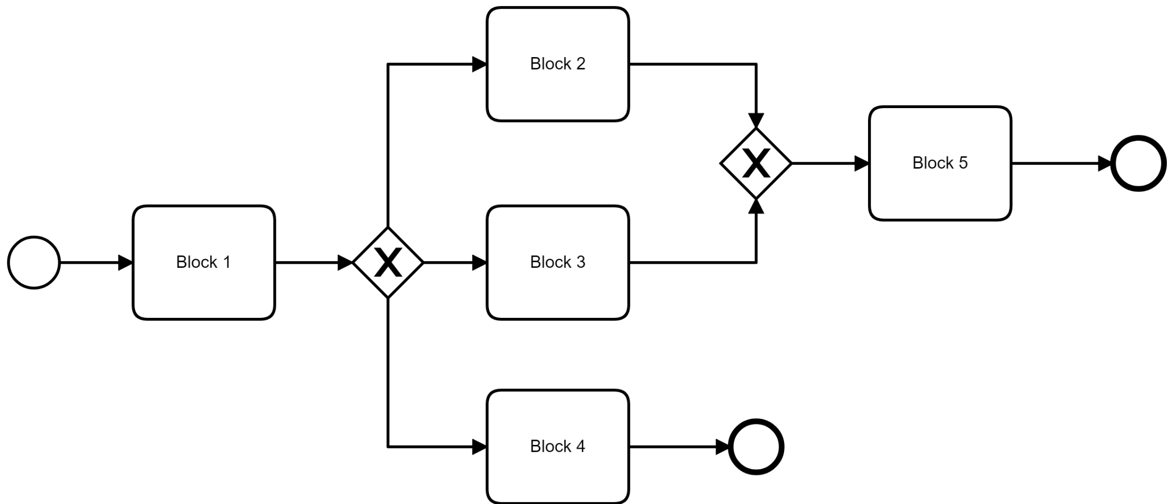

Abbildung 3.24: XOR-Block: Verschachtelt und nicht Zusammengeführt

Es ist auch möglich, dass sich Verzweigungen überkreuzen, wie es im Geschäftsprozess aus Abbildung 3.25 der Fall ist.

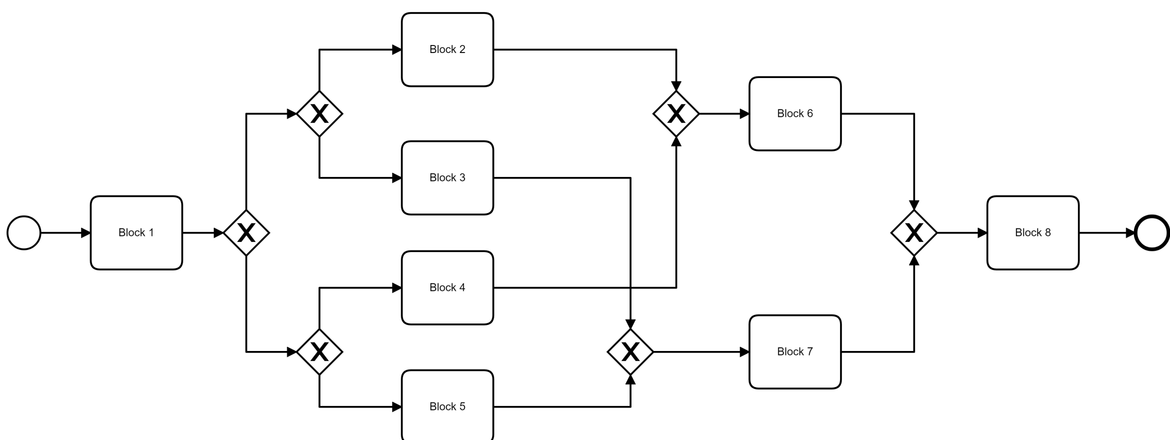

Abbildung 3.25: XOR-Block: Überkreuzungen

3.2.2 Schlussfolgerung

Die Methode zur Berechnung der Unsicherheit von Prozessen von Jung et al. [2] sieht vor, dass der Geschäftsprozess schrittweise vereinfacht wird, in dem in einem iterativen Prozess vordefinierte Blöcke identifiziert und berechnet werden, um sie durch ein einzelnes Element, welches einen berechneten Wert enthält, ersetzt. Dieser iterative Prozess wird dabei so lange wiederholt, bis der Prozess so weit vereinfacht wurde, sodass die Unsicherheit des gesamten Prozesses berechnet werden kann. In ihrer Arbeit stellen Jung et al. [2] Formeln für fünf Muster (Sequenzen, parallele Blöcke, exklusive Blöcke, inklusive Blöcke und Schleifen) zur Verfügung. In diesem Kapitel wurde jedoch bereits darauf aufmerksam gemacht, dass wesentlich mehr Muster als diese fünf in modellierten Geschäftsprozessen angetroffen werden können.

Auf vergleichbare Hürden ist man bereits in vergangenen Arbeiten gestoßen, in denen die Möglichkeit der Übersetzung von BPMN (Business Process Model and Notation) Prozessen in BPEL (Business Process Execution Language) Prozesse untersucht wurde. Recker und Mendling [7] erläutern in ihrer Arbeit, dass unter anderem die unterschiedlichen Paradigmen der Sprachen BPMN und BPEL zu Schwierigkeiten in der Übersetzung eines BPMN Prozesses

in einen BPEL Prozess führen kann. Dies liegt daran, dass BPMN eine graphenorientierte Sprache ist, während es sich bei BPEL um eine blockorientierte Sprache handelt. Recker und Mendling schreiben des Weiteren, dass Herausforderungen in der Übersetzung dann auftreten können, wenn es sich bei dem BPMN Prozess um einen unstrukturierten Prozess handelt. Als strukturierten Prozess bezeichnen Recker und Mendling einen Prozess, bei welchem unter anderem zu einem jedem verzweigenden Gateway ein zusammenführendes Gateway vom selben Typ gehört. Hier können gewisse Parallelen zu den Erkenntnissen gezogen werden, welche in Kapitel 3.2.1 gewonnen wurden. Wie anhand von einigen Beispielen erläutert, wird die Unterteilung von BPMN Prozessen in Blöcken dann erschwert, wenn es sich um einen "unstrukturierten" Prozess handelt, in welchem nicht ein jedes verzweigende Gateway über ein eigenes passendes Gegenstück verfügt.

Eine weitere Herausforderung in der Automatisierung eines Algorithmus, welcher einen graphenorientierten Prozess in einen blockorientierten Prozess übersetzen soll, ist die Identifizierung von blockartigen Strukturen innerhalb des graphenorientierten Prozesses. Auf diese Erkenntnis sind auch Nguyen et al. [8] in ihrer Arbeit gestoßen, in welcher sie ebenfalls die Übersetzung von BPMN Prozessen in BPEL Prozesse untersuchten. Hierbei stellen Nguyen et al. ebenfalls fest, dass die Identifizierung von gut strukturierten Blöcken einfacher ist als die Identifizierung von schlecht strukturierten Blöcken. Soll ein solcher Algorithmus jedoch auch in der Praxis tauglich sein, so muss er dazu in der Lage sein, jeden beliebigen Prozess in Blöcke zu unterteilen. Ist dies nicht der Fall, so kann der Algorithmus nicht auf alle Prozesse angewandt werden.

Dies bedeutet nicht, dass die Methode von Jung et al. nicht auf alle Geschäftsprozesse anwendbar ist. Es bedeutet lediglich, dass für ein jedes neues Muster, welches mit keinem der bisher definierten Muster übereinstimmt, eine neue Formel entwickelt werden muss. Für eine automatisierte Berechnung der Unsicherheit von Geschäftsprozessen ist dies in der Praxis jedoch ungeeignet, da es vermutlich nicht möglich ist, eine Anwendung zu entwickeln, welche alle möglichen Muster identifizieren und auch korrekt berechnen kann. Dieses Problem entsteht dadurch, dass Blöcke, welche aus mehreren Elementen bestehen, identifiziert und berechnet werden. Solange die Berechnungsmethode darauf beruht, die Unsicherheit von Blöcken zu berechnen, wird dieses Problem bestehen bleiben. Als logische Konsequenz hieraus stellt sich nun die Frage, ob eine Berechnungsmethode, in welcher die Unsicherheit von einzelnen Elementen berechnet wird, sich besser für eine automatisierte Berechnung eignet.

3.3 Berechnung der Unsicherheit von einzelnen Elementen

Wie im vorherigen Kapitel erwähnt, eignet sich eine Berechnungsmethode, in der die Unsicherheit von einzelnen Elementen berechnet wird, womöglich besser für die automatisierte Berechnung der Unsicherheit von Geschäftsprozessen als eine Berechnungsmethode, in welcher die Unsicherheit von vordefinierten Blöcken berechnet wird. In diesem Kapitel wird eine Methode vorgeschlagen, welche auf dieser Überlegung beruht.

Werden nur einzelne Elemente betrachtet, so stehen folgende Informationen für die Berechnung zur Verfügung:

- Typ des Elements (Task, exklusives Gateway, inklusives Gateway, etc.)
- Anzahl der eingehenden Sequenzflüsse
- Anzahl der ausgehenden Sequenzflüsse
- Ausführungswahrscheinlichkeiten der ausgehenden Sequenzflüsse

- Anliegende Elemente

Aus diesen Informationen lassen sich weitere Informationen ableiten: die möglichen Sequenzflüsse innerhalb des Geschäftsprozesses sowie die Wahrscheinlichkeiten, mit welchen die jeweiligen Elemente des Geschäftsprozesses ausgeführt werden.

3.3.1 Notation

Um für mehr Klarheit bei der Erläuterung der Berechnungsmethode in den folgenden Kapiteln zu sorgen, soll eine Notation für bestimmte Variablen definiert werden. Die zu definierenden Variablen lauten wie folgt:

- G: Der betrachtete Geschäftsprozess.
- E: Ein Element innerhalb eines Geschäftsprozesses. Ein Element ist eine ausführbahre Einheit, wie etwa ein Task oder ein Gateway. Sequenzflüsse gehören dabei nicht zu den Elementen.
- r: Die relative Ausführungswahrscheinlichkeit eines ausgehenden Zeiges eines Elements. Betrachtet man etwa ein exklusives Gateway, so verfügen die ausgehenden Zweige über eine relative Ausführungswahrscheinlichkeit. Die relative Ausführungswahrscheinlichkeit r eines ausgehenden Zweiges eines Elements E beschreibt die Wahrscheinlichkeit, mit welcher der jeweilige Zweig ausgeführt wird, nachdem das Element E ausgeführt wurde.
- S: Ein Sequenzfluss innerhalb eines Geschäftsprozesses.
- $P(E)$: Die Ausführungswahrscheinlichkeit eines Elements. Dieser Wert beschreibt die Wahrscheinlichkeit, mit welcher ein Element E ausgeführt wird.
- $P(S)$: Die Ausführungswahrscheinlichkeit eines Sequenzflusses. Dieser Wert beschreibt die Wahrscheinlichkeit, mit welcher ein Sequenzfluss S ausgeführt wird.
- $U(G)$: Die Unsicherheit des gesamten Geschäftsprozesses.
- $U_r(E)$: Die relative Unsicherheit eines Elements E. Dieser Wert beschreibt die Unsicherheit eines Elements bei isolierter Betrachtung, ohne Berücksichtigung der Ausführungswahrscheinlichkeit $P(E)$ des Elements.
- $U_a(E)$: Die absolute Unsicherheit eines Elements E. Dieser Wert beschreibt die Unsicherheit eines Elements unter der Berücksichtigung der Ausführungswahrscheinlichkeit $P(E)$ des Elements.

3.3.2 Exklusive Verzweigungen

Zunächst soll die Methode anhand von exklusiven Gateways vorgestellt werden. Die Methode kann anhand eines Beispiels demonstriert werden, welches in Abbildung 3.26 dargestellt wird. Um für eine erhöhte Übersichtlichkeit zu sorgen, wurden die Tasks in diesem Prozess bewusst ausgelassen. Es befinden sich acht exklusive Gateways in diesem Prozess. Vier davon sind teilende Gateways und die verbleibenden vier sind zusammenführende Gateways. Ein jedes teilende Gateway verfügt über Wahrscheinlichkeiten, mit welchen die Entscheidung über die Weiterführung des Sequenzflusses über einen bestimmten ausgehenden Zweig getroffen werden. Diese Wahrscheinlichkeiten werden durch die variablen r dargestellt. Des Weiteren verfügen Sequenzflüsse über eine Wahrscheinlichkeit, mit welcher sie beim Durchlaufen des gesamten Geschäftsprozesses ausgeführt werden.

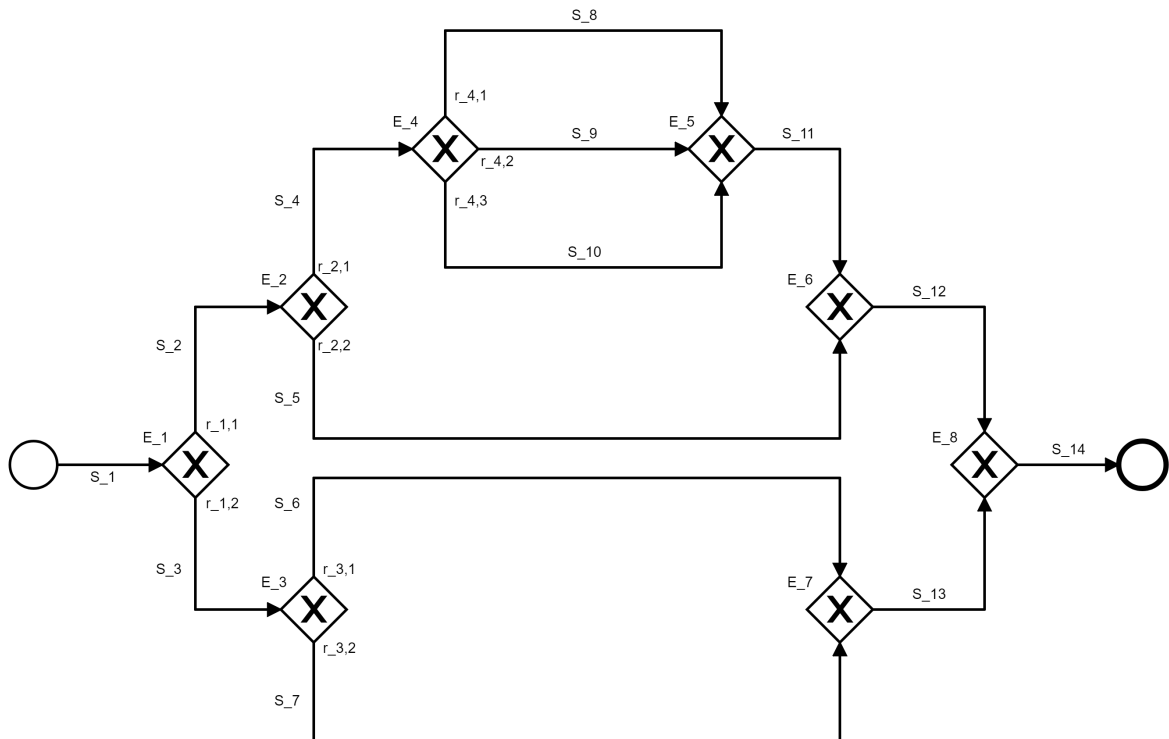

Abbildung 3.26: Beispiel: Berechnung von einzelnen Elementen

Somit kann man die Ausführungswahrscheinlichkeit eines jeden einzelnen Elements in diesem Prozess ermitteln, insofern die Werte für r bekannt sind. Für dieses Beispiel werden folgende Werte angenommen: $r_{1,1} = 0.4$, $r_{1,2} = 0.6$, $r_{2,1} = 0.7$, $r_{2,2} = 0.3$, $r_{3,1} = 0.8$, $r_{3,2} = 0.2$, $r_{4,1} = 0.2$, $r_{4,2} = 0.3$, $r_{4,3} = 0.5$

Die Ausführungswahrscheinlichkeit des ersten Sequenzflusses beträgt automatisch einhundert Prozent: $P(S_1) = 1$

Bei jedem teilenden Gateway müssen die absoluten Wahrscheinlichkeiten $P(S)$ der ausgehenden Sequenzflüsse berechnet werden, indem die Ausführungswahrscheinlichkeit des Gateways $P(E)$ (entspricht der absoluten Wahrscheinlichkeit des eingehenden Sequenzflusses) mit den relativen Wahrscheinlichkeiten r für die jeweiligen ausgehenden Zweige multipliziert werden. Diese Vorgehensweise beruht auf dem Multiplikationssatz von Kolmogoroff [9, S. 6]. Bei zusammenführenden Gateways werden die eingehenden absoluten Wahrscheinlichkeiten der jeweiligen Zweige nach dem Additionssatz von Kolmogoroff [9, S. 6] addiert, um die absolute Wahrscheinlichkeit des ausgehenden Zweigs zu berechnen. Die Berechnung der absoluten Wahrscheinlichkeiten lautet demnach wie folgt:

$$
\begin{aligned}
P(S_1) &= 1 \\
P(S_2) &= P(S_1) \times r_{1,1} = 1 \times 0.4 = 0.4 \\
P(S_3) &= P(S_1) \times r_{1,2} = 1 \times 0.6 = 0.6 \\
P(S_4) &= P(S_2) \times r_{2,1} = 0.4 \times 0.7 = 0.28 \\
P(S_5) &= P(S_2) \times r_{2,2} = 0.4 \times 0.3 = 0.12 \\
P(S_6) &= P(S_3) \times r_{3,1} = 0.6 \times 0.8 = 0.48 \\
P(S_7) &= P(S_3) \times r_{3,2} = 0.6 \times 0.2 = 0.12 \\
P(S_8) &= P(S_4) \times r_{4,1} = 0.28 \times 0.2 = 0.056 \\
P(S_9) &= P(S_4) \times r_{4,2} = 0.28 \times 0.3 = 0.084 \\
P(S_{10}) &= P(S_4) \times r_{4,3} = 0.28 \times 0.5 = 0.14 \\
P(S_{11}) &= P(S_8) + P(S_9) + P(S_{10}) = 0.056 + 0.084 + 0.14 = 0.28 \\
P(S_{12}) &= P(S_{11}) + P(S_5) = 0.28 + 0.12 = 0.4 \\
P(S_{13}) &= P(S_6) + P(S_7) = 0.48 + 0.12 = 0.6 \\
P(S_{14}) &= P(S_{12}) + P(S_{13}) = 0.4 + 0.6 = 1
\end{aligned}
\tag{3.10}
$$

Diese Wahrscheinlichkeiten können nun verwendet werden, um die Unsicherheit von Elementen zu gewichten. Diese Gewichtung geschieht in der Methode von Jung et al. [2] (siehe auch Kapitel 2.2) implizit über die Formeln sowie dem Ansatz, verschachtelte Blöcke von innen heraus zu berechnen.

Zur Berechnung der relativen Unsicherheit von exklusiven Gateways $U_r(E_{XOR})$ kann eine abgewandelte Version der Formel von Jung et al. [2] (siehe auch Kapitel 2.2.4) verwendet werden. Es wird nur der erste Teil der Formel benötigt, in welchem die Unsicherheit des Gateways selbst berechnet wird. Somit lautet die Formel für die Berechnung der relativen Unsicherheit eines exklusiven Gateways:

$$U_r(E_{XOR}) = -\sum_{n=1}^{N} r_n \log_2 r_n \tag{3.11}$$

Um die absolute Unsicherheit eines exklusiven Gateways zu berechnen, muss die relative Unsicherheit des Gateways noch mit der Ausführungswahrscheinlichkeit des Gateways multipliziert werden:

$$U_a(E_{XOR}) = P(E_{XOR}) \times -\sum_{n=1}^{N} r_n \log_2 r_n \tag{3.12}$$

Die Unsicherheit eines gesamten Geschäftsprozesses ergibt sich aus der Summe der absoluten Unsicherheiten aller Elemente des Geschäftsprozesses. Unsicherheit kann nur von teilenden Elementen erzeugt werden. Aus diesem Grund wird in diesem Beispiel nur die Unsicherheit der teilenden exklusiven Gateways berechnet. Somit berechnet sich die Unsicherheit des Geschäftsprozesses aus Abbildung 3.26 wie folgt:

$$
\begin{aligned}
U(G) = \sum_{i}^{N} U_a(E_i) \\
&= 1 \times -(0.4 \log_2 0.4 + 0.6 \log_2 0.6) \\
&+ 0.4 \times -(0.7 \log_2 0.7 + 0.3 \log_2 0.3) \\
&+ 0.6 \times -(0.8 \log_2 0.8 + 0.2 \log_2 0.2) \\
&+ 0.28 \times -(0.2 \log_2 0.2 + 0.3 \log_2 0.3 + 0.5 \log_2 0.5) \\
&= 2.17255
\end{aligned} \tag{3.13}
$$

Berechnet man die Unsicherheit des Geschäftsprozesses stattdessen nach der Methode von Jung et al. [2], welche in Kapitel 2.2 behandelt wurde, so geht man wie folgt vor:

Im ersten Schritt werden zwei exklusive Blöcke identifiziert, welche berechnet werden können.

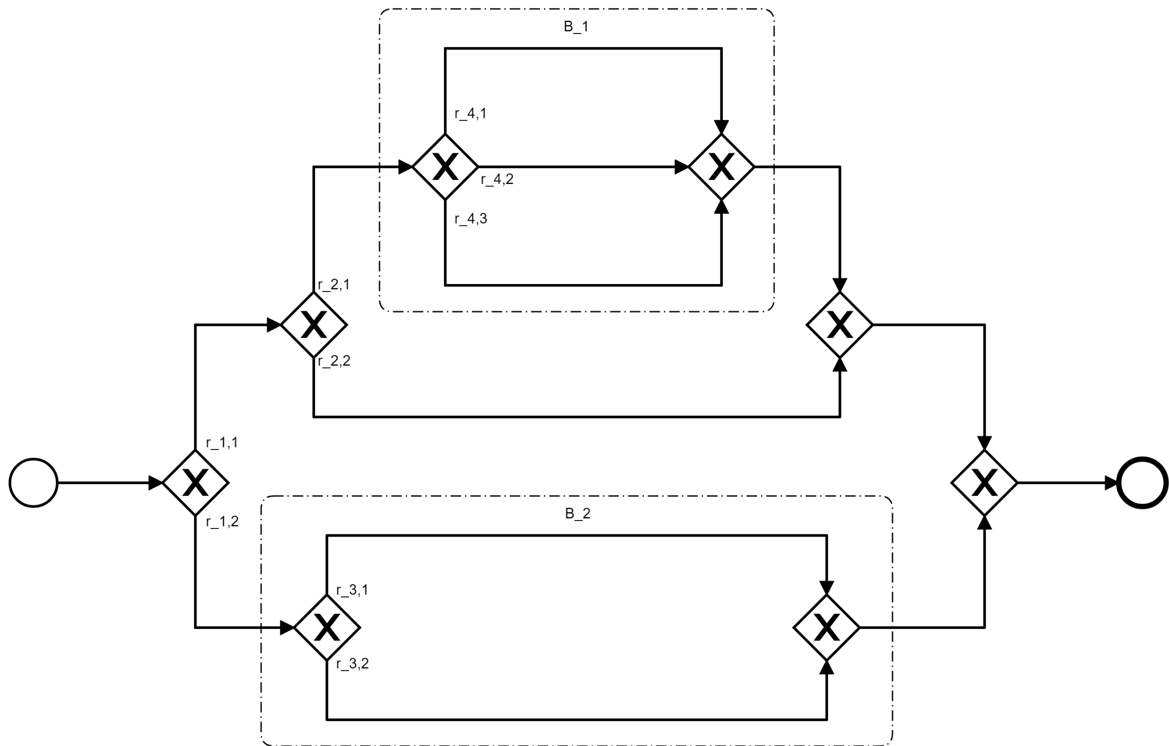

Abbildung 3.27: Beispiel: erster Schritt

Die Berechnung der beiden Blöcke:

$$
\begin{aligned}
U(B_1) = -\sum_{g=1}^{N} r_{0,g} \log_2 r_{0,g} + \sum_{g=1}^{N} r_{0,g} \times U(B_g) \\
&= -(0.2 \log_2 0.2 + 0.3 \log_2 0.3 + 0.5 \log_2 0.5) \\
&+ 0.2 \times 0 + 0.3 \times 0 + 0.5 \times 0 \\
&= 1.48547
\end{aligned} \tag{3.14}
$$

$$
\begin{aligned}
U(B_2) &= -\sum_{g=1}^{N} r_{0,g} \log_2 r_{0,g} + \sum_{g=1}^{N} r_{0,g} \times U(B_g) \\
&= -(0.8 \log_2 0.8 + 0.2 \log_2 0.2) + 0.8 \times 0 + 0.2 \times 0 \\
&= 0.72192
\end{aligned}
\tag{3.15}
$$

Im zweiten Schritt werden die Blöcke B_1 und B_2 vereinfacht. Des Weiteren wird ein neuer zu berechnender Block identifiziert. Der vereinfachte Prozess kann in Abbildung 3.28 betrachtet werden.

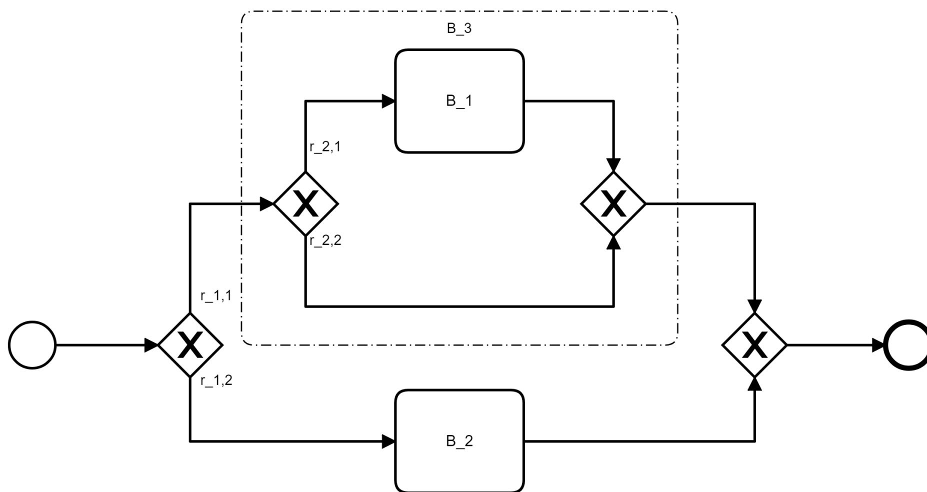

Abbildung 3.28: Beispiel: zweiter Schritt

B_3 wird wie folgt berechnet:

$$
\begin{aligned}
U(B_3) &= -\sum_{g=1}^{N} r_{0,g} \log_2 r_{0,g} + \sum_{g=1}^{N} r_{0,g} \times U(B_g) \\
&= -(0.7 \log_2 0.7 + 0.3 \log_2 0.3) + 0.7 \times 1.48547 + 0.3 \times 0 \\
&= 1.92111
\end{aligned}
\tag{3.16}
$$

Im letzten Schritt wird der Block B_3 vereinfacht und es wird ein exklusiver Block identifiziert, wie es in Abbildung 3.29 dargestellt wurde.

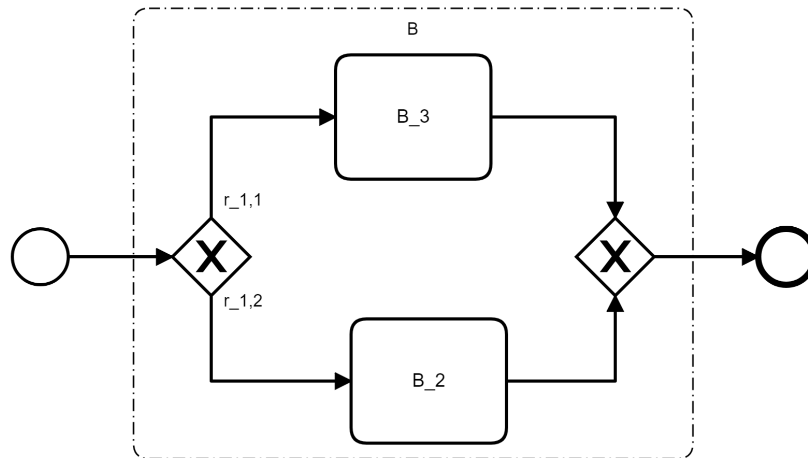

Abbildung 3.29: Beispiel: dritter Schritt

Die Unsicherheit des Prozesses wird somit wie folgt berechnet:

$$
\begin{aligned}
U(B) &= -\sum_{g=1}^{N} r_{0,g} \log_2 r_{0,g} + \sum_{g=1}^{N} r_{0,g} \times U(B_g) \\
&= -(0.4 \log_2 0.4 + 0.6 \log_2 0.6) + 0.4 \times 1.92111 + 0.6 \times 0.72192 \\
&= 2.17254
\end{aligned}
\tag{3.17}
$$

Wie hier gezeigt wird, führen beide Methoden zum selben Ergebnis. Jedoch ist die Methode, in welcher die Unsicherheit von einzelnen Elementen berechnet wird, nicht auf bestimmte Block-Muster angewiesen.

3.3.3 Parallele Verzweigungen

Wird diese Methode auf Geschäftsprozesse angewandt, in welchen sich parallele Verzweigungen befinden, so muss dies bei der Ermittlung der absoluten Ausführungswahrscheinlichkeiten der jeweiligen Elemente berücksichtigt werden. Bei der Ausführung eines parallelen Gateways werden stets alle ausgehenden Zweige ausgeführt [4, S. 294]. Somit entspricht die Ausführungswahrscheinlichkeit eines jeden ausgehenden Sequenzflusses eines parallelen Gateways denselben Wert wie die Ausführungswahrscheinlichkeit des Gateways selbst. Bei der Zusammenführung von parallelen Verzweigungen ist zu berücksichtigen, dass die Ausführungswahrscheinlichkeit des zusammengeführten Sequenzflusses ebenfalls der Ausführungswahrscheinlichkeit des teilenden parallelen Gateways entspricht. Die Ausführungswahrscheinlichkeiten der einzelnen Zweige werden also bei der Zusammenführung nicht addiert, wie es bei den exklusiven Verzweigungen der Fall war. Dies lässt sich am besten anhand eines Beispiels (siehe Abbildung 3.30) demonstrieren. Für eine bessere Übersicht, wurde auf die Modellierung von Tasks erneut verzichtet.

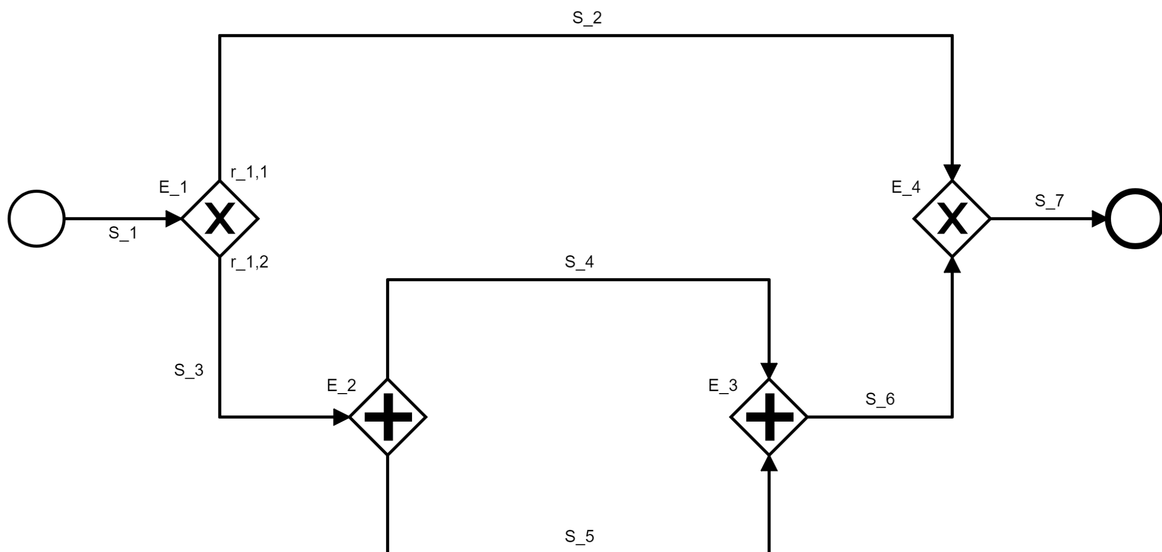

Abbildung 3.30: Beispiel: Berechnung von einzelnen parallelen Elementen

Für dieses Beispiel werden die folgenden Werte für die relativen Wahrscheinlichkeiten r des exklusiven Verzweigungen angenommen: $r_{1,1} = 0.4$, $r_{1,2} = 0.6$ Für die Ausführungswahrscheinlichkeit der ersten Sequenz gilt: $P(S_1) = 1$

Wie bereits in Kapitel 3.3.2 erläutert wurde, wird die absolute Ausführungswahrscheinlichkeit von ausgehenden Sequenzflüssen eines teilenden exklusiven Gateways berechnet, indem die absolute Ausführungswahrscheinlichkeit des eingehenden Sequenzflusses mit der jeweiligen relativen Ausführungswahrscheinlichkeit des ausgehenden Sequenzflusses multipliziert wird.

$$\begin{aligned} &P(S_1) = 1 \\ &P(S_2) = P(S_1) \times r_{1,1} = 1 \times 0.4 = 0.4 \\ &P(S_3) = P(S_1) \times r_{1,2} = 1 \times 0.6 = 0.6 \end{aligned} \tag{3.18}$$

Wie vorhin bereits erwähnt, beeinflusst eine parallele Verzweigung die Ausführungswahrscheinlichkeit von Sequenzflüssen nicht. Somit entsprechen die Werte der absoluten Ausführungswahrscheinlichkeiten der ausgehenden Sequenzflüsse des parallelen Gateways E_2 dem Wert der absoluten Ausführungswahrscheinlichkeit des eingehenden Sequenzflusses von E_2.

$$\begin{aligned} &P(S_4) = P(S_3) = 0.6 \\ &P(S_5) = P(S_3) = 0.6 \end{aligned} \tag{3.19}$$

Bei der Zusammenführung von parallelen Verzweigungen gilt dieselbe Regel. Die absolute Ausführungswahrscheinlichkeit von Sequenzflüssen werden nicht durch parallele Verzweigungen beeinflusst. Bei der Zusammenführung einer parallelen Verzweigung enspricht $P(S)$ des ausgehenden Sequenzflusses den Werten $P(S)$ der eingehenden Sequenzflüsse.

$$P(S_6) = P(S_4) = P(S_5) = 0.6 \tag{3.20}$$

In Kapitel 3.3.2 wurde erläutert, dass bei der Zusammenführung von exklusiven Verzweigungen die absoluten Wahrscheinlichkeiten der eingehenden Sequenzflüsse addiert werden, um die absolute Wahrscheinlichkeit des ausgehenden Sequenzflusses zu berechnen.

$$P(S_7) = P(S_2) + P(S_6) = 0.4 + 0.6 = 1 \tag{3.21}$$

3.3.4 Inklusive Verzweigungen

Inklusive Verzweigungen weisen ein ähnliches Verhalten auf wie exklusive Verzweigungen. Jedoch besteht ein essenzieller Unterschied zwischen den beiden. Bei exklusiven Verzweigungen wird stets nur einer der verzweigten Sequenzflüsse ausgeführt [4, S. 290]. Die Ausführung eines verzweigten Sequenzflusses ist somit von der Ausführung der anderen verzweigten Sequenzflüsse abhängig. Bei einer inklusiven Verzweigung hingegen wird über die Ausführung eines jeden ausgehenden Sequenzflusses individuell bestimmt [4, S. 292]. Um die absolute Ausführungswahrscheinlichkeit eines ausgehenden Sequenzflusses eines teilenden inklusiven Gateways zu berechnen, geht man identisch vor wie bei teilenden exklusiven Gateways. Um die absolute Ausführungswahrscheinlichkeit eines inklusiven Zweigs zu berechnen, muss also die absolute Ausführungswahrscheinlichkeit des eingehenden Sequenzflusses des teilenden Gateways mit der jeweiligen relativen Ausführungswahrscheinlichkeit des verzweigten Sequenzflusses multipliziert werden. Die Zusammenführung von inklusiven Verzweigungen unterscheidet sich jedoch von der Zusammenführung von exklusiven Verzweigungen. Aufgrund der Tatsache, dass inklusive Zweige unabhängig voneinander zu betrachten sind, ist es möglich, dass kein Zweig, ein Zweig, mehrere Zweige oder auch alle Zweige ausgeführt werden. Es besteht somit die Möglichkeit, dass eine inklusive Verzweigung aus drei Sequenzflüssen besteht, welche jeweils eine Ausführungswahrscheinlichkeit von beispielsweise siebzig Prozent aufweisen und dennoch nach Ausführung des inklusiven Gateways keine dieser Sequenzflüsse ausgeführt wird. Daher kann die Ausführungswahrscheinlichkeit des ausgehenden Sequenzflusses des zusammenführenden inklusiven Gateways nicht berechnet werden, indem die Ausführungswahrscheinlichkeiten der verzweigten Sequenzflüsse addiert werden. Hierfür muss eine Methode aus der Wahrscheinlichkeitsrechnung angewandt werden. Die konkrete Vorgehensweise soll erneut anhand eines Beispiels demonstriert werden. Das Beispiel ist in Abbildung 3.31 dargestellt. Auf die Modellierung von Tasks wurde aufgrund der Übersichtlichkeit verzichtet.

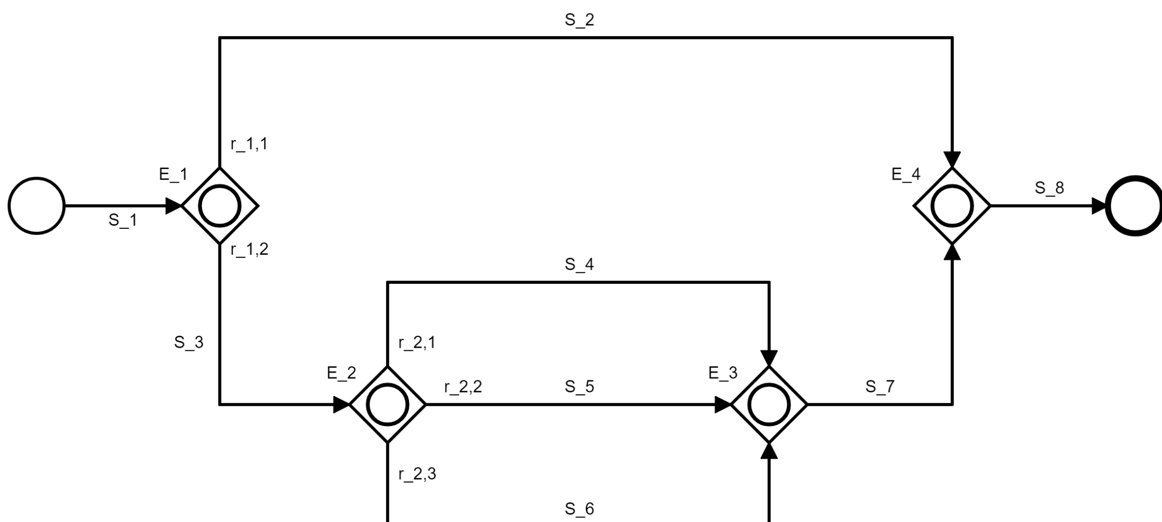

Abbildung 3.31: Beispiel: Berechnung von einzelnen Inklusiven Elementen

Für dieses Beispiel werden die folgenden Werte für die relativen Wahrscheinlichkeiten r

der inklusiven Verzweigungen angenommen: $r_{1,1} = 0.8$, $r_{1,2} = 0.7$, $r_{2,1} = 0.4$, $r_{2,2} = 0.8$, $r_{2,3} = 0.5$ Für die Ausführungswahrscheinlichkeit der ersten Sequenz gilt: $P(S_1) = 1$
Wie bereits erwähnt, wird die absolute Ausführungswahrscheinlichkeit eines inklusiven Zweigs von Element E berechnet, indem die absolute Ausführungswahrscheinlichkeit des eingehenden Sequenzflusses von Element E mit der relativen Ausführungswahrscheinlichkeit des inklusiven Zweigs multipliziert wird.

$$\begin{aligned}
&P(S_1) = 1 \\
&P(S_2) = P(S_1) \times r_{1,1} = 1 \times 0.8 = 0.8 \\
&P(S_3) = P(S_1) \times r_{1,2} = 1 \times 0.7 = 0.7 \\
&P(S_4) = P(S_3) \times r_{2,1} = 0.7 \times 0.4 = 0.28 \\
&P(S_5) = P(S_3) \times r_{2,2} = 0.7 \times 0.8 = 0.56 \\
&P(S_6) = P(S_3) \times r_{2,3} = 0.7 \times 0.5 = 0.35
\end{aligned} \tag{3.22}$$

Um die Ausführungswahrscheinlichkeit des ausgehenden Sequenzflusses eines zusammenführenden inklusiven Gateways zu berechnen, müssen Methoden aus der Wahrscheinlichkeitsrechnung angewandt werden. Grundsätzlich kann behauptet werden, dass S_7 dann ausgeführt wird, wenn einer der Zweige S_4, S_5 oder S_6 ausgeführt wurde. Es ist nun möglich, die Wahrscheinlichkeiten aller Fälle zu berechnen, in welchen zumindest einer der drei Zweige ausgeführt wird, um die Wahrscheinlichkeiten all dieser Fälle zu addieren. Einfacher ist es jedoch, die Wahrscheinlichkeit des Falles zu berechnen, in welchem keiner der drei Zweige ausgeführt wird. Ein Sequenzfluss kann nur einen von zwei möglichen Zuständen annehmen. Er kann entweder ausgeführt werden oder nicht ausgeführt werden. Somit können die Wahrscheinlichkeiten, dass die jeweiligen Sequenzflüsse nicht ausgeführt werden, unter Berücksichtigung von Kolmogoroffs Axiomen [9, S. 6] der Wahrscheinlichkeitsrechnung wie folgt berechnet werden:

$$\begin{aligned}
\neg P(S_4) = 1 - P(S_4) = 1 - 0.28 = 0.72 \\
\neg P(S_5) = 1 - P(S_5) = 1 - 0.56 = 0.44 \\
\neg P(S_6) = 1 - P(S_6) = 1 - 0.35 = 0.65
\end{aligned} \tag{3.23}$$

Soll nun die Wahrscheinlichkeit berechnet werden, dass jedes dieser Szenarien gleichzeitig Eintritt, so muss nach Kolmogoroffs Multiplikationssatz [9, S. 6] das Produkt dieser Wahrscheinlichkeiten gebildet werden:

$$\begin{aligned}
\neg P(S_7) &= (1 - P(S_4)) \times (1 - P(S_5)) \times (1 - P(S_6)) \\
&= 0.72 \times 0.44 \times 0.65 \\
&= 0.206
\end{aligned} \tag{3.24}$$

Da nun die Wahrscheinlichkeit berechnet wurde, mit welcher S_7 nicht ausgeführt wird, kann erneut auf Kolmogoroffs Axiome [9, S. 6] zurückgegriffen werden, um die Wahrscheinlichkeit zu berechnen, mit welcher S_7 ausgeführt wird:

$$\begin{aligned}
P(S_7) &= 1 - \neg P(S_7) = 1 - [(1 - P(S_4)) \times (1 - P(S_5)) \times (1 - P(S_6))] \\
&= 1 - [0.72 \times 0.44 \times 0.65] \\
&= 0.794
\end{aligned} \tag{3.25}$$

Allgemein lässt sich die absolute Ausführungswahrscheinlichkeit des ausgehenden Sequenzflusses eines inklusiven Blocks mit folgender Formel berechnen:

$$P(S) = 1 - \prod_{i=1}^{N} (1 - P(S_i)) \tag{3.26}$$

Dies ist eine bereits etablierte Methodik im Feld der Wahrscheinlichkeitsrechnung. Mithilfe dieser Formel lässt sich nun auch die absolute Ausführungswahrscheinlichkeit von S_8 berechnen:

$$\begin{aligned} P(S_8) &= 1 - [(1 - P(S_2)) \times (1 - P(S_7))] \\ &= 1 - [0.2 \times 0.206] \\ &= 0.9588 \end{aligned} \tag{3.27}$$

3.3.5 Schleifen

Schleifen sind Konstrukte, welche nicht an ein bestimmtes Element gebunden sind. Schleifen können entstehen, wenn ein Sequenzfluss zu einem vorgelagerten Element rückgeführt wird. Üblicherweise findet diese Rückführung mithilfe eines Gateways, wie beispielsweise einem exklusiven Gateway, statt. In einem Schleifen-Konstrukt können bestimmte Elemente wiederholend ausgeführt werden. Die Anzahl der Wiederholungen hängt primär davon ab, wann die Bedingung zur Ausführung des rückführenden Sequenzflusses nicht mehr erfüllt ist. Wird die maximale Anzahl der Wiederholungen nicht durch ein Limit beschränkt, kann die Schleife beliebig oft durchlaufen werden.

In Abbildung 3.33 kann eine Schleife betrachtet werden. Block 1 und Block 2 sind Teil des Schleifen-Konstrukts und können mehrmals ausgeführt werden. Des Weiteren kann festgestellt werden, dass Block 1 stets einmal mehr ausgeführt wird als Block 2.

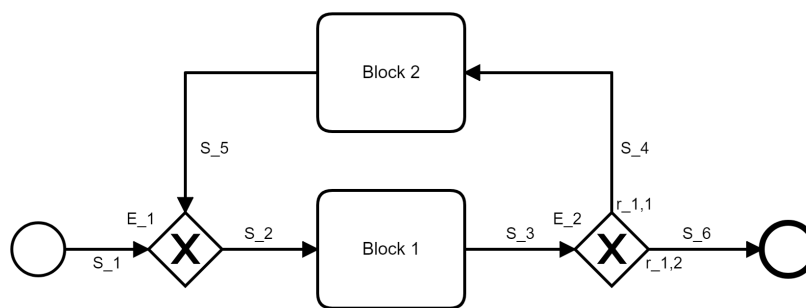

Abbildung 3.32: Schleife

Jung et al. [2] berechnen die Unsicherheit einer Schleife in ihrer Arbeit mit folgender Formel:

$$\begin{aligned} U(B_{LOOP}) = &- \left[1 - (r_{1,1})^L\right] \times \left[\frac{r_{1,1} \log_2(r_{1,1})}{r_{1,2}} + \log_2(r_{1,2})\right] \\ &+ \left[\frac{1 - (r_{1,1})^{L+1}}{r_{1,2}}\right] U(B_1) + \left[\frac{r_{1,1} - (r_{1,1})^{L+1}}{r_{1,2}}\right] U(B_2) \end{aligned} \tag{3.28}$$

Dabei entspricht $U(B_1)$ der Unsicherheit von Block 1, $U(B_2)$ entspricht der Unsicherheit von Block 2 und L entspricht dem Limit für die Anzahl der Wiederholungen der Schleife.

Bei genauerer Betrachtung kann die Formel von Jung et al. [2] in drei Teile unterteilt werden. Der erste Teil der Formel beschreibt die Unsicherheit, welche von der Schleife selbst ausgeht: $-\left[1-(r_{1,1})^L\right] \times \left[\frac{r_{1,1}\log_2(r_{1,1})}{r_{1,2}} + \log_2(r_{1,2})\right]$. Der zweite Teil der Formel beinhaltet den Faktor, mit welchem die Unsicherheit von Block 1 multipliziert werden muss: $\left[\frac{1-(r_{1,1})^{L+1}}{r_{1,2}}\right] U(B_1)$. Der dritte Teil der Formel beinhaltet den Faktor, mit welchem die Unsicherheit von Block 2 multipliziert werden muss: $\left[\frac{r_{1,1}-(r_{1,1})^{L+1}}{r_{1,2}}\right] U(B_2)$.

Diese Faktoren für Block 1 und Block 2 entsprechen jeweils dem Erwartungswert, wie oft der jeweilige Block ausgeführt wird. Dies kann anhand eines einfachen Beispiels demonstriert werden. Als Basis für dieses Beispiel wird erneut der Geschäftsprozess aus Abbildung 3.33 betrachtet. Es werden folgende Werte für die relativen Ausführungswahrscheinlichkeiten der ausgehenden Zweige des Gateways E_2 angenommen: $r_{1,1} = 0.8, r_{1,2} = 0.2$. Somit ergibt sich bei jedem Durchlauf der Schleife eine Chance von 80 %, dass die Schleife ein weiteres Mal durchlaufen wird. Des Weiteren wird angenommen, dass für die Anzahl der Iterationen der Schleife kein Limit vorhanden ist. Dies wird mit $\lim_{L\to\infty}$ abgebildet. Berechnet man nun die Faktoren, welche mit den Unsicherheiten der jeweiligen Blöcke multipliziert werden sollen, so erhält man die Werte $\lim_{L\to\infty} \frac{1-0.8^{L+1}}{0.2} = 5$ und $\lim_{L\to\infty} \frac{0.8-0.8^{L+1}}{0.2} = 4$.

Bei einer Rückführungswahrscheinlichkeit von 80 % kann man erwarten, dass die Schleife in einem durchschnittlichen Prozessablauf viermal rückgeführt wird. Bei der fünften Abfrage der Rückführbedingung wird die Schleife im Durchschnitt beendet. Bei einem durchschnittlichen Prozessablauf wird also Block 1 fünfmal und Block 2 viermal ausgeführt. Dies entspricht ebenfalls den berechneten Faktoren, welche mithilfe des entsprechenden Teils der Formel von Jung et al. berechnet wurden.

Um die Formel von Jung et al. für die Methode zur Berechnung der Unsicherheit von einzelnen Elementen zu verwenden, müssen die Teile der Formel entsprechend verwendet werden. Wie in Abbildung 3.33 betrachtet werden kann, beginnt ein Schleifen-Konstrukt mit einem zusammenführenden Element, wie etwa einem exklusiven Gateway. Um die Ausführungswahrscheinlichkeit des ausgehenden Sequenzflusses des zusammenführenden Gateways innerhalb des Schleifen-Konstrukts zu berechnen, muss die Ausführungswahrscheinlichkeit des Sequenzflusses, welcher zum Schleifen-Konstrukt führt, mit dem Faktor $\left[\frac{1-(r_{1,1})^{L+1}}{r_{1,2}}\right]$ multipliziert werden. Dabei kann ein Wert entstehen, welcher größer ist als der Wert 1. Dies liegt in der Natur der Schleife und stellt für die Berechnung kein Problem dar.

Um die Ausführungswarscheinlichkeit von Sequenzflüssen innerhalb von Schleifen zu berechnen, werden die Faktoren aus der Formel von Jung et al. [2] verwendet. Die absolute Ausführungswahrscheinlichkeit eines Sequenzflusses innerhalb einer Schleife wird berechnet, indem die Ausführungswahrscheinlichkeit des eingehenden Sequenzflusses des Schleifen-Blocks mit den entsprechenden Faktoren multipliziert werden. Bei Schleifen ist zu beachten, dass die Ausführungswahrscheinlichkeit des verzweigenden Gateways $P(E)$ der Ausführungswahrscheinlichkeit des Sequenzflusses entspricht, welcher zum Schleifen-Block führt. Dies kann anhand des Beispiels demonstriert werden, welches in Abbildung 3.33 dargestellt ist.

Für das Beispiel werden folgende Werte angenommen: $P(S_1) = 1$, $r_{1,1} = 0.3$, $r_{1,2} = 0.7$ Des Weiteren wird angenommen, dass kein Limit für die Anzahl der Wiederholungen der Schleife vorhanden ist.

Somit kann die Ausführungswahrscheinlichkeit der jeweiligen Sequenzflüsse innerhalb der Schleife wie folgt berechnet werden:

$$\begin{aligned} P(S_2) = P(S_3) = P(S_1) \times \left[\frac{1-(r_{1,1})^{L+1}}{r_{1,2}}\right] \\ = \lim_{L\to\infty} 1 \times \left[\frac{1-0.3^{L+1}}{0.7}\right] = 1.4286 \\ P(S_4) = P(S_5) = P(S_1) \times \left[\frac{r_{1,1}-(r_{1,1})^{L+1}}{r_{1,2}}\right] \\ = \lim_{L\to\infty} 1 \times \left[\frac{0.3-0.3^{L+1}}{0.7}\right] = 0.4286 \end{aligned} \tag{3.29}$$

Die Ausführungswahrscheinlichkeit des ausgehenden Sequenzflusses des Schleifen-Blocks entspricht der Ausführungswahrscheinlichkeit des eingehenden Sequenzflusses des Schleifen-Blocks:

$$P(S_6) = P(S_1) = 1 \tag{3.30}$$

Die Ausführungswahrscheinlichkeit des verzweigenden Gateways entspricht der Ausführungswahrscheinlichkeit des Sequenzflusses, welcher zum Schleifen-Block führt:

$$P(E_2) = P(S_1) = 1 \tag{3.31}$$

3.3.6 Berechnung von Prozessen

In Kapitel 3.3.2 wurde bereits kurz darauf eingegangen, wie die Unsicherheit eines gesamten Geschäftsprozesses berechnet wird. Um die Unsicherheit eines Geschäftsprozesses zu berechnen, muss die Summe der absoluten Unsicherheiten aller der sich im Geschäftsprozess befindlichen Elemente gebildet werden.

$$U(G) = \sum_{i}^{N} U_a(E_i) \tag{3.32}$$

Dabei gilt zu beachten, dass nur verzweigende Elemente über eine Unsicherheit verfügen. Für andere Elemente wie beispielsweise Tasks oder zusammenführende Gateways gilt: $U(E) = 0$

Für die Berechnung der Unsicherheit der verzweigenden Elemente werden die Formeln von Jung et al. [2] verwendet. Wie bereits in Kapitel 3.3.2 erwähnt wurde, bestehen diese Formeln aus verschiedenen Teilen. Für die Berechnung der Unsicherheit von einzelnen Elementen werden lediglich jene Teile verwendet, welche die Unsicherheit der konkreten Gateways berechnen. Diese Formeln entsprechen den Formeln aus Kapitel 2.2, wenn angenommen wird, dass sich innerhalb der Blöcke keine weiteren Blöcke befinden.

Verzweigende parallele Gateways verfügen über keine Unsicherheit. Für sie gilt:

$$U_r(E_{AND}) = U_a(E_{AND}) = 0 \tag{3.33}$$

Die Unsicherheit von verzweigenden exklusiven Gateways wird wie folgt berechnet:

$$U_r(E_{XOR}) = -\sum_{n=1}^{N} r_n \log_2 r_n \tag{3.34}$$

$$U_a(E_{XOR}) = P(E_{XOR}) \times -\sum_{n=1}^{N} r_n \log_2 r_n \tag{3.35}$$

Die Unsicherheit von verzweigenden inklusiven Gateways wird wie folgt berechnet:

$$U_r(E_{OR}) = -\sum_{n=1}^{N} \left[r_n \log_2(r_n) + (1-r_n)\log_2(1-r_n)\right] \tag{3.36}$$

$$U_a(E_{OR}) = P(E_{OR}) \times -\sum_{n=1}^{N} \left[r_n \log_2(r_n) + (1-r_n)\log_2(1-r_n)\right] \tag{3.37}$$

Die Unsicherheit von verzweigenden Gateways von Schleifen wird wie folgt berechnet:

$$U_r(E_{LOOP}) = -\left[1-(r_{1,1})^L\right] \times \left[\frac{r_{1,1}\log_2(r_{1,1})}{r_{1,2}} + \log_2(r_{1,2})\right] \tag{3.38}$$

$$U_a(E_{LOOP}) = P(E_{LOOP}) \times -\left[1-(r_{1,1})^L\right] \times \left[\frac{r_{1,1}\log_2(r_{1,1})}{r_{1,2}} + \log_2(r_{1,2})\right] \tag{3.39}$$

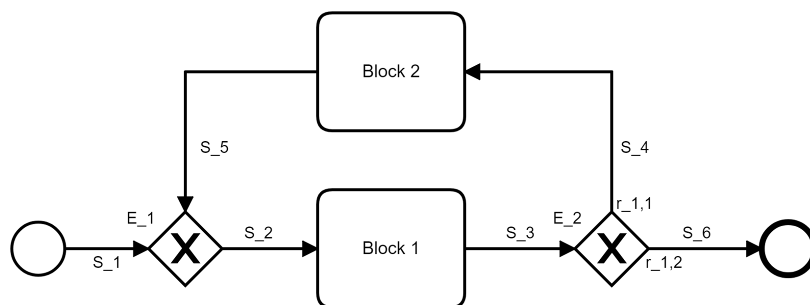

Abbildung 3.33: Schleife

Abschließend wird diese Methode anhand eines Geschäftsprozesses demonstriert. Der Geschäftsprozess kann in Abbildung 3.34 betrachtet werden. Es werden folgende Werte für das Beispiel angenommen: $P(S_1) = 1$, $r_{2,1} = 0.7$, $r_{2,2} = 0.3$, $r_{4,1} = 0.4$, $r_{4,2} = 0.6$, $r_{7,1} = 0.8$, $r_{7,2} = 0.6$, $r_{9,1} = 0.2$, $r_{9,2} = 0.8$ Außerdem wird angenommen, dass kein Limit für Wiederholungen innerhalb von Schleifen vorhanden ist.

Das Beispiel wird wie folgt berechnet:

$$
\begin{aligned}
P(S_2) &= P(S_1) = 1 \\
P(S_3) &= P(S_1) \times \left[\frac{1-(r_{4,1})^{L+1}}{r_{4,2}}\right] \\
&= \lim_{L\to\infty} 1 \times \left[\frac{1-0.4^{L+1}}{0.6}\right] = 1.6667 \\
P(E_2) &= P(S_3) = 1.6667 \\
U_a(E_2) &= P(E_2) \times -\sum_{n=1}^{N} r_n \log_2 r_n \\
&= 1.6667 \times -(0.7 \log_2 0.7 + 0.3 \log_2 0.3) = 1.4688 \\
P(S_4) &= P(S_3) \times r_{2,1} = 1.6667 \times 0.7 = 1.1667 \\
P(S_4) &= P(S_3) \times r_{2,1} = 1.6667 \times 0.3 = 0.5 \\
P(S_6) &= P(S_4) = 0.5 \\
P(S_7) &= P(S_5) = 1.1667 \\
P(S_8) &= P(S_6) + P(S_7) = 1.6667 \\
P(E_4) &= P(S_2) = 1 \\
U_a(E_4) &= P(E_4) \times -\left[1-(r_{4,1})^{L}\right] \times \left[\frac{r_{4,1}\log_2(r_{4,1})}{r_{4,2}} + \log_2(r_{4,2})\right] \\
&= \lim_{L\to\infty} 1 \times -\left[1-0.4^{L}\right] \times \left[\frac{0.4\log_2 0.4}{0.6} + \log_2 0.6\right] \\
&= 1.6183
\end{aligned}
\tag{3.40}
$$

$$
\begin{aligned}
P(S_9) &= P(S_1) \times \left[\frac{r_{4,1}-(r_{4,1})^{L+1}}{r_{4,2}}\right] \\
&= \lim_{L\to\infty} 1 \times \left[\frac{0.4-0.4^{L+1}}{0.6}\right] = 0.6667 \\
P(S_{10}) &= P(S_9) = 0.6667 \\
P(S_{11}) &= P(S_9) = 0.6667 \\
P(S_{12}) &= P(S_{11}) = 0.6667 \\
P(S_{13}) &= P(S_{10}) = 0.6667 \\
P(S_{14}) &= P(S_{13}) = P(S_{12}) = 0.6667
\end{aligned}
\tag{3.41}
$$

$$
\begin{aligned}
P(S_{15}) &= P(S_2) = 1 \\
P(E_7) &= P(S_{15}) = 1 \\
U_a(E_7) &= P(E_7) \times -\sum_{n=1}^{N} \left[r_n \log_2(r_n) + (1-r_n)\log_2(1-r_n)\right] \\
&= 1 \times -(0.8 \log_2 0.8 + 0.2 \log_2 0.2 + 0.6 \log_2 0.6 + 0.4 \log_2 0.4) \\
&= 1.6929 \\
P(S_{16}) &= P(S_{15}) \times r_{7,1} = 0.8 \\
P(S_{17}) &= P(S_{15}) \times r_{7,2} = 0.6 \\
P(S_{18}) &= P(S_{16}) = 0.8 \\
P(S_{19}) &= P(S_{16}) = 0.8 \\
P(S_{20}) &= P(S_{18}) = 0.8 \\
P(S_{19}) &= P(S_{21}) = 0.8 \\
P(S_{22}) &= P(S_{20}) = P(S_{21}) = 0.8
\end{aligned}
\tag{3.42}
$$

$$
\begin{aligned}
P(S_{23}) &= P(S_{17}) \times \left[\frac{1-(r_{9,1})^{L+1}}{r_{9,2}}\right] \\
&= \lim_{L\to\infty} 0.6 \times \left[\frac{1-0.2^{L+1}}{0.8}\right] = 0.75 \\
P(S_{24}) &= P(S_{23}) = 0.75 \\
P(E_9) &= P(S_{17}) = 0.6 \\
U_a(E_9) &= P(E_9) \times -\left[1-(r_{9,1})^{L}\right] \times \left[\frac{r_{9,1}\log_2(r_{9,1})}{r_{9,2}} + \log_2(r_{9,2})\right] \\
&= \lim_{L\to\infty} 0.6 \times -\left[1-0.2^{L}\right] \times \left[\frac{0.2\log_2 0.2}{0.8} + \log_2 0.8\right] \\
&= 0.5414 \\
P(S_{25}) &= P(S_{17}) \times \left[\frac{r_{9,1}-(r_{9,1})^{L+1}}{r_{9,2}}\right] \\
&= \lim_{L\to\infty} 0.6 \times \left[\frac{0.2-0.2^{L+1}}{0.8}\right] = 0.15 \\
P(S_{26}) &= P(S_{17}) = 0.6 \\
P(S_{27}) &= 1 - \left[(1-P(S_{22})) \times (1-P(S_{26}))\right] = 0.92
\end{aligned}
\tag{3.43}
$$

Die Unsicherheit des Prozesses setzt sich aus der Summe der absoluten Unsicherheiten der einzelnen Elemente zusammen:

$$
\begin{aligned}
U(G) &= \sum_{i}^{N} U_a(E_i) \\
&= U_a(E_2) + U_a(E_4) + U_a(E_7) + U_a(E_9) \\
&= 1.4688 + 1.6183 + 1.6929 + 0.5414 = 5.3214
\end{aligned}
\tag{3.44}
$$

Das Ergebnis dieser Berechnungsmethode deckt sich mit dem Ergebnis, welches durch die Berechnungsmethode von Jung et al. [2] erzielt wurde (siehe Kapitel 3.1.4).

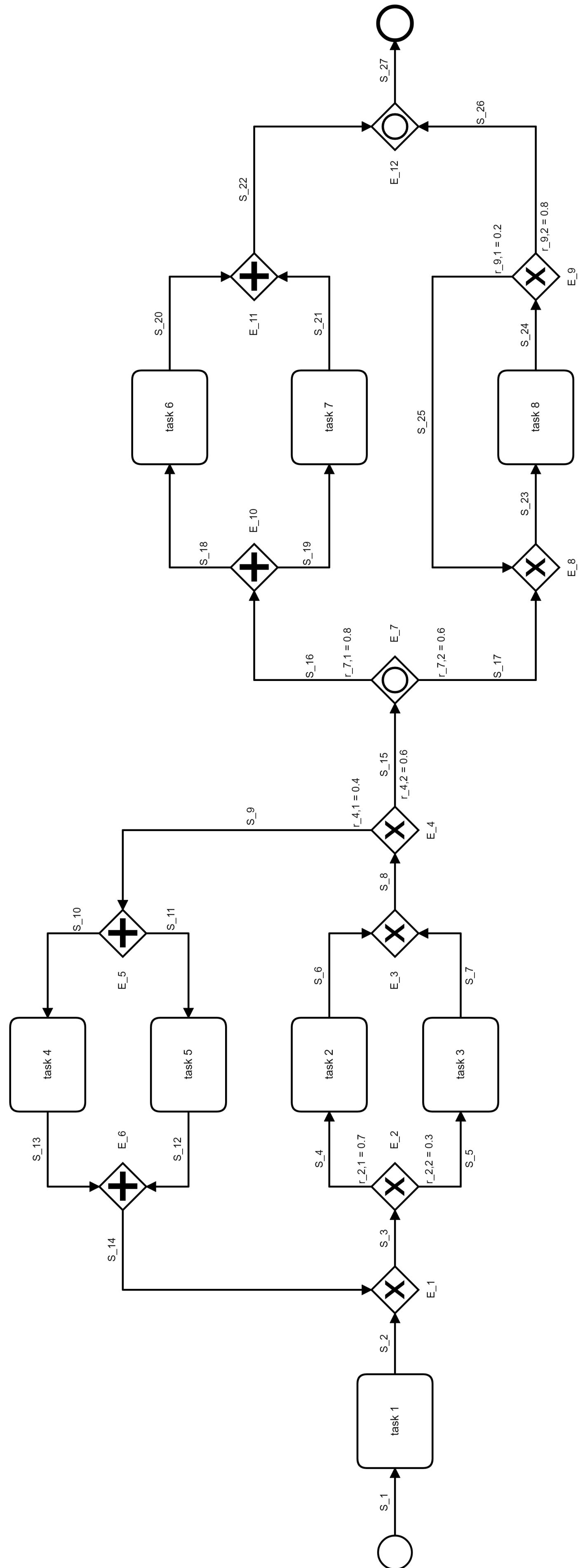

Abbildung 3.34: Beispiel: Berechnung von Elementen

4 Fazit

Im Zuge dieser Arbeit wurde festgestellt, dass die Methode zur Berechnung der Unsicherheit von Prozessen, welche von Jung et al. [2] veröffentlicht wurde, für eine Automatisierung ungeeignet ist. Wie in Kapitel 3.2 erwähnt wurde, liegt dies hauptsächlich an der Tatsache, dass es sich bei BPMN um eine graphenorientierte Sprache handelt [7]. Dies führt dazu, dass eine zahlreiche Menge an unterschiedlichen Mustern entstehen können. In der Methode von Jung et al. [2] werden für bestimmte Blöcke Formeln für die Berechnung der Unsicherheit definiert. Aufgrund der Vielzahl an möglichen Mustern ist es jedoch möglich, dass ein neues Muster entdeckt wird, welches nicht in eines der Schemas der vordefinierten Formeln passt. In einem solchen Fall muss eine neue Formel definiert werden, welches die Berechnung des neu entdeckten Musters ermöglicht.

Diese Eigenschaft ist besonders problematisch, wenn die Methode automatisiert werden soll. Eine programmierte Webapplikation ist üblicherweise nicht dazu in der Lage, selbstständig neue Muster zu erkennen und eine neue Formel für die Berechnung dieses Musters zu entwickeln. Aus diesem Grund wurde in dieser Arbeit eine Methode vorgestellt, in welcher die Unsicherheit von einzelnen Elementen anstatt von vordefinierten Blöcken berechnet wird. Diese neue Methode zur Berechnung der Unsicherheit beruht auf der Methode von Jung et al. [2] sowie den grundlegendenden Gesetzen der Wahrscheinlichkeitsrechnung.

Wie in Kapitel 3.3.6 gezeigt wurde, führt die neue Berechnungsmethode zum selben Ergebnis wie die Methode von Jung et al. [2]. Des Weiteren ist diese Methode auf alle möglichen Muster anwendbar. Formeln werden grundsätzlich für einzelne Elemente anstatt für Blöcke definiert. Auch wenn der Vorgang der Berechnung aufwendiger und umfangreicher erscheint, so stellt dies für eine Automatisierung kein Hindernis dar.

Ein weiterer Aspekt, welcher im Zuge einer Automatisierung von besonderer Relevanz ist, ist die Erkennung von Mustern. Die Methode von Jung et al. [2] schreibt vor, die Unsicherheit von Blöcken zu berechnen. Blöcke enthalten dabei mehrere Elemente, welche ein bestimmtes Muster bilden. Eine Automatisierung dieser Methode setzt somit voraus, dass jene Blöcke von einem Algorithmus korrekt identifiziert werden können. Die automatisierte Identifikation der Blöcke, welche von Jung et al. [2] in ihrer Arbeit behandelt wurden, ist definitiv möglich. Dies wird durch die Webapplikation, welche im Zuge dieser Arbeit entwickelt wurde, bestätigt. Jedoch muss beachtet werden, dass eine Vielzahl von weiteren Mustern in modellierten Geschäftsprozessen vorkommen können, wie in Kapitel 3.2.1 gezeigt wird. Die automatisierte Erkennung von Mustern innerhalb von Geschäftsprozessen ist eine herausfordernde Aufgabe, wie auch bereits vergangene Projekte zeigten, in welchen BPMN Prozesse in BPEL Prozesse transformiert werden sollten [8].

Die in dieser Arbeit vorgestellte Berechnungsmethode ist grundsätzlich nicht darauf angewiesen, bestimmte Muster zu erkennen, da die Unsicherheit von einzelnen Elementen berechnet wird. Jedoch muss an dieser Stelle angemerkt werden, dass dies nur bedingt zutrifft. Schleifen müssen auch bei der neuen Berechnungsmethode identifiziert werden, bevor eine korrekte Berechnung des Prozesses möglich ist. Dies liegt daran, dass für die Modellierung von Schleifen kein eigenes Element definiert ist. Dies bedeutet, dass auch bei der neuen Berechnungsmethode eine Form der Mustererkennung notwendig ist.

In dieser Arbeit wird gezeigt, wie der Informationsgehalt von Geschäftsprozessen berechnet werden kann. Ein Ziel dieser Arbeit ist es zu evaluieren, ob sich dieses Maß ebenfalls für die

Quantifizierung von Komplexität eignet. Wie Mitchell [3, pp. 13-14] bereits schreibt, kann eine Quantifizierung von Komplexität erst dann stattfinden, wenn der Begriff ausreichend definiert ist. Auch wenn bereits unterschiedliche Definitionen für Komplexität vorhanden sind, so ist laut Mitchell keine dieser Definitionen allgemein anerkannt. Somit lässt sich nicht klar beantworten, ob der Informationsgehalt nach Shannon [1] ein geeignetes Maß für die Komplexität darstellt. Den Informationsgehalt eines Systems als Grundlage für die Berechnung der Komplexität dieses Systems zu verwenden, wurde jedoch auch schon in anderen Bereichen der Wissenschaft untersucht, wie die Arbeit von Adami [10] zeigt.

Einer der Gründe, weshalb eine universelle Definition des Begriffes ”Komplexität” eine besonders herausfordernde Aufgabe ist, wird von Simon [11] in seiner Arbeit ausführlich behandelt. Simon schreibt, dass komplexe Strukturen in verschiedenen Systemen anzutreffen sind. So können komplexe Strukturen etwa in sozialen Systemen, biologischen und physikalischen Systemen oder auch in symbolischen Systemen entdeckt werden. Um also eine universelle Definition des Begriffs ”Komplexität” zu erhalten, müssen die Gemeinsamkeiten der komplexen Strukturen innerhalb der unterschiedlichen Systeme näher untersucht werden.

Unabhängig von der Fragestellung, ob der Informationsgehalt eines Geschäftsprozesses tatsächlich ein Maß für die Komplexität des Geschäftsprozesses darstellt, kann eine Methode zur Berechnung der Unsicherheit von Prozessen für Unternehmen von Nutzen sein. Wie Jung et al. [2] schreiben, kann die Reduktion der Unsicherheit eines Geschäftsprozesses dazu führen, dass die Ressourcenplanung innerhalb des Unternehmens erleichtert wird. Eine Applikation, welche dazu in der Lage ist, die Unsicherheit von Geschäftsprozessen mithilfe eines Algorithmus zu berechnen, kann dazu eingesetzt werden, Prozesse mit einer hohen Unsicherheit zu identifizieren. Des Weiteren können Prozesse anhand eines berechneten Wertes miteinander verglichen werden. Dies ermöglicht auch weitere Untersuchungen über mögliche Zusammenhänge zwischen der Unsicherheit von Prozessen und anderen relevanten Kenngrößen.

5 Anhang

```
/*
  Part of this code was used from the following projects:
  bpmn-js:
  https://github.com/bpmn-io/bpmn-js-examples/tree/master/modeler
  bpmn-engine:
  https://github.com/paed01/bpmn-engine
*/

import $ from 'jquery';
import BpmnViewer from 'bpmn-js/lib/Viewer';
const { Engine } = require('bpmn-engine');
const BpmnModdle = require('bpmn-moddle').default;
const elements = require('bpmn-elements');
const { default: Serializer, TypeResolver } = require('moddle-context
    -serializer');


/**
 * BPMN Viewer for the visualization of the imported process
 * as well as the functionality to import .bpmn files
 **/

var container = $('#js-drop-zone');
var viewer = new BpmnViewer({
  container: $('#js-canvas'),
  //height: 600
});

async function openDiagram(xml) {
  try {
    await viewer.importXML(xml);
    container
      .removeClass('with-error')
      .addClass('with-diagram');
  } catch (err) {
    container
      .removeClass('with-diagram')
      .addClass('with-error');
    container.find('.error pre').text(err.message);
    console.error(err);
  }

  /**
   * BPMN-ENGINE
   * Important Note!!!: The Startevent of the BPMN-Process
   * has to have the id "start", otherwise the loading process
   * will fail
   *
   * The bpmn-engine imports the process from the XML file
   * and structures the possible paths of the process in
   * different arrays
   */
  (async function IIFE() {
```

```
53      const moddleContext = await (new BpmnModdle({
54        camunda: require('camunda-bpmn-moddle/resources/camunda.json'),
55      })).fromXML(xml);
56
57      /**
58       * The elementRegistry contains information that the bpmn-engine
59       * already pre-filters
60       * Some of this information is needed later
61       * The element registry is not part of the bpmn-engine, but
           rather
62       * is a part of the bpmn-js library
63       */
64      const elementRegistry = viewer.get('elementRegistry');
65      const businessElements = elementRegistry._elements;
66      const sourceContext = Serializer(moddleContext, TypeResolver(
          elements));
67      const engine = Engine({
68        sourceContext,
69      });
70
71      const [definition] = await engine.getDefinitions();
72
73      /**
74       * shakenStarts is the product of the bpmn-engine and provides
           the sequences
75       * of the process along with some meta data of the bpmn process
76       */
77      const shakenStarts = definition.shake();
78
79      // filtering the process meta data since it is not needed
80      const shakenStartsSequences = shakenStarts.start.map(e => e.
          sequence);
81
82      /**
83       * Removing Tasks, Sequenceflows, StartEvents and EndEvents from
           the sequences
84       * The only elements needed for the algorithm are the gateways
85       */
86      const shakenStartsFiltered = shakenStartsSequences
87        .map(e => e.filter(e => !e.type.includes("EndEvent")
88          && !e.type.includes("StartEvent") && !e.type.includes("Task")
89          && !e.type.includes("SequenceFlow")));
90
91      /**
92       * Applying a custom function to each remaining element in order
           to add
93       * some necessary information
94       */
95      const convertedElements = shakenStartsFiltered
96        .map(m => m.map(e => convertElements(e)));
97
98      /**
99       * Applying a custom function to the arrays in order to pair the
           elements
100      * This is necessary in order to identify "blocks" within the
           bpmn process
101      * A pair consists of two gateways
102      * Each of the gateways marks the start and the end of the block
           respectively
103      * Paired elements share the same pairid
```

```
104     */
105     const pairedElements = pairElements(convertedElements);
106 
107     /**
108      * Applying a custom function to the arrays in order to classify
           the blocks
109      * within the bpmn process
110      * The blocks will be classified as one of the following: XOR, OR
           , AND, LOOP
111      */
112     const classifiedElements = classifyElements(pairedElements);
113 
114     /**
115      * Applying a custom function to the arrays in order to calculate
            the
116      * uncertainty of the entire process
117      */
118     const uncertainty = calculateUncertainty(classifiedElements);
119 
120     /**
121      * changeLabel visualizes the calculated uncertainty of the bpmn
           process
122      * on the HTML page
123      */
124     changeLabel();
125     function changeLabel() {
126       let lbl = document.getElementById('calculatedUncertainty');
127       lbl.innerText = "The uncertainty of the business process: " +
              uncertainty;
128     }
129 
130     /**
131      * This function adds necessary information to each element of
           the bpmn process
132      * This function also determines, whether a gateway is a
           splitting gateway
133      * or a merging gateway (attribute "sm")
134      * Some of these attributes are permanent and remain unchanged
           like the id
135      * attribute or the sm attribute, while other attributes might be
            changed
136      * as needed
137      * The boolean attributes "removed" and "calculated" store the
           state of each
138      * element and are necessary for the calculation
139      */
140     function convertElements(e) {
141       const fullElement = businessElements[e.id];
142       const incoming = fullElement.element.incoming.length;
143       const outgoing = fullElement.element.outgoing.length;
144       if (incoming == 1 && outgoing > 1) {
145         return ({
146           id: e.id, type: e.type, sm: "split", outgoing: outgoing,
147           uncertainty: 0, uncertaintyOfBranches: [], recordedBranches
                : [],
148           removed: false, calculated: false, counter: 0, paired:
                false, pairid: "",
149           isLoop: false, classification: ""
150         });
151       } else if (incoming > 1 && outgoing == 1) {
```

```
152         return ({
153           id: e.id, type: e.type, sm: "merge", incoming: incoming,
154           uncertainty: 0, uncertaintyOfBranches: [], recordedBranches
                : [],
155           removed: false, calculated: false, counter: 0, paired:
                false,
156           pairid: "", isLoop: false, classification: ""
157         });
158       }
159     };
160 
161     /**
162      * This function is responsible for pairing the gateways in order
            to identify
163      * blocks within the process
164      * Gateways will be paired if a merging gateway follows a
           splitting gateway
165      * in the same sequence
166      * In the next iteration paired elements will be removed in order
            to pair
167      * gateways that contain closed blocks
168      * This method only works correctly if the process only contains
           "clean" blocks
169      * that are properly closed and do not overlap with other blocks
170      */
171     function pairElements(listsOfElements) {
172       var pairedList = listsOfElements;
173       for (let x = 0; x < listsOfElements.length; x++) {
174         var filteredUnpairedList = pairedList
175           .map(e => e.filter(m => m.paired != true));
176         if (filteredUnpairedList.length >= 0) {
177           for (let i = 0; i < filteredUnpairedList.length; i++) {
178             const listOfElements = filteredUnpairedList[i];
179             for (let j = 0; j < listOfElements.length; j++) {
180               if (checkIfSequence(filteredUnpairedList,
181                 listOfElements[j],
182                 listOfElements[j + 1])) {
183                 pairedList = setAttributes(pairedList,
184                   listOfElements[j],
185                   listOfElements[j + 1]);
186               }
187             }
188           }
189         }
190       }
191       const pairidArray = pairedList.map(m => m.map(e => e.pairid));
192       /**
193        * Because loops have an inverted pattern (merging gateway
             comes first)
194        * this can cause the function to pair some elements
             incorrectly if loops
195        * are present in the process
196        * However, loops can be identified because the merging gateway
              appears twice
197        * in one of the sequences
198        * This way the incorrect pairings can be reverted
199        */
200       const duplicates = pairidArray
201         .map(e => e.filter((elem, index) => e.indexOf(elem) !== index
              ));
```

```
202       var duplicatesConverted = [];
203       for (let i = 0; i < duplicates.length; i++) {
204         duplicatesConverted = duplicatesConverted.concat(duplicates[i
              ]);
205       }
206       for (let i = 0; i < pairedList.length; i++) {
207         for (let j = 0; j < pairedList[i].length; j++) {
208           if (!duplicatesConverted.includes(pairedList[i][j].pairid))
                {
209             pairedList[i][j].pairid = "";
210             pairedList[i][j].paired = false;
211           }
212         }
213       }
214       return (pairedList);
215     }
216
217     /**
218      * This function checks if a splitting gateway is followed by a
           merging
219      * gateway
220      */
221     function checkIfSequence(unpairedElements, a, b) {
222       var check = false;
223       if (b != undefined) {
224         for (let i = 0; i < unpairedElements.length; i++) {
225           const elements = unpairedElements[i];
226           for (let j = 0; j < elements.length - 1; j++) {
227             if ((elements[j].id == a.id && elements[j + 1].id == b.id
                  ) &&
228               (a.sm == "split" && b.sm == "merge")) {
229               check = true;
230             }
231           }
232         }
233       }
234       return check;
235     }
236
237     /**
238      * This function creates a unique pairid for two elements that
           have been
239      * detected as a pair
240      * To ensure that the pairid is unique, the id of both elements
           is combined
241      * into the pairid, while both ids are separated by a "_"
           character
242      * Afterwards the function sets the pairids for both elements and
            also
243      * sets the bolean attribute "ispaired" to "true"
244      */
245     function setAttributes(elementArray, elem1, elem2) {
246       var id1 = elem1.id;
247       var newElementArray = elementArray;
248       if (elem2 != undefined) {
249         var id2 = elem2.id;
250       }
251       for (let i = 0; i < elementArray.length; i++) {
252         for (let j = 0; j < elementArray[i].length; j++) {
253           if (elem2 != undefined) {
```

```
254             if (elementArray[i][j].id == id1 || elementArray[i][j].id
                    == id2) {
255               newElementArray[i][j].paired = true;
256               newElementArray[i][j].pairid = id1.concat("_").concat(
                      id2);
257             }
258           } else {
259             if (elementArray[i][j].id == id1) {
260               newElementArray[i][j].paired = true;
261               newElementArray[i][j].pairid = id1;
262             }
263           }
264         }
265       }
266       return newElementArray;
267     }
268 
269     /**
270      * This function classifies each element of the business process
271      * Because loops are not bound to a specific type of element,
           loops have to
272      * be detected first
273      * After a loop has been detected, the "isloop" attribute is set
           to "true" for
274      * the involved elements
275      * Then the elements can be classified
276      * If the "isloop" attribute is "true", then the element is
           classified as part
277      * of a loop block
278      * If the "isloop" attribute is false, then the elements will be
           classified
279      * according to their gateway type (exclusive, parallel,
           inclusive)
280      */
281     function classifyElements(elemArray) {
282       const idArray = elemArray.map(m => m.map(e => e.id));
283       var toclassifyArray = elemArray;
284       const duplicates = idArray
285         .map(e => e.filter((elem, index) => e.indexOf(elem) !== index
              ));
286       var duplicatesConverted = [];
287       for (let i = 0; i < duplicates.length; i++) {
288         duplicatesConverted = duplicatesConverted.concat(duplicates[i
              ]);
289       }
290       for (let i = 0; i < toclassifyArray.length; i++) {
291         for (let j = 0; j < toclassifyArray[i].length; j++) {
292           if (duplicatesConverted.includes(toclassifyArray[i][j].id))
                {
293             toclassifyArray[i][j].isLoop = true;
294             for (let x = 0; x < toclassifyArray.length; x++) {
295               for (let y = 0; y < toclassifyArray[x].length; y++) {
296                 if (toclassifyArray[x][y].pairid == toclassifyArray[i
                      ][j].pairid) {
297                   toclassifyArray[x][y].isLoop = true;
298                 }
299               }
300             }
301           }
302         }
```

```
303         }
304         for (let i = 0; i < toclassifyArray.length; i++) {
305           for (let j = 0; j < toclassifyArray[i].length; j++) {
306             const classifyElem = toclassifyArray[i][j];
307             if (classifyElem.isLoop) {
308               classifyElem.classification = "LOOP";
309             } else if (classifyElem.type.includes("Exclusive")) {
310               classifyElem.classification = "XOR";
311             } else if (classifyElem.type.includes("Parallel")) {
312               classifyElem.classification = "AND"
313             } else if (classifyElem.type.includes("Inclusive")) {
314               classifyElem.classification = "OR"
315             }
316           }
317         }
318         return toclassifyArray;
319       }
320
321       /**
322        * This function is responsible for calculating the uncertainty
             of the
323        * entire business process
324        * Blocks can only be calculated if they do not contain any
             uncalculated blocks
325        * For this reason the calculation is an iterative process and
             many
326        * requirements have to be checked before the actual calculation
327        */
328       function calculateUncertainty(classifiedElements) {
329         var calculateArray = classifiedElements;
330         for (let m = 0; m < calculateArray.length; m++) {
331           for (let n = 0; n < calculateArray[m].length; n++) {
332             // filterCalculated only contains elements that have not
333             // been calculated yet by checking the "calculated"
                    attribute
334             var filterCalculated = calculateArray
335               .map(e => e.filter(m => m.calculated == false));
336             // filterRemoved only contains elements that have not
337             // been removed yet by checking the "removed" attribute
338             var filterRemoved = calculateArray
339               .map(e => e.filter(m => m.removed == false));
340             /**
341              * At first it is necessary to iterate over all arrays in
                   order to
342              * detect which blocks can be calculated
343              * A block can only be calculated if it doesn't contain any
                    other
344              * uncalculated blocks
345              * Since each array only stores one path of the business
                   process,
346              * a counter has to be introduced in order to keep track of
                    any
347              * uncalculated blocks that reside inside of any other
                   block in any path
348              * of the business process
349              * The counter needs to be increased for the relevant
                   element in each
350              * array
351              * The counter is stored in the splitting element, since
                   this is also the
```

```
352                  * center of interest for the calculation
353                  */
354                 for (let i = 0; i < filterCalculated.length; i++) {
355                   for (let j = 0; j < filterCalculated[i].length; j++) {
356                     if (filterCalculated[i][j + 1] != undefined &&
357                       filterCalculated[i][j].pairid != filterCalculated[i][
                              j + 1].pairid &&
358                       filterCalculated[i][j].sm == "split" &&
359                       filterCalculated[i][j].classification != "LOOP") {
360                       for (let x = 0; x < calculateArray.length; x++) {
361                         for (let y = 0; y < calculateArray[x].length; y++)
                                {
362                           if (filterCalculated[i][j].id == calculateArray[x
                                  ][y].id) {
363                             calculateArray[x][y].counter += 1;
364                           }
365                         }
366                       }
367                       /**
368                        * Loops have to checked separately, because blocks
                              can be
369                        * inside of a loop block while being positioned
                              before or
370                        * after the splitting gateway of the loop block
371                        */
372                     } else if (filterCalculated[i][j].sm == "split" &&
373                       filterCalculated[i][j].classification == "LOOP") {
374                       for (let x = 0; x < filterCalculated.length; x++) {
375                         var filterPairID = filterCalculated[x]
376                           .filter(m => m.pairid == calculateArray[i][j].
                                  pairid);
377                         /**
378                          * Only the array that contains the repatriating
                                path of the loop
379                          * contains information of both relevant sides of
                                the splitting
380                          * gateway
381                          * Therefore the array that contains the duplicate
                                merging
382                          * gateway is identified before applying the
                                counter
383                          */
384                         if (filterPairID.length >= 3) {
385                           for (let y = 0; y < filterCalculated[x].length; y
                                  ++) {
386                             if (filterCalculated[x][y].id == calculateArray
                                    [i][j].id &&
387                               (filterCalculated[x][y + 1].pairid !=
388                                 filterCalculated[i][j].pairid ||
389                                 filterCalculated[x][y - 1].pairid !=
390                                 filterCalculated[i][j].pairid)) {
391                               for (let t = 0; t < filterCalculated.length;
                                      t++) {
392                                 for (let z = 0; z < filterCalculated[t].
                                        length; z++) {
393                                   if (calculateArray[i][j].id ==
                                          calculateArray[t][z].id) {
394                                     calculateArray[t][z].counter += 1;
395                                   }
396                                 }
```

```
397                         }
398                       }
399                     }
400                   }
401                 }
402               }
403             }
404           }
405
406           /**
407            * Next, the uncertainty of a calculated block that resides
                   inside of
408            * another block is added to a storage array in the outer
                 block
409            * That way the uncertainty of inner blocks can be
                 considered when
410            * calculating the uncertainty of outer blocks
411            * Furthermore, the probability of the according path is
                 stored along
412            * with the uncertainty of that path
413            */
414           for (let i = 0; i < calculateArray.length; i++) {
415             for (let j = 0; j < calculateArray[i].length; j++) {
416               if (calculateArray[i][j].counter == 0 &&
417                 calculateArray[i][j + 1] != undefined &&
418                 !calculateArray[i][j].recordedBranches
419                   .includes(calculateArray[i][j + 1].id) &&
420                 calculateArray[i][j].sm == "split" &&
421                 calculateArray[i][j].classification != "LOOP") {
422                 /**
423                  * The probability of a path has to be stored within
                       the id of
424                  * the according sequence by adding "_x.y_" in front
                       of the sequence
425                  * id, while x.y is the probability as a decimal
                       number
426                  * For example: the id of the sequence is "
                       Sequence_eireg35j"
427                  * Then the probability can be added
428                  * like this: "_0.7_Sequence_eireg35j"
429                  * If there is no valid number defined as probability
                         , then
430                  * the probability 0.5 is set automatically
431                  */
432                 for (let k = 0; k < shakenStartsSequences[i].length;
                      k++) {
433                   if (shakenStartsSequences[i][k].id ==
                        calculateArray[i][j].id) {
434                     var id = shakenStartsSequences[i][k + 1].id;
435                     var probabilityArray = id.split('_');
436                     if (!isNaN(probabilityArray[1])) {
437                       var probability = Number(probabilityArray[1]);
438                     } else {
439                       var probability = 0.5;
440                     }
441                   }
442                 }
443                 for (let x = 0; x < calculateArray.length; x++) {
444                   for (let y = 0; y < calculateArray[x].length; y++)
                        {
```

```
445                     if (calculateArray[i][j].id == calculateArray[x][
                          y].id) {
446                       /**
447                        * If there is a calculated block within a
                             another block,
448                        * then the uncertainty will be added to the
                             storage of
449                        * the outer block
450                        */
451                       if (calculateArray[i][j].pairid !=
452                         calculateArray[i][j + 1].pairid) {
453                         if (!calculateArray[x][y].
                              uncertaintyOfBranches
454                           .map(e => e.id).includes(id)) {
455                           calculateArray[x][y].uncertaintyOfBranches
456                             .push({
457                               id: id,
458                               uncertainty: calculateArray[i][j + 1].
                                    uncertainty,
459                               probability: probability
460                             });
461                         }
462                         calculateArray[x][y].recordedBranches
463                           .push(calculateArray[i][j + 1].pairid);
464                         /**
465                          * If there is no block inside of this block
                               at the current
466                          * path, then the uncertainty 0 is added to
                               the storage
467                          * This occurs when only Tasks reside within
                               a block
468                          * Tasks do not increase the uncertainty
469                          */
470                       } else if (calculateArray[i][j].pairid ==
471                         calculateArray[i][j + 1].pairid) {
472                         if (!calculateArray[x][y].
                              uncertaintyOfBranches
473                           .map(e => e.id).includes(id)) {
474                           calculateArray[x][y].uncertaintyOfBranches
475                             .push({
476                               id: id,
477                               uncertainty: 0,
478                               probability: probability
479                             });
480                         }
481                       }
482                     }
483                   }
484                 }
485                 /**
486                  * The process of adding the uncertainty of inner
                       blocks to the
487                  * storage of outer blocks is different for loops,
                       because it is
488                  * necessary to define whether the inner block is
                       positioned before
489                  * or after the splitting gateway
490                  */
491               } else if (calculateArray[i][j].counter == 0 &&
492                 calculateArray[i][j].recordedBranches.length == 0 &&
```

```
493                 calculateArray[i][j].sm == "split" &&
494                 calculateArray[i][j].classification == "LOOP") {
495                 var pairidArray = filterRemoved[i].map(e => e.pairid)
                       ;
496                 var filteredPairArray = pairidArray
497                   .filter(m => m == calculateArray[i][j].pairid);
498                 /**
499                  * Like before, the repatriating path of the loop has
                        to be
500                  * identified first, before the uncertainty of the
                       inner blocks
501                  * can be added to the storage of the splitting
                       gateway of the
502                  * loop block
503                  * In the repatriating path, the merge gateway
                       appears twice
504                  * This means that there will be 3 gateways with the
                       same pairid
505                  * in this path
506                  */
507                 if (filteredPairArray.length >= 3) {
508                   for (let y = 0; y < filterRemoved[i].length; y++) {
509                     /**
510                      * The first case checks if there is a calculated
                           block
511                      * before the splitting gateway of the loop block
512                      * If there is a calculated block before the
                          splitting gateway
513                      * of the loop block, then it is added to the
                          storage and stored
514                      * with the note that this is branch 1
515                      */
516                     if (filterRemoved[i][y].id == calculateArray[i][j
                          ].id) {
517                       if (filterRemoved[i][y].id == calculateArray[i
                            ][j].id &&
518                         filterRemoved[i][y - 1].pairid !=
519                         filterRemoved[i][y].pairid &&
520                         filterRemoved[i][y - 2].pairid ==
521                         filterRemoved[i][y].pairid &&
522                         !calculateArray[i][j].uncertaintyOfBranches
523                           .map(e => e.branch).includes(1)) {
524                         for (let f = 0; f < calculateArray.length; f
                              ++) {
525                           for (let g = 0; g < calculateArray[f].
                                length; g++) {
526                             if (calculateArray[i][j].id ==
                                  calculateArray[f][g].id) {
527                               calculateArray[f][g].
                                    uncertaintyOfBranches
528                                 .push({
529                                   id: filterRemoved[i][y - 1].id,
530                                   branch: 1,
531                                   uncertainty: filterRemoved[i][y -
                                        1].uncertainty
532                                 });
533                             }
534                           }
535                         }
536                         /**
```

```
537                    * If there is no block before the splitting
                         gateway (for
538                    * example if only a Task resides there),
                         then the
539                    * uncertainty 0 is added to the storage for
                         branch 1
540                    */
541                } else if (filterRemoved[i][y].id ==
542                  calculateArray[i][j].id &&
543                  filterRemoved[i][y - 1].pairid ==
544                  filterRemoved[i][y].pairid &&
545                  !calculateArray[i][j].uncertaintyOfBranches
546                    .map(e => e.branch).includes(1)) {
547                  for (let f = 0; f < calculateArray.length; f
                       ++) {
548                    for (let g = 0; g < calculateArray[f].
                         length; g++) {
549                      if (calculateArray[i][j].id ==
                           calculateArray[f][g].id) {
550                        calculateArray[f][g].
                             uncertaintyOfBranches
551                          .push({
552                            id: filterRemoved[i][y - 1].id,
553                            branch: 1,
554                            uncertainty: 0
555                          });
556                      }
557                    }
558                  }
559                }
560                /**
561                 * Now the same procedure has to be done for
                      branch 2
562                 * Branch 2 describes the loop-back path
563                 * Here, also the loop-back probability is
                      stored along
564                 * with the uncertainty of any inner blocks
565                 * The probability is detected by examining the
                       id of
566                 * the according sequence, as it was already
                      explained before
567                 */
568                if (filterRemoved[i][y].id == calculateArray[i
                     ][j].id &&
569                  filterRemoved[i][y + 1].pairid !=
570                  filterRemoved[i][y].pairid &&
571                  filterRemoved[i][y + 2].pairid ==
572                  filterRemoved[i][y].pairid &&
573                  !calculateArray[i][j].uncertaintyOfBranches
574                    .map(e => e.branch).includes(2)) {
575                  for (let k = 0; k < shakenStartsSequences[i].
                       length; k++) {
576                    if (shakenStartsSequences[i][k].id ==
577                      calculateArray[i][j].id) {
578                      var probabilityArray =
                           shakenStartsSequences[i][k + 1]
579                        .id.split('_');
580                      if (!isNaN(probabilityArray[1])) {
581                        var probability = Number(
                             probabilityArray[1]);
```

```
582                 } else {
583                   var probability = 0.5;
584                 }
585               }
586             }
587             for (let f = 0; f < calculateArray.length; f
                    ++) {
588               for (let g = 0; g < calculateArray[f].
                      length; g++) {
589                 if (calculateArray[i][j].id ==
                        calculateArray[f][g].id) {
590                   calculateArray[f][g].
                          uncertaintyOfBranches
591                     .push({
592                       id: filterRemoved[i][y + 1].id,
593                       branch: 2,
594                       uncertainty: filterRemoved[i][y +
                              1].uncertainty,
595                       probability: probability
596                     });
597                 }
598               }
599             }
600             /**
601              * And again, if there is no block within
                    branch 2, then
602              * the uncertainty 0 is stored along with the
                      probabilty
603              * of the loop-back path
604              */
605           } else if (filterRemoved[i][y].id ==
606             calculateArray[i][j].id &&
607             filterRemoved[i][y + 1].pairid ==
608             filterRemoved[i][y].pairid &&
609             !calculateArray[i][j].uncertaintyOfBranches
610               .map(e => e.branch).includes(2)) {
611             for (let k = 0; k < shakenStartsSequences[i].
                    length; k++) {
612               if (shakenStartsSequences[i][k].id ==
613                 calculateArray[i][j].id) {
614                 var probabilityArray =
                        shakenStartsSequences[i][k + 1]
615                   .id.split('_');
616                 if (!isNaN(probabilityArray[1])) {
617                   var probability = Number(
                          probabilityArray[1]);
618                 } else {
619                   var probability = 0.5;
620                 }
621               }
622             }
623             for (let f = 0; f < calculateArray.length; f
                    ++) {
624               for (let g = 0; g < calculateArray[f].
                      length; g++) {
625                 if (calculateArray[i][j].id ==
                        calculateArray[f][g].id) {
626                   calculateArray[f][g].
                          uncertaintyOfBranches
627                     .push({
```

```
628                             id: filterRemoved[i][y + 1].id,
629                             branch: 2, uncertainty: 0,
630                             probability: probability
631                           });
632                       }
633                     }
634                   }
635                 }
636               }
637             }
638           }
639         }
640       }
641     }
642
643     /**
644      * In this part, the actual calculation of the uncertainty
           of each block
645      * takes place
646      * A block can only be calculated if there are no other
           uncalculated
647      * blocks residing inside of the block
648      * The calculation takes place when reaching the splitting
           gateways
649      * The calculated uncertainty of inner blocks is also
           stored within
650      * the splitting gateways
651      * Each different block type comes with its own formula for
            the
652      * calculation of its uncertainty
653      */
654     for (let i = 0; i < calculateArray.length; i++) {
655       for (let j = 0; j < calculateArray[i].length; j++) {
656         if (calculateArray[i][j].counter == 0 &&
657           calculateArray[i][j].removed == false &&
658           calculateArray[i][j].sm == "split" &&
659           calculateArray[i][j].calculated == false) {
660           /**
661            * XOR blocks are calculated here
662            */
663           if (calculateArray[i][j].classification == "XOR") {
664             /**
665              * The probabilityArray stores the probabilities of
                   each
666              * outgoing paths of the splitting gateway
667              */
668             const probabilityArray = calculateArray[i][j]
669               .uncertaintyOfBranches.map(e => e.probability);
670             /**
671              * Here the probabilities are used to calculate a
                 part
672              * of the uncertainty of the XOR block
673              */
674             const calculatedProbabilityArray = probabilityArray
675               .map(e => e * Math.log2(e));
676             /**
677              * In order to keep the formulas shorter, the
                 formulas were
678              * split into parts
679              * Here the uncertainty of the XOR block is
```

```
                calculated in 2
680           * parts
681           * The first part is calculated in uncertainty1
682           * The second part is calculated in uncertainty2
683           */
684          const uncertainty1 = -1 *
                calculatedProbabilityArray
685            .reduce((prev, cur) => prev + cur, 0);
686          const weightedUncertaintyArray = calculateArray[i][
                j]
687            .uncertaintyOfBranches.map(e => e.uncertainty * e
                  .probability);
688          const uncertainty2 = weightedUncertaintyArray
689            .reduce((prev, cur) => prev + cur, 0);
690          for (let g = 0; g < calculateArray.length; g++) {
691            for (let h = 0; h < calculateArray[g].length; h
                  ++) {
692              if (calculateArray[i][j].recordedBranches
693                .includes(calculateArray[g][h].pairid)) {
694                /**
695                 * If the uncertainty of a block has been
                     calculated,
696                 * every inner block has to be removed from
                     all arrays
697                 * This happens by marking them with the
                     according attribute
698                 * These elements will be filtered in future
                     iterations
699                 */
700                calculateArray[g][h].removed = true;
701              }
702              if (calculateArray[i][j].id == calculateArray[g
                  ][h].id) {
703                /**
704                 * Here both parts of the calculated
                     uncertainty are
705                 * added together and stored in the according
                      element
706                 * It is necessary to iterate over all arrays
                     , because
707                 * the element in which the uncertainty has
                     to be stored
708                 * can be present in multiple arrays (paths)
709                 */
710                calculateArray[g][h].uncertainty =
711                  uncertainty1 + uncertainty2;
712              }
713            }
714          }
715          /**
716           * OR blocks are calculated here
717           * The process is the same as with XOR blocks, but
718           * the formula is different
719           */
720        } else if (calculateArray[i][j].classification == "OR
              ") {
721          const probabilityArray = calculateArray[i][j]
722            .uncertaintyOfBranches.map(e => e.probability);
723          const calculatedProbabilityArray1 =
                probabilityArray
```

```
724                  .map(e => e * Math.log2(e));
725                const calculatedProbabilityArray2 =
                     probabilityArray
726                  .map(e => (1 - e) * Math.log2((1 - e)));
727                /**
728                 * The formula for OR blocks is split into 3 parts:
                        uncertainty1,
729                 * uncertainty2 and uncertainty3
730                 * They will be combined before adding the
                       calculated value to
731                 * the according elements
732                 */
733                const uncertainty1 = -1 *
                     calculatedProbabilityArray1
734                  .reduce((prev, cur) => prev + cur, 0);
735                const uncertainty2 = -1 *
                     calculatedProbabilityArray2
736                  .reduce((prev, cur) => prev + cur, 0);
737                const weightedUncertaintyArray = calculateArray[i][
                     j]
738                  .uncertaintyOfBranches.map(e => e.uncertainty * e
                        .probability);
739                const uncertainty3 = weightedUncertaintyArray
740                  .reduce((prev, cur) => prev + cur, 0);
741                for (let g = 0; g < calculateArray.length; g++) {
742                  for (let h = 0; h < calculateArray[g].length; h
                        ++) {
743                    if (calculateArray[i][j].recordedBranches
744                      .includes(calculateArray[g][h].pairid)) {
745                      /**
746                       * Removing inner blocks from the arrays
747                       */
748                      calculateArray[g][h].removed = true;
749                    }
750                    if (calculateArray[i][j].id == calculateArray[g
                         ][h].id) {
751                      /**
752                       * Combining the different parts of the
                           uncertainty and
753                       * storing it in the according elements
754                       */
755                      calculateArray[g][h].uncertainty =
756                        uncertainty1 + uncertainty2 + uncertainty3;
757                    }
758                  }
759                }
760                /**
761                 * LOOP blocks are calculated here
762                 */
763              } else if (calculateArray[i][j].classification == "
                  LOOP") {
764                const loopProbability = findLoopProbability(
                     calculateArray[i][j]
765                  .uncertaintyOfBranches);
766                /**
767                 * The formula has been split into 3 parts here as
                       well
768                 * As the formulas are defined right now, it
                       calculates the
769                 * uncertainty for the case that the loop has no
```

```
         looping limit
770   * If the uncertainty should be calculated for a
         specific looping
771   * limit, then the commented parts of the formulas
         have to be
772   * activated
773   * The looping limit goes where the 10 and 11 is
         right now
774   * If the looping limit is 5, then 10 has to be
         replaced by 5
775   * and 11 has to be replaced by 6
776   * See the actual formulas in the master thesis for
         more
777   * detailed information
778   */
779  const uncertainty1 = -1 * (1 /*- Math.pow(
        loopProbability, 10)*/)
780    * ((loopProbability * Math.log2(loopProbability))
         /
781      (1 - loopProbability) + Math.log2((1 -
           loopProbability)));
782  const uncertainty2 = ((1 /*- Math.pow(
        loopProbability, 11)*/)
783    / (1 - loopProbability)) *
784    findUncertaintyOfBranch(calculateArray[i][j]
785      .uncertaintyOfBranches, 1);
786  const uncertainty3 =
787    ((loopProbability /*- Math.pow(loopProbability,
         11)*/)
788      / (1 - loopProbability)) *
789    findUncertaintyOfBranch(calculateArray[i][j]
790      .uncertaintyOfBranches, 2);
791  for (let g = 0; g < calculateArray.length; g++) {
792    for (let h = 0; h < calculateArray[g].length; h
         ++) {
793      if (calculateArray[i][j].uncertaintyOfBranches
794        .map(m => m.id).includes(calculateArray[g][h
             ].id)) {
795        /**
796         * Removing inner blocks from the arrays
797         */
798        calculateArray[g][h].removed = true;
799      }
800      if (calculateArray[i][j].pairid ==
801        calculateArray[g][h].pairid) {
802        /**
803         * Combining the parts of the uncertainty and
              storing it
804         */
805        calculateArray[g][h].uncertainty =
806          uncertainty1 + uncertainty2 + uncertainty3;
807      }
808    }
809  }
810  /**
811   * AND blocks are calculated here
812   */
813 } else if (calculateArray[i][j].classification == "
      AND") {
814  for (let g = 0; g < calculateArray.length; g++) {
```

```
815                     for (let h = 0; h < calculateArray[g].length; h
                          ++) {
816                       if (calculateArray[i][j].recordedBranches
817                         .includes(calculateArray[g][h].pairid)) {
818                         /**
819                          * Removing inner blocks from the arrays
820                          */
821                         calculateArray[g][h].removed = true;
822                       }
823                       if (calculateArray[i][j].id == calculateArray[g
                            ][h].id) {
824                         /**
825                          * Calculating the uncertainty of the AND
                              block and
826                          * storing it
827                          * The formula for AND blocks is very simple
                              and therefore
828                          * is wasn't necessary to split it into parts
829                          */
830                         calculateArray[g][h].uncertainty =
                              calculateArray[i][j]
831                           .uncertaintyOfBranches.map(e => e.
                                uncertainty)
832                           .reduce((prev, cur) => prev + cur, 0);
833                       }
834                     }
835                   }
836                 }
837                 /**
838                  * Here the element which has been calculated in this
                      iteration
839                  * is marked by setting the boolean attribute "
                     calculated" to "true"
840                  * Also, only the splitting gateway of a calculated
                     block remains
841                  * in the arrays
842                  * Therefore, the merging gateways with the same
                     pairid are removed
843                  * by setting their boolean "removed" attribute to "
                     true"
844                  */
845                 for (let x = 0; x < calculateArray.length; x++) {
846                   for (let y = 0; y < calculateArray[x].length; y++)
                        {
847                     if (calculateArray[x][y].pairid == calculateArray
                          [i][j].pairid) {
848                       calculateArray[x][y].calculated = true;
849                     }
850                     if (calculateArray[x][y].id != calculateArray[i][
                          j].id &&
851                       calculateArray[x][y].pairid == calculateArray[i
                            ][j].pairid &&
852                       calculateArray[i][j].removed != true) {
853                       calculateArray[x][y].removed = true;
854                     }
855                   }
856                 }
857               }
858             }
859           }
```

```
860          /**
861           * Since a block has been calculated before, the counter
                attribute is
862           * reset for all elements in all arrays
863           * The counter attribute stores the information if
                uncalculated blocks
864           * reside in an outer block
865           * In the next iteration, this has to be evaluated again
                for every element
866           * Therefore this reset is necessary
867           */
868          for (let i = 0; i < calculateArray.length; i++) {
869            for (let j = 0; j < calculateArray[i].length; j++) {
870              calculateArray[i][j].counter = 0;
871            }
872          }
873          /**
874           * If all elements have been calculated, but there are
                still multiple
875           * elements in the arrays, then the remaining blocks are in
                 a sequence
876           * to each other
877           * Therefore, if all elements are marked as calculated, the
                 uncertainty
878           * of the remaining elements is combined and stored in one
                element
879           * This leads to the circumstance, that in the end, the
                total uncertainty
880           * of the business process is stored in one element
881           * All other elements are being removed in this process
882           */
883          for (let i = 0; i < filterRemoved.length; i++) {
884            for (let j = 0; j < filterRemoved[i].length; j++) {
885              if (filterRemoved[i][j + 1] != undefined &&
886                filterRemoved[i][j].calculated == true &&
887                filterRemoved[i][j].removed == false &&
888                filterRemoved[i][j + 1].calculated == true &&
889                filterRemoved[i][j + 1].removed == false &&
890                !filterRemoved[i][j].recordedBranches
891                  .includes(filterRemoved[i][j + 1].id)) {
892                for (let x = 0; x < calculateArray.length; x++) {
893                  for (let y = 0; y < calculateArray[x].length; y++)
                       {
894                    if (calculateArray[x][y].id == filterRemoved[i][j
                         ].id) {
895                      calculateArray[x][y].uncertainty +=
896                        filterRemoved[i][j + 1].uncertainty;
897                    }
898                    if (calculateArray[x][y].id == filterRemoved[i][j
                          + 1].id) {
899                      calculateArray[x][y].removed = true;
900                    }
901                  }
902                }
903              }
904            }
905          }
906        }
907      }
908      const filteredArray = calculateArray
```

```
909         .map(e => e.filter(m => m.removed == false));
910         const totalUncertainty = filteredArray[0].map(e => e.
              uncertainty)
911         .reduce((prev, cur) => prev + cur, 0);
912         return totalUncertainty;
913       }
914 
915       /**
916        * Helper function to return the uncertainty of a specific branch
              of a loop block
917        */
918       function findUncertaintyOfBranch(array, branch) {
919         for (let i = 0; i < array.length; i++) {
920           if (array[i].branch == branch) {
921             return array[i].uncertainty;
922           }
923         }
924         return 0;
925       }
926 
927       /**
928        * Helper function to return the loop-back probability of a loop
             block
929        */
930       function findLoopProbability(array) {
931         for (let i = 0; i < array.length; i++) {
932           if (array[i].branch == 2) {
933             return array[i].probability;
934           }
935         }
936         return 0.5;
937       }
938 
939     })();
940   }
941 
942   /**
943    * This section enables the drag and drop functionality of this
         application and was
944    * developed in the bpmn.io project that has been mentioned at the
         beginning
945    */
946   function registerFileDrop(container, callback) {
947     function handleFileSelect(e) {
948       e.stopPropagation();
949       e.preventDefault();
950       var files = e.dataTransfer.files;
951       var file = files[0];
952       var reader = new FileReader();
953       reader.onload = function (e) {
954         var xml = e.target.result;
955         callback(xml);
956       };
957       reader.readAsText(file);
958     }
959 
960     function handleDragOver(e) {
961       e.stopPropagation();
962       e.preventDefault();
963       e.dataTransfer.dropEffect = 'copy';
```

```
964     }
965     container.get(0).addEventListener('dragover', handleDragOver, false
          );
966     container.get(0).addEventListener('drop', handleFileSelect, false);
967   }
968
969   if (!window.FileList || !window.FileReader) {
970     window.alert(
971       'Looks like you use an older browser that does not support drag
            and drop. ' +
972       'Try using Chrome, Firefox or the Internet Explorer > 10.');
973   } else {
974     registerFileDrop(container, openDiagram);
975   }
```

Literatur

[1] C. E. Shannon, „A Mathematical Theory of Communication“, *Bell System Technical Journal*, Jg. 27, Nr. 3, S. 379–423, Juli 1948. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[2] J.-Y. Jung, C.-H. Chin und J. Cardoso, „An entropy-based uncertainty measure of process models“, *Information Processing Letters*, Jg. 111, Nr. 3, S. 135–141, Jan. 2011. DOI: 10.1016/j.ipl.2010.10.022.

[3] M. Mitchell, *Complexity a guided tour*. Oxford University Press, 2009.

[4] OMG, *Business Process Model and Notation (BPMN) Version 2.0*, Jan. 2011.

[5] Camunda, *Web-based tooling for BPMN, DMN and CMMN*. Adresse: https://bpmn.io/.

[6] O. Foundation, *About*. Adresse: https://nodejs.org/en/about/.

[7] J. Recker und J. Mendling, „On the translation between BPMN and BPEL: Conceptual mismatch between process modeling languages“, in *Proceedings of the Workshops and Doctoral Consortium*, Presses universitaires de Namur (Namur University Press), 2006, S. 521–532.

[8] B. T. Nguyen, D. H. Nguyen und T. T. Nguyen, „Translation from BPMN to BPEL, current techniques and limitations“, in *Proceedings of the Fifth Symposium on Information and Communication Technology*, 2014, S. 21–30.

[9] A. N. Kolmogoroff, *Grundbegriffe der Wahrscheinlichkeitsrechnung*. Springer-Verl, 1973.

[10] C. Adami, „What is complexity?“, *BioEssays*, Jg. 24, Nr. 12, S. 1085–1094, 2002.

[11] H. A. Simon, „The architecture of complexity“, in *Facets of systems science*, Springer, 1991, S. 457–476.