Effiziente Traversierung von Zustandsbäumen zur Prüfung von Wahrscheinlichkeitsgrenzen in Hybriden Petri Netzen

Bachelorarbeit
zur Erlangung des akademischen Grades
Bachelor of Science

Westfälische Wilhelms-Universität Münster
Fachbereich Mathematik und Informatik
Institut für Informatik

Betreuung:
Prof. Dr. Remke
Prof. Dr. Vahrenhold

Eingereicht von:
Christopher Distelkämper

Münster, April 2015

Zusammenfassung

Das bereits bestehende DFPN-Tool, welches der Berechnung von Wahrscheinlichkeiten zu Ereignissen innerhalb eines HPnG-Netzes dient, wurde im Rahmen dieser Bachelorarbeit untersucht.

Statt wie bisher, zu jedem möglichen Ereignis im Petri-Netz die Wahrscheinlichkeit zu berechnen, ist der Ansatzpunkt eine Aussage über den Wahrscheinlichkeitsbereich des Ereignisses zu treffen. Die für die Berechnung der Wahrscheinlichkeit notwendigen Zustände, des zum Petri-Netz gehörigen Zustandsraumes, werden dabei nur bei Bedarf berechnet. Zum Auswählen der richtigen Zustände wurden verschiedene Auswahlverfahren untersucht und analysiert.

Das Konzept wurde implementiert und getestet, sodass eine deutliche Steigerung der Effizienz des Tools festgestellt werden konnte.

Danksagung

An dieser Stelle möchte ich mich zunächst bei alldenjenigen bedanken, die mich während der Anfertigung meiner Bachelor-Arbeit unterstützt haben.

Ein besonderer Dank gilt dabei Frau Prof. Dr. Remke, die mich während der Arbeit betreut hat. Sie hat mich in die bestehenden Strukturen eingeführt und mir während der Anfertigung der Arbeit wertvolle Hinweise und Ratschläge gegeben. Desweiteren möchte ich mich bei Herrn Prof. Dr. Vahrenhold bedanken, der sich mir angenommen und das Thema für die Bachelorarbeit vermittelt hat.

Auch möchte ich mich bei meiner Freundin, Jana Jüttner, bedanken, die mir bei der Korrektur meiner Arbeit geholfen hat und mich die ganze Zeit über seelisch unterstützt hat.

Desweiteren möchte ich mich bei meinen Eltern bedanken, die mich während meines ganzen Studiums nicht nur finanziell unterstützt haben, sondern mir immer mit Rat und Tat zur Seite standen.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Motivation . . . 1
1.2. Zielsetzung . . . 1
1.3. Aufbau der Arbeit . . . 2

2. Petri-Netze **3**
2.1. Diskrete Petri-Netze . . . 3
2.2. Kontinuierliche Petri-Netze . . . 6
2.3. Hybride Petri-Netze . . . 8

3. Modellklasse HPnG **11**
3.1. Erweiterung des hybriden Petri-Netzes . . . 11
3.2. Mathematische Beschreibung . . . 12
3.3. Beispiel . . . 13

4. Zustandsraum **15**
4.1. Einführung . . . 15
4.2. Berechnung des Zustandsraumes . . . 15
4.3. Rekursive Berechnung des Zustandsraumes . . . 17

5. Effiziente Zustandsraumerzeugung **19**
5.1. Wahrscheinlichkeitsschranken . . . 19
5.2. Berechnung von Wahrscheinlichkeiten . . . 20
5.2.1. Berechnung des Gültigkeitsbereiches . . . 21
5.2.2. Berechnung der Wahrscheinlichkeit . . . 22
5.3. Eulertour . . . 24
5.4. Heuristiken . . . 26

6. Implementierung **29**
6.1. Tour durch den Zustandsgraphen . . . 29
6.2. Ausgeben von Zuständen . . . 30
6.3. Berechnung des Gültigkeitsbereiches . . . 31
6.4. Berechnung von Wahrscheinlichkeiten . . . 35
6.5. Berechnung des Wahrscheinlichkeitsbereiches . . . 37

7. Case Study **41**
7.1. Allgemeines . . . 41

7.2. Direkter Vergleich von Heuristiken 41
7.3. Allgemeiner Vergleich 44
7.3.1. Vergleiche im modifizierten Algorithmus 44
7.3.2. Vergleich mit dem ursprünglichen Algorithmus 46

8. Zusammenfassung 47

A. Anhang 49

Abbildungsverzeichnis 51

Tabellenverzeichnis 53

Listings 55

Literatur 57

1. Einleitung

In diesem Kapitel wird zunächst die Motivation besprochen, sich mit der Thematik zu beschäftigen, ehe die Zielsetzung dieser Bachelorarbeit erklärt wird. Zu guter Letzt liefert dieses Kapitel einen kurzen Überblick über den Aufbau der Arbeit.

1.1. Motivation

Die Informatik versucht anhand von vereinfachten Modellen die reale Welt so gut es geht darzustellen. Mit ihr sollen Problemstellungen vereinfacht werden können und Analysen sollen aufgrund ihrer Berechnungen helfen, Lösungen für Probleme der realen Welt zu finden.

So stellen die sogenannten Petri-Netze Arbeitsabläufe in der realen Welt dar. Eine Erweiterung der Petri-Netze, die sogenannten HPnG-Netze, werden benutzt um zum Beispiel Versorgungswerke, wie Stromkraft- oder Wasserwerke dazustellen. Da durch Wartung oder Defekte z.B. Maschinen ausfallen können und dennoch der Verbraucher ausreichend versorgt sein soll, ist es wichtig die Konsequenzen eines Ausfalls zu kennen. Die Simulation dafür kann durch die HPnG-Netze geschehen.

1.2. Zielsetzung

Im Rahmen dieser Bachelorarbeit soll das bereits bestehende DFPN-Tool, welches für die Analyse der HPnG-Netze genutzt wird, erweitert werden.

Statt der exakten Berechnung der Wahrscheinlichkeit jedes möglichen Ereignisses des Petri-Netzes, soll eine Aussage über den Wahrscheinlichkeitsbereich eines Ereignisses getroffen werden können. Der Vorteil dieses Konzeptes ist, dass die Zustände des Petri-Netzes nur bei Bedarf berechnet werden müssen. So wird der Zustandsraum nicht komplett berechnet, was zur Verringerung der Rechenzeit führt. Die zu berechnenden Zustände innerhalb des Zustandsraumes sollen möglichst gut gewählt werden, um nur wenige Zustände berechnen zu müssen. Dazu sollen verschiedene Auswahlverfahren implementiert und bewertet werden.

1.3. Aufbau der Arbeit

In den folgenden Kapiteln werden zunächst die Grundlagen für die Thematik behandelt. Kapitel 2 beschäftigt sich mit den Varianten von Petri-Netzen. Kapitel 3 setzt auf diesen auf und erweitert das hybride Petri-Netz um stochastische Transitionen. In Kapitel 4 wird erklärt, wie Zustandsräume zu HPnG-Netzen berechnet werden. Kapitel 5 stellt das Konzept zur Umsetzung der Zielsetzung vor und erklärt die Berechnung von Wahrscheinlichkeiten in einem Zustandsraum, ehe in Kapitel 6 die Implementierung des Konzeptes vorgestellt wird. Im Anschluss daran vergleicht Kapitel 7 die verschiedenen Heuristiken untereinander und weißt auf Vor- und Nachteile der Heuristiken hin. Das letzte Kapitel fasst noch einmal die Bachelorarbeit zusammen und gibt einen kurzen Ausblick.

2. Petri-Netze

Dieses Kapitel gibt eine Einführung in das Modell der Petri-Netze. Petri-Netze lassen sich in drei verschiedene Klassen einteilen, diskret, kontinuierlich und hybrid. Die nachfolgenden Abschnitte erläutern die verschiedenen Modellklassen und dienen als Grundlage für die Modellklasse der *Hybriden Petri-Netze mit stochastischen Transitionen* (kurz HPnG), welche im folgenden Kapitel 3 auf Seite 11 betrachtet werden.

2.1. Diskrete Petri-Netze

Dieser Abschnitt erklärt den Aufbau und die Regeln des klassischen diskreten Modells des Petri-Netzes. Dazu wird nach einer kurzen Einführung, der Aufbau eines Petri-Netzes erklärt. Anschließend werden die Schaltregeln erläutert. Ein Beispiel zum Abschluss veranschaulicht die Theorie. Im Anschluss an diesen Abschnitt, werden die leicht abgewandelten Varianten des Petri-Netzes, welche auf diesem diskreten Modell basieren, vorgestellt.

Historie
Das ursprüngliche Modell des Petri-Netzes wurde von dem Informatiker C. Petri[1] im Jahr 1962 entwickelt und basiert auf dem Modell des endlichen Automaten. Das Modell beschreibt den Ablauf von Schaltvorgängen und dient als Unterstützung in der Modellierung von Prozess- oder Arbeitsabläufen. Neben der Verwendung in der Informatik, wird das Modell des Petri-Netzes, zum Teil angepasst, auch in anderen Fachbereichen, wie zum Beispiel der theoretischen Biologie, aber auch im Maschinenbau, eingesetzt. Zusätzlich verwenden zum Beispiel Aktivitätsdiagramme, aber auch andere Modellierungsformen, Prinzipien des Petri-Netzes (nach [DA05a]).

Aufbau
Das Petri-Netz ist ein gerichteter Graph. Es besteht aus sogenannten Plätzen und Transitionen. Diese Objekte stellen die Knoten in dem Graphen dar. Plätze und Transitionen können durch gerichtete Kanten miteinander verbunden werden, wobei ein Knoten ohne ein- bzw. ausgehende Kanten dabei nicht erlaubt ist. Außerdem gibt es sogenannte Marken, die in den Plätzen liegen. Grafik 2.1 zeigt die Symbole, welche im Graphen eines diskreten Petri-Netzes verwendet werden. Eine Marke kann dafür stehen, dass der durch die Plätze

[1] Carl Adam Petri * 12. Juli 1926 in Leipzig

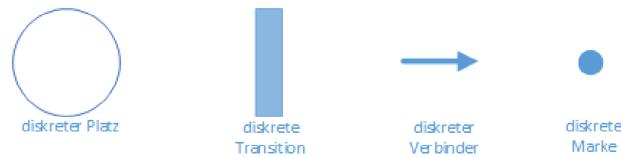

Abbildung 2.1.: Symbole des diskreten Petri-Netzes

symbolisierte Zustand aktiv ist. Die Transitionen stellen in dem Netz Schalter dar. Sie können die Marke von einem Platz zu einem anderen bewegen.

Mathematische Beschreibung

Mathematisch kann das obige Petri-Netz durch ein Tripel (S, T, F) beschrieben werden. S gibt die endliche Menge aller Plätze an, die in dem System vorhanden ist. T bezeichnet die endliche Menge aller Transitionen des Graphen und F ist die Menge der benutzen Kanten, also ist $F \subset S \times T \cup T \times S$. Eine Markenbelegung des Petri-Netz ist eine Abbildung $\sigma : S \to \mathbb{N}$, welcher jedem Platz des Graphen eine Anzahl an Marken zuordnet.

Definition 2.1. *Der* ***Vorbereich*** *eines Platzes oder einer Transition ist die Menge aller Knoten, welche eine Kante in Richtung des beobachteten Knoten besitzen.*

Den Vorbereich eines Platzes oder einer Transition gibt die Menge $\bullet v = \{w \in S \cup T \mid (w, v) \in F\}$ an, wobei v aus der Menge $S \cup T$ stammt. Äquivalent wird der Nachbereich eines Platzes oder einer Transition durch die Menge $v\bullet = \{w \in S \cup T \mid (w, v) \in F\}$ angegeben.

Schaltregeln

Allgemeine Schaltregeln: Im Petri-Netz kann jeder Platz des Graphen nur maximal eine Marke enthalten. Eine Transition kann genau dann schalten, wenn jeder Platz im Vorbereich eine Marke enthält und jeder Platz im Nachbereich leer ist. Es muss also gelten:

$$\forall s \in \bullet t : \sigma(s) = 1 \wedge \forall s \in t\bullet : \sigma(s) = 0.$$

Hat eine Transition geschaltet, wird aus jedem Platz im Vorbereich die vorhandene Marke herausgenommen und zu jedem Platz des Nachbereichs eine Marke hinzugefügt. Der Zustand, der aus σ folgt, wird folgendermaßen berechnet:

$$\sigma'(s) = \begin{cases} 0, & \text{falls } s \in \bullet t \\ 1, & \text{falls } s \in t\bullet \\ \sigma(s), & \text{sonst.} \end{cases}$$

Spezialfälle: Hat die Transition einen leeren Vorbereich, also gibt es keine Kante, welche von einem Platz zu der Transition hinführt, so wird, beim Schalten

der Transition, eine neue Marke erzeugt und in die Stellen des Nachbereiches gelegt. Besitzt eine Transition keine Plätze im Nachbereich, so werden lediglich Marken aus dem Vorbereich entfernt (vgl. [DA05b]).

Beispiel
Anhand von folgendem Beispiel (Abb. 2.2) wird die Funktionsweise eines Petri-Netzes noch einmal dargestellt. Das Netz besteht aus den Plätzen P1, P2 und

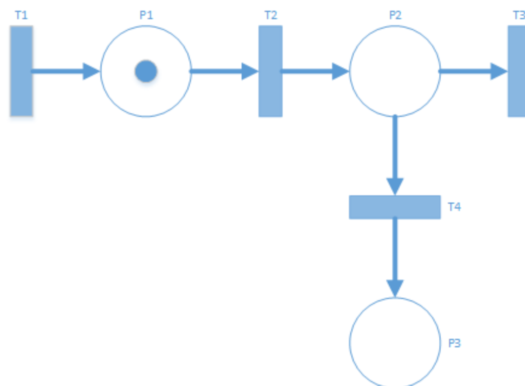

Abbildung 2.2.: Allgemeines Petri-Netz: Ausgangssituation

P3, sowie der Transitionen T1, T2, T3 und T4. Lediglich der Platz P1 hält eine Marke. Um den nächsten Zustand des Petri-Netzes zu berechnen, wird überprüft, welche Transitionen als nächstes schalten können. T1 kann nicht schalten, da der Platz P1 im Nachbereich bereits eine Marke hält. Das Schalten der Transition T2 ist möglich, denn der Platz im Vorbereich P1 enthält eine Marke und der Platz des Nachbereiches P2 ist leer. Die beiden Transitionen T3 und T4 können nicht schalten, da der Platz des Vorbereichs P2 keine Marke hält. Die einzige Möglichkeit ist somit die Transition T2 zu schalten. Abbildung 2.3 zeigt den daraus resultierenden Zustand des Petri-Netzes.

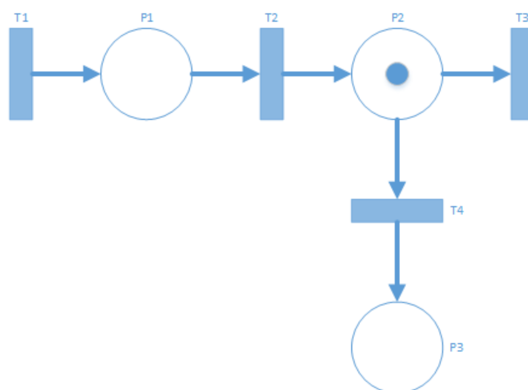

Abbildung 2.3.: Allgemeines Petri-Netz: erster Schritt

Da die Transition T2 geschaltet hat, wurde die Marke aus dem Platz P1 entfernt und P2 hinzugefügt. Nun ist es möglich, dass sowohl T1, T3 und T4

schalten, lediglich T2 darf nicht schalten, da der Vorbereich leer ist und der Nachbereich eine Marke enthält. In diesem Fall darf beliebig eine der Transitionen T1, T3 oder T4 schalten. Durch Priorisieren der Transitionen könnte jedoch eine Reihenfolge festgelegt werden.

2.2. Kontinuierliche Petri-Netze

Das kontinuierliche Petri-Netz ist dem diskreten Petri-Netz sehr ähnlich. Der große Unterschied ist, dass der Inhalt von Plätzen stetig steigt oder sinkt. Der Inhalt der Plätze lässt sich somit nicht mehr durch natürliche Zahlen, sondern nur noch durch reelle Zahlen beschreiben. Diese Veränderung zieht einige Änderungen am Aufbau des Netzes und an den Schaltregeln nach sich, welche in diesem Abschnitt erklärt werden. Abgeschlossen wird dieser Abschnitt durch ein kurzes Beispiel.

Aufbau
Der Aufbau des kontinuierlichen Petri-Netzes unterscheidet sich nur geringfügig von dem des diskreten Petri-Netzes. Es besteht aus Plätzen und Transitionen, sowie einer Kante, welche Plätze und Transitionen miteinander verbindet. Um die diskreten und kontinuierlichen Symbole unterscheiden zu können, sind die kontinuierlichen Symbole doppelt umrandet, siehe dazu die nachfolgende Grafik 2.4. Da es für das kontinuierliche Petri-Netz keine vorgeschriebenen

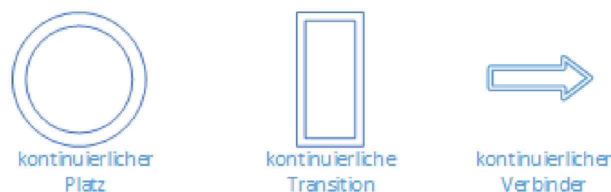

Abbildung 2.4.: Symbole des kontinuierlichen Petri-Netzes

Grenzen für den Inhalt der Plätze gibt, muss für jeden Platz eine obere Kapazitätsgrenze B angegeben werden. Die Transitionen können ebenfalls einen unterschiedlichen Bedarf haben, dieser wird ebenfalls über eine Angabe von einer Rate R angegeben.

Schaltregeln
Die Schaltregeln des kontinuierlichen Petri-Netzes sind ähnlich denen des diskreten Netzes. Der Unterschied ist eigentlich nur, das kontinuierlich die Inhalte der Plätze verändert werden. Die Transitionen schalten also nicht diskret, sondern man kann sie sich als Rohre in einem Brunnensystem vorstellen. Ein kontinuierliches Modell kann mit Hilfe eines diskreten Modells approximiert werden, indem die diskreten Transitionen in Intervallen schalten. Je kürzer

das Intervall ist, desto näher kommt das diskrete Modell an das Kontinuierliche heran (nach [DA05c]).

Beispiel
Das folgende Beispiel 2.5 zeigt ein System mit 2 Plätzen, welche durch eine Transition verbunden sind, eine weitere Transition leert den Platz P2. P1 hält zu Anfang 9 Einheiten, P2 ist noch leer. Die Transition T1 übergibt pro Zeitschritt 3 Einheiten von P1 nach P2, T2 leert P2 um eine Einheit pro Zeitschritt. Von Beginn an wird P1 geleert. P1 hat einen Inhalt von 9 Einheiten, es

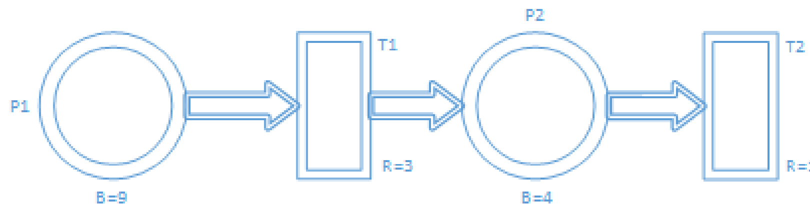

Abbildung 2.5.: Beispiel eines kontinuierlichen Petri-Netzes

besitzt keine eingehenden Kanten und wenn möglich leert Transition 1 P1 um 3 Einheiten pro Zeitschritt. Platz P2 nimmt die Einheiten von P1 auf und gibt eine Einheit pro Zeitschritt ab. Jedoch kann P2 nur 4 Einheiten aufnehmen. Zum Zeitpunkt $T = 2$ ist die Obergrenze erreicht, denn $2 * (3 - 1) = 4$. Ab diesem Zeitpunkt kann T1 nur noch eine Einheit pro Zeitschritt überführen, da P2 nicht mehr Einheiten aufnehmen kann. Die folgende Grafik 2.6 zeigt den Verlauf des Inhaltes eines Platzes zur Zeit. Dabei stellt die x-Achse die Zeit und die y-Achse die Menge des Inhaltes eines Platzes dar. Der Inhalt des

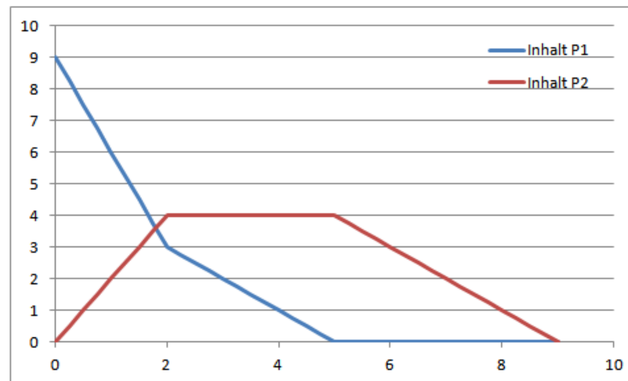

Abbildung 2.6.: Verlauf des Inhaltes der Plätze P1 und P2

Platzes P1 sinkt kontinuierlich, jedoch ab dem Zeitpunk 2 langsamer, da die obere Kapazitätsgrenze von P2 erreicht ist. Der Inhalt von P2 steigt zunächst an, solange bis die Kapazitätsgrenze erreicht ist. Zum Zeitpunkt 5, wo P1 leer ist, beginnt sich der Inhalt des Platzes P2 zu reduzieren. Zum Zeitpunkt 9 sind beide Plätze leer.

2.3. Hybride Petri-Netze

Das hybride Petri-Netz vereint das diskrete Petri-Netz mit dem im vorherigen Abschnitt vorgestellten kontinuierlichen Bereich. Neben den in beiden Netzen verwendeten Symbolen wird noch ein weiteres neues Symbol, der Tester, benötigt. Alle möglichen Symbole zeigt die Grafik 2.7. Der Tester dient dazu,

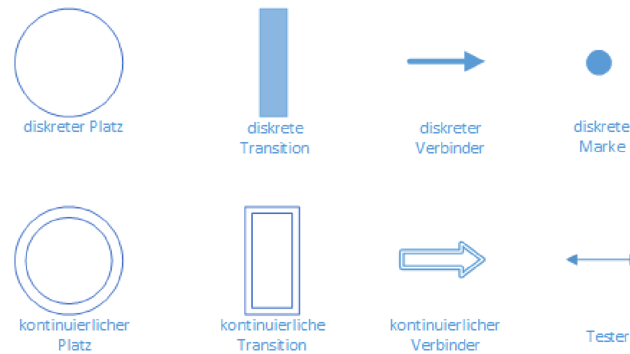

Abbildung 2.7.: Symbole des hybriden Petri-Netzes

Abhängigkeiten zwischen dem diskreten und kontinuierlichen Bereich des Netzes darzustellen. Sie verbindet einen diskreten Platz mit einer kontinuierlichen oder diskreten Transition. Falls in dem diskreten Platz keine Marke enthalten ist, so ist die Transition auch nicht aktiv, ist jedoch eine Marke enthalten, ist die Transition, wenn alle weiteren Bedingungen erfüllt sind, aktiv.

Schaltregeln

Die diskreten und kontinuierlichen Transitionen schalten wie in den vorherigen Abschnitten beschrieben. Enthält der diskrete Platz, welcher durch einen Tester mit einer kontinuierlichen Transition verbunden ist, eine Marke, so kann diese kontinuierliche Transition aktiv sein. Äquivalent dazu verhält es sich mit den, mit einem Tester verbundenen, diskreten Transitionen (vgl. [DA05d]).

Beispiel

Die folgende Abbildung 2.8 zeigt ein einfaches hybrides Petri-Netz. In der Mitte der Grafik ist der kontinuierliche Bereich des Petri-Netzes zu sehen. Die diskreten Bereiche oben und unten steuern die Pumpen. Dargestellt wird ein Wasserkreislauf, das Wasser ist enthalten in P1 mit 10 Einheiten und wird durch T1 zu P2 gepumpt. P2 enthält zu Beginn noch kein Wasser. Über die Pumpe T2 wird das Wasser wieder zurück nach P1 gepumpt. Die Pumpe T2 ist aktiv, solange die Transition T3 nicht schaltet, da dann die Marke aus dem Platz *Pump 2* entfernt wird. Die Pumpe T1 kann manuell, durch die Transitionen T on und T off, ein- und ausgestellt werden.
Schaltet T on, so beginnt Wasser von P1 nach P2 zu fließen, da jedoch T2 auch aktiv ist, wird wieder ein Teil zurückgepumpt. Da die Pumpe T1 mit 2 Wassereinheiten/Zeiteinheit pumpt und T2 nur mit 1 Wassereinheit/Zeiteinheit,

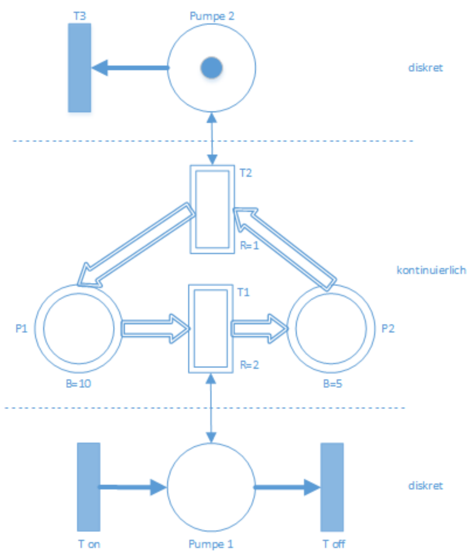

Abbildung 2.8.: Hybrides Petri-Netz

füllt sich P2 langsam. Ändert sich nichts an der Konstellation, so ist P2 nach 5 Zeiteinheiten mit Wasser gefüllt. Von diesem Augenblick an, passt sich die Förderrate von T1 an die von T2 an. Durch Schalten von Pumpe 2, kann kein Wasser mehr aus P2 abgepumpt werden, dadurch liegt nun auch die Förderrate von T1 bei 0 Wassereinheiten/Zeiteinheit. Es sind durch Veränderungen der Schaltvorschriften zu T on und T off andere Szenarien möglich.

3. Modellklasse HPnG

In diesem Kapitel wird die Klasse der *hybriden Petri-Netze mit stochastischen Transitionen* (HPnG) vorgestellt. Das Modell der HPnG basiert hauptsächlich auf den hybriden Petri-Netzen, welche in Abschnitt 2.3 auf Seite 8 definiert wurden. Zunächst werden die Unterschiede zum hybriden Petri-Netz aufgezeigt, ehe der Formalismus in einem Beispiel näher betrachtet wird.

3.1. Erweiterung des hybriden Petri-Netzes

Die Modellklasse der HPnGs unterscheidet sich nur geringfügig von dem Modell der hybriden Petri-Netzen. Zu der Funktionalität des hybriden Petri-Netzes wurde lediglich die stochastische Transition hinzugefügt. Alle für das HPnG-Modell relevanten Symbole zeigt die Grafik 3.1. Die stochastischen Transitio-

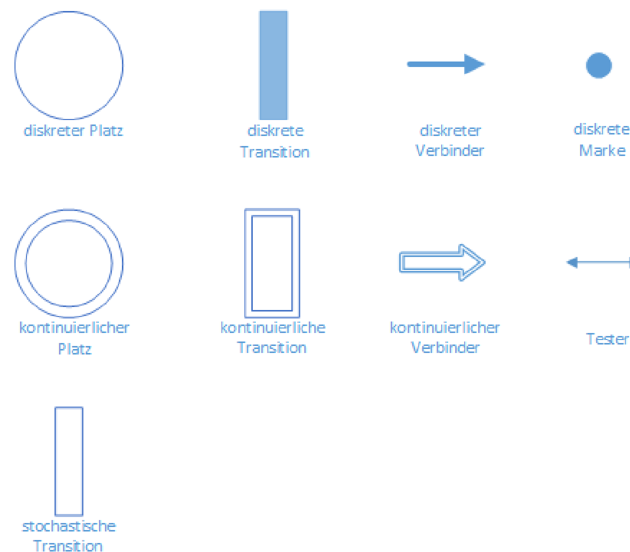

Abbildung 3.1.: Symbole des HPnG-Modells

nen werden im diskreten Bereich des Petri-Netzes eingesetzt. Das besondere an diesen Transitionen ist, dass sie, nicht zu einem bestimmten Zeitpunkt, sondern zufällig, nach einer beliebigen Wahrscheinlichkeitsverteilung schalten. Stochastische Transitionen schalten wie diskrete Transitionen, können jedoch nur einmal im Verlauf der Simulation schalten. Desweiteren ist noch eine direkt schaltende (immediate) Transition möglich, diese wird jedoch vernachlässigt,

da sie durch eine deterministische Transition mit der Schaltzeit $T = 0$ dargestellt werden kann.

Definition 3.1. *Der Ablauf einer Simulation ist* ***deterministisch****, wenn der folgende Handlungsschritt eindeutig durch das Modell definiert ist (vgl. [Bol06]).*

Durch diese Änderung, wird aus den nicht deterministischem hybriden Petri-Netz, die Modellklasse HPnG, in welcher sich meistens durch Wahrscheinlichkeiten die nicht Determiniertheit auflösen lässt.

Nicht Determinismus - Determinismus
Im bisher betrachteten hybriden Petri-Netz, konnten die Schaltungen in beliebiger Reihenfolge beliebig oft schalten, vorausgesetzt die Bedingungen für die Transitionen sind erfüllt. Da also der nächste Schritt nicht unbedingt vorhersagbar ist, handelt es sich um ein nicht deterministisches Modell. Das HPnG-Modell ist zwar nicht immer deterministisch, jedoch lässt sich dies durch Wahrscheinlichkeiten auflösen, was in folgendem Beispiel 3.2 gezeigt wird. Die Tran-

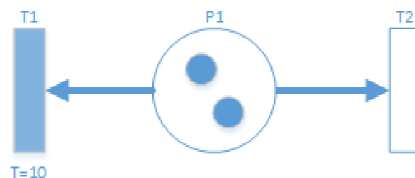

Abbildung 3.2.: Beispiel zu den Bedingungsfolgen

sition T1 schaltet zum Zeitpunkt $T = 10$, jedoch ist es nicht bekannt, wann T2 schaltet. Die Transition T2 kann somit vor oder nach T1 schalten, was nicht deterministisch ist. Falls T1 zuerst schaltet, muss T2 als nächstes Schalten, genauso im umgekehrten Fall.
Die Reihenfolge der Schaltungen kann durch Wahrscheinlichkeiten angegeben werden. So lässt sich eine Aussage treffen, zu welcher Zeit welcher Platz einen bestimmten Wert mit welcher Wahrscheinlichkeit hält. Wie die Wahrscheinlichkeiten berechnet werden können, wird in Abschnitt 5.2 beschrieben.

3.2. Mathematische Beschreibung

Das HPnG-Modell ist als Tupel $(\mathcal{P}, \mathcal{T}, \mathcal{A}, m_0, x_0, \phi)$ definiert. $\mathcal{P}$ steht dabei für die Menge der Plätze, $\mathcal{T}$ für die Menge der Transitionen und $\mathcal{A}$ für die Menge der Verbindungen zwischen den Transitionen und Plätzen im betrachteten Netz. Die Initialinhalte der Plätze werden durch m_0 und x_0 angegeben, wobei m_0 die Inhalte der diskreten und x_0 die Inhalte der kontinuierlichen Plätze beschreibt. ϕ gibt die verschiedenen Eigenschaften der Plätze und Transitionen,

es besteht aus 9 Funktionen, welche unter anderem die Obergrenze der Inhalte eines Platzes, die Priorität oder die Gewichtung einer Transition angeben (vlg. [GR10]).

3.3. Beispiel

Im folgenden wird ein leicht abgewandeltes Petri-Netz aus 2.3 betrachtet. So wurden die beiden diskreten Transitionen T on und T off durch stochastische Transitionen ersetzt. Außerdem wurde zur diskreten Transition T3 ein fester Schaltzeitpunkt $T = 10$ hinzugefügt. Abbildung 3.3 zeigt das Petri-Netz. Die

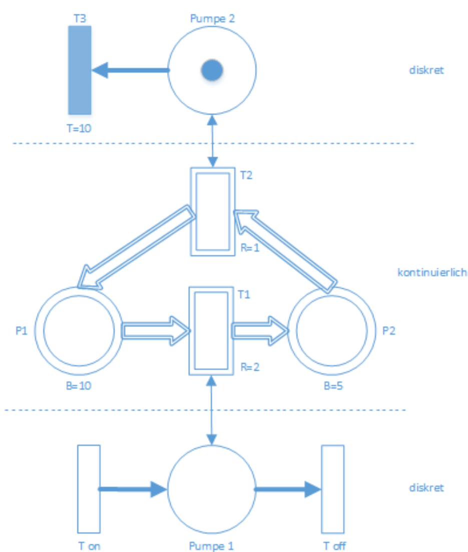

Abbildung 3.3.: Petri-Netz der Modellklasse HPnG

Pumpe T1 kann manuell, durch die Transitionen T on und T off, ein- und ausgestellt werden. Schaltet T on, so beginnt Wasser von P1 nach P2 zu fließen, da jedoch T2 auch aktiv ist, wird wieder ein Teil zurückgepumpt. Da die Pumpe T1 mit 2 Wassereinheiten/Zeiteinheit pumpt und T2 nur mit 1 Wassereinheit/Zeiteinheit, füllt sich P2 langsam. Ändert sich nichts an der Konstellation, so ist P2 nach 5 Zeiteinheiten mit Wasser gefüllt. Von diesem Augenblick an, passt sich die Förderrate von T1 an die von T2 an. Nach 10 Zeiteinheiten wird T2 deaktiviert, da die diskrete Transition T3 schaltet. Da kein Wasser mehr aus P2 abgepumpt werden kann, liegt nun auch die Förderrate von T1 bei 0 Wassereinheiten/Zeiteinheit.

4. Zustandsraum

Dieses Kapitel handelt von Zustandsräumen, die aus den, im vorherigen Kapitel 3 vorgestellten, hybriden Petri-Netzen mit stochastischen Transitionen berechnet werden können. In diesem Kapitel wird zu Anfang erklärt, was ein Zustandsraum ist, ehe die Berechnung des Zustandsraumes beschrieben wird. Zu guter Letzt wird der rekursive Algorithmus des *DFPN*-Tools vorgestellt, auf dem diese Bachelorarbeit aufsetzt und beschrieben, wie das Tool die Berechnung der Zustandsräume durchführt.

4.1. Einführung

Ein Zustandsraum besteht aus mehreren Zuständen. Dabei enthält ein Zustand innerhalb des Zustandsraum enthält verschiedene Informationen. So wird zu jedem Platz der momentane Inhalt gespeichert, sowie zu den stochastischen Ttransitionen unter anderem, ob sie bereits geschaltet haben. Außerdem erhält ein Zustand Informationen über das Intervall in dem dieser Zustand gültig ist, sowie der Zeitpunkt, ab dem der Zustand gültig ist.
Allgemein wird ein Zustandsraum durch ein Tupel $[\mathcal{N}, \mathcal{A}, \mathcal{S}, \mathcal{G}]$ dargestellt. $\mathcal{N}$ ist dabei die Menge aller Zustände und $\mathcal{A}$ die Menge aller Verbindungskanten der Zustände. $\mathcal{S}$ ist der Startzustand des HPnGs. $\mathcal{G}$ bezeichnet die Menge aller Zielzustände.

4.2. Berechnung des Zustandsraumes

Zur Berechnung eines Zustandsraumes ist es notwendig das zugehörige HPnG-Netz zu kennen. Abbildung 4.1 zeigt das Petri-Netz, welches im weiteren

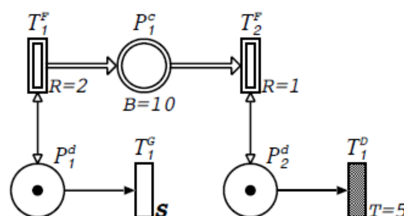

Abbildung 4.1.: HPnG Petri-Netz[1]

[1]aus dem Paper "Hybrid Petri nets with general one-shot transitions" Seite 23

Verlauf dieses Kapitels betrachtet wird. Aus diesem HPnG-Netz lässt sich ein Zustandsraum ableiten. Der erste Schritt ist die Initialbelegung als Wurzelknoten des Zustandsraumes zu wählen. Die direkt erreichbaren Zustände ergeben sich aus Schaltvorgängen innerhalb des Petri-Netzes oder durch das Erreichen von Ober- oder Untergrenzen bei kontinuierlichen Plätzen. Die folgende Abbildung 4.2 zeigt den Zustandsraum zum HPnG aus 4.1 Die einzelnen Zustände

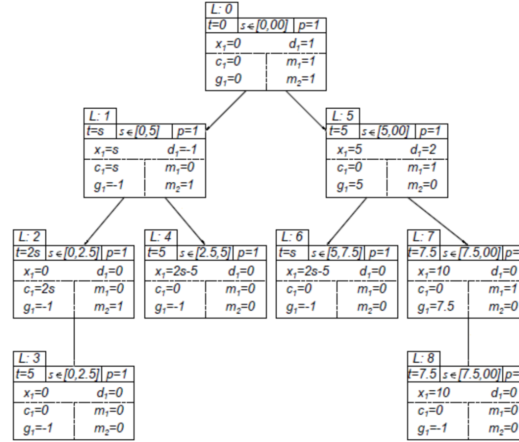

Abbildung 4.2.: Zustandsbaum berechnet durch das DFPN-Tool[2]

des Zustandsraumes enthalten alle für die Analyse oder die Berechnung von erreichbaren Zuständen wichtigen Informationen. Jeder Zustand erhält eine in L gespeicherte eindeutige Zustandsnummer, daneben gibt t den Zeitpunkt des Zustandes an. Ein Hilfsintervall für die Berechnung von Wahrscheinlichkeiten wird durch s angegeben und die Wahrscheinlichkeit, mit der man diesen Zustand besucht, gibt p an. Der Inhalt der Plätze P_1^c, P_1^d und P_2^d wird durch die Variablen x_1, m_1 und m_2 angegeben. Die Variable d_1 gibt die momentane Änderungsrate des kontinuierlichen Platzes P_1^c an, c_1 gibt an, wie viele Zeiteinheiten die Transition T_1^D schon wartet und g_1 gibt an, ob die stochastische Transition bereits geschaltet hat, dabei steht der Wert -1 dafür, dass die Transition geschaltet hat. Der berechnete Zustandsraum hat eine Baumstruktur, welche typisch für Zustandsräume der HPnG's ist. Der Wurzelknoten gibt den Initialzustand des Petri-Netzes an. Da es nicht sicher ist, ob die deterministische Transition T_1^D zum Zeitpunkt $T = 5$ oder die stochastische Transition T_1^G als erstes schaltet, zweigen vom Startzustand zwei Zustände ab. Von diesen Zuständen zweigen wiederum weitere Zustände ab, weil entweder die jeweils andere Transition schaltet oder weil eine Unter- bzw. Obergrenze des kontinuierlichen Platzes P_1^c erreicht wird.

[2]aus dem Paper "Hybrid Petri nets with general one-shot transitions" Seite 24

4.3. Rekursive Berechnung des Zustandsraumes

Die Berechnung des Zustandsraumes erfolgt am einfachsten rekursiv.

Definition 4.1. *Eine **Rekursion** ist, wenn eine Funktion in ihrer Definition sich selbst aufruft. Das Ziel einer Rekursion ist es, ein zu lösendes Problem bei jedem Aufruf zu verringern, bis es im Verlauf einfach lösbar wird. Theoretisch lassen sich auf diese Weise auch endlose Folgen von Funktionsaufrufen einfach umsetzten (nach [Har15]).*

Dazu wird ausgehend vom Initialzustand getestet, welche Folgezustände berechnet werden können. Für jeden Folgezustand wird dann wieder erneut getestet, welche Zustände berechnet werden können. Dies geschieht solange, bis alle Zustände berechnet wurden und keine weiteren Folgezustände möglich sind, oder bis eine festgelegte maximale Zeit im Modell erreicht wurde.
Der Algorithmus zur Berechnung ist in der Funktion *produceNext* des, für diese Bachelorarbeit zugrunde liegenden, *DFPN*-Tools implementiert. Das Programm wurde von Frau Prof. A. Remke und den Herren B. Postema, sowie H. Ghasemieh entwickelt und wird weiterhin verbessert und um neue Funktionen bereichert. Es kann unter der folgenden URL[3] heruntergeladen werden und ist in den Programmiersprachen C/C++ geschrieben.
Die Funktion *produceNext* arbeitet mit zwei verschiedenen implementierten Datentypen, dem State, welches einen Zustand darstellt und dem Model, welches das Petri-Netz bereitstellt. Das State, welcher in folgendem Listing 4.1 zu sehen ist, enthält wie schon weiter oben beschrieben alle wichtigen Informationen über das Petri-Netz.

```
1  typedef struct State_tag {
2  Marking *M; // Marking in the state
3  double t0; // t0 and t1 determines the time T at which the system entered this
       state:
4  double t1; // if s is the general transition sample, then
5  // the state was entered at T = t1 * s + t0
6  double leftInt; // Left bound of the validity region
7  double rightInt; // Right bound of the validity region
8  int stateID; // id of the state (for printing purposes);
9  struct State_tag *next; // pointer to the "next" state (for printing purposes)
10 struct StateTimeAlt_tag *nextStateList; // Next states
11 } State;
```

Listing 4.1: Struct des States

Die Informationen über die Inhalte der Plätze und dem Zustand der Transitionen sind dabei in der Variable *M (Zeile 2) gespeichert. Der Zeitpunkt wird über die Werte t0 und t1 anhand der Formel $t = t1 * s + t0$ berechnet (siehe Zeile 3 und 4), s stellt in dieser Formel einen Wert des Hilfsintervalls dar, welches durch die Variablen leftInt und richtInt (Zeile 6 und 7) beschrieben

[3] `https://code.google.com/p/fluid-survival-tool/`

wird. Die Variable *nextStateList (Zeile 9) hält alle von diesem Zustand aus erreichbaren Zustände.
Den Datentypen des Models zeigt das folgende Listing 4.2.

```
typedef struct {
int N_places; // number of places
Place *places; // list of places
int N_transitions; // number of transitions
Transition *transitions; // list of transitions
int N_arcs; // number of arcs
Arc *arcs; // list of arcs
double MaxTime;
////computed field
int N_discretePlaces;
int N_fluidPlaces;
int N_determTransitions;
int N_fluidTransitions;
int N_generalTransitions;
State *initialState; // initial marking
} Model;
```

Listing 4.2: Struct des Models

Die Variablen *places, *transitions und *arcs (Zeile 3, 5 und 7) halten dazu die Elemente des HPnG in einer entsprechenden Liste. Maximal dürfen Schaltvorgänge in der Simulation bis zur Zeit MaxTime (Zeile 8) geschehen. Außerdem erhält das Model noch den Initialzustand des Petri-Netzes in der Variablen *initialState (Zeile 15).

produceNext

Die Funktion *produceNext* ist in *produceNext.c* enthalten. Diese Funktion liefert den kompletten Zustandsbaum. Die Funktion erhält als Parameter das Model, einen State und zwei Zahlwerte leftInt und rightInt, die das Hilfsintervall s darstellen. Zum ersten Mal wird die Funktion mit dem Initialzustand aufgerufen. Danach berechnet es anhand von verschiedenen Schaltungen der Transitionen die möglichen Kinderknoten. Für jeden Kinderknoten wird dann die Funtion *produceNext* rekursiv aufgerufen. Zusätzlich werden die möglichen Folgezustände zu dem State in dem Parameter *nextStateList* abgespeichert. (nach [Gha+12])

5. Effiziente Zustandsraumerzeugung

Die Berechnung des Zustandsraumes, welche im vorherigen Kapitel 4 vorgestellt wurde, ist gerade bei größeren Petri-Netz Modellen sehr aufwendig. Deshalb ist es unter anderem das Ziel dieser Bachelorarbeit möglichst wenige Zustände zu berechnen und somit nicht den gesamten Zustandsraum zu erzeugen. Das Konzept zur effizienten Berechnung von Zuständen wird in diesem Kapitel vorgestellt. Desweiteren wird erklärt wie zu Eigenschaften im Petri-Netz Wahrscheinlichkeiten berechnet werden können. Die Umsetzung in Quellcode wird danach in folgendem Kapitel 6 auf Seite 29 besprochen.

5.1. Wahrscheinlichkeitsschranken

Die Wahrscheinlichkeiten zu Eigenschaften im HPnG-Netz wurden im bisherigen Algorithmus immer exakt berechnet. Im DFPN-Tool ist dafür die Funktion *computeMeasures* zuständig.

computeMeasures
Die Funktion *computeMeasures* berechnet zu jeder Eigenschaft eines Platzes die Wahrscheinlichkeiten zu verschiedenen Zeitpunkten verschiedene Werte zu erhalten. Sie ist in *Measures.c* implementiert und wird im Anschluss an die Funktion produceNext aufgerufen, wenn der Zustandsbaum komplett berechnet ist. Alle Wahrscheinlichkeiten werden nach und nach iterativ über mehrere for-Schleifen berechnet und nach jeder Berechnung im Array *Properties* aktualisiert. Im Anschluss werden die Wahrscheinlichkeiten in einer Ausgabedatei abgespeichert.

Da die exakte Berechnung der Wahrscheinlichkeiten sehr aufwändig ist und meistens nicht die exakte Wahrscheinlichkeit von Bedeutung ist, sondern es ausreicht eine Aussage über das Intervall zu treffen, indem die Wahrscheinlichkeit gilt, wird nun das Konzept der Wahrscheinlichkeitsgrenzen betrachtet.
Mit den Wahrscheinlichkeitsschranken ist gemeint, dass während der Berechnung nach jedem Zustand überprüft wird, ob die vorgegebene Wahrscheinlichkeitsgrenze schon erreicht ist. Ist die Gleichung $\tilde{\pi}^{\psi}(\tau) > \mu$ noch nicht erfüllt, wobei $\tilde{\pi}$ für die akkumulierte Wahrscheinlichkeit und μ für die Wahrscheinlichkeitsgrenze steht, so muss ein nächster Zustand betrachtet werden um weiter

Wahrscheinlichkeitsmasse zu sammeln. Ist die Gleichung jedoch erfüllt kann der Wahrscheinlichkeitsbereich akzeptiert werden. Dies hat den Vorteil, dass nur so viele Zustände berechnet werden müssen, wie auch wirklich benötigt werden und nicht wie vorher der gesamte Zustandsraum. Um noch effizienter eine Aussage zu treffen, wird nicht nur die Wahrscheinlichkeitsmasse dafür gesammelt, dass die Eigenschaft erfüllt ist, sondern auch die des Gegenereignisses. Ist die Gleichung $\tilde{\pi}^{\neg\psi}(\tau) \geq 1 - \mu$ erfüllt, kann die Berechnung ebenfalls abgebrochen werden und der Wahrscheinlichkeitsbereich abgelehnt werden.

Anpassung des Programmablaufes

Bisher war berechnete der Algorithmus zunächst den kompletten Zustandsbaum, ehe in diesem die Berechnung der Wahrscheinlichkeiten durchführte, Abbildung 5.1 zeigt den alten Programmablauf. Aufgrund der Anpassung der

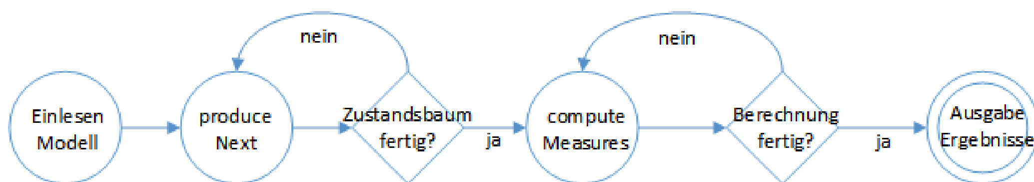

Abbildung 5.1.: Alter Programmablauf des DFPN-Tools

Berechnung von Wahrscheinlichkeiten kann der Programmablauf geändert werden, sodass der Algorithmus effizienter ist. Gestartet wird weiterhin mit dem Einlesen des Modells und mit dem Initialzustand muss auch zuerst die Funktion *produceNext* aufgerufen werden, um die Folgezustände des Initialzustandes zu berechnen. Doch anstatt mit den Folgezuständen rekursiv die Funktion *produceNext* aufzurufen wird nun erst einmal die Wahrscheinlichkeit des Initialzustandes berechnet, ehe mit einem der beiden Folgezuständen fortgefahren wird. Den neuen Programmablauf zeigt die Abbildung 5.2.

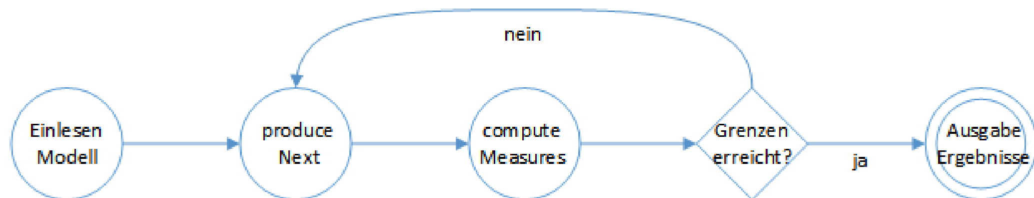

Abbildung 5.2.: Neu gestalteter Programmablauf

5.2. Berechnung von Wahrscheinlichkeiten

Mithilfe eines berechneten Zustandsraumes ist es möglich Wahrscheinlichkeiten dafür zu berechnen, dass ein Platz zu einer bestimmten Zeit einen bestimmten

Wert hält. Die Eigenschaft, zu der die Wahrscheinlichkeit berechnet werden soll wird durch ψ dargestellt. Die Wahrscheinlichkeit zu einem Zeitpunkt τ gibt die Funktion $\pi^{\psi}(\tau)$ an. Die Wahrscheinlichkeit wird anhand der Zustände des Zustandsraumes berechnet, dafür ist es zunächst notwendig zu wissen, über welchen Zeitraum ein Zustand gültig ist.

5.2.1. Berechnung des Gültigkeitsbereiches

Für die Berechnung des Gültigkeitsbereiches eines Zustandes ist es nicht nur von Interesse den eigenen Gültigkeitsbereich zu kennen, sondern es müssen auch die Folgezustände betrachtet werden. Der Gültigkeitsbereich eines Zustandes $[G_{min}, G_{max}]$ ist das Intervall für das man in diesem Zustand ist. Es gilt dabei $[G_{min}, G_{max}] \subseteq [S_{min}, S_{max}]$, wobei $[S_{min}, S_{max}]$ das Hilfsintervall aus Abschnitt 4.2 darstellt. Der Gültigkeitsbereichdes Zustandes berechnet sich durch das Schneiden des Intervalls $[S_{min}, S_{max}]$ mit den Intervallen der erreichbaren Folgezustände. Die Berechnung des Gültigkeitsbereiches wird anhand eines Beispiels erklärt.

Beispiel zur Berechnung des Gültigkeitsbereichs
Als Beispiel zur Berechnung des Gültigkeitsbereiches dient der Zustandsraum aus Abbildung 4.2. Der in diesem Beispiel betrachtete Zeitpunkt ist $T = 4$. Wir berechnen den Gültigkeitsbereich von dem Zustand 0 (oben). Zuerst müssen wir testen, ob der Zustand 0 überhaupt erreichbar ist. Wie dies überprüft wird, erklärt der folgende Einschub.

Überprüfen, ob ein Zustand erreichbar ist
Ein Zustand ist erreichbar, wenn folgende Bedingung erfüllt ist:

$$\exists s \in [S_{min}, S_{max}] : t \leq \tau.$$

Algorithmus
Um zu prüfen, ob ein Zustand erreichbar ist, wird zunächst untersucht, ob der Zeitpunkt des Zustandes unabhängig ist von s, s stellt das Hilfsintervall des Zustandes dar. Ist ein Zustand unabhängig von s, brauchen lediglich der Zeitpunkt des Zustandes, im Zustand durch t angegeben und der betrachtete Zeitpunkt verglichen werden, wenn der betrachtete Zeitpunkt größer oder gleich dem Zeitpunkt des Zustands ist, so ist der Zustand erreichbar, falls nicht, ist der Zustand nicht erreichbar.
Ist der Zeitpunkt des Zustandes abhängig von s, so muss überprüft werden, für welches s die Gleichung $T = t_1 * s + t_0$ erfüllt ist (t_1 und t_0 sind Parameter des Zustandes (Listing 4.1 Zeile 3 und 4) und T der betrachtete Zeitpunkt). Dazu reicht es die Gleichung nach s umzustellen, es ergibt sich somit

$$s = \frac{T - t_0}{t_1}.$$

Nun muss nur noch überprüft werden, ob der Zeitpunkt ober- oder unterhalb des Grenzwertes erfüllt ist. Je nachdem, wie das Hilfsintervall zu s definiert ist, ist der Zustand erreichbar oder nicht, außerdem muss das Hilfsintervall ggf. angepasst werden, falls der Grenzwert von s innerhalb des Intervalls des Hilfsbereiches liegt.

Da $t = 0 < 4$ gilt, ist der Zustand 0 erreichbar. Der angegebene Bereich $s \in [0, \infty]$ ist der Ausgangsbereich, dieser wird durch die beiden Zustände 1 und 5 jedoch weiter eingeschränkt. Zuerst betrachten wir den Zustand 1, dieser ist für $s \in [0, 4]$ gültig, da dann $t = s \leq 4$ gilt. Dies ist dann auch gleich der Gültigkeitsbereich von Zustand 1. Somit gilt für den Gültigkeitsbereich von Zustand 0 folglich $s \in [0, \infty] \setminus [0, 4]$ also $s \in [4, \infty]$. Nun betrachten wir den Zustand 5. Dieser ist jedoch nicht möglich, da $t = 5 > 4$. Deshalb bleibt der Gültigkeitsbereich bei $[4, \infty]$.
Der Algorithmus zur Berechnung des Gültigkeitsbereiches zeigt der folgende Pseudocode 5.1.

```
1  Berechne Gültigkeitsbereich{
2    Falls Zustand erreichbar, dann{
3      Falls Zustand keine Folgezustände hat, dann{
4        Gebe Hilfsintervall des Zustandes zurück;
5      } sonst{
6        Wiederhole für alle Folgezustände{
7          Falls Folgezustand erreichbar, dann{
8            schneide Hilfsintervall des Folgezustandes mit dem verbliebenen
                 Hilfsintervall des betrachteten Zustandes;
9          }
10       }
11       Gebe verbliebenes Hilfsintervall zurück;
12     }
13   }
14 }
```

Listing 5.1: Pseudocode zur Berechnung des Gültigkeitsbereiches

Dabei sind die Folgezustände nur die direkten Kinderknoten des betrachteten Zustandes und das Hilfsintervall wird von Folgezustand zu Folgezustand immer weiter eingeschränkt.

5.2.2. Berechnung der Wahrscheinlichkeit

Die Wahrscheinlichkeit lässt sich durch die folgende Formel berechnen:

$$\pi^{\psi}(\tau) = \int \xi^{\psi}_{L_i}(\tau|s)g(s)ds.$$

Dabei steht $g(s)$ für die Verteilung der Wahrscheinlichkeit über den Zeitraum, L_i für den betrachteten erreichbaren Zustand und die Funktion $\xi^{\psi}_{L_i}(\tau|s)$ für die Wahrscheinlichkeit in dem Zustand zu sein. Es wird dabei über alle erreichbaren Zustände integriert. Anschaulich wird die Berechnung an folgendem Beispiel erklärt.

Beispielberechnung der Wahrscheinlichkeit eines Zustandes
Wir berechnen, wie schon im obigen Beispiel des Abschnitts 5.2.1 die Wahrscheinlichkeit zum Zeitpunkt $T = 4$. Wir gehen von einem maximalen Zeitraum bis 10 aus und gehen davon aus, dass die Wahrscheinlichkeit über das Intervall von$[0, 10]$ gleichverteilt ist. Berechnet werden soll die Wahrscheinlichkeit davon, dass der kontinuierliche Platz, also x_1 einen Inhalt von mindestens 2 aufweist. Die Wahrscheinlichkeit soll bei mindestens 75% liegen.
Zunächst wird der Wurzelknoten mit der ID 0 betrachtet. Es gilt nun zu überprüfen, ob der Zustand möglich ist. Da der Zeitpunkt $T = 4 > 0$ ist, ist der Zustand möglich. Nun muss das Intervall des Gültigkeitsbereiches berechnet. Die Berechnung des Intervalls ist in Abschnitt 6.3 ab Seite 31 erklärt. Das Ergebnis ist das Intervall von $[4, \infty]$, da jedoch der Maximalwert von s bei 10 liegt, wird das Intervall auf $[4, 10]$ reduziert. Der letzte Schritt ist zu überprüfen, ob die Bedingung, zu der die Wahrscheinlichkeit berechnet werden soll, erfüllt oder nicht erfüllt ist. Der Zustand gibt für den Inhalt von x_1 den Wert 0 an, somit ist die Bedingung nicht erfüllt. Die Wahrscheinlichkeit, dass die Bedingung nicht erfüllt ist, beträgt nach Betrachtung des ersten Zustandes

$$\frac{|[4, 10]|}{|[0, 10]|} = 0,6 = 60\%$$

Da die Wahrscheinlichkeitsgrenze der Bedingung bei 75% liegen sollte und nun die Gegenwahrscheinlichkeit schon einen Wert von 60% hat, kann die Aussage abgelehnt werden.
Den Algorithmus zum Berechnen der Wahrscheinlichkeit zeigt der folgende Pseudocode 5.2.

```
1  Berechne Wahrscheinlichkeit{
2    Falls Zustand möglich, dann{
3      Berechne Gültigkeitsbereich;
4      Falls Bedingung erfüllt, dann{
5        Addiere Wahrscheinlichkeitsmasse des Gültigkeitsbereiches zur bisher
             berechneten Wahrscheinlichkeit;
6      } sonst, falls Bedingung nicht erfüllt, dann{
7        Addiere Wahrscheinlichkeitsmasse des Gültigkeitsbereiches zur bisher
             berechneten Gegenwahrscheinlichkeit;
8      } sonst{
9        // Bedingung teilweise erfüllt, bei kontinuierlichen Plätzen und Abhä
             ngigkeit von s möglich
10       Berechne Bereiche in denen die Bedingung erfüllt ist und in denen die
             Bedingung nicht erfüllt ist;
11       Addiere Wahrscheinlichkeitsmasse des Bereiches erfüllt zur bisher
             berechneten Wahrscheinlichkeit;
12       Addiere Wahrscheinlichkeitsmasse des Bereiches nicht erfüllt ist zur
             bisher berechneten Gegenwahrscheinlichkeit;
13     }
14   }
15 }
```

Listing 5.2: Pseudocode zur Berechnung Wahrscheinlichkeit eines Zustandes

Die Bedingung steht dabei dafür, ob der gewünschte Mindestinhalt erreicht ist oder nicht. Die Wahrscheinlichkeitsmasse wird immer zu den bisher berechneten Wahrscheinlichkeiten hinzuaddiert, da die Wahrscheinlichkeit über den gesamten Wahrscheinlichkeitsraum berechnet werden soll.

5.3. Eulertour

Die Eulertour oder auch das Eulerkreisproblem geht auf den Mathematiker Leonhard Euler[1] zurück. Er beschäftigte sich unter anderem mit der Fragestellung des Königsberger Brückenproblems (Abb. 5.3), welche als Ursprung der Graphentheorie gilt.

Die eigentliche Fragestellung dieses Problems war: Gibt es einen Weg, bei

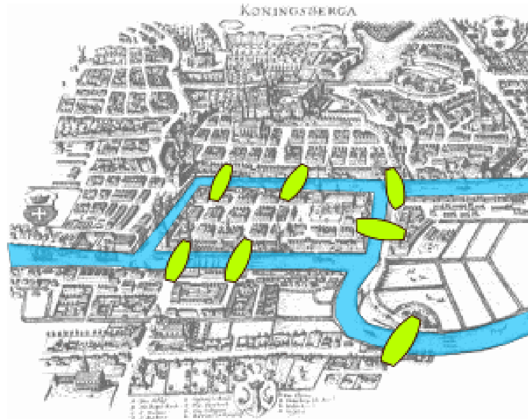

Abbildung 5.3.: Das Königsberger Brückenproblem[2]

dem alle Brücken genau einmal überquert werden oder gar einen Rundweg, bei dem der Start- gleich dem Endpunkt der Route ist?
Das ursprüngliche Problem lässt sich vereinfachen, indem man die Grafik als Graphen (Abb. 5.4) darstellt. Da die Brücken in beide Richtungen begehbar sind, handelt es sich um einen ungerichteten Graphen. Die Stadt Königsberg wurde hierbei in die 4 Gebiete Nord, Mitte, Süd und Ost geteilt, da nur die Übergange zwischen den einzelnen Bereichen betrachtet werden.

Leonhard Euler löste die Problematik und führte folgende Begriffe, die später nach Ihm benannt wurden, ein. Dabei betrachtete er nicht nur das Problem mit ungerichteten Graphen, sondern verallgemeinerte es auch mit gerichteten Graphen.

Eulertour
Als Eulertour bezeichnet man einen Rundgang durch den Graphen, indem alle Kanten besucht werden und der Startpunkt gleich dem Endpunkt ist. Der

[1]Leonhard Euler * 15. April 1707 in Basel, † 18. September 1783 in St. Petersburg
[2]Königsberger Brückenproblem - Bogdan Giuşcă

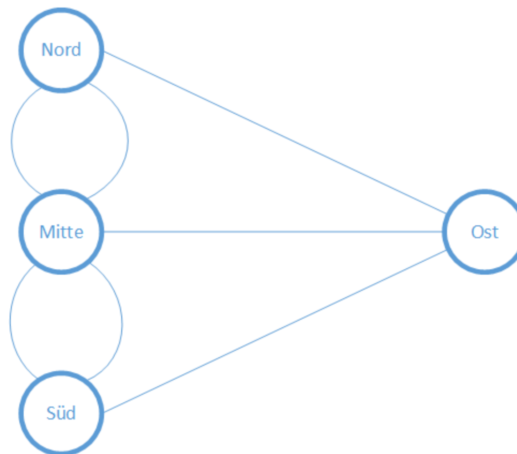

Abbildung 5.4.: Das Königsberger Brückenproblem als Graph

Graph wird dann ebenfalls eulersch genannt.
Ein **ungerichteter** Graph besitzt genau dann einen Eulerkreis, wenn der Graph zusammenhängend ist und wenn alle seine Knoten einen geraden Grad besitzen.
Ein **gerichteter** Graph ist dann eulersch, wenn er stark zusammenhängend ist, es gibt also von jedem Knoten u einen gerichteten Weg nach v, und in jedem Knoten der Grad der aus- und eingehenden Kanten gleich groß ist.

Offene Eulertour
Im Gegensatz zur Eulertour ist es bei der offenen Eulertour nicht interessant, ob der Start- gleich dem Endpunkt ist. Somit ergeben sich auch andere Voraussetzungen, die der Graph zu erfüllen hat.
Eine offene Eulertour ist in jedem **ungerichteten** Graphen zu finden, falls alle bis auf zwei Knoten einen geraden Knotengrad besitzen und der Graph zusammenhängend ist.
Ein **gerichteter** Graph besitzt genau dann eine offene Eulertour, wenn der Grad der eingehenden Kanten über alle Knoten gleich dem Grad der ausgehenden Kanten über alle Knoten ist und es Knoten gibt, in denen der Grad der ausgehenden Kanten um 1 größer ist, als der Grad der eingehenden Kanten oder der Grad der eingehenden Kanten um 1 größer ist, als der Grad der ausgehenden Kanten. (nach [Ste07])

Verwendbarkeit in dem Programm
Die Zustandsbäume, die der Algorithmus berechnet, werden zur Auswertung bisher rekursiv betrachtet und die Wahrscheinlichkeiten der Ereignisse so berechnet. Einen beispielhaften Zustandsbaum zeigt die Abbildung 4.2 auf Seite 16.
Um Wahrscheinlichkeiten zu Zuständen kostengünstig zu berechnen, ist es notwendig die Zustände der Reihe nach zu berechnen. Es wird eine Route gesucht,

die alle Knoten abdeckt. Dazu wird die Eulertour genutzt.
Hierfür wird der Zustandsbaum als gerichteter Graph betrachtet, es werden, wie in Abb. 5.5 zu sehen, alle Kanten durch gerichtete Kanten in beide Richtungen ersetzt.
Der Zustandsbaum ist eulersch, da der Graph stark zusammenhängend ist,

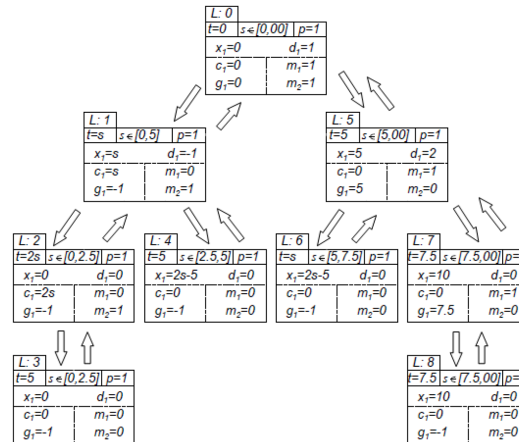

Abbildung 5.5.: Der Zustandsbaum als gerichteter Graph

man kommt von jedem Zustand zu jedem anderen, da man von der Wurzel zu jedem Knoten und von jedem Knoten zur Wurzel gelangt und der Knotengrad der eingehenden Kanten gleich dem der ausgehenden Kanten ist. Es ist notwendig immer erst in die Tiefe die Zustände zu besuchen, da die Kanten für eine Eulertour nur einmal zu benutzen sind.

5.4. Heuristiken

Um möglichst wenige Zustände berechnen zu müssen , ist es notwendig den möglichst optimalen nächsten Zustand zu wählen, um viel Wahrscheinlichkeitsmasse zu sammeln und schnell eine Aussage über die Bedingungen $\tilde{\pi}^{\psi}(\tau) > \mu$ oder $\tilde{\pi}^{\neg\psi}(\tau)$ treffen zu können. Für die Wahl des nächsten Zustandes ist die Heuristik zuständig.

Definition 5.1. *Eine* ***Heuristik*** *bezeichnet eine Funktion, welche der Problemlösung dient. Sie bewertet die möglichen nächsten Schritte auf dem Weg zur Lösung und versucht anhand von Schätzungen oder Vermutungen den bestmöglichen Schritt zu wählen.*

Eine optimale Heuristik wählt trotz geringer eigener Rechenzeit einen guten Zustand aus. Jedoch stehen die Qualität und die geringe eigene Rechenzeit häufig im Widerspruch zueinander, deshalb gilt es einen guten Kompromiss zu

#	Heuristik
1	Bewertung anhand der Zustandsnummer
2	Bewertung durch die Länge des Gültigkeitsbereiches
3	Bewertung anhand der Wahrscheinlichkeitsmasse
4	Bewertung aufgrund der Nähe der aktuellen Wahrscheinlichkeit/Gegenwahrscheinlichkeit zur Wahrscheinlichkeitsgrenze unter Berücksichtigung der Wahrscheinlichkeitsmasse

Tabelle 5.1.: Übersicht über verschiedene mögliche Heuristiken

finden. Die folgende Tabelle 5.1 zeigt alle betrachteten Heuristiken. Die zuerst genannte Bewertung ist im Grunde genommen keine Heuristik, sie dient lediglich dazu eine komplette Eulertour durch den Graphen zum Vergleich umzusetzen. Die zweite Funktion ist eine Heuristik, sie bewertet die Zustände anhand der Länge des Gültigkeitsbereiches, sie ist eine vereinfachte Form der dritten Heuristik. Diese bewertet die Zustände anhand der möglichen Wahrscheinlichkeitsmasse. Bei einer gleichverteilten Wahrscheinlichkeitsmassenfunktion ist Heuristik 2 der Heuristik 3 gleichzusetzen. Bei anders verteilten Wahrscheinlichkeitsdichten berechnet die Heuristik die Wahrscheinlichkeitsmasse anhand des Gültigkeitsbereiches, wie z.B. in Abbildung 5.6 zu sehen, können so Unterschiede in den beiden Heuristiken vorkommen. Heuristik 2 würde in diesem

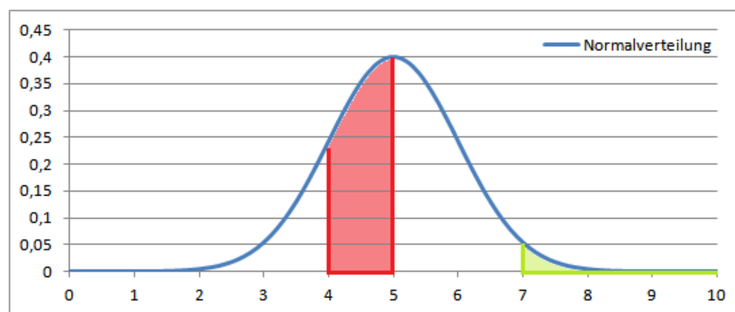

Abbildung 5.6.: Normalverteilung Vergleich zweier Gültigkeitsbereiche

Fall den grün markierten Bereich wählen, weil dieser die Intervalllänge 3 hat, der rote Bereich hat nur die Länge 1. Jedoch wäre es in diesem Fall sinnvoll das rote Intervall zu wählen, da in diesem eine höhere Wahrscheinlichkeitsmasse gesammelt werden kann.

Die letzte Heuristik ist die von der Berechnung her aufwendigste. Falls eine Wahrscheinlichkeitsgrenze fast erreicht ist, ist es sinnvoll einen Zustand zu wählen, der dazu führt eine Wahrscheinlichkeitsgrenze zu erreichen und damit danach die Berechnung abbrechen zu können. Dazu wird getestet, ob der Zustand Wahrscheinlichkeitsmasse für ψ oder für $\neg\psi$ und anhand der folgenden Formel wird die Berechnung vorgenommen, wobei L_0 den betrachteten

Zustand darstellt.

$$\frac{\alpha * \tilde{\pi}^{\psi}(\tau)}{\mu} * \int \xi^{\psi}_{L_0}(\tau|s)g(s)ds \text{ bzw.}$$

$$\frac{\alpha * \tilde{\pi}^{\neg\psi}(\tau)}{1-\mu} * \int \xi^{\psi}_{L_0}(\tau|s)g(s)ds$$

Dabei stellt μ die bisher berechnete Wahrscheinlichkeit bzw. $1-\mu$ die bisher berechnete Gegenwahrscheinlichkeit dar. Der Faktor α kann verändert werden um die Gewichtung der Nähe zur Wahrscheinlichkeitsgrenze anzupassen. Je höher α, desto wahrscheinlicher ist es, dass der Zustand ausgewählt wird, welcher mehr Wahrscheinlichkeitsmasse zur Wahrscheinlichkeit mit geringerem Abstand zur Wahrscheinlichkeitsgrenze hinzufügen kann.

Heuristiken im Vergleich

Heuristik 2: Länge des Gültigkeitsbereiches - Heuristik 3: Maximale Wahrscheinlichkeitsmasse
Diese beiden Heuristiken unterscheiden sich lediglich, wenn die Funktion zur Wahrscheinlichkeitsdichte keine Gleichverteilung ist, da sonst beide Heuristiken nach der Länge des Gültigkeitsbereiches entscheiden. In dem Fall der Gleichverteilung, ist grundsätzlich Heuristik 2 zu bevorzugen, da die Berechnung durch Heuristik 3, einen höheren Aufwand darstellt und sich so die Performance verschlechtert.
Die Heuristik 3 hat bei anderen Funktionen für Wahrscheinlichkeitsdichten z.B. bei der Gaußschen Glockenkurve den Vorteil, dass ggf. nicht die Randbereiche, die kaum Wahrscheinlichkeitsmasse halten, sondern die Bereiche um den Erwartungswert, welche viel Wahrscheinlichkeitsmasse halten, betrachtet werden.

Heuristik 3: Maximale Wahrscheinlichkeitsmasse - Heuristik 4: Nähe zur Wahrscheinlichkeitsgrenze in Abhängigkeit der Wahrscheinlichkeitsmasse
Heuristik 4 ist eine Weiterentwicklung der dritten Heuristik. Sie ist jedoch deutlich komplizierter und zeitaufwändiger, da nach den Eigenschaften im Zustand getestet werden muss. Einen deutlichen Vorteil bietet Heuristik 4, wenn eine der Wahrscheinlichkeitsgrenzen fast erreicht ist und durch eine kleine Wahrscheinlichkeitsmasse die Grenze erreicht wird. Jedoch ist es auch möglich, dass die Heuristik zu sehr die fast erreichte Wahrscheinlichkeitsgrenze erreichen will, obwohl diese gar nicht erreicht werden kann und somit viele unnötige Zustände berechnet.
In Kapitel 7 auf Seite 7 wird beschrieben, wie sich die verschiedenen Heuristiken auf die Performance des Algorithmus auswirken.

6. Implementierung

In diesem Kapitel wird erklärt, wie die verschiedenen Anpassungen des Programms implementiert worden sind. Wie sich die Veränderungen am Quelltext auf die Performance des Programmes ausgewirkt hat, finden Sie im Kapitel 7 auf Seite 41.

6.1. Tour durch den Zustandsgraphen

Die Zustände werden, wie bereits beschrieben, erst nach und nach berechnet. Den Anfang bildet der Wurzelknoten, der durch den Initialzustand angegeben wird. Ausgehend von diesem Knoten können dessen Folgezustände besucht werden. Da die Anzahl der möglichen auswählbaren Zustände immer größer wird, je mehr Knoten besucht wurden, vorausgesetzt der Zustandsbaum ist ausreichend groß, müssen die möglichen nächsten Zustände verwaltet werden. Dafür wurde das Struct *Zustandskette* implementiert, welches im folgenden Quellcode 6.1 dargestellt wird.

```
1 typedef struct Zustandsketten_Element{
2 State *Zustand;
3 struct Zustandsketten_Element *next;
4 } Zustandskette;
```

Listing 6.1: Struct der Zustandskette

Das Struct *Model* erhält zwei dieser Zustandsketten. Eine für die bereits besuchten Zustände des Zustandsbaumes und eine für die möglichen nächsten Zustände.
Jedes Element der Zustandskette erhält einen Zeiger auf den Zustand den dieses Element darstellt und außerdem einen Zeiger auf das nächste Element der Zustandskette. Anhand dieser Ketten ist es möglich, den Zustandsbaum der Reihe nach durchzugehen. Die Kette der möglichen nächsten Zuständen arbeitet nach dem *Teleskop*-Prinzip. So enthält die Kette zunächst nur die erreichbaren Zustände, wird jedoch ein Zustand ausgewählt, wird dieser Zustand durch die Zustände der Kinderknoten im Zustandsraum ersetzt. Wichtig ist es außerdem, den nächsten Zustand möglichst optimal zu wählen. Dies ist durch die Kettenstuktur ermöglicht. Wie der optimale Zustand gewählt wird, wird im folgenden Abschnitt 6.2 beschrieben.

6.2. Ausgeben von Zuständen

Ist die Wahrscheinlichkeitsgrenze μ bzw. die Wahrscheinlichkeitsgrenze des Gegenereignisses $1-\mu$ noch nicht erreicht, so muss der nächste Zustand ausgewählt und berechnet werden. Um einen Zustand zu erhalten wird die Funktion *giveNextState* in Euler.c aufgerufen, Quelltext 6.2 zeigt die Funktion.

```
1  State* giveNextState(Model *M){
2      struct Zustandsketten_Element *Naechste_iter = M->Naechste;
3      struct Zustandsketten_Element *Best = M->Naechste;
4      State *Element;
5      double momBewertung;
6      double *S_left, *S_right;
7
8      S_left = (double *)malloc(sizeof(double));
9      S_right = (double *)malloc(sizeof(double));
10
11     // Suche nach dem bestmoeglichen Zustand, anhand einer Charakteristik,
           testen, ob dieser moeglich ist, ansonsten wird der Zustand entfernt
12     momBewertung = bewerteZustand(Best->Zustand, M);
13     while(Naechste_iter->next != NULL){
14         Naechste_iter = Naechste_iter->next;
15         *S_left = Naechste_iter->Zustand->leftInt;
16         *S_right = Naechste_iter->Zustand->rightInt;
17         if(possibleState(Naechste_iter->Zustand, M->T, S_left, S_right) == 1){
18             // Falls der Zustand moeglich ist, teste, ob der Zustand besser
                   ist
19             if(bewerteZustand(Naechste_iter->Zustand, M) > momBewertung){
20                 Best = Naechste_iter;
21             }
22         } else {
23             // Falls der Zustand nicht moeglich ist, entferne ihn aus der
                   Liste Naechste
24             deleteElement(M, Naechste_iter, 0);
25         }
26     }
27
28     Element = Best->Zustand;
29     // Produziere die Folgezustaende des Knotens
30     produceNext(M, Element, Element->leftInt, Element->rightInt);
31     saveChilds(M, Element);
32
33     // Speichern des Zustandes zu den abgearbeiteten
34     saveState(M, Element, 1);
35
36     // Loeschen des Elements Best
37     deleteElement(M, Best, 0);
38
39     return Element;
40 }
```

Listing 6.2: Funktion giveNextState

Die Funktion testet zunächst alle Elemente der Zustandskette, der noch nicht betrachteten Zustände, Naechste, welches von ihnen das am besten geeignete Element ist (Zeile 13 -26). Dazu überprüft es zunächst, ob der Zustand erreichbar ist (Zeile 17). Falls dies nicht der Fall ist, wird das Element aus der Kette gelöscht (Zeile 24), um im weiteren Verlauf der Berechnung nicht unnötig Rechenzeit zu verbrauchen. Ist der Zustand möglich, wird er, anhand der in der Funktion *bewerteZustand* verwendeten Heuristik, bewertet (Zeile 19). Die Ta-

belle 5.1 auf Seite 27 zeigt die verschiedenen implementierten Heuristiken. In der Funktion *bewerteHeuristik* kann durch auskommentieren die gewünschte Heuristik gewählt werden.
Ist der, je nach Heuristik am besten bewertete, nächste Zustand gefunden, ist es wichtig die Funktion *produceNext* mit dem optimalen Zustand als Parameter aufzurufen (Zeile 30). Denn für die Berechnung der Wahrscheinlichkeiten ist es wichtig zu wissen, welche Folgezustände der Zustand hat. Diese werden von *produceNext* berechnet und in dem Parameter *nextStateList* des States gespeichert. Um im nächsten Aufruf der Funktion *giveNextState* auch die Folgezustände des ausgewählten Knotens betrachten zu können, werden diese über die Funktion *saveChilds* in der Zustandskette Naechste gespeichert (Zeile 31). Zu guter Letzt wird der ausgewählte Zustand von der Kette Naechste in die Kette Besucht verschoben (Zeile 34 + 37) und der Zustand an die aufrufende Funktion zurückgegeben.

6.3. Berechnung des Gültigkeitsbereiches

In diesem Abschnitt wird die Umsetzung der Berechnung des Gültigkeitsbereiches erklärt. Der Algorithmus wurde bereits in Abschnitt 5.2.1 erläutert. Dieser Abschnitt betrachtet die Umsetzung des dort in Pseudocode festgehaltenen Algorithmus, siehe Listing 5.1.
Die erste Abfrage der Funktion ist das Überprüfen, ob ein Zustand möglich ist. Der folgende Einschub zeigt, wie diese Funktion implementiert wurde.

Überprüfen, ob ein Zustand möglich ist
Der folgende Quellcode 6.3 zeigt die Funktion possibleState, welche überprüft, ob der Zustand, für den im Model abgespeicherten Zeitpunkt, möglich ist. Die Funktion liefert für den Fall, dass der Zustand möglich ist den Wert 1, sowie die möglicherweise angepassten Grenzen des Gültigkeitsbereichs zurück. Falls der Zustand nicht möglich ist, ist der Rückgabewert 0.

```
1  int possibleState(State *S, double T, double *S_left, double *S_right){
2
3      if (S->t1 == 0.0){
4          // Ergebnis unabhaengig von s
5          if (T >= S->t0){
6              // State moeglich ueber den gesamten Bereich von S_left bis
                   S_right
7              return 1;
8          } else {
9              // State nicht moeglich
10             return 0;
11         }
12     } else {
13         // Ergebnis abhaengig von s: T = t1 * s + t0 | (T - t0)/ t1 = s
14         double border;
15
16         border = computeBorder(T, S->t1, S->t0);
17
```

```
18          // Testen, on die Border zwischen S_left und S_right liegt
19          if ((border >= *S_left) && (*S_right >= border)){
20              // Testen, auf welcher Seite von border die Bedingung erfuellt ist
21              if (T > (S->t1 * (border - 1) + S->t0)){
22                  // Bedingung ist im Bereich kleiner der Border erfuellt
23                  // Testen, ob ein Teilbereich zwischen S_left und S_right
                        liegt
24                  if (border > *S_left){
25                      // State ist moeglich im Bereich zwischen S_left und
                            border
26                      *S_right = border;
27                      return 1;
28                  } else {
29                      // State nicht moeglich
30                      return 0;
31                  }
32              } else {
33                  // Bedingung ist im Bereich groesser der Border erfuellt
34                  // Testen, ob ein Teilbereich zwischen S_left und S_right
                        liegt
35                  if (border < *S_right){
36                      // State ist moeglich im Bereich zwischen border und
                            S_right
37                      *S_left = border;
38                      return 1;
39                  } else {
40                      // State nicht moeglich
41                      return 0;
42                  }
43              }
44          } else {
45              // Entweder ist der Zustand nicht moeglich, oder ueber den
                    gesamten Bereich von S_left bis S_right
46              if (S->t1 > 0){
47                  // Das Intervall muss unterhalb der border liegen, damit es
                        erfuellt ist
48                  if (*S_right < border){
49                      return 1;
50                  } else {
51                      return 0;
52                  }
53              } else {
54                  // Das Intervall muss oberhalb der border liegen, damit es
                        erfuellt ist
55                  if (border < *S_left){
56                      return 1;
57                  } else {
58                      return 0;
59                  }
60              }
61          }
62      }
63  }
```

Listing 6.3: Funktion possibleState in Measures.c

Zuerst wird dafür überprüft, ob der Zeitpunkt des Zustandes unabhängig ist von s (Zeile 3). Ist ein Zustand unabhängig von s, brauchen lediglich der Zeitpunkt des Zustandes und der Zeitpunkt des Models verglichen werden, wenn der betrachtete Zeitpunkt größer oder gleich dem Zeitpunkt des Zustands ist, so ist der Zustand erreichbar, falls nicht, ist der Zustand nicht erreichbar (Zeile 5 - 11) . Der Gültigkeitsbereich muss in diesem Fall nicht verändert werden, da der Zeitpunkt des Zustandes unabhängig vom Gültigkeitsbereich ist.

Ist der Zeitpunkt des Zustandes abhängig von s, so muss überprüft werden, für welches s die Gleichung $T = t_1 * s + t_0$ erfüllt ist (t_1 und t_0 sind Parameter des Zustandes, siehe Listing 4.1, und T der betrachtete Zeitpunkt). Dies berechnet die Hilfsfunktion *computeBorder* (Zeile 16). Nun muss nur noch überprüft werden, ob der Zeitpunkt ober- oder unterhalb des Grenzwertes erfüllt ist. Je nachdem, wie der Gültigkeitsbereich zu s definiert ist, ist der Zustand erreichbar oder nicht. Außerdem muss der Gültigkeitsbereich ggf. angepasst werden, falls der Grenzwert von s innerhalb des Intervalls des Gültigkeitsbereiches liegt (Zeile 26 + 37).

Der Algorithmus aus Listing 5.1 wurde ähnlich im Quellcode umgesetzt. Die innere while-Schleife in der Funktion *computeMeasuresEuler* berechnet dabei das Intervall des Gültigkeitsbereiches S. Dazu wurde ein neues Struct *Intervall*, wie folgt angelegt.

```
1  typedef struct Gesamtintervall{
2  double Gesamtlaenge;
3  double *leftInts;
4  double *rightInts;
5  int Anzahl_Intervalle;
6  } Intervall;
```

Listing 6.4: Struct des Intervalls

Das Struct trägt dabei neben den Parametern der Gesamtlänge (Zeile 2) und der Anzahl der Teilintervalle (Zeile 5) eine Liste mit den linken und eine Liste mit den rechten Intervallgrenzen (Zeile 3 + 4). Die Gesamtlaenge bezeichnet die Länge aller Teilintervalle zusammenaddiert.
Zunächst wird das Intervall des Gültigkeitsbereiches, mit den Werten des aktuell betrachteten Zustandes initialisiert. In der Folge, werden nachher, wie im Algorithmus die einzelnen Folgezustände betrachtet und deren Hilfsintervall mit dem Hilfsintervall des betrachteten Zustandes geschnitten. Für das Schneiden der Intervalle ist die Funktion *intersect* verantwortlich, siehe Quellcode 6.5.

```
1  void intersect(Intervall *I, double left, double right) {
2      int i, x; // Hilfsvariablen fuer die Schleifen
3
4      // Gehe alle Teilintervalle durch
5      for (i = 0; i < I->Anzahl_Intervalle; i++){
6
7          // Testen, ob das Intervall dieses Teilintervall schneidet
8          if ((I->leftInts[i] <= right) && (I->rightInts[i] >= left)){
9
10             // Testen, ob gesamtes Teilintervall abgedeckt ist
11             if((I->leftInts[i] >= left) && (I->rightInts[i] <= right)){
12                 I->Gesamtlaenge = I->Gesamtlaenge - (I->rightInts[i] - I->
                       leftInts[i]);
13                 I->rightInts[i] = I->leftInts[i];
14             } else {
15
16                 // Testen, ob das Intervall nicht komplett innerhalb vom
                       Teilintervall liegt
```

```
17                  if ((I->leftInts[i] >= left) || (I->rightInts[i] <= right)){
18
19                      // Testen, ob das Intervall ueber der linken Grenze liegt
20                      if (I->leftInts[i] <= left){
21                          I->Gesamtlaenge = I->Gesamtlaenge - (right - I->
                                leftInts[i]);
22                          I->leftInts[i] = right;
23                      } else {
24                          // Intervall liegt ueber der rechten Grenze
25                          I->Gesamtlaenge = I->Gesamtlaenge - (I->rightInts[i] -
                                 left);
26                          I->rightInts[i] = left;
27                      }
28
29                  } else {
30                      // Das Intervall liegt innerhalb dieses Teilintervalls
31                      // Gesamtlaenge reduzieren
32                      I->Gesamtlaenge = I->Gesamtlaenge - (right - left);
33
34                      // Teilintervall anpassen und neues Teilintervall anlegen
35                      I->Anzahl_Intervalle++;
36
37                      // Hilfsarrays anlegen und mit Inhalt der vorherigen
                            Arrays belegen
38                      double lefts[I->Anzahl_Intervalle];
39                      double rights[I->Anzahl_Intervalle];
40
41                      for (x = 0; x < (I->Anzahl_Intervalle -1); x++){
42                         lefts[x] = I->leftInts[x];
43                         rights[x] = I->rightInts[x];
44                      }
45
46                      free(I->leftInts);
47                      free(I->rightInts);
48
49                      I->leftInts = lefts;
50                      I->leftInts = rights;
51                      I->rightInts[I->Anzahl_Intervalle -1] = I->rightInts[i];
52                      I->leftInts[I->Anzahl_Intervalle -1] = right;
53                      I->rightInts[i] = left;
54                  }
55              }
56          }
57      }
58  }
```

Listing 6.5: Funktion intersect

Die Funktion bekommt als Parameter einen Zeiger auf das betrachtete Intervall, sowie die Grenzen des Intervalls des betrachteten Folgezustandes. Danach schneidet die Funktion alle Teilintervalle mit diesem Intervall (for-Schleife Zeile 5). Liegt das Intervall inmitten eines Teilintervalls, wird dieses in zwei Teilintervalle geteilt, eines vor und eines nach dem Intervall des Folgezustandes (Zeile 30 - 53).
Sind alle Folgezustände betrachtet worden, hält das Struct alle wichtigen Informationen für die Berechnung der Wahrscheinlichkeiten zu diesem Zustand. Ist während der Berechnung des Intervalls von S die Gesamtlänge des Intervalls 0 geworden, wird die Berechnung abgebrochen, da keine Wahrscheinlichkeitsmasse gesammelt werden kann.

6.4. Berechnung von Wahrscheinlichkeiten

Der Funktion zur Berechnung von Wahrscheinlichkeiten wurde in Anlehnung an den Pseudocode 5.2 auf Seite 23 programmiert. Eine genauere Erklärung zu diesem Thema findet sich in Abschnitt 5.2.2.
Ein Teil des Algorithmus ist direkt in der Funktion *computeProb* umgesetzt worden, welche sich in *Measures.c* befindet. Die Funktion berechnet im Zusammenhang mit einem Teil der Funktion *computeMeasuresEuler*, die die Funktion *computeProb* aufruft, die Wahrscheinlichkeit. In diesem Absatz wird zunächst die Funktion *computeProb*, welche in folgendem Quellcode 6.6 zu sehen ist, erklärt. Eine Erklärung zur aufrufenden Funktion *computeMeasuresEuler* finden sie im nachfolgenden Abschnitt 6.5 auf Seite 37.

```
1  void computeProb(State *MomState,Model *M, double S_left, double S_right){
2
3      if (M->isFluidPlace == 1){
4          // Es handelt sich um einen kontinuierlichen Platz, testen, ob
               Bedingung von Anfang an erfuellt ist
5          double minContent, maxContent;
6          if (MomState->t1 >= 0){
7              minContent = MomState->M->fluid1[M->PlaceNumber] * S_left +
                   MomState->M->fluid0[M->PlaceNumber];
8              maxContent = MomState->M->fluid1[M->PlaceNumber] * S_right +
                   MomState->M->fluid0[M->PlaceNumber];
9          } else {
10             minContent = MomState->M->fluid1[M->PlaceNumber] * S_right +
                   MomState->M->fluid0[M->PlaceNumber];
11             maxContent = MomState->M->fluid1[M->PlaceNumber] * S_left +
                   MomState->M->fluid0[M->PlaceNumber];
12         }
13         if (minContent >= M->Value){
14             // Bedingung ist ueber den ganzen Bereich von S erfuellt
15             // Wahrscheinlichkeiten zu Prob addieren
16             *M->Prob = *M->Prob + HeuristikMaxWMasse(M, S_left, S_right);
17
18         } else {
19             // Testen, ob Bedingung niemals erfuellt ist
20             if (maxContent <= M->Value){
21                 // Bedingung ist niemals erfuellt
22                 // Wahrscheinlichkeit zu CounterProb addieren
23                 *M->CounterProb = *M->CounterProb + HeuristikMaxWMasse(M,
                       S_left, S_right);
24
25             } else {
26                 // Bedingung ist in einem bestimmten Bereich erfuellt
27                 // Wahrscheinlichkeit vom ersten Bereich zu CounterProb und
                       vom zweiten Bereich zu Prob addieren
28                 double border;
29                 border = computeBorder(M->Value, MomState->M->fluid1[M->
                       PlaceNumber], MomState->M->fluid0[M->PlaceNumber]);
30                 if (MomState->t1 >= 0){
31                     *M->CounterProb = *M->CounterProb + HeuristikMaxWMasse(M,
                           S_left, border);
32                     *M->Prob = *M->Prob + HeuristikMaxWMasse(M, border,
                           S_right);
33                 } else {
34                     *M->Prob = *M->Prob + HeuristikMaxWMasse(M, S_left, border
                           );
35                     *M->CounterProb = *M->CounterProb + HeuristikMaxWMasse(M,
                           border, S_right);
```

```
36                  }
37              }
38          }
39      } else {
40          // Es handelt sich um einen diskreten Platz
41          if (MomState->M->tokens[M->PlaceNumber] >= (int)M->Value){
42              // Die Bedingung ist erfuellt
43              *M->Prob = *M->Prob + HeuristikMaxWMasse(M, S_left, S_right);
44          } else {
45              // Die Bedingung ist nicht erfuellt
46              *M->CounterProb = *M->CounterProb + HeuristikMaxWMasse(M, S_left,
                    S_right);
47          }
48      }
49  }
```

Listing 6.6: Funktion computeProb

Als Eingangsbedingung für die Funktion *computeProb* gilt, dass der mitgelieferte Zustand möglich ist, sowie, dass der Gültigkeitsbereich, der durch S_left und S_right angegeben ist, richtig ist.
Abhängig davon, ob der betrachtete Platz ein kontinuierlicher oder diskreter Platz ist, wird in den aufwendigeren (then-Fall Zeile 4 - 38) oder einfachen Berechnungspfad (else-Fall Zeile 40 - 48) verzweigt. Der diskrete Fall ist deutlich einfacher, da die Markierung in einem Zustand konstant ist. So muss in diesem Fall lediglich überprüft werden, ob die Bedingung erfüllt ist oder nicht (Zeile 41). Im kontinuierlichen Fall kann sich die Markierung abhängig von s verändern. Es gilt zu überprüfen, ob der Mindestinhalt des Platzes schon größer als der vorgegebene Inhalt ist (Zeile 13). In diesem Fall ist die Bedingung ψ immer erfüllt. Wenn der Maximalinhalt kleiner als der vorgegebene Inhalt ist (Zeile 20), ist die Bedingung ψ niemals erfüllt. Wenn es im Intervall $[S_{left}, S_{right}]$ eine Grenze gibt, für die die Bedingung gilt,l wird die Wahrscheinlichkeit des Intervalls der jeweiligen Teilbereiche zur Wahrscheinlichkeit $\tilde{\pi}^{\psi}$ und Gegenwahrscheinlichkeit $\tilde{\pi}^{\neg\psi}$ hinzuaddiert (Zeile 30 -36). In allen anderen Fällen wird die Wahrscheinlichkeit des Intervalls entweder zur Wahrscheinlichkeit $\tilde{\pi}^{\psi}$ oder Gegenwahrscheinlichkeit $\tilde{\pi}^{\neg\psi}$ hinzuaddiert.
Die Wahrscheinlichkeit des Intervalls berechnet sich abhängig von der Wahrscheinlichkeitsdichte. In der Funktion *WStammfunktion* wurden die Stammfunktionen für die Wahrscheinlichkeitsdichte der Gleichverteilung und der Gaußschen Glockenkurve implementiert. Durch auskommentieren, kann eine der beiden Funktionen ausgewählt werden. Die Wahrscheinlichkeitsmasse berechnet sich durch einsetzen der beiden Intervallgrenzen in die Funktion nach der folgenden Formel.

$$\int_{y}^{x} f(x)dx = F(x) - F(y)$$

Wobei y für die untere Grenze und x für die obere Grenze des Intervalls und $f(x)$ für die Funktion der Wahrscheinlichkeitsdichte steht.

6.5. Berechnung des Wahrscheinlichkeitsbereiches

Um die Aussage über den Wahrscheinlichkeitsbereich einer Eigenschaft treffen zu können, sind alle, in den vorherigen Abschnitten vorgestellten Funktionen, notwendig. Die Funktion *computeMeasuresEuler*, welche in *Measures.c* implementiert ist, vereint all diese Bereiche und wird im folgenden vorgestellt.

Implementierung

Die Funktion *computeMeasuresEuler* besteht im wesentlichen aus 3 Bereichen: Initialisierung, While-Schleife für die Berechnung der Wahrscheinlichkeit und der Ausgabe der Ergebnisse. Die für diesen Abschnitt interessante While-Schleife ist im folgenden Quellcode 6.7 zu sehen.

```
1    while ((*M->Prob < M->ProbLimit) && (*M->CounterProb <= (1 - M->ProbLimit)
         ) && (M->Naechste != NULL)) {
2
3        printf("\n Nach %d besuchten Zustaenden lag die Wahrscheinlichkeit bei
              %g.\n", M->StatesVisited, *M->Prob);
4        printf("\n Nach %d besuchten Zustaenden lag die
             Gegenwahrscheinlichkeit bei %g.\n", M->StatesVisited, *M->
             CounterProb);
5
6        // Naechsten Zustand liefern
7        MomState = giveNextState(M);
8        M->StatesVisited++;
9
10       *S_left = MomState->leftInt;
11       *S_right = MomState->rightInt;
12
13       // Testen, ob man in dem Zustand sein kann, dabei werden S_left und
             S_right gesetzt
14       if (possibleState(MomState, M->T, S_left, S_right) == 1){
15
16           // Berechnen, fuer welchen Wert von S man in diesem Zustand ist
17           if (MomState->nextStateList == NULL){
18               // Es gibt keine Folgezustaende, berechne die
                     Wahrscheinlichkeiten ueber den gesamten Bereich von S_left
                      bis S_right
19               computeProb(MomState, M, *S_left, *S_right);
20
21           } else {
22               // Zu analysieren, welchen Wertebereich S haben kann
23               // Es gibt mehrere moegliche Zustaende, der Bereich von S wird
                      kontinuierlich eingeschraenkt
24               struct StateTimeAlt_tag *altStatesList = MomState->
                     nextStateList;
25
26               double leftInts[1];
27               double rightInts[1];
28
29               Intervall *inter;
30               inter = (Intervall *)malloc(sizeof(Intervall));
31               inter->Anzahl_Intervalle = 1;
32               inter->leftInts = leftInts;
33               inter->rightInts = rightInts;
34
35               if (*S_left <= M->S_Max){
36                   inter->leftInts[0] = *S_left;
37               } else {
38                   inter->leftInts[0] = M->S_Max;
```

```
39                  }
40
41                  if (*S_right <= M->S_Max){
42                     inter->rightInts[0] = *S_right;
43                  } else {
44                      inter->rightInts[0] = M->S_Max;
45                  }
46
47                  inter->Gesamtlaenge = inter->rightInts[0] - inter->leftInts
                        [0];
48
49                  // Teste alle nachfolgenden Zustaende, hoere auf, falls die
                        Gesamtlaenge des Intervalls = 0 ist
50                  while ((altStatesList != NULL) && (inter->Gesamtlaenge > 0)){
51                      StateProbAlt *altState = altStatesList->sa;
52                      State *state = altState->S;
53
54                      *S_left_alt = state->leftInt;
55                      *S_right_alt = state->rightInt;
56
57                      // Bereich von S weiter eingrenzen
58                      // Testen ob der Zustand moeglich ist
59                      if (possibleState(state, M->T, S_left_alt, S_right_alt)){
60                          // Berechnen des Bereiches
61                          intersect(inter, *S_left_alt, *S_right_alt);
62                      }
63                      altStatesList = altStatesList->next;
64                  }
65
66                  // Wenn die Gesamtlaenge des Intervalls nicht 0 ist, berechne
                        die Wahrscheinlichkeit
67                  if (inter->Gesamtlaenge > 0){
68
69                      // Wahrscheinlichkeiten dazu addieren, dazu wird jedes
                            Teilintervall von inter getestet
70                      for (i = 0; i < inter->Anzahl_Intervalle; i++){
71                          if (inter->leftInts[i] != inter->rightInts[i]){
72                              computeProb(MomState, M, inter->leftInts[i], inter
                                    ->rightInts[i]);
73                          }
74                      }
75                  }
76
77                  free(inter);
78
79              }
80          }
81      }
```

Listing 6.7: While-Schleife zur Berechnung der Wahrscheinlichkeit

Ist keine der Abbruchbedingungen (Erreichen der Wahrscheinlichkeitsgrenze zum Akzeptieren des Wahrscheinlichkeitsbereichs, zum Ablehnen des Wahrscheinlichkeitsbereichs und das keine Zustände mehr vorhanden sind) der While-Schleife erfüllt (Zeile 1), so wird zunächst ein neuer Zustand durch die Funktion *giveNextState* ausgewählt und berechnet (Zeile 7). Ist der Zustand erreichbar, so kann über diesem Zustand, Wahrscheinlichkeitsmasse gesammelt werden (then-Fall Zeile 15 - 81).

Besitzt der Zustand keine Folgezustände, so kann bereits die Wahrscheinlichkeit des Zustands über dem Gültigkeitsbereich des Zustandes berechnet werden, da kein Folgezustand den Gültigkeitsbereich einschränken kann (Zeile 19). Sind jedoch Folgezustände vorhanden, muss der Gültigkeitsbereich des momen-

tan betrachteten Zustandes ggf. eingeschränkt werden. Dazu wird das Struct *Intervall* angelegt (Zeile 29 - 48), welches dazu gedacht ist, die Einschränkungen darzustellen. Genauere Informationen dazu befinden sich auf Seite 31 in Abschnitt 6.3.
In der folgenden While-Schleife, wird der ursprüngliche Gültigkeitsbereich eingeschränkt. Dazu wird jeder Folgezustand des betrachteten Zustands überprüft. Falls ein Folgezustand erreichbar ist (then-Fall Zeile 61 - 63), wird die Funktion *intersect* aufgerufen (Zeile 62), diese schneidet das ursprüngliche Intervall mit dem Intervall des Gültigkeitsbereiches des Folgezustandes, falls es Überschneidungen gibt. Ist dies für alle Folgezustände durchgeführt worden, wird die übrig gebliebene Wahrscheinlichkeitsmasse auf die bisherigen Wahrscheinlichkeiten $\tilde{\pi}^{\psi}$ und $\tilde{\pi}^{\neg\psi}$ hinzuaddiert. Dies geschieht, falls, überhaupt Wahrscheinlichkeitsmasse vorhanden ist, in der for-Schleife (Zeile 70 - 75). Hier wird für jedes Teilintervall die Funktion *computeProb* aufgerufen (Zeile 73).
Zu guter Letzt, werden nach Beendigung der While-Schleife die Ergebnisse ausgegeben und der Speicher, der für die Berechnung verwendet wurde, wieder freigegeben.

7. Case Study

Dieses Kapitel dient dazu die, in Abschnitt 5.4 auf Seite 26 beschriebenen, Heuristiken miteinander zu vergleichen. Als Referenz dient dabei den ursprünglichen Algorithmus[1] und das Berechnen der Wahrscheinlichkeit mit Hilfe der Eulertour, auf dem vollständigen Zustandsraum.
Da es zu Schwankungen in der Rechenzeit der Tests kommen kann, wurden alle Tests mehrfach durchgeführt und ein Mittelwert gebildet.

7.1. Allgemeines

Die implementierten Heuristiken haben alle unterschiedliche Eigenschaften und unterschiedlich aufwendige Algorithmen. Grundsätzlich kann man jedoch sagen, dass je größer ein Zustandsraum wird, desto größer kann der Vorteil einer aufwendigen Heuristik sein. Jedoch ist auch nicht immer garantiert, dass die aufwendigere Heuristik den besseren Zustand liefert. Dies wird an dem Zustandsraum 4.2 aus Kapitel 4 auf Seite 16 klar. Angenommen die Zustände 0, 1 und 2 wurden bereits betrachtet und die Zustände 3, 4 und 5 sind mögliche nächste Zustände. So würden bis auf eine Heuristik alle Zustand 5 liefern, da dort eine größere Wahrscheinlichkeitsmasse vorhanden ist. Im Gegensatz dazu würde die einfachste Heuristik, die nach der Zustandsnummer entscheidet, Zustand 3 wählen und dort die vielleicht entscheidende Masse zum treffen einer Aussage finden. Zustand 5 hingegen kann keine Wahrscheinlichkeitsmasse liefern, da durch die beiden Folgezustände der komplette Gültigkeitsbereich abgedeckt wurde.

7.2. Direkter Vergleich von Heuristiken

Heuristik 2: Länge des Gültigkeitsbereiches - Heuristik 3: Maximale Wahrscheinlichkeitsmasse
Wie schon in Abschnitt 5.4 beschrieben unterscheiden sich diese beiden Heuristiken nur geringfügig. Anhand von verschiedenen Tests sollen die Heuristiken nun anhand ihrer benötigten Zustände und der benötigten Rechenzeit miteinander verglichen werden.
Die folgende Tabelle 7.1 zeigt die Tests, die mithilfe verschiedener Modelle, verschiedenen Eigenschaften und einer gleichverteilten Wahrscheinlichkeits-

[1] Verfügbar unter `https://code.google.com/p/fluid-survival-tool/` (B.F. Postema, A.K.I. Remke, H. Ghasemieh)

#	Model	Platz	Inhalt	Grenze	Zeit	Heuristik	Zustände	Rechenzeit (in μsec)
1	model.m	pumpOn	1	0.7	4	2 3	5 5	494 479
2	BestWBorder.m	reservoir	5	0.55	10	2 3	9 9	811 829
3	model3.m	reservoir	3	0.85	7	2 3	2 2	231 233
4	dsn_easy.m	soft1	6	0.7	7	2 3	5 5	409 411
5	dsn.m	filt1	10	0.2	4	2 3	4 4	367 373
6	dsn_very_easy.m	filt1	2	0.6	10	2 3	7 7	314 324

Tabelle 7.1.: Vergleich der Heuristiken 2 und 3 bei gleichverteilter Wahrscheinlichkeitsdichte

dichte durchgeführt wurden.
Wie bereits vermutet, wurden immer exakt gleich viele Zustände betrachtet, da das Bewertungskriterium das Gleiche ist. Bis auf eine Ausnahme in Test 1 war die zweite Heuristik immer geringfügig schneller, dies kann darauf zurückgeführt werden, dass Heuristik 3 etwas aufwendiger in der Bewertung der Zustände ist. Jedoch macht dies im Mittel nur einen Unterschied von ca. 4 Mikrosekunden, weshalb der Unterschied zu vernachlässigen ist.

Ändert man die Funktion der Wahrscheinlichkeitsdicht, so ergibt sich für die Heuristiken eine unterschiedliche Bewertungsgrundlage. Tabelle 7.2 zeigt die

#	Model	Platz	Inhalt	Grenze	Zeit	Heuristik	Zustände	Rechenzeit (in μsec)
1	dsn_easy.m	soft1	9	0.6	15	2 3	14 14	1340 2050
2	dsn_very_easy.m	filt1	1	0.45	4	2 3	4 4	231 245
3	dsn.m	filt1	5	0.6	10	2 3	9 13	1086 1724
4	model-f2.m	fp1	4	0.4	3	2 3	4 3	517 428
5	BestWMasse.m	reservoir	3	0.6	10	2 3	9 5	1176 448
6	model-extendet.m	reservoir	6	0.7	9	2 3	4 3	541 417

Tabelle 7.2.: Vergleich der Heuristiken 2 und 3 bei gauß-verteilter Wahrscheinlichkeitsdichte

durchgeführten Tests mit einer Wahrscheinlichkeitsdichte, welche nach der Gaußschen Glockenkurve mit dem Erwartungswert 5 und der Standartabweichung 1 verteilt wurde.
Insgesamt schneidet die Heuristik 2, bei den berechneten Zuständen, nur in Test 3 besser ab, dort jedoch mit 4 Zuständen. Über alle Tests hat Heuristik 3 für die Tests 2 Zustände weniger berechnet. Im Vergleich der Rechenzeiten ist die Heuristik 2 deutlich vorzuziehen. Durchschnittlich benötigte Heuristik 2 815 Mikrosekunden, Heuristik 3 jedoch 885 Mikrosekunden, was etwa 8,5%

mehr sind.
Anhand der Ergebnisse der Tests ist die Heuristik 2 zu bevorzugen, da sie durchschnittlich weniger Rechenzeit über alle Tests benötigt. Jedoch ist Heuristik 3 etwas effizienter, was die Anzahl der betrachteten Zustände betrifft.

Heuristik 3: Maximale Wahrscheinlichkeitsmasse - Heuristik 4: Nähe zur Wahrscheinlichkeitsgrenze in Abhängigkeit der Wahrscheinlichkeitsmasse

Die Heuristik 4 ist von allen betrachteten Heuristiken die, von der Berechnung, Aufwendigste. Nun soll überprüft werden, ob sich dieser Aufwand auch in der Effizienz auswirkt.
In der folgenden Tabelle 7.3 sind die Tests mit gleichverteilter Wahrscheinlich-

#	Model	Platz	Inhalt	Grenze	Zeit	Heuristik	Zustände	Rechenzeit (in musec)
1	model.m	pumpOn	1	0.7	4	3	5	479
						4	5	467
2	BestWBorder.m	reservoir	5	0.55	10	3	9	829
						4	8	776
3	dsn_easy.m	soft1	6	0.7	7	3	5	411
						4	5	446
4	dsn.m	filt1	10	0.2	4	3	4	373
						4	4	389
5	dsn_very_easy.m	filt1	2	0.6	10	3	7	324
						4	7	339

Tabelle 7.3.: Vergleich der Heuristiken 3 und 4 bei gleichverteilter Wahrscheinlichkeitsdichte

keitsdichte aufgeführt.
Wie zu sehen ist, unterscheiden sich die beiden Heuristiken nur in Test 2 voneinander. Somit ist die Heuristik 4 leicht besser, was die betrachteten Zustände angibt. Wenn man jedoch die Rechenzeit miteinander vergleicht stellt man fest, dass die Tests durchschnittlich gleich lange benötigen. In diesem Fall sind also Heuristik 3 und 4 gleichzusetzen.
Nun sollen die beiden Heuristiken auch mit einer anderen Wahrscheinlichkeitsdichte verglichen werden. So wird wieder die gauß-verteilte Wahrscheinlichkeitsdichte betrachtet. Tabelle 7.4 zeigt die durchgeführten Testaufrufe.

#	Model	Platz	Inhalt	Grenze	Zeit	Heuristik	Zustände	Rechenzeit (in μsec)
1	BestWMasse.m	reservoir	3	0.6	10	3	5	448
						4	6	583
2	model-extendet.m	reservoir	6	0.7	9	3	3	417
						4	5	635
3	model-f2.m	fp1	4	0.4	3	3	3	428
						4	5	661
4	dsn.m	filt1	5	0.6	10	3	13	1724
						4	8	997
5	dsn_easy.m	soft1	9	0.6	15	3	14	2050
						4	19	2157

Tabelle 7.4.: Vergleich der Heuristiken 3 und 4 bei gauß-verteilter Wahrscheinlichkeitsdichte

Trotz der aufwendigeren Berechnung, werden in Heuristik 4 mehr Zustände berechnet, was daran liegen könnte, dass zu sehr versucht wurde eine Wahrscheinlichkeitsschranke zu erreichen. Obwohl so durchschnittlich 1 Zustand mehr betrachtet wurde, ist die Rechenzeit im Vergleich zu Heuristik 3 geringer. Dies liegt vermutlich daran, dass im dsn.m-Model, weniger Zustände berechnet werden mussten, da dieses Model sehr groß ist, ist die Berechnung der Zustände sehr aufwendig.
Anhand dieser Testergebnisse kann keine Empfehlung für oder gegen eine der beiden Heuristiken ausgesprochen werden, da die Rechenzeit jeweils nahezu identisch ist. Jedoch ist festzustellen, dass die Heuristik 3 weniger Zustände benötigte um eine Aussage über den Wahrscheinlichkeitsbereich der Eigenschaft zu treffen.

7.3. Allgemeiner Vergleich

In diesem Abschnitt werden alle Heuristiken, sowie die Berechnung auf dem vollständigen Zustandsraum und der ursprüngliche Algorithmus miteinander verglichen. Dazu werden im ersten Unterabschnitt dieses Abschnittes die Berechnungen auf dem in dieser Bachelorarbeit entwickelten Algorithmus verglichen. Bevor zu guter Letzt die Rechenzeiten des modifizierten Algorithmus mit denen des ursprünglichen Algorithmus verglichen werden. Die einzelnen Testergebnisse sind in der Tabelle A.1 auf Seite 50 zu sehen.

7.3.1. Vergleiche im modifizierten Algorithmus

Vergleich der berechneten Zustände
Das folgende Diagramm 7.1 zeigt, wie viele Zustände durchschnittlich berechnet werden mussten, um eine Wahrscheinlichkeitsgrenze zu erreichen. Einen deutlichen Unterschied macht es, ob die Abbruchbedingungen verwendet oder weggelassen wurden. So erhöhte sich die Anzahl der berechneten Zustände um ca. 240%. Beim Vergleich der verschiedenen Heuristiken ist zu erkennen, dass die Heuristik 2, die die Länge des Gültigkeitsbereiches liefert, geringfügig schlechter abschneidet, als die Heuristik 3, welche die Wahrscheinlichkeitsmasse berechnet. Dies ist darauf zurückzuführen, dass gerade bei der Verteilung der Wahrscheinlichkeitsmasse in der Glockenkurve die großen Randbereiche kaum Wahrscheinlichkeitsmasse halten und es somit sinnvoller ist die kleineren Bereiche zu wählen, welche näher am Erwartungswert liegen.
Die aufwendige Heuristik 4 benötigt mehr Zustände als die Heuristik 3, obwohl diese eine Modifikation darstellt. Dies liegt daran, dass Heuristik 4 versucht hat die nähere Wahrscheinlichkeitsgrenze zu erreichen, obwohl dies nicht möglich war.
Die einfachste Heuristik 1, welche die Zustände nach der Zustandsnummer auswählt, lieferte das schlechteste Ergebnis der Heuristiken ab. Dies ist darauf

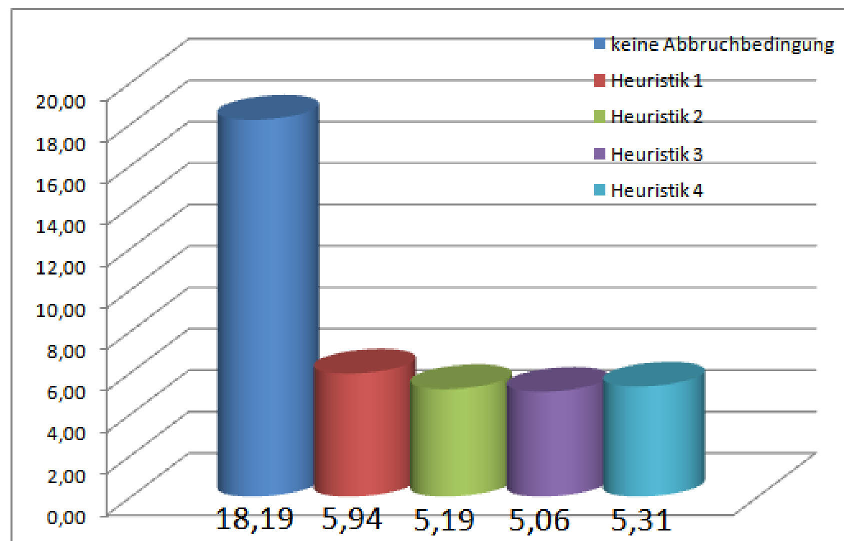

Abbildung 7.1.: Vergleich der verschiedenen Heuristiken über die Anzahl der berechneten Zustände

zurückzuführen, dass die Zustände nicht gut gewählt werden.

Vergleich der Rechenzeit

Die für die Berechnung, unter den verschiedenen Heuristiken, benötigte Rechenzeit zeigt das folgende Diagramm 7.2. Die Werte zu den Heuristiken sind in Mikrosekunden angegeben. Das Diagramm weißt eine ähnliche Charakteris-

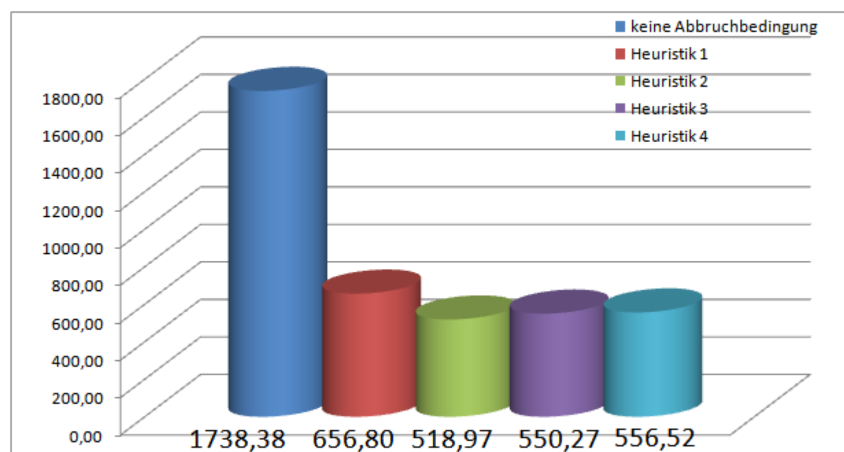

Abbildung 7.2.: Vergleich der verschiedenen Heuristiken über die benötigte Rechenzeit

tik wie Diagramm 7.1 auf. Jedoch verdrängt Heuristik 2 Heuristik 3 von der besten Platzierung. Dies ist darauf zurückzuführen, dass Heuristik 2 weniger Rechenzeit für die Bewertung von Zuständen benötigt. Die Rechenzeit ohne Abbruchbedingung dauert deutlich am längsten. Jedoch ist zu erkennen, dass

durch die aufwendige Berechnung in den Heuristiken, die Rechenzeit nur noch ca. 200% über dem Durchschnitt der Heuristiken liegt.
Für die effiziente Berechnung ist also Heuristik 3, welche nach der Wahrscheinlichkeitsmasse entscheidet, zu wählen. Jedoch kann in kleinen Petri-Netzen mit einem kleinen Zustandsraum Heuristik 1 und in großen Petri-Netzen mit einem großen Zustandsraum unter Umständen Heuristik 4 eine bessere Effizienz aufweisen.

7.3.2. Vergleich mit dem ursprünglichen Algorithmus

Zum Vergleich zu den Werten der modifizierten Berechnung, wurde ebenfalls eine Zeitmessung im ursprünglichen Programmcode, mit allen in Tabelle A.1 verwendeten Modellen, durchgeführt. Die Zeitmessung wurde vor der Funktion *VisitStates* gestartet und endet nach Abschluss aller Berechnungen. Da das ursprüngliche Tool alle Zustände berechnet und zu jedem Platz, jedem Inhalt und jeder Zeit eine Berechnung der Wahrscheinlichkeit durchführt ist die Rechenzeit entsprechend langsamer. Im Durchschnitt über alle Modelle, benötigte der ursprüngliche Algorithmus 7:30 Minuten, gerade die Berechnung des großen dsn.m-Models benötigte ungefähr 1:21 Stunden. Dies entspricht ca. dem 560.000-fachen der durchschnittlich benötigten Zeit mithilfe des neuen Programmablauf.
Somit gilt, sind nur bestimmte Wahrscheinlichkeitsgrenzen für den Nutzer interessant, so ist die Berechnung über den in dieser Bachelorarbeit implementierten Algorithmus vorzuziehen. Lediglich, wenn die gesamten Wahrscheinlichkeiten benötigt werden, ist das ursprüngliche *DFPN*-Tool weiterhin zu verwenden.

Alle angegebenen Ergebnisse wurden auf einem System mit einem Intel Core i7-4702MQ Quadcoreprozessor berechnet. Als Betriebssystem wurde über eine virtuelle Maschine Linux in der Version Debian 7 64-bit verwendet, dem 4GB Arbeitsspeicher zugewiesen worden sind.

8. Zusammenfassung

In dieser Bachelorarbeit wurden zunächst die Grundlagen von hybriden Petri-Netzen erklärt. Dazu wurde das ursprüngliche Petri-Netz sowohl mathematisch beschrieben, als auch an einem Beispiel erklärt. Die koninuierliche Variante des Petri-Netzes veranschaulichte den zweiten Bereich des hybriden Petri-Netzes.

Anschließend wurde erklärt, wie für hybride Petri-Netze mit einer stochastischen Transition ein Zustandsraum berechnet werden kann. Es wurde ein Konzept vorgestellt, wie die Berechnung des Zustandsraumes effizienter gestaltet werden kann. Dazu wurde von der exakten Berechnung der Wahrscheinlichkeit abgerückt und auf die Angabe eines Wahrscheinlichkeitsbereiches beschränkt. Im Anschluss wurde erklärt, wie die Berechnung einer Wahrscheinlichkeit zu einem Ereignis durchgeführt werden kann und konzeptionell ein Algorithmus, zur Berechnung von Wahrscheinlichkeiten zu einem Ereignis, entwickelt. Durch das Konzept der Wahrscheinlichkeitsschranken musste nicht mehr der gesamte Zustandsraum berechnet werden, sondern nur noch einzelne Zustände. Zum Wählen des, als nächstes zu betrachtenen, Zustandes, zum Sammeln weiterer Wahrscheinlichkeitsmasse, wurden verschiedene Heuristiken vorgestellt.

Die Implementierung des Konzeptes wurde in verschiedene Bereiche aufgeteilt und getrennt voneinander betrachtet. Die im Vorhinein entwickelten Algorithmen wurden umgesetzt und erklärt. Anschließend wurden die verschiedenen Heuristiken miteinander verglichen und eine Empfehlung für die weitere Verwendung gegeben. Desweiteren wurde aufgezeigt unter welchen Umständen der ursprüngliche Algorithmus von Vorteil ist und wann, der in dieser Bachelorarbeit erarbeitete, Algorithmus zu nutzen ist.

Ausblick
Die Bachelorarbeit hat gezeigt, dass viel Rechenzeit durch das Betrachten von Wahrscheinlichkeitsintervallen gespart werden kann, jedoch ist auch der Informationsverlust durch diesen Algorithmus nicht zu vernachlässigen.

Um mehr Informationen über das Petri-Netz zu gewinnen, wäre es möglich zu mehreren Eigenschaften oder Wahrscheinlichkeitsintervallen eine Aussage zu treffen. Dazu müssten neue Heuristiken entwickelt werden, die eine Bewertung der Zustände anhand von mehreren Eigenschaften vornehmen.

Desweiteren können auch für den jetzigen Algorithmus noch effizientere Heuristiken entwickelt werden, um noch weniger Zustände betrachten zu müssen.

A. Anhang

#	Model	Platz	Inhalt	Grenze	Zeit	Verteilung	Heuristik	Zustände	Rechenzeit (in μsec)
				-			0	5	455
							1	4	423
1	model.m	pumpOn	1	0.7	4	1	2	5	494
							3	5	479
							4	5	467
				-			0	9	762
							1	6	601
2	BestWMasse.m	reservoir	3	0.6	10	2	2	9	1176
							3	5	448
							4	6	583
				-			0	4	335
							1	3	268
3	model2.m	reservoir	3	0.7	6	2	2	3	271
							3	3	288
							4	3	357
				-			0	9	802
							1	8	725
4	BestWBorder.m	reservoir	5	0.55	10	1	2	9	811
							3	9	829
							4	8	776
				-			0	4	397
							1	2	214
5	model3.m	reservoir	3	0.85	7	1	2	2	231
							3	2	233
							4	2	219
				-			0	5	501
							1	3	345
6	model_dis.m	reservoir	2.5	0.6	3	2	2	2	246
							3	2	274
							4	2	311
				-			0	6	734
							1	4	524
7	model-extendet.m	reservoir	6	0.7	9	2	2	4	541
							3	3	417
							4	5	635
				-			0	6	743
							1	4	477
8	model-f2.m	fp1	4	0.4	3	2	2	4	517
							3	3	428
							4	5	661
				-			0	2	211
							1	1	137
9	model-fc.m	fp	4	0.6	7	2	2	1	146
							3	1	150
							4	1	155
				-			0	1	149
							1	1	157
10	model-feedback.m	fp0	2	0.8	5	2	2	1	124
							3	1	131
							4	1	142
				-			0	64	9871
							1	16	2014
11	dsn.m	filt1	5	0.6	10	2	2	9	1086
							3	13	1724
							4	8	997
				-			0	17	1322
							1	7	594
12	dsn_easy.m	soft1	6	0.7	7	1	2	5	409
							3	5	411
							4	5	446
				-			0	44	1873
							1	4	517
13	dsn_very_easy.m	filt1	1	0.45	4	2	2	4	231
							3	4	245
							4	4	270
				-			0	37	3998
							1	4	342
14	dsn.m	filt1	10	0.2	4	1	2	4	367
							3	4	373
							4	4	389
				-			0	34	3167
							1	21	2873
15	dsn_easy.m	soft1	9	0.6	15	2	2	14	1340
							3	14	2050

#	Model	Platz	Inhalt	Grenze	Zeit	Verteilung	Heuristik	Zustände	Rechenzeit (in μsec)
							4	19	2157
16	dsn_very_easy.m	filt1	2	-	10	1	0	44	2494
				0.6			1	7	298
							2	7	314
							3	7	324
							4	7	339

Tabelle A.1.: Alle Heuristiken im Vergleich

Abbildungsverzeichnis

2.1. Symbole des diskreten Petri-Netzes 4
2.2. Allgemeines Petri-Netz: Ausgangssituation 5
2.3. Allgemeines Petri-Netz: erster Schritt 5
2.4. Symbole des kontinuierlichen Petri-Netzes 6
2.5. Beispiel eines kontinuierlichen Petri-Netzes 7
2.6. Verlauf des Inhaltes der Plätze P1 und P2 7
2.7. Symbole des hybriden Petri-Netzes 8
2.8. Hybrides Petri-Netz 9

3.1. Symbole des HPnG-Modells 11
3.2. Beispiel zu den Bedingungsfolgen 12
3.3. Petri-Netz der Modellklasse HPnG 13

4.1. HPnG Petri-Netz 15
4.2. Zustandsbaum berechnet durch das DFPN-Tool 16

5.1. Alter Programmablauf des DFPN-Tools 20
5.2. Neu gestalteter Programmablauf 20
5.3. Das Königsberger Brückenproblem 24
5.4. Das Königsberger Brückenproblem als Graph 25
5.5. Der Zustandsbaum als gerichteter Graph 26
5.6. Normalverteilung Vergleich zweier Gültigkeitsbereiche 27

7.1. Vergleich der verschiedenen Heuristiken über die Anzahl der berechneten Zustände 45
7.2. Vergleich der verschiedenen Heuristiken über die benötigte Rechenzeit 45

Tabellenverzeichnis

5.1. Übersicht über verschiedene mögliche Heuristiken 27

7.1. Vergleich der Heuristiken 2 und 3 bei gleichverteilter Wahrscheinlichkeitsdichte 42
7.2. Vergleich der Heuristiken 2 und 3 bei gauß-verteilter Wahrscheinlichkeitsdichte 42
7.3. Vergleich der Heuristiken 3 und 4 bei gleichverteilter Wahrscheinlichkeitsdichte 43
7.4. Vergleich der Heuristiken 3 und 4 bei gauß-verteilter Wahrscheinlichkeitsdichte 43

A.1. Alle Heuristiken im Vergleich 50

Listings

4.1. Struct des States 17
4.2. Struct des Models 18

5.1. Pseudocode zur Berechnung des Gültigkeitsbereiches 22
5.2. Pseudocode zur Berechnung Wahrscheinlichkeit eines Zustandes . . . 23

6.1. Struct der Zustandskette 29
6.2. Funktion giveNextState 30
6.3. Funktion possibleState in Measures.c 31
6.4. Struct des Intervalls 33
6.5. Funktion intersect 33
6.6. Funktion computeProb 35
6.7. While-Schleife zur Berechnung der Wahrscheinlichkeit 37

Literatur

[Bol06] Dietrich Boles. „Programmierung“. In: *Programmieren spielend gelernt*. Teubner, 2006, S. 28. URL: http://dx.doi.org/10.1007/978-3-8351-9002-3_1.

[DA05a] René David und Hassane Alla. *Discrete, Continuous, and Hybrid Petri Nets*. Springer Science & Business Media, 2005, S. V–IX.

[DA05b] René David und Hassane Alla. *Discrete, Continuous, and Hybrid Petri Nets*. Springer Science & Business Media, 2005, S. 3–4.

[DA05c] René David und Hassane Alla. *Discrete, Continuous, and Hybrid Petri Nets*. Springer Science & Business Media, 2005, S. 111–114.

[DA05d] René David und Hassane Alla. *Discrete, Continuous, and Hybrid Petri Nets*. Springer Science & Business Media, 2005, S. 122–123.

[Gha+12] H. Ghasemieh u. a. „Region-Based Analysis of Hybrid Petri Nets with a Single General One-Shot Transition“. In: *Formal modeling and analysis of timed systems*. Bd. 7595. Lecture Notes in Computer Science. London, UK: Springer Verlag, Sep. 2012, S. 139–154. URL: http://doc.utwente.nl/83495/.

[GR10] Marco Gribaudo und Anne Remke. „Hybrid petri nets with general one-shot transitions for dependability evaluation of fluid critical infrastructures“. In: *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*. IEEE. 2010.

[Har15] Peter Hartmann. „Natürliche Zahlen, vollständige Induktion, Rekursion“. In: *Mathematik für Informatiker*. Springer Fachmedien Wiesbaden, 2015, S. 62–63. URL: http://dx.doi.org/10.1007/978-3-658-03416-0_3.

[Ste07] Angelika Steger. „Graphentheorie“. In: *Diskrete Strukturen*. Springer-Lehrbuch. Springer Berlin Heidelberg, 2007, S. 57–102. URL: http://dx.doi.org/10.1007/978-3-540-46664-2_3.

Eidesstattliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit über

Effiziente Traversierung von Zustandsbäumen zur Prüfung von Wahrscheinlichkeitsgrenzen in Hybriden Petri Netzen

selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommenen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Christopher Distelkämper, Münster, 28. April 2015

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in eine Datenbank einverstanden.

Christopher Distelkämper, Münster, 28. April 2015