technische universität
dortmund

Masterarbeit

Funktionale Analyse von Queueing-Petri-Netzen durch Reduktion der Netzstruktur

Ella Albrecht
2. Februar 2015

Gutachter:
Dr. Falko Bause
Prof. Dr. Peter Buchholz

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für praktische Informatik (LS 4)
http://ls4-www.cs.tu-dortmund.de

Zusammenfassung

Petri-Netze sind ein beliebtes Modell, wenn es um die Modellierung von nebenläufigen Prozessen und Synchronisationsvorgängen geht. Die Analyse solcher Netze beruht häufig auf der Untersuchung des Zustandsraumes. Leider tritt dabei häufig das Problem der Zustandsraumexplosion auf. Eine Möglichkeit, mit diesem Problem umzugehen ist die Reduktion der Netzstruktur. Diese Arbeit betrachtet gängige Reduktionsregeln und passt sie an die Reduktion von Queueing-Petri-Netzen an. Anschließend werden die Regeln in das Open-Source-Tool QPME eingebaut und deren Nutzen für die funktionale Analyse von Queueing-Petri-Netzen untersucht.

Abstract

Petri nets are a popular model for modelling synchronisation and concurrent systems. These nets are often analysed by state space exploration. Unfortunatly for bigger nets the so-called problem of state explosiion arises. One approach to cope with the problem is to reduce the net before state space exploration. In this thesis common reduction rules are regard and adapted for queueing petri nets. After that the rules are implemented into the open-source tool QPME and their practical effect for the qualitative analysis of queueing petri nets is examined on a few examples.

Inhaltsverzeichnis

5

Mathematische Notation

Allgemein

Notation	Bedeutung
$\mathbb{N}$	Natürliche Zahlen ohne Null
$\mathbb{Z}$	Ganze Zahlen
$\mathbb{R}$	Reelle Zahlen
$\mathbb{R}^+$	Positive reelle Zahlen
$\mathbf{x}, \mathbf{y}, \mathbf{c}$	Ein n-dimensionaler Vektor mit den Elementen x_1,...,x_n
a, b, c	Eine eindimensionale, skalare Variable
A, B, C	Eine Menge
$\|M\|$	Mächtigkeit einer Menge M
$\mathcal{A}, \mathcal{B}, \mathcal{C}$	Ein Tupel
A_{MS}	Multimenge über eine Menge A

Petri-Netze

Notation	Bedeutung
$\mathcal{P}$	Petri-Netz $\mathcal{P} = (\mathcal{N}, m_0)$
$\mathcal{N}$	Netzstruktur $\mathcal{N} = (P, T, I^-, I^+)$
P	Menge von Stellen $p_1, p_2, \ldots, p_m$
T	Menge von Transitionen $t_1, t_2, \ldots, t_n$
I^-	Rückwärtsinzidenzfunktion
I^+	Vorwärtsinzidenzfunktion
m_0	Startmarkierung
m	Markierung
${}^\bullet x$	Vorbereich einer Stelle/Transition x

$x^\bullet$	Nachbereich einer Stelle/Transition x
$EN_T(m)$	Menge der aktivierten Transitionen in einer Markierung m
$m \xrightarrow{t}$	Transition t ist in Markierung m aktiviert
$m \xrightarrow{t} m'$	Markierung m' kann von Markierung m nach Feuern von Transition t erreicht werden
σ	Feuerfolge
T^*	Menge aller Feuerfolgen eines Petri-Netzes
$RS(\mathcal{N}, m))$	Menge aller von Markierung m erreichbaren Markierungen in einem Petri-Netz $\mathcal{P}$
$RG(\mathcal{P}$	Erreichbarkeitsgraph eines Petri-Netzes $\mathcal{P}$
V	Knotenmenge eines Graphen
E	Kantenmenge eines Graphen
(m, t, m')	Kanten von Knoten m nach Knoten m' mit Label t
C	Farbfunktion
$\mathcal{N}_C$	gefärbtes Netz $\mathcal{N}_C = (P, T, C, I^-, I^+)$
$\mathcal{CPN}$	gefärbtes Petri-Netz $\mathcal{CPN} = (\mathcal{N}_C, m_0)$
$\mathcal{GSPN}$	Generalised-Stochastic-Petri-Netz $\mathcal{GSPN} = (P, T_1, T_2, W, m_0)$
T_1	Menge von zeitbehafteten Transitionen
T_2	Menge von zeitlosen Transitionen
W	Feuergewicht/Feuerrate vonm Transitionen $(w_1, \ldots, w_{\lvert T\rvert})$ mit $w_i \in \mathbb{R}^+$
$\mathcal{QPN}$	Queueing-Petri-Netz $\mathcal{QPN} = (\mathcal{CPN}, Q, W, m_0)$
$\tilde{Q}_1$	Menge von Queueing-Stellen
$\tilde{Q}_2$	Menge von gewöhnlichen Stellen
$\tilde{W}_1$	Menge von zeitbehafteten Transitionen
$\tilde{W}_2$	Menge von zeitlosen Transitionen

Abkürzungsverzeichnis

CPN	gefärbtes Petri-Netz (*coloured petri net*)
EFC	Extended-Free-Choice-Netz
ESPL	Extended-Simple-Netz
FC	Free-Choice-netz
FCFS	First-Come-First-Served
GSPN	Generalised-Stochastic-Petri-Netz
IS	Infinite Server
LCFS	Last-Come-First-Served
MG	Synchronistaionsgraph (*marked graph*)
PRIO	Priority-Scheduling
PS	Processor Sharing
QN	Warteschlangennetz (*queueing network*)
QPN	Queueing-Petri-Netz
SM	Zustandsmaschine (*state machine*)
SPL	Simple-Netz
SZK	starke Zusammenhangskomponente

Kapitel 1

Einleitung

1.1 Motivation

Petri-Netze [39] sind ein beliebtes mathematisches Modell für nebenläufige Prozesse. Zum einen lassen sie sich durch ihre klare mathematische Definition auf zahlreiche Möglichkeiten analysieren und zum anderen bieten sie eine einfache grafische Darstellung. Die meisten Analyseverfahren basieren dabei auf der Untersuchung des Zustandsraums. Leider kann dieser bereits für einfache Netze sehr groß werden, wodurch die Berechnung aufgrund von Zeit- oder Speicherschranken unmöglich sein kann. Dieses Problem ist allgemein als Zustandsraumexplosion [45] bekannt. Um dieses Problem zu umgehen, wurden zahlreiche Ansätze entwickelten. Ein Ansatz ist, nur Netze mit bestimmten strukturellen Eigenschaften zu betrachten und für diese Analysemöglichkeiten ohne die Betrachtung des Zustandsraumes durchzuführen. Ein anderer Ansatz ist, den Zustandsraum zu verkleinern. Zum einen wurden Verfahren entwickelt, die nur einen für die Analyse relevanten Teil des Zustandsraumes erzeugen [46]. Zum anderen gibt es den Ansatz, das Petri-Netz selbst unter Beibehaltung der Eigenschaften, die man untersuchen möchte, zu reduzieren [9]. Ziel dabei ist, dass der Zustandsraum des reduzierten Netzes um ein Vielfaches kleiner ist als der des ursprünglichen Netzes.

Wenn von Petri-Netzen die Rede ist, sind in der Regel Stellen-Transition-Netze (S/T-Netze) gemeint. Neben dieser relativ einfachen Form von Petri-Netzes gibt es noch viele andere, komplexere Formen. Gefärbte Petri-Netze [28] beispielsweise ermöglichen es, zwischen verschieden Arten von Tokens zu unterscheiden. Generalised-Stochastic-Petri-Netze [36] hingegen erweitern S/T-Netze um einfache zeitliche Aspekte, indem zwischen zeitlosen und zeitbehafteten Transitionen unterschieden wird. In [5] wurden sogenannte Queueing-Petri-Netze vorgestellt. Diese vereinen gefärbte Generalised-Stochastic-Petri-Netze mit Warteschlangensystemen. Dadurch wird es ermöglicht, die qualitative und quantitative Analyse eines Systems mit nur einem Modell zu verbinden.

1.2 Ziel der Arbeit

Ziel der Arbeit ist es, die qualitative Analyse von Queueing-Petri-Netzen durch die Reduktion ihrer Netzstruktur zu ermöglichen. Dazu sollen zunächst gängige Reduktionsregeln [9; 8; 37; 18] für die Reduktion von S/T-Netzen betrachtet werden. Da Beschränktheit, Lebendigkeit und die Existenz von Heimatzuständen essenziell für die quantitative Analyse von Queueing-Petri-Netzen sind [7], wird sich nur auf Regeln beschränkt, die diese Eigenschaften unter der Reduktion beibehalten.

Durch den zeitlichen Aspekt in Queueing-Petri-Netzen können die meisten Regeln nicht einfach übernommen werden, da dabei bestimmte Eigenschaften nicht mehr erhalten werden würden. Daher sollen geeignete Regeln ausgewählt und für die Verwendung in zeitbehafteten Netzen angepasst werden. Anschließend soll das Open-Source-Tool QPME [33] um die funktionale Analyse von Queueing-Petri-Netzen erweitert werden. Dabei sollen auch die vorher entwickelten Reduktionsregeln mit eingebunden werden. Abschließend soll exemplarisch untersucht werden, ob und in welchem Maße die Reduktion Einfluss auf die Analyse hat.

1.3 Aufbau der Arbeit

Kapitel 2 gibt zunächst eine Einführung in Petri-Netze, deren Eigenschaften und die funktionale Analyse mithilfe des Erreichbarkeitsgraphen. Dann werden strukturelle Kriterien vorgestellt, mit denen sich Petri-Netze näher klassifizieren lassen. Für die meisten dieser sogenannten Netzklassen gibt es Methoden, mit denen sich die Netze ohne die Konstruktion des Erreichbarkeitsgraphen untersuchen lassen. Anschließend werden gefärbte und Generalised-Stochstic-Petri-Netze vorgestellt, die die S/T-Netze um bestimmte Aspekte erweitern und die Grundlage für die im darauffolgenden Kapitel präsentierten Queueing-Petri-Netze bilden. Kapitel 4 behandelt die Reduktion von Petri-Netzen. Dazu werden zunächst gängige Regeln zur Reduktion von S/T-Netzen erörtert. Anschließend werden die Regeln für die Reduktion von Queueing-Petri-Netzen angepasst. Kapitel 5 beschreibt die Implementierung der Regeln und der gesamten qualitativen Analyse in QPME. In Kapitel 6 wird der Nutzen der implementierten Reduktionsregeln schlussendlich an einigen Beispielen untersucht.

Kapitel 2

Petri-Netze

Petri-Netze wurden 1962 von Carl Adam Petri in seiner Dissertation [39] eingeführt. Sie ermöglichen es, Eigenschaften wie Konkurrenz, Nebenläufigkeit, Synchronisation, Parallelität und kausale Abhängigkeit darzustellen. Daher bieten sie ein breites Spektrum an Anwendungsgebieten, wie beispielsweise die Verifikation von Kommunikationsprotokollen [10; 19], die Modellierung von Arbeitsabläufen und Geschäftsprozessen [1; 40] oder der Entwurf von Fertigungssystemen [42].

Dieses Kapitel behandelt die Grundlagen zu Petri-Netzen. Dazu erläutert Abschnitt 2.1 zunächst die Struktur und das dynamische Verhalten von Petri-Netzen anhand von Stellen-Transitions-Netzen. Anschließend werden in Abschnitt 2.2 wichtige Eigenschaften von Petri-Netzen vorgestellt und dargestellt, wie diese mithilfe eines sogenannten Erreichbarkeitsgraphen bestimmt werden können. Abschnitt 2.3 behandelt, wie Petri-Netze anhand ihrer Struktur in verschiedene Klassen unterteilt werden können und welche effizienten Methoden es für die Analyse der einzelnen Netzklassen gibt. Neben den Stellen-Transitions-Netzen, die eine sehr elementare Form von Petri-Netzen sind, wurden auch viele Erweiterungen zu diesen Netzen entwickelt. Der letzte Abschnitt stellt einige dieser sogenannten High-Level-Petri-Netze vor. Dazu gehören gefärbte Petri-Netze, in denen Tokens anhand ihrer Farbe unterschieden werden können, Generalised-Stochastic-Petri-Netze, die zwischen zeitbehafteten und zeitlosen Transitionen unterscheiden können und somit einen zeitlichen Aspekt in Petri-Netze integrieren, und Queuing-Petri-Netze, die im nächsten Kapitel vorgestellt werden. Die Ausführungen in diesem Kapitel sind dabei – soweit nicht anders benannt – an [7] angelehnt.

2.1 Struktur und Verhalten

Häufig sind, wenn von Petri-Netzen die Rede ist, Stellen-Transitions-Netze (S/T-Netze) gemeint. Sie sind eine sehr einfache Form von Petri-Netzen und bilden die

Basis für die meisten sogenannten High-Level-Petri-Netze. Ein S/T-Netz ist ein bipartiter Graph mit zwei Arten von Knoten, den *Stellen* (dargestellt als Kreise) und den *Transitionen* (dargestellt als Balken). Stellen repräsentieren Zustandsparameter wie beispielsweise Ressourcen oder Systembedingungen. Transitionen hingegen stellen Ereignisse oder Aktionen dar. Die Kanten zwischen den Knoten werden *Bögen* genannt und haben ein bestimmtes *Gewicht*, auch *Vielfachheit* genannt (an Bögen annotiert). Ein Gewicht von k entspricht k parallelen Bögen. Bei einem Gewicht von 1 wird die Beschriftung häufig auch weggelassen. Bögen und deren Gewichte werden durch sogenannte *Inzidenzfunktionen* beschrieben. Die *Vorwärtsinzidenzfunktion* beschreibt alle Bögen, die aus einer Transition herausgehen, die *Rückwärtsinzidenzfunktion* alle in eine Transition eingehenden Bögen. Jeder Stelle können sogenannte *Marken* oder *Tokens* (dargestellt als schwarze Punkte) zugeordnet werden. Sie beschreiben, inwieweit die durch die Stelle repräsentierte Bedingung erfüllt ist.[43]

Definition 2.1.1 (Petri-Netz). Ein *Petri-Netz* ist ein Tupel $\mathcal{P} = (\mathcal{N}, m_0)$. Dabei beschreibt $\mathcal{N} = (P, T, I^-, I^+)$ mit

- einer endlichen Menge von Stellen $P = \{p_1, p_2, \cdots, p_m\}$,
- einer endlichen Menge von Transitionen $T = \{t_1, t_2, \cdots, t_n\}$,
- einer Rückwärtsinzidenzfunktion $I^- : P \times T \to \mathbb{N}_0$,
- einer Vorwärtsinzidenzfunktion $I^+ : P \times T \to \mathbb{N}_0$ und
- $P \cap T = \emptyset$

die Netzstruktur und $m_0 : P \to \mathbb{N}_0$ die initiale Markierung des Petri-Netzes.

Falls in einem Petri-Netz alle Kanten ein Gewicht von 1 haben, d.h. für alle Stellen p und alle Transitionen t gilt $I^-(p,t) \leq 1$ und $I^+(p,t) \leq 1$, so wird das Petri-Netz *gewöhnlich* genannt.

Definition 2.1.2 (Markierung). Eine *Markierung* zu einem Netz $\mathcal{N}$ ist eine Funktion $m : P \to \mathbb{N}_0$.

Die Markierung eines Netzes beschreibt, wieviele Tokens an welcher Stelle vorhanden sind. Sie gibt somit den Systemzustand des Petri-Netzes an. Falls die Stellen geordnet sind, kann eine Markierung alternativ auch als Vektor dargestellt werden, wobei die i-te Komponente jeweils der Markenanzahl der i-ten Stelle entspricht:

$$\mathbf{m} = \begin{pmatrix} m(p_1) \\ m(p_2) \\ \cdots \\ m(p_n) \end{pmatrix} \tag{2.1}$$

Definition 2.1.3 (Vor- und Nachbereich). Für ein Petri-Netz $\mathcal{P} = (P, T, I^-, I^+, m_0)$ sind

- ${}^\bullet t = \{p \in P \mid (I^-(p,t) > 0\}$ die *Vorstellen* einer Transition t ,
- $t^\bullet = \{p \in P \mid I^+(p,t) > 0\}$ die *Nachstellen* einer Transition t,
- ${}^\bullet p = \{t \in T \mid (I^+(p,t) > 0\}$ die *Vortransitionen* einer Stelle p und
- $p^\bullet = \{t \in T \mid I^-(p,t) > 0\}$ die *Nachtransitionen* einer Stelle p.

Der Vorbereich einer Stelle p (resp. Transition t) enthält alle Transitionen (resp. Stellen), von denen aus ein Bogen zu p (resp. t) verläuft. Der Nachbereich einer Stelle p (resp. Transition t) enthält entsprechend alle Transitionen (resp. Stellen), zu denen ein Bogen von p (resp. t) führt.

Das dynamische Verhalten des Petri-Netzes ergibt sich durch das Schalten von Transitionen und der daraus resultierenden Änderung der Markierung. Eine Transition t ist *aktiviert* in einer Markierung m, man schreibt $m \xrightarrow{t}$, falls alle Stellen genügend Tokens entsprechend des Bogengewichts enthalten, d.h.

$$\forall p \in {}^\bullet t : \; m_i(p) \geq I^-(p,t). \tag{2.2}$$

Wenn eine Transition aktiviert ist, kann sie feuern. Es darf jedoch immer nur jeweils eine Transition gleichzeitig feuern. Die Entscheidung, welche Transition tatsächlich feuert, ist nicht-deterministisch. Die Folgemarkierung m' nach Feuern von Transition t ergibt sich daraus, dass aus den Stellen aus dem Vorbereich von t Marken entsprechend des Gewichts des Bogens von der Stelle zu t entfernt werden und Stellen aus dem Nachbereich Marken entsprechend des Bogengewichts hinzugefügt werden:

$$m'(p) = m(p) - I^-(p,t) + I^+(p,t) \tag{2.3}$$

$m \xrightarrow{t} m'$ beschreibt, dass Markierung m' von Markierung m nach Feuern von Transition t erreicht werden kann.

Beispiel 2.1.4. Abbildung 2.1 zeigt das Petri-Netz zu Dijkstras Philosophenproblem:

> *„Five philosophers, numbered from 0 to 4 are living in a house where the table laid for them, each philosopher having his own place at the table. Their only problem — besides those of philosophy — is that the dish served is a very difficult kind of spaghetti, that has to be eaten with two forks. There are two forks next to each plate, so that presents no*

> *difficulty: as a consequence, however, no two neighbours may be eating simultaneously.“*[20]

Der Übersicht halber wurde das Problem statt für fünf nur für drei Philosophen modelliert. Die Menge der Stellen ist:

$$P = \{think_0, think_1, think_2, eat_0, eat_1, eat_2, fork_0, fork_1, fork_2\}$$

Dabei repräsentiert eine Stelle $think_i$, dass Philosoph i sich im Zustand *denkend* befindet. Die Stelle eat_i beschreibt, dass Philosoph i gerade isst und $fork_i$ gibt an, ob Gabel i auf dem Tisch liegt. Die Transitionsmenge ist:

$$T = \{t_0, t_1, t_2, r_0, r_1, r_2\}$$

Transition t_i beschreibt das Aufnehmen der Gabeln von Philosoph i und r_i beschreibt, dass Philosoph i mit dem Essen fertig ist und die Gabeln wieder zurücklegt. Die Anfangsmarkierung ist:

$$\mathbf{m_0} = (1\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 1)^T$$

d.h. am Anfang befinden sich alle Philosophen im Zustand *denkend* und alle Gabeln liegen auf dem Tisch. Es sind die Transitionen t_0, t_1 und t_2 aktiviert, es gilt also:

$$m_0 \xrightarrow{t_0}$$
$$m_0 \xrightarrow{t_1}$$
$$m_0 \xrightarrow{t_2}$$

Nach Schalten von Transition t_1 wird die Markierung

$$\mathbf{m'} = (1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 0)^T$$

erreicht (siehe Abbildung 2.2).

2.2 Analyse von Petri-Netzen

Die Modellierung von Systemen durch Petri-Netze ermöglicht es, das System auf gewünschte Eigenschaften zu untersuchen. Im Laufe der Zeit wurde eine Vielzahl von Analysemethoden für Petri-Netze entwickelt. Es lässt sich grundsätzlich zwischen zwei Arten von Analysen unterscheiden – statischen und dynamischen. Bei der statischen Analyse werden nur Eigenschaften untersucht, die auf der Struktur

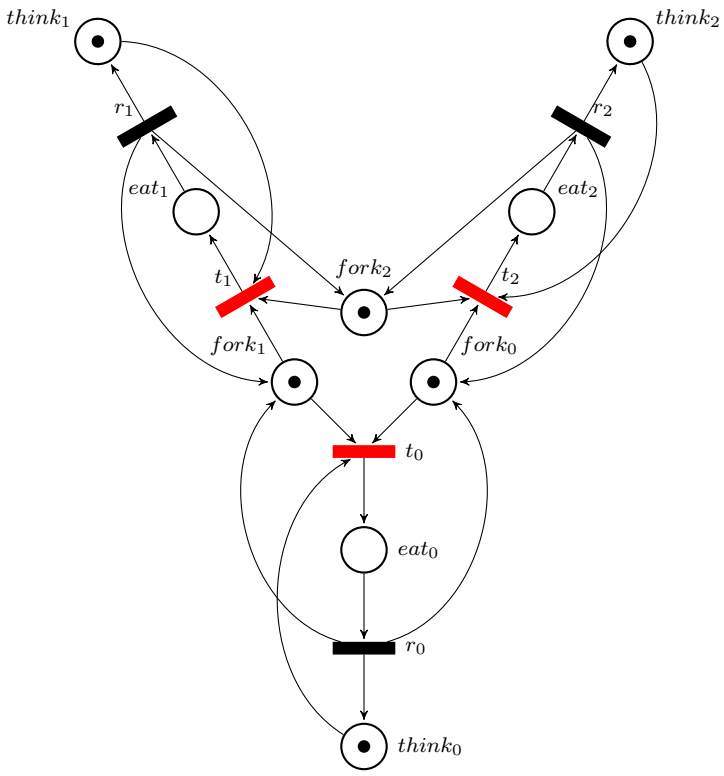


Abbildung 2.1: Petri-Netz zum Philosophenproblem (aktivierbare Transitionen rot)

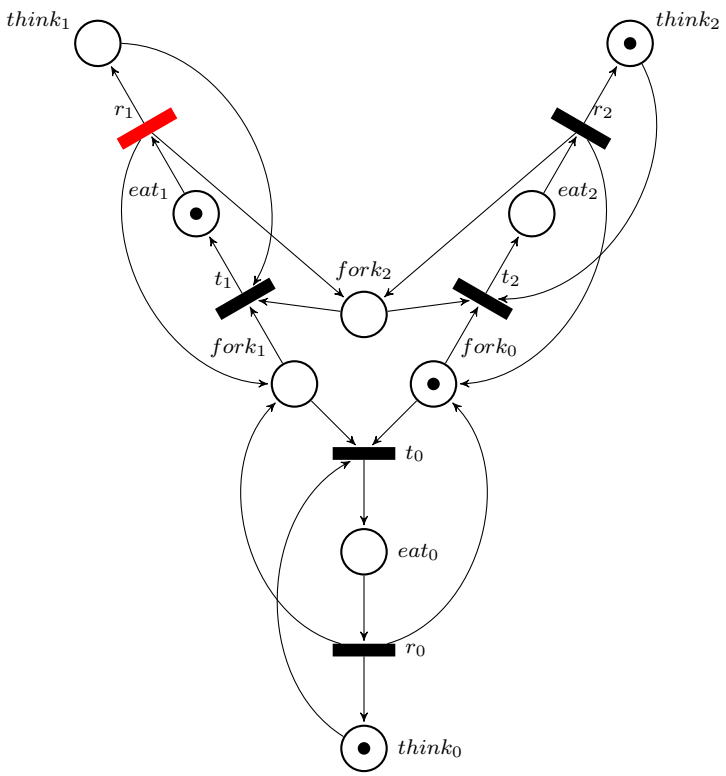


Abbildung 2.2: Petri-Netz zum Philosophenproblem nach Feuern von t_1 (aktivierbare Transitionen rot)

des Petri-Netzes basieren, z. B. ob ein Netz gewöhnlich ist oder sogenannte statische Konflikte[1] existieren. Ebenfalls zur statischen Analyse gehört die sogenannte *Invariantenanalyse*. Invarianten stellen Eigenschaften eines Systems dar, die beim Betrieb des modellierten Systems stets erhalten bleiben, wie beispielsweise die Anzahl der Marken im gesamten Netz oder nur in einem Teilnetz. Die Invarianten ergeben sich aus dem Lösen bestimmter linearer Gleichungssysteme.

Bei der dynamischen Analyse wird das Verhalten von Petri-Netzes untersucht, z. B. ob ein Zustand erreicht werden kann, in dem keine weitere Transition aktiviert ist. Die Analyse von dynamischen Eigenschaften basiert daher in der Regel auf der Untersuchung der *Erreichbarkeitsmenge* bzw. des Zustandsraumes.

Definition 2.2.1 (Feuerfolge). Eine *Feuerfolge* (*firing sequence*) eines Petri-Netzes $\mathcal{P}$ ist eine endliche Folge von Transitionen $\sigma = t_1 \cdots t_n$ mit $n \geq 0$, sodass es Markierungen $m_1, \cdots, m_{n+1}$ gibt, für die gilt: $m_i \xrightarrow{t_i} m_{i+1}$ für alle $i = 1, \cdots n$. Man schreibt $m_1 \xrightarrow{\sigma}$. T^* ist die Menge aller Feuerfolgen.

Definition 2.2.2 (Erreichbarkeitsmenge [18]). Die *Erreichbarkeitsmenge* (*reachability set*) RS zu einem Petri-Netz $\mathcal{P} = (\mathcal{N}, m_0)$ ist

$$RS(\mathcal{N}, m_0) = \{m \mid \exists \sigma \in T^* \; m_0 \xrightarrow{\sigma} m\}$$

Die Erreichbarkeitsmenge enthält alle Markierungen, die durch das Schalten einer Folge von Transitionen von der Anfangsmarkierung m_0 erreicht werden können. Sie besteht daher aus allen möglichen Zuständen des durch das Petri-Netz modellierten Systems.

2.2.1 Eigenschaften

In diesem Abschnitt werden einige dynamische Eigenschaften von Petri-Netzen vorgestellt, die insbesondere für die Analyse von Queueing-Petri-Netzen (siehe Abschnitt 3) wichtig sind, da sie die Voraussetzungen für deren quantitative Analyse bilden.

Beschränktheit

Die Erreichbarkeitsmenge ist genau dann endlich, wenn es eine positive, endliche Zahl k gibt, die die obere Schranke für die Zahl der Marken an jeder Stelle im Netz bildet. Ein solches Netz nennt man *beschränkt*.

[1] Ein statischer Konflikt liegt vor, wenn sich zwei Transitionen eine Vorstelle teilen

Definition 2.2.3 (Beschränktheit). Ein Petri-Netz $\mathcal{P} = (P, T, I^-, I^+, m_0)$ ist *beschränkt* gdw. es ein $k \in \mathbb{N}_0$ gibt, sodass für alle Stellen $p \in P$ und alle Markierungen $m \in RS(\mathcal{N}, m_0)$ gilt: $m(p) \leq k$.

Ein Netz, das beschränkt mit $k = 1$ ist, heißt *sicher*.

Beispiel 2.2.4. Beim Philosophenproblem repräsentieren die Stellen binäre Bedingungen, deshalb darf jede Stelle maximal ein Token enthalten. Die Prüfung auf Beschränktheit bzw. Sicherheit kann feststellen, ob das Netz richtig modelliert wurde.

Lebendigkeit

Die Lebendigkeit einer Transition beschreibt, dass von jeder erreichbaren Markierung irgendwann eine Markierung erreicht werden kann, in der diese Transition aktiviert ist. Ein Netz ist lebendig, wenn alle seine Transitionen lebendig sind. Die Lebendigkeit stellt also sicher, dass alle Funktionalitäten eines Systems in jedem Systemzustand erhalten bleiben.

Definition 2.2.5 (Lebendigkeit). Ein Petri-Netz $\mathcal{P} = (\mathcal{N}, m_0)$ ist *lebendig*, wenn für alle Markierungen $m \in RS(\mathcal{N}, m_0)$ und alle Transitionen $t \in T$ gilt: es existiert ein $m' \in RS(\mathcal{N}, m)$, sodass $m' \xrightarrow{t}$.

Neben der Lebendigkeit, die eine sehr strikte Forderung ist, gibt es noch zwei abgeschwächtere Lebendigkeitsbegriffe. Quasi-Lebendigkeit stellt sicher, dass jede Transition potenziell aktivierbar ist, d.h. ein Systemzustand erreicht werden kann, in dem die Transition aktiviert ist.

Definition 2.2.6 (Quasi-Lebendigkeit [18]). Eine Transition ist *tot* in einem Petri-Netz $\mathcal{P} = (\mathcal{N}, m_0)$, wenn es keine Markierung $m \in RS(\mathcal{N}, m_0)$ gibt, sodass $m \xrightarrow{t}$.

Ein Petri-Netz ist *quasi-lebendig*, wenn keine seiner Transitionen tot ist.

Pseudo-Lebendigkeit bedeutet, dass in jeder erreichbaren Markierung immer mindestens eine Transition aktiviert ist, d.h. es gibt keine Deadlocks. Ist ein Netz nicht pseudo-lebendig, so kann es irgendwann stoppen.

Definition 2.2.7 (Pseudo-Lebendigkeit [18]). Ein Petri-Netz $\mathcal{P} = (\mathcal{N}, m_0)$ ist *pseudo-lebendig* oder *deadlock-frei*, wenn für alle Markierungen $m \in RS(\mathcal{N}, m_0)$ eine Transition $t \in T$ existiert, sodass $m \xrightarrow{t}$.

Beispiel 2.2.8. Beim Philosophenproblem (siehe Beispiel 2.1.4, S. 5) bedeutet Lebendigkeit , dass unabhängig vom Verlauf des Abends, jeder Philosoph die Möglichkeit hat irgendwann zu essen und auch irgendwann die Gabeln wieder zurückzulegen.

Quasi-Lebendigkeit bedeutet, dass nicht schon zu Beginn des Abends ausgeschlossen wird, dass ein Philosoph irgendwann essen bzw. seine Gabeln zurücklegen kann. Allerdings könnte es sich im Laufe des Abends ändern.

Pseudo-Lebendigkeit bedeutet, dass im Laufe des Abends niemals ein Zustand auftreten kann, in dem keiner der Philosophen etwas tun kann.

Pseudo-Lebendigkeit und Quasi-Lebendigkeit sind unabhängig voneinander, es kann also Netze geben, die pseudo-lebendig sind, aber nicht quasi-lebendig, und umgekehrt. Ein Netz, dass lebendig ist, ist immer auch pseudo- und quasi-lebendig. Der umgekehrte Fall gilt in der Regel nicht.[18]

Reversibilität und Heimatzustände

Als *Heimatzustände* (*home states*) werden Zustände bezeichnet, zu denen immer wieder zurückgekehrt werden kann.

Definition 2.2.9 (Heimatzustand [18], Reversibilität). Ein Petri-Netz $\mathcal{P} = (\mathcal{N}, m_0)$ hat einen *Heimatzustand* m_h, wenn für alle $m \in RS(\mathcal{N}, m_0)$ gilt: es existiert $\sigma \in TM*$ mit $m \xrightarrow{\sigma} m_h$.

Ein Petri-Netz ist *reversibel*, wenn es die initiale Markierung m_0 als Heimatzustand hat.

Beispiel 2.2.10. Fordert man beim Philosophenproblem Reversibilität, so heißt das, dass die Möglichkeit besteht, dass wieder irgendwann im Laufe des Abends keiner der Philosophen isst und alle Gabeln wieder auf dem Tisch liegen.

2.2.2 Analyse mithilfe des Erreichbarkeitsgraphen

Das Verhalten eines Petri-Netzes kann durch einen sogenannten *Erreichbarkeitsgraphen* (*reachability graph*) dargestellt werden. Die Knoten des Graphen entsprechen den erreichbaren Markierungen und Kanten repräsentieren den Übergang von einer Markierung zur anderen durch das Feuern von Transitionen.

Definition 2.2.11 (Erreichbarkeitsgraph). Der *Erreichbarkeitsgraph* zu einem Petri-Netz $\mathcal{P} = (P, T, I^-, I^+, m_0)$ ist ein gerichteter Graph $RG(\mathcal{P}) = (V, E)$ mit

- Knotenmenge $V = RS(\mathcal{N}, m_0)$ und

- Kantenmenge $E = \{(m, t, m') \mid m, m' \in R_N(\mathcal{N}, m_0), m \xrightarrow{t} m'\}$

Algorithmus 2.1 beschreibt, wie der Erreichbarkeitsgraph zu einem Petri-Netz bestimmt werden kann. Dazu werden alle Transitionen, die in einer bestimmten Markierung m aktiviert sind, betrachtet und geschaut, welche Markierungen m' durch das Schalten der Transitionen erreicht werden können. Falls es bereits eine Markierung gibt, die m' entspricht, so wird einfach nur eine Kante von m zu m' (mit der Bezeichnung der Transition als Kantenbeschriftung) hinzugefügt. Andernfalls wird ein neuer Knoten m' und eine entsprechende Kante von m zu diesem eingefügt. Gilt jedoch für alle Stellen p von m', dass $m'(p) \geq m''(p)$ für einen bereits vorhanden Knoten m'', so wird m'' von m' *überdeckt*. Falls in dem Fall m'' und m' nicht gleich sind, wird der Algorithmus abgebrochen und liefert einen Fehler. Die Erreichbarkeitsmenge ist dann unendlich und es kann daher kein Erreichbarkeitsgraph erzeugt werden. Der Algorithmus endet erfolgreich, wenn alle Markierungen und deren aktivierte Transitionen bzw. Nachfolgemarkierungen betrachtet wurden und kein Fehler geliefert wurde.

Algorithmus 2.1 Generierung des Erreichbarkeitsgraphen (vgl. [25])

Eingabe: Petri-Netz $\mathcal{P} = (\mathcal{N}, m_0)$
Ausgabe: Gerichteter Graph $RG(\mathcal{P}) = (V, E)$

```
1:  Initialisiere RG(P) = ({m_0}, ∅), m_0 ist nicht markiert
2:  while es gibt noch unmarkierte Knoten in V do
3:    wähle unmarkierten Knoten m ∈ V und markiere ihn
4:    for jede in m aktivierte Transition t do
5:      bestimme m', sodass m -t-> m'
6:      if es gibt m'' ∈ V mit m'' -σ-> m' und m' > m'' then
7:        return ERROR
8:      end if
9:      if es gibt kein m'' ∈ V mit m'' = m' then
10:       V := V ∪ {m'}
11:     end if
12:     E := E ∪ {m, t, m'}
13:   end for
14: end while
15: return RG(P)
```

Wenn ein Petri-Netz beschränkt ist, so ist sein Erreichbarkeitsgraph endlich und kann mit Algorithmus 2.1 eindeutig bestimmt werden. Für unbeschränkten Petri-Netze gibt der Algorithmus einen Fehler zurück. In diesem Fall wird der sogenannte *Überdeckungsgraph* (*coverability graph*) [29] zur graphischen Repräsentation des Verhaltens genutzt. Dieser Graph ist immer endlich und verwendet das spezielle Symbol ω, um unbeschränkte Stellen in Markierungen kennzuzeichnen. Ein Petri-Netz

ist genau dann beschränkt, wenn keine Markierung im Überdeckungsgraphen ω enthält. Im Fall, dass das Petri-Netz beschränkt ist, stimmen Erreichbarkeitsgraph und Überdeckungsgraph überein [43].

Algorithmus 2.2 Generierung des Überdeckungsgraphen

Eingabe: Petri-Netz $\mathcal{P} = (\mathcal{N}, m_0)$
Ausgabe: Gerichteter Graph $CG(\mathcal{P}) = (V, E)$

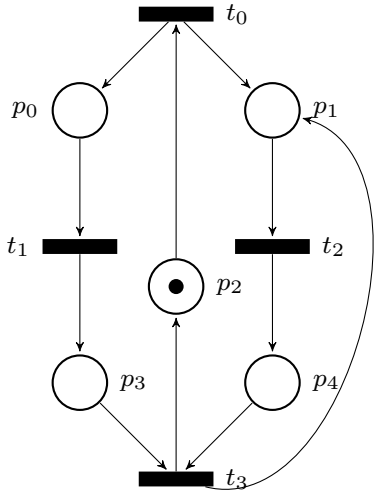
```
1:  Initialisiere RG(P) = ({m_0}, ∅), m_0 ist nicht markiert
2:  while es gibt noch unmarkierte Knoten in V do
3:    wähle unmarkierten Knoten m ∈ und markiere ihn
4:    for jede in m aktivierte Transition t do
5:      bestimme m', sodass m -t-> m'
6:      if es gibt m'' ∈ V mit m'' -σ-> m' und m' > m'' then
7:        for jede Stelle p mit m'(p) > m''(p) do
8:          m'(p) := ω
9:        end for
10:     else if es gibt kein m'' ∈ V mit m'' = m' then
11:       V := V ∪ {m'}
12:     end if
13:     E := E ∪ {m, t, m'}
14:   end for
15: end while
16: return CG(P)
```

Algorithmus 2.2 beschreibt die Generierung des Überdeckungsgraphen zu einem Petri-Netz. Der Algorithmus basiert auf Algorithmus 2.1 und unterscheidet sich darin, was geschieht, wenn eine überdeckende Markierung gefunden wird (Zeilen 6-9). Während der Algorithmus für Entscheidungsgraphen dann abbricht und einen Fehler meldet, wird beim Überdeckungsgraphen in der überdeckenden Markierung m' für jede Stelle, die in m' mehr Marken enthält als in m'', ω als Markenzahl eingetragen. ω wird bei der Bestimmung der aktivierten Transitionen wie ∞ behandelt.

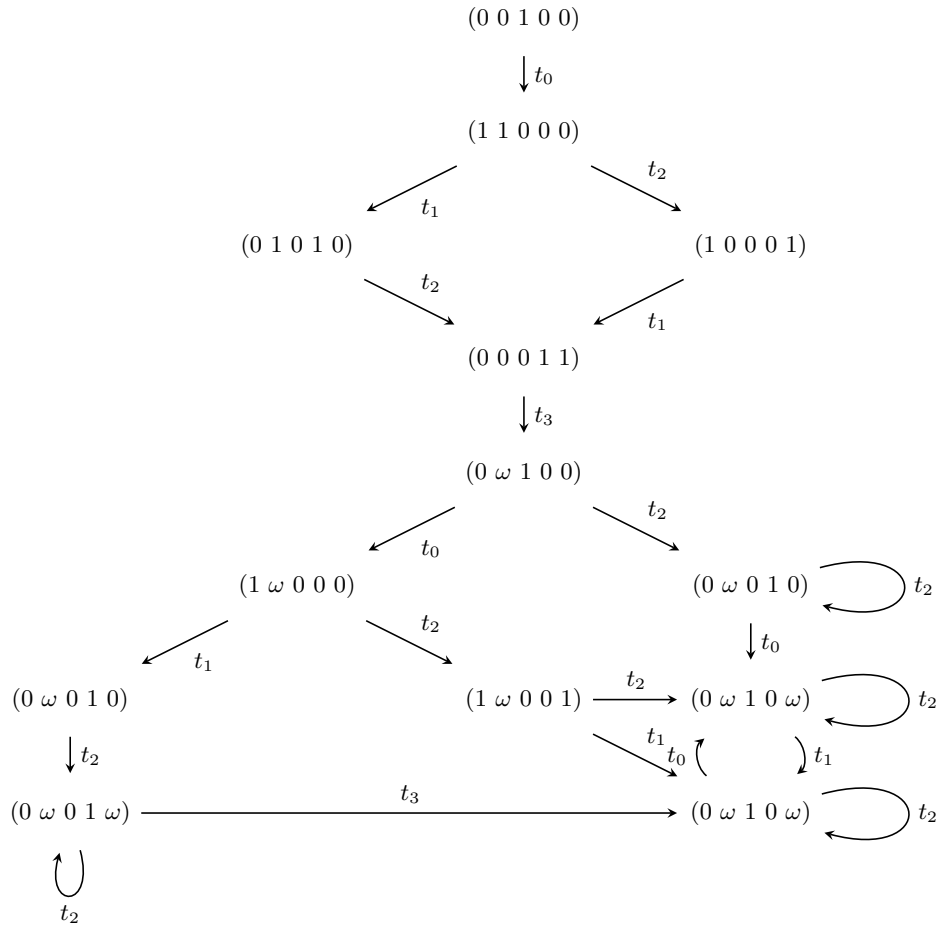
Beispiel 2.2.12. Abbildung 2.3 zeigt ein unbeschränktes Petri-Netz und den zugehörigen Überdeckungsgraphen. Aus dem Graphen lässt sich ablesen, dass die Stelle p_1 unbeschränkt ist.

Beispiel 2.2.13. Abbildung 2.4 zeigt den Erreichbarkeitsgraphen zum Petri-Netz des Philosophenproblems aus Beispiel 2.1.4. Da der Erreichbarkeitsgraph endlich ist, ist das Petri-Netz beschränkt. Der Erreichbarkeitsgraph zeigt außerdem, dass in jeder Markierung die Zahl der Tokens an jeder Stelle nie höher als eins ist und das Petri-Netz somit auch sicher ist.

Auch die anderen Eigenschaften lassen sich mithilfe des Erreichbarkeitsgraphen analysieren. Ein Petri-Netz ist genau dann lebendig, wenn in seinem Erreichbarkeitsgra-



(a) Petri-Netz



(b) Überdeckungsgraph

Abbildung 2.3: Ein unbeschränktes Petri-Netz und sein Überdeckungsgraph

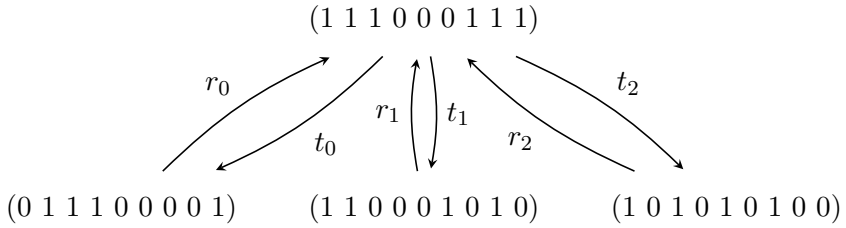


Abbildung 2.4: Erreichbarkeitsgraph zum Philosophenproblem

phen in jeder terminalen starken Zusammenhangskomponente[2] (SZK) jede Transition mindestens einmal als Kante auftritt [18].

Beispiel 2.2.14. Der Erreichbarkeitsgraph zum Philosophenproblem (siehe Abbildung 2.4) besteht aus einer terminalen starken Zusammenhangskomponente, nämlich dem gesamten Graphen. Da jede Transition als Kantenbeschriftung auftaucht, ist das Petri-Netz zum Philosophenproblem lebendig.

Quasi-Lebendigkeit kann daran erkannt werden, dass jede Transition eines Petri-Netzes mindestens einmal als Kantenbeschriftung auftaucht, und Pseudo-Lebendigkeit daran, dass jeder Knoten im Graphen mindestens einen Nachfolger hat.[18]

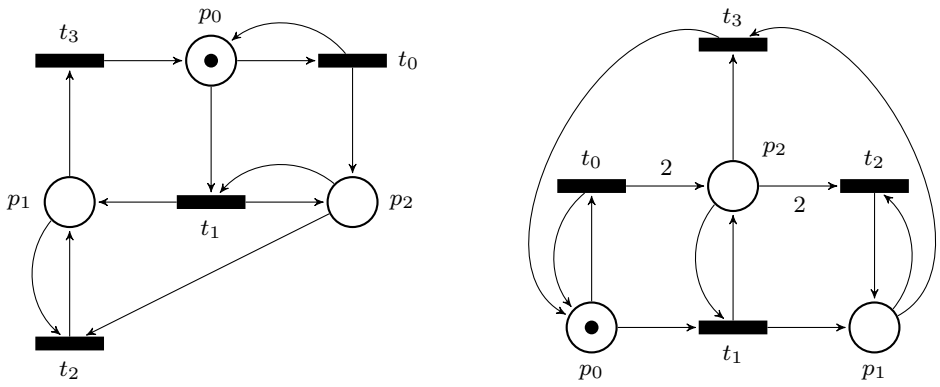
Beispiel 2.2.15. Im Erreichbarkeitsgraphen des Philosophenproblems taucht jede Transition mindestens einmal als Kantenbeschriftung auf und jeder Knoten hat einen Nachfolger, daher ist das zugehörige Petri-Netz quasi- und pseudo-lebendig.

In einem Petri-Netz existiert ein Heimatzustand, wenn der zugehörige Erreichbarkeitsgraph genau eine terminale starke Zusammenhangskomponente enthält [7]. Die Heimatzustände sind dann diejenigen Markierungen, die zu der terminalen SZK gehören.

Beispiel 2.2.16. Da der gesamte Erreichbarkeitsgraph zum Philosophenproblem eine terminale SZK ist, ist stellt jede Markierung/jeder Zustand einen Heimatzustand dar. Somit ist das Petri-Netz auch reversibel.

Durch die Einführung von ω im Überdeckungsgraphen gehen Informationen verloren. Dadurch lassen sich einige Eigenschaften, die am Erreichbarkeitsgraphen zeigen lassen, für unbeschränkte Netzte nicht aus dem Überdeckungsgraphen ableiten.

[2] Eine SZK ist ein Teilgraph $G' = (V', E')$ eines Graphen $G = (V, E)$ mit $V' \subseteq V$ und $E' \subseteq E$, bei dem jeder Knoten von jedem anderen Knoten aus erreichbar ist. Eine SZK ist terminal, wenn für jeden Knoten $v' \in V'$ gilt: $\nexists v \in V$ mit $v \notin V'$ und $(v', v) \in E$.



(a) Lebendiges Petri-Netz

(b) Nicht lebendiges Petri-Netz

Abbildung 2.5: Zwei Petri-Netze mit dem selben Überdeckungsgraphen (vgl. [37, S.551])

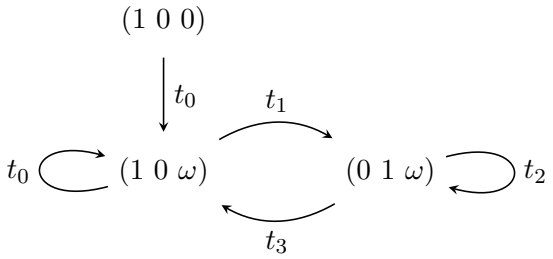


Abbildung 2.6: Überdeckungsgraph zu den beiden Petri-Netzen aus Abbildung 2.5 (vgl. [37, S. 551])

Insbesondere sind dies Eigenschaften, bei denen die Erreichbarkeit von bestimmten Markierungen eine Rolle spielt, wie Lebendigkeit oder die Existenz von Heimatzuständen. Eigenschaften wie Beschränktheit, Quasi-Lebendigkeit und Pseudo-Lebendigkeit hingegen lassen sich auch anhand eines Überdeckungsgraphen für ein unbeschränktes Netz zeigen.[37]

Beispiel 2.2.17 (vgl. [37]). Abbildung 2.5 zeigt zwei Petri-Netze, die den selben Überdeckungsgraphen haben. Würde der Überdeckungsgraph (siehe Abbildung 2.6) bei der Analyse wie ein Erreichbarkeitsgraph behandelt werden, so würden beide Petri-Netze als lebendig eingestuft werden. Zwar ist das Petri-Netz aus Abbbildung 2.5(a) lebendig, das Petri-Netz aus Abbildung 2.5(b) jedoch nicht, da nach dem Feuern der Transitionsfolge $\sigma = t_0\ t_1\ t_3$ keine Transition mehr aktiviert ist:

$$(1\ 0\ 0) \xrightarrow{t_0} (1\ 0\ 2) \xrightarrow{t_1} (0\ 1\ 2) \xrightarrow{t_2} (0\ 1\ 0)$$

Anzahl Philosophen	3	5	10	20
Anzahl Markierungen im Erreichbarkeitsgraphen	4	11	123	>15000

Tabelle 2.1: Anzahl der Zustände für das Philosophenproblem in Abhängigkeit von der Anzahl der Philosophen

Bei Betrachtung des Philosophenproblems für eine verschiedene Zahl von Philosophen (siehe Tabelle 2.1), kann festgestellt werden, dass die Zahl der Zustände exponentiell mit der Zahl der Philosophen wächst. Es kann also passieren, dass bereits für kleine Netze (z.B. gemessen an der Zahl der Stellen und Transitionen) der Erreichbarkeitsgraph unhandhabbar groß wird, d.h. der Erreichbarkeitsgraph kann aufgrund seiner Größe von keinem Computer der Welt berechnet werden. Dieses Problem wird als *Zustandsraumexplosion* [45] bezeichnet.

Um dieses Problem zu umgehen, wurden verschiedene Lösungsansätze entwickelt. Beispielsweise wurden für Petri-Netze mit einer bestimmten Struktur einfachere Verfahren zur Analyse entwickelt, die nicht auf dem Erreichbarkeitsgraphen arbeiten (siehe Abschnitt 2.3). Wiederum andere Verfahren wie z.B. die Stubborn-Set-Methode [46] bauen nur den Teil des Zustandsraums auf, der für die Analyse von bestimmten Eigenschaften nötig ist. Eine andere Methode zielt darauf ab, das Petri-Netz unter Beibehaltung bestimmter Eigenschaften zu verkleinern, um so auch den Zustandsraum zu verkleinern (siehe Kapitel 4).

2.3 Netzklassen

Petri-Netze lassen hinsichtlich struktureller Eigenschaften in verschiedene Klassen unterteilen. Für einige Netzklassen existieren spezielle Verfahren, mit denen sich das Netz effizienter analysieren lässt. In diesem Abschnitt werden die Netzklassen

- Zustandsmaschine (*state machine*, SM),
- Synchronisationsgraph (*marked graph*, MG),
- Free-Choice-Netz (FC),
- Extended-Free-Choice-Netz (EFC),
- Simple-Netz (SPL) und
- Extended-Simple-Netz (ESPL)

und deren Analyse bezüglich der im vorherigen Abschnitt behandelten Eigenschaften vorgestellt. Abbildung 2.7 zeigt die Beziehungen zwischen den einzelnen Netzklassen

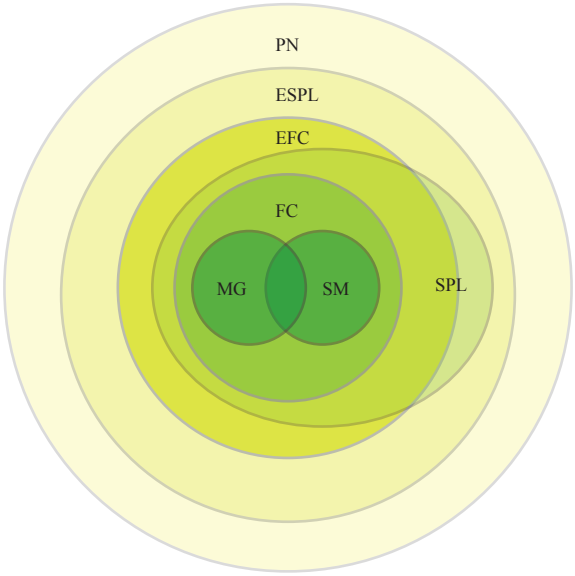


Abbildung 2.7: Beziehungen zwischen den einzelnen Netzklassen (vgl. [7, S. 104])

auf. Die Darstellungen in diesem Abschnitt orientieren sich – soweit nicht anders benannt – an [7, Abschnitt 5.4.3] und [37, S. 553ff]

2.3.1 Zustandsmaschine

Eine *Zustandsmaschine* (*state machine*) [24] ist ein gewöhnliches Petri-Netz, bei dem jede Transition genau eine Vorstelle und eine Nachstelle hat.

Definition 2.3.1 (Zustandsmaschine). Eine *Zustandsmaschine* ist ein gewöhnliches Petri-Netz $\mathcal{P} = (\mathcal{N}, m_0)$, bei dem für alle Transitionen t gilt: $|{}^\bullet t| = |t^\bullet| = 1$.

Dadurch, dass jede Transition genau einen Vorplatz und einen Nachplatz hat und das Netz gewöhnlich ist, bleibt die Zahl der Tokens in einem Netz konstant. Eine Zustandsmaschine ist somit immer beschränkt. Eine Stelle mit einem Token aktiviert alle seine Nachtransitionen, ein Token kann also beliebig durch das Netz „wandern “. Ist die Zustandsmaschine stark zusammenhängend, kann jede Stelle von jeder anderen Stelle aus erreicht und alle Transitionen aktiviert werden, vorausgesetzt es gibt in der Startmarkierung eine Stelle mit mindestens einem Token. Eine Zustandsmaschine ist also genau dann lebendig, wenn sie stark zusammenhängend ist und sich in der Startmarkierung auf mindestens einer Stelle mindestens ein Token befindet. Ferner kann jedes Token dann zu seiner Startstelle (oder jede anderen beliebigen Stelle) zurückkehren. Das Netz ist dann also auch reversibel und jede erreichbare Markierung ist ein Heimatzustand.

2.3.2 Synchronisationsgraph

Der *Synchronisationsgraph* (*marked graph*) [16] ist das Gegenstück zur Zustandsmaschine. Dort hat jede Stelle genau eine Vor- und eine Nachtransition.

Definition 2.3.2 (Synchronisationsgraph). Ein *Synchronisationsgraph* ist ein gewöhnliches Petri-Netz $\mathcal{P} = (\mathcal{N}, m_0)$, bei dem für alle Plätze p gilt: $|{}^\bullet p| = |p^\bullet| = 1$.

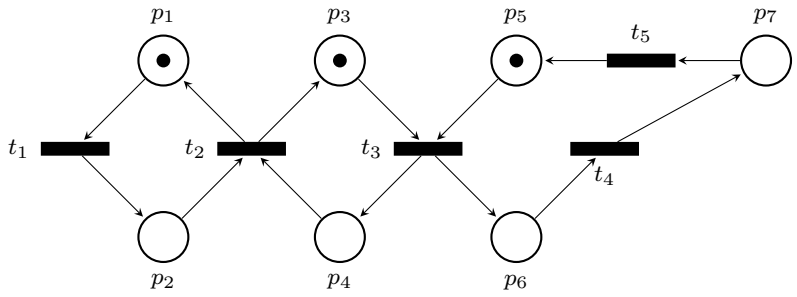
Dadurch, dass jede Stelle genau eine Vor- und eine Nachtransition besitzt, lässt sich das Petri-Netz $\mathcal{P} = (\mathcal{N}, m_0)$ auch als gerichteter Graph $\mathcal{G} = (V, E)$ darstellen. Dabei entsprechen die Knoten des Graphen den Transitionen und die Kanten den Stellen des Petri-Netzes. Eine Kante zu einer Stelle p hat als Startknoten die Vortransition ${}^\bullet p$, als Endknoten die Nachtransition $p^\bullet$ und ist mit der Zahl der Tokens $m(p)$ auf der Stelle beschriftet. Wenn eine Transition feuert, wird von jeder eingehenden Kante ein Token entfernt und auf jede ausgehende Kante eins hinzugefügt. In einem einfachen Zyklus[3] bleibt die Zahl der Tokens immer erhalten [16].

Für die Lebendigkeit des Synchronisationsgraphen muss sichergestellt werden, dass im zugehörigen gerichteten Graphen $\mathcal{G}$ jede Kante irgendwann ein Token erhalten kann. Andernfalls können die Transitionen, deren eingehende Kanten das sind, niemals gefeuert werden. Da in Zyklen die Zahl der Tokens konstant bleibt, muss also jeder einfache Zyklus mindestens ein Token enthalten. Ferner muss man noch die Kanten betrachten, die in einen Zyklus eingehen und die aus einem Zyklus ausgehen[4]. Die eingehenden Kanten eines Zyklus können entweder nachfolgende Kanten[5] eines anderen Zyklus oder nachfolgende Kanten eines Knotens t ohne eingehende Kanten sein. In diesem Fall kann die zu dem Knoten gehörende Transition immer Feuern und ein Token über alle nachfolgenden Kanten „wandern“. Bei ausgehenden Kanten eines Zyklus wird ein Token aus dem Zyklus heraus erzeugt und kann über alle nachfolgenden Kanten „wandern“. Ein Synchronisationsgraph ist also genau dann lebendig, wenn die Startmarkierung mindestens einer Stelle aus jedem einfachen Zyklus mindestens ein Token zuordnet. Besteht der gerichtete Graph zu einem Synchronisationsgraphen nur aus Zyklen, kann also jeder Knoten mindestens einem einfachen Zyklus zugeordnet werden, so ist der Synchronisationsgraph beschränkt, da die Zahl der Tokens im Graphen dann konstant bleibt. Wenn der Synchronisationsgraph lebendig und beschränkt ist, dann ist er auch reversibel [7].

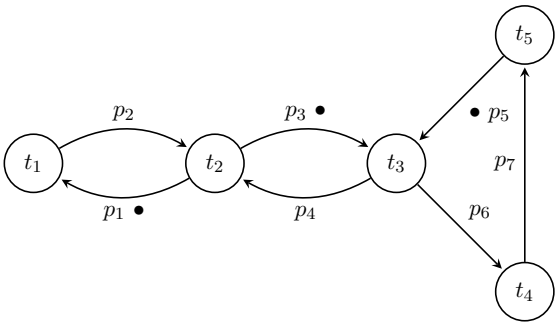
[3] Ein Zyklus heißt einfach, wenn er keinen kleineren Zyklus enthält

[4] Mit ein- und ausgehenden Kanten eines Zyklus sind Kanten gemeint, die ein- oder ausgehende Kanten eines Knoten im Zyklus sind

[5] Mit nachfolgenden Kanten eines Knoten v sind Kanten gemeint, die auf einem zyklenfreien Weg vom Knoten v aus liegen. Nachfolgende Kanten eines Zyklus sind nachfolgende Kanten eines Knoten, der zum Zyklus gehört



(a) Synchronisationsgraph [7]



(b) Zugehöriger gerichteter Graph

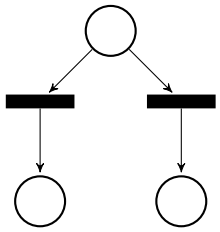
Abbildung 2.8: Ein Synchronisationsgraph und der zugehörige gerichtete Graph

Beispiel 2.3.3. Abbildung 2.8 zeigt einen Synchronisationsgraphen (a) und den gerichteten Graphen (b) dazu. Der Synchronisationsgraph hat die drei einfachen Zyklen $\langle t_1, p_2, t_2, p_1, t_1 \rangle$, $\langle t_2, p_4, t_3, p_3, t_2 \rangle$ und $\langle t_3, p_6, t_4, p_7, p_5, t_3 \rangle$. Die Anfangsmarkierung ist (1 0 1 0 1 0 0). Damit ist in jedem einfachen Zyklus ein Token enthalten und das Petri-Netz lebendig. Ferner gehört jeder Knoten zu einem Zyklus, sodass das Netz auch beschränkt und reversibel ist.

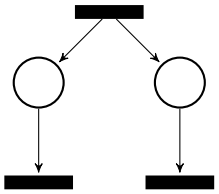
Es gibt Petri-Netze, die sowohl Zustandsmaschinen als auch Synchronisationsgraphen sind, es gibt aber auch Petri-Netze, die zwar jeweils das Eine sind, jedoch nicht das Andere (siehe Abbildung 2.9).

2.3.3 (Extended-)Free-Choice-Netz

Haben bei einem *Extended-Free-Choice-Netz* [13] mehrere Transitionen gemeinsame Vorstellen, so haben sie alle die gleichen Vorstellen. *Free-Choice-Netze* sind eine Einschränkung davon. Transitionen mit gleichen Vorstellen, dürfen nur eine Vorstelle (nämlich die geteilte) haben.

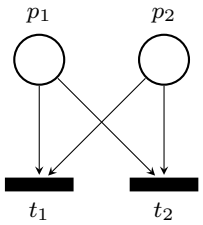


(a) Zustandsmaschine, aber kein Synchronisationsgraph

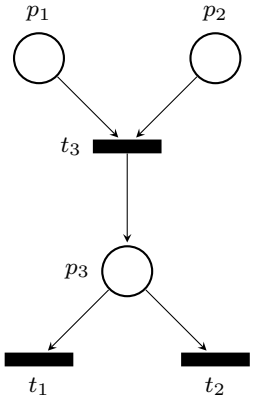


(b) Synchronisationsgraph, aber keine Zustandsmaschine

Abbildung 2.9: Vergleich zwischen Synchronisationsgraph und Zustandsmaschine (vgl. [37, S. 554])



(a) Extended-Free-Choice-Netz



(b) Free-Choice-Netz

Abbildung 2.10: EFC- und FC-Netz [37, S. 554]

Definition 2.3.4 (Free-Choice-Netz, Extended-Free-Choice-Netz). Ein gewöhnliches Petri-Netz $\mathcal{P} = (\mathcal{N}, m_0)$ ist genau dann ein *Free-Choice-Netz*, wenn für alle $t_1, t_2 \in T$ mit $t_1 \neq t_2$ gilt: wenn ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$, dann $|{}^\bullet t_1| = |{}^\bullet t_2| = 1$.

Das Petri-Netz ist ein *Extended-Free-Choice-Netz* genau dann, wenn für alle $t_1, t_2 \in T$ gilt: wenn ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$, dann ${}^\bullet t_1 = {}^\bullet t_2$.

Beispiel 2.3.5. Abbildung 2.10 zeigt ein Extended-Free-Choice-Netz (a) und ein dazu äquivalentes Free-Choice-Netz (b).

Für die Analyse von EFC-Netzen werden die Begriffe *Siphon*[6] und *Falle* (*trap*) benötigt.

Definition 2.3.6 (Siphon). Ein *Siphon* ist eine nicht-leere Menge von Stellen $P' \subseteq P$ mit ${}^\bullet P' \subseteq P'^\bullet$.

[6] wird in der Literatur häufig auch als *Deadlock* bezeichnet

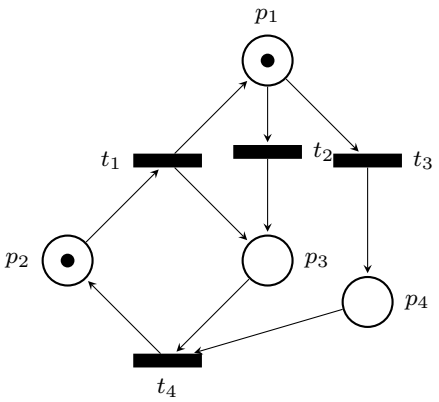


Abbildung 2.11: FC-Netz [37, S. 557]

Enthält der Siphon die Nachstelle einer Transition, so muss er auch deren Vorstelle enthalten.

Definition 2.3.7 (Falle). Eine *Falle* ist eine nicht-leere Menge von Stellen $P' \subseteq P$ mit $P'^{\bullet} \subseteq {}^{\bullet}P'$.

Die Falle ist das Gegenstück zum Siphon. Enthält eine Falle die Vorstelle einer Transition, so muss sie auch deren Nachstelle enthalten.

Beispiel 2.3.8. Das FC-Netz aus Abbildung 2.11 hat die Siphons $S_1 = \{p_1, p_2, p_3\}$, $S_2 = \{p_1, p_2, p_4\}$ und $S_3 = \{p_1, p_2, p_3, p_4\}$. Die Fallen in dem Netz sind $F_1 = \{p_2, p_3\}$, $F_2 = \{p_2, p_3, p_4\}$ und $F_3 = S_3 = \{p_1, p_2, p_3, p_4\}$.

Wenn alle Tokens irgendwann einen Siphon verlassen haben, es also eine Markierung $m \in RS(\mathcal{N}, m_0)$ gibt, in der alle Stellen des Siphons keine Tokens mehr haben, so bleiben die Stellen des Siphon für immer leer. In einer Falle sind Tokens hingegen „gefangen“, d. h. in jeder Markierung $m \in RS(\mathcal{N}, m_0)$ enthält mindestens eine Stelle der Falle ein Token. Damit ein Netz lebendig ist, dürfen Tokens einen Siphon niemals verlassen. Dies ist genau dann der Fall, wenn jeder Siphon eine Falle enthält, der in der Startmarkierung mindestens einer Stelle ein Token zugeordnet wird. Ein EFC-Netz ist also genau dann lebendig, wenn jeder Siphon S eine Falle F enthält, $F \subseteq S$, und es eine Stelle $p \in F$ gibt, sodass $m_0(p) > 0$.

Beispiel 2.3.9. Da der Siphon $S_2 = \{p_1, p_2, p_4\}$ im FC-Netz aus Abbildung 2.11 keine Falle enthält, ist das FC-Netz nicht lebendig.

Für die Analyse der Beschränktheit von EFC-Netzen wird der Begriff der *P-Komponente* gebraucht.

Definition 2.3.10 (Subnetz). Ein von $X \subseteq P \cup T$ erzeugtes *Subnetz* eines Petrinetzes $\mathcal{P} = (\mathcal{N}, m_0)$ ist ein Petri-Netz $\mathcal{P}' = (\mathcal{N}', m_0')$ mit

- $P' = (X \cap P) \cup {}^\bullet(X \cap T) \cup (X \cap T)^\bullet$,
- $T' = (X \cap T) \cup {}^\bullet(X \cap P) \cup (X \cap P)^\bullet$,
- $I'^-(p,t) = I'^-(p,t)$ für alle $p \in P', t \in T'$,
- $I'^+(p,t) = I'^+(p,t)$ für alle $p \in P', t \in T'$ und
- $m_0'(p) = m_0(p)$ für alle $p \in P'$.

Ein von X generiertes Subnetz enthält alle Stellen aus X, deren Vor- und Nachtransitionen, alle Transitionen aus X und deren Vor- und Nachstellen. Die Beziehungen zwischen den Stellen und Transitionen bleiben genauso wie die Startmarkierung der Stellen erhalten.

Definition 2.3.11 (P-Komponente). Ein von P' generiertes Subnetz $\mathcal{P}' \subseteq \mathcal{P}$ ist eine *P-Komponente* eines Petri-Netzes $\mathcal{P}$ gdw. für alle $t' \in T'$ in $\mathcal{P}'$ gilt: $|{}^\bullet t' \cap P'| \leq 1$ und $|t'^\bullet \cap P'| \leq 1$.

Ein lebendiges EFC-Netz ist genau dann beschränkt, wenn das EFC-Netz von stark zusammenhängenden P-Komponenten überdeckt wird. Ein EFC-Netz ist wird genau dann von P-Komponenten überdeckt, wenn es für jede Stelle und jede Transition eine P-Komponente mit einem Token in der Startmarkierung gibt, zu der die Stelle bzw. Transition gehört.

Beispiel 2.3.12. Das FC-Netz aus Abbildung 2.11 hat keine stark zusammenhängenden P-Komponenten. Da das Netz aber nicht lebendig ist, hat das keine Aussagekraft zur Beschränktheit des Netzes. Wird stattdessen der Überdeckungsgraph zur Analyse der Beschränktheit genutzt, so kann festgestellt werden, dass das Netz beschränkt ist.

Ein lebendiges und beschränktes EFC-Netz besitzt immer Heimatzustände. Die Heimatzustände in solchen Netzen sind die Markierungen, in denen alle Fallen markiert sind.[12]

2.3.4 (Extended-)Simple-Netz

Teilen sich in einem *Simple-Netz* zwei Stellen eine Nachtransition, so hat mindestens eine der Stellen nur die gemeinsame Transition als Nachtransition. Beim *Extended-Simple-Netz*[7] muss mindestens eine von je zwei Stellen, die sich Nachtransitionen

[7] auch als *Asymmetric-Choice-Netz* bekannt

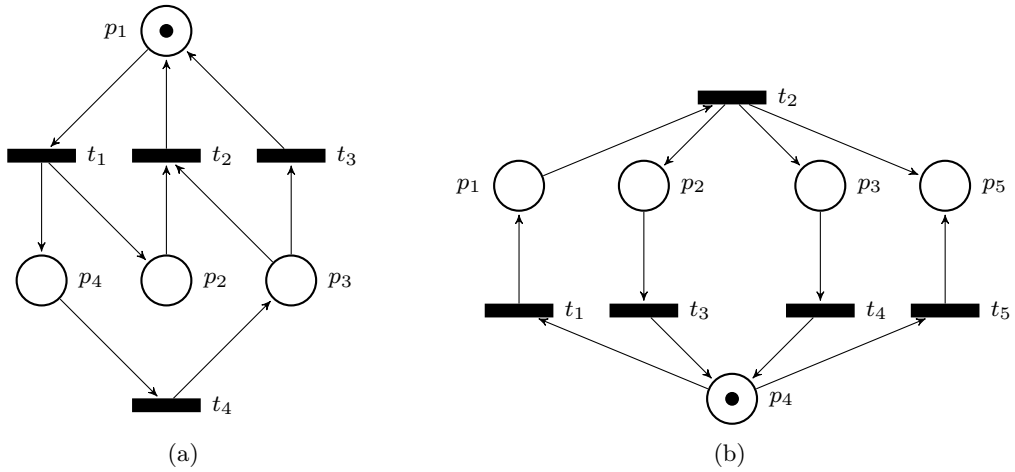


Abbildung 2.12: Zwei Extended-Simple-Netze [37, S. 557f.]

teilen, alle Transitionen, die die andere Stelle als Vortransitionen hat, ebenfalls als Vortransitionen haben.

Definition 2.3.13 (Simple-Netz, Extended-Simple-Netz). Ein Petri-Netz $\mathcal{P} = (\mathcal{N}, m_0)$ ist genau dann ein *Simple-Netz*, wenn für alle $p, p' \in P$ mit $p \neq p'$ gilt: wenn $p^\bullet \cap p'^\bullet \neq \emptyset$, dann ist $|p^\bullet| \leq 1$ oder $|p'^\bullet| \leq 1$.

Das Petri-Netz ist genau dann ein *Extended-Simple-Netz*, wenn für alle $p, p' \in P$ gilt: wenn $p^\bullet \cap p'^\bullet \neq \emptyset$, dann ist $p^\bullet \subseteq p'^\bullet$ oder $p'^\bullet \subseteq p^\bullet$.

Ein Extended-Simple-Netz ist lebendig, wenn jeder Siphon eine Falle enthält. Dies ist aber nur eine hinreichende und keine notwendige Bedingung, d. h. auch wenn nicht jeder Siphon eine Falle enthält, kann das Netz lebendig sein.[11]

Beispiel 2.3.14 (vgl. [37]). Die beiden Netze aus Abbildung 2.12 sind nicht nur Extended-Simple-Netze, sondern auch Simple-Netze. Das Netz aus Abbildung 2.12(a) hat die Siphons $\{p_1, p_3, p_4\}$ und $\{p_1, p_2, p_3, p_4\}$. Der erste Siphon enthält die in p_1 markierete Falle $\{p_1, p_3, p_4\}$. Der zweite Siphon enthält die selbe Falle und zusätzlich noch die in p_1 markierte Falle $\{p_1, p_2\}$. Somit ist das Netz lebendig.

Das Netz aus Abbildung 2.12(b) hat den Siphon $\{p_1, p_2, p_3, p_4\}$. Obwohl der Siphon keine Falle enthält, ist das Netz trotzdem lebendig.

2.4 Petri-Netz-Erweiterungen

Stellen-Transitions-Netze sind eine der einfachsten Formen von Petri-Netzen. Im Laufe der Zeit wurden viele verschiedene Petri-Netz-Arten entwickelt, die die Dar-

stellung erleichtern und Petri-Netze um weitere Funktionen, wie beispielsweise Zeitaspekte, erweitern. In diesem Abschnitt werden zunächst die gefärbten Petri-Netze (CPN) vorgestellt. In CPNs können Tokens unterschieden werden, indem sie verschiedene Farben zugeordnet bekommen. Anschließend werden Generalalised-Stochastic-Petri-Netze (GSPN) erläutert. Diese erweitern Petri-Netze um einfach zeitliche Aspekte. Zum Schluss werden die auf CPNs und GSPNs aufbauenden Queuing-Petri-Netze (QPN) vorgestellt, die Petri-Netze mit sogenannten Warteschlangennetzen verknüpfen.

2.4.1 Gefärbtes Petri-Netz

Wird das Petri-Netz zum Philosophenproblem aus Beispiel 2.1.4 betrachtet, so fallen starke Symmetrien auf, Teile des Petri-Netzes verhalten sich gleich. Da es für größere Beispiele schnell unübersichtlich werden kann, wurden sogenannte *gefärbte Petri-Netze* [28] eingeführt. In diesen können Tokens anhand ihrer Farbe unterschieden werden. Jede Transition hat verschiedene Schaltmodi, die ebenfalls als Farben bezeichnet werden.

Definition 2.4.1 (Multimenge). Eine *Multimenge* ist eine Funktion $s = S \to \mathbb{N}_0$, die jeder Menge S eine natürliche Zahl zuordnet.

Eine Multimenge beschreibt eine Menge, in der Elemente auch mehrfach vorkommen können. Der Funktionswert $m(s)$ gibt dabei an, wie oft ein Element $s \in S$ in der Multimenge auftaucht. Die Menge aller Multimengen über einer Menge S wird im Weiteren mit S_{MS} gekennzeichnet.

Definition 2.4.2 (Gefärbtes Petri-Netz). Ein *gefärbtes Petri-Netz* ist ein Tupel $\mathcal{CPN} = (\mathcal{N}_C, m_0)$. Dabei beschreibt $\mathcal{N}_C = (P, T, C, I^-, I^+)$ mit

- einer endlichen Menge von Stellen P
- einer endlichen Menge von Transition T,
- einer Farbfunktion C, die jeder Stelle und jeder Transition eine nicht-leere Menge von Farben zuordnet, und
- Inzidenzfunktionen I^+ und I^-, wobei $I^+(p,t), I^-(p,t) : C(t) \to C(p)_{MS}$ für alle $p \in P, t \in T$

die Netzstruktur und $m_0 : P \to C(p)_{MS}$ die initiale Markierung des gefärbten Petri-Netzes.

Die Farbfunktion beschreibt für Stellen, welche Farben die Tokens darin haben können, und für Transitionen, welche verschiedenen Schaltmodi die Transition hat. Markierungen und Vor- und Nachbereich können analog zu den S/T-Netzen (siehe S. 4) definiert werden:

Definition 2.4.3 (Markierung in CPN). Eine *Markierung* zu einem gefärbten Netz $\mathcal{N}_C$ ist eine Funktion $m : P \to C(p)_{MS}$.

Bei CPNs beschreibt die Markierung, wieviele Tokens jeder Farbe eine Stelle jeweils enthält.

Definition 2.4.4 (Vor und Nachstellen, Vor- und Nachtransitionen). Für eine Transition $t \in T$ und eine Farbe $c' \in C(t)$ sind

- ${}^\bullet(t, c') = \{(p, c) \mid p \in P, c \in C(p), I^-(p, t)(c')(c) \neq \emptyset\}$ die Vorstellen und
- $(t, c')^\bullet = \{(p, c) \mid p \in P, c \in C(p), I^+(p, t)(c')(c) \neq \emptyset\}$ die Nachstellen.

Für eine Stelle $p \in P$ und eine Farbe $c \in C(p)$ sind

- ${}^\bullet(p, c) = \{(t, c') \mid t \in T, c' \in C(t), I^+(p, t)(c')(c) \neq \emptyset\}$ die Vortransitionen und
- ${}^\bullet(p, c) = \{(t, c') \mid t \in T, c' \in C(t), I^-(p, t)(c')(c) \neq \emptyset\}$ die Nachtransitionen.

Eine Transition t ist in einer Markierung m bezüglich einer bestimmten Farbe/einem bestimmten Modus $c \in C(t)$ *aktiviert*, wenn die für den Schaltmodus benötigten Tokens in den Vorstellen vorhanden sind, d. h.

$$\forall (p, c) \in {}^\bullet(t, c') : m(p)(c) \geq I^-(p, t)(c')(c). \tag{2.4}$$

Die Folgemarkierung m' ergibt sich dadurch, dass diese Tokens entsprechend aus den Vorstellen entfernt und den Nachstellen die vom Schaltmodus definierten Tokens hinzugefügt werden:

$$m'(p, c) = m(p)(c) + I^+(p, t)(c')(c) - I^-(p, t)(c')(c) \tag{2.5}$$

$m \xrightarrow{(t,c')} m'$ beschreibt, dass Markierung m' von Markierung m nach Feuern von Transition t bezüglich der Transitionsfarbe c' erreicht wird.

Beispiel 2.4.5. Abbildung 2.13 zeigt das Philosophenproblem aus Beispiel 2.1.4 auf Seite 5 als CPN. Die ursprünglichen drei Stellen eat_1, eat_2, eat_3 wurden zu einer Stelle eat zusammengefasst. Auf die gleiche Weise wurden auch die Stellen $think_i$ und $fork_i$ mit $i \in \{1, 2, 3\}$ zu je einer Stelle $think$ und $forks$ vereinigt. Damit ergibt sich die Stellenmenge:

$$P = \{eat, think, forks\}$$

Auch die Transitionen t_i und r_i mit $i \in \{1, 2, 3\}$ können zu Transitionen t und r zusammengelegt werden, sodass sich die Transitionsmenge

$$T = \{t, r\}$$

ergibt.

Als Farben für Tokens können die einzelnen Philosophen p_1, p_2 und p_3 und die einzelnen Gabeln f_1, f_2 und f_3 verwendet werden. Die Stellen *eat* und *think* können dann Tokens der Farben p_i und die Stelle $forks$ Tokens der Farben f_i mit $i \in \{1, 2, 3\}$ enthalten:

$$\begin{aligned} C(eat) &= \{p_1, p_2, p_3\} \\ C(think) &= \{p_1, p_2, p_3\} \\ C(forks) &= \{f_1, f_2, f_3\} \end{aligned}$$

Als Transitionsfarben werden ebenfalls die einzelnen Philosophen p_1, p_2 und p_3 unterschieden. Die Farbe p_i beschreibt dabei eine Aktion, die von Philosoph i durchgeführt wird. Die Farbfunktion wird dadurch ergänzt zu:

$$\begin{aligned} C(t) &= \{p_1, p_2, p_3\} \\ C(r) &= \{p_1, p_2, p_3\} \end{aligned}$$

Beim Feuern der Transition t bezüglich einer Farbe p_i wird aus der Stelle *think* ein Token der Farbe p_i und aus der Stelle $forks$ je ein Token der Farbe f_i und $f_{(i+1) \ mod \ 3}$ entfernt und ein Token der Farbe p_i der Stelle *eat* hinzugefügt:

$$I^-(think, t)(p_i)(c) = \begin{cases} 1 & \text{wenn } c = p_i \\ 0 & \text{sonst} \end{cases}$$

$$I^-(forks, t)(p_i)(c) = \begin{cases} 1 & \text{wenn } c = f_i \text{ oder } c = f_{(i+1) \ mod \ 3} \\ 0 & \text{sonst} \end{cases}$$

$$I^+(eat, t)(p_i)(c) = \begin{cases} 1 & \text{wenn } c = p_i \\ 0 & \text{sonst} \end{cases}$$

Dies beschreibt, dass Philosoph i die Gabeln rechts und links von ihm aufnimmt und in den Zustand *essend* wechselt. Das Beenden des Essens und Zurücklegen der

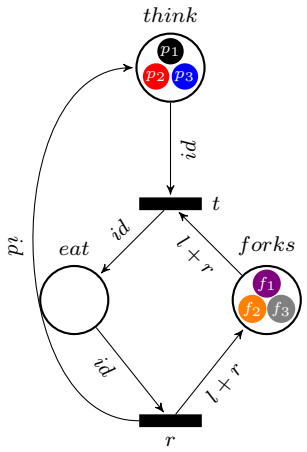


Abbildung 2.13: CPN zum Philosophenproblem

Gabeln von Philosoph i wird durch das Feuern der Transition r bezüglich der Farbe p_i beschrieben:

$$I^-(eat, r)(p_i)(c) = \begin{cases} 1 & \text{wenn } c = p_i \\ 0 & \text{sonst} \end{cases}$$

$$I^+(think, r)(p_i)(c) = \begin{cases} 1 & \text{wenn } c = p_i \\ 0 & \text{sonst} \end{cases}$$

$$I^+(forks, r)(p_i)(c) = \begin{cases} 1 & \text{wenn } c = f_i \text{ oder } c = f_{(i+1) \ mod \ 3} \\ 0 & \text{sonst} \end{cases}$$

Die Inzidenzfunktionen lassen sich vereinfacht durch drei Funktionen $id(p_i) = p_i$ und $r(p_i) = f_i$ und $l(p_i) = f_{(i+1) \ mod \ 3}$ beschreiben. Die initiale Markierung ist:

$$\begin{aligned} m(eat) &= \emptyset \\ m(think) &= \{p_1, p_2, p_3\} \\ m(forks) &= \{f_1, f_2, f_3\} \end{aligned}$$

Analyse von CPNs

Auch CPNs lassen sich bezüglich der Eigenschaften aus Abschnitt 2.2 analysieren. Dies kann auf Ebene der CPNs selbst erfolgen oder mithilfe der bereits vorgestellten Methoden, indem das CPN zu einem S/T-Netz *entfaltet* wird. Das entfaltete S/T-

Netz $\mathcal{P} = (\mathcal{N}, m_0)$ zu einem gefärbten Petri-Netz $\mathcal{CPN} = (\mathcal{N}_\mathcal{C}, m_{0C})$ erhält man mit:

- $P := \{(p, c) \mid p \in P_C, c \in C(p)\}$,
- $T := \{(t, c') \mid t \in T_C, c' \in C(t)\}$,
- $I^-((p, c)(t, c')) := I_C^-(p, t)(c')(c)$,
- $I^+((p, c)(t, c')) := I_C^+(p, t)(c')(c)$ und
- $m_0((p, c)) := m_{0C}(p)(c)$ für alle $p \in P_C$ und $c \in C(p)$.

Beispiel 2.4.6. Wird das CPN zum Philosophenproblem aus Abbildung 2.13 entfaltet, so wird wieder das bereits bekannte S/T-Netz aus Abbildung 2.2 auf Seite 8 erhalten. Das entfaltete Netz unterscheidet sich nur in der Bezeichnung der Stellen und Transitionen, so heißen beispielsweise die Stellen $(think, p_1)$, $(think, p_2)$ und $(think, p_3)$ statt $think_1$, $think_2$ und $think_3$. Die Transitionen werden mit (t, p_i) und (r, p_i) statt t_i und r_i benannt.

2.4.2 Generalised-Stochastic-Petri-Netze

Generalised-Stochastic-Petri-Netze [36] erweitern S/T-Netze um einen einfachen zeitlichen Aspekt, indem zwischen zwei Arten von Transitionen unterschieden wird. *Zeitlose Transitionen* (*immediate transitions*) (dargestellt als ausgefüllte Balken) können ohne zeitliche Verzögerung gefeuert werden, wenn sie aktiviert sind, und verhalten sich somit wie die bisher aus S/T-Netzen bekannten Transitionen. Die zweite Sorte von Transitionen sind die *zeitbehafteten Transitionen* (*timed transitions*) (dargestellt als unausgefüllte Balken). Sie können erst nach einer zufälligen, exponential-verteilten Zeit gefeuert werden, wenn sie aktiviert sind.

Definition 2.4.7 (Generalised-Stochastic-Petri-Netz [7]). Ein *Generalised-Stochastic-Petri-Netz* ist ein 4-Tupel $\mathcal{GSPN} = (\mathcal{P}, T_1, T_2, W)$. Dabei beschreibt

- $\mathcal{P} = (P, T, I^-, I^+, m_0)$ das zugrundeliegende S/T-Netz,
- $T_1 \subseteq T$ die Menge der zeitbehafteten Transitionen mit $T_1 \neq \emptyset$,
- $T_2 \subset T$ die Menge der zeitlosen Transitionen mit $T_1 \cap T_2 = \emptyset$ und $T = T_1 \cup T_2$ und
- $W = (w_1, \ldots, w_{|T|})$ ist ein Array, dessen Eintrag $w_i \in \mathbb{R}^+$
  - die Feuerrate angibt, wenn t_i eine zeitbehaftete Transition ist,
  - ein Feuergewicht angibt, wenn t_i eine zeitlose Transition ist.

Zeitlose Transitionen haben gegenüber zeitbehafteten Transitionen eine höhere Priorität beim Feuern. Sind nach Gleichung 2.2 (siehe S. 5) sowohl zeitlose als auch zeitbehaftete Transitionen aktiviert, so wird auf jeden Fall eine zeitlose Transition feuern. Die Menge der aktivierten Transitionen $EN_T(m)$ in einer Markierung m reduziert sich zu:

$$EN_T(m) := \{t \in T \mid m \xrightarrow{t} \text{ und wenn } \exists t' \in T_2 : m \xrightarrow{t}, \text{ dann } t \in T_2\} \tag{2.6}$$

Sind mehrere zeitlose Transitionen aktiviert, so kann über das Feuergewicht w_i, die Wahrscheinlichkeit der Transition $t_i \in EN_T(m)$ in Markierung m zu feuern bestimmt werden:

$$p(t_i, m) = \frac{w_i}{\sum\limits_{t_j \in EN_T(m)} w_j} \tag{2.7}$$

Bei zeitbehafteten Transitionen wird die exponentielle Verteilung über die Zufallsvariable χ_i der Feuerzeit von t_i durch die Feuerrate w_i festgelegt:

$$F_{\chi_i}(x) = 1 - e^{-w_i x} \tag{2.8}$$

Sind mehrere zeitbehaftete Transitionen in einer Markierung m aktiviert, so entspricht die Wahrscheinlichkeit dafür, dass Transition $t_i \in EN_T(m)$ feuert, der Wahrscheinlichkeit, dass die Feuerzeit von t_i kleiner ist als die Feuerzeit aller anderen in m aktivierten Transitionen $t_j \in EN_T(m)$:

$$p(t_i, m) = P[\bigwedge_{\substack{t_j \in EN_T(m) \\ i \neq j}} \chi_i < \chi_j] = \frac{w_i}{\sum\limits_{t_j \in EN_T(m)} w_j} \tag{2.9}$$

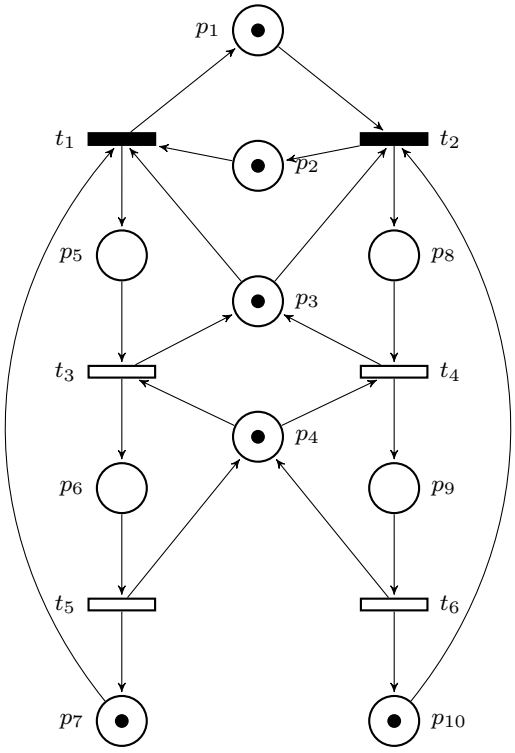
Analyse von GSPNs

Neben qualitativen Eigenschaften wie Lebendigkeit, Beschränktheit und Existenz von Heimatzustände, können bei GSPNs auch quantitave Eigenschaften untersucht werden, wie beispielsweise die durchschnittliche Anzahl von Tokens in einer Stelle p_i oder die Wahrscheinlichkeit, dass eine Transition t_i feuert. Ein GSPN beschreibt einen stochastischen Prozess. Dieser kann als zeitdiskrete Markow-Kette betrachtet werden, bei der einzelne Punkte in der Zeit zusammenfallen. Zur quantitativen Analyse wird die stationäre Verteilung des zugrundeliegenden stochastischen Prozesses untersucht. Als notwendige Voraussetzung für die Existenz einer solchen stationären Verteilung ist es wichtig, dass das GSPN beschränkt ist und Heimatzustände besitzt [7]. Da sich diese Masterarbeit mit der Analyse von qualitativen Eigenschaften

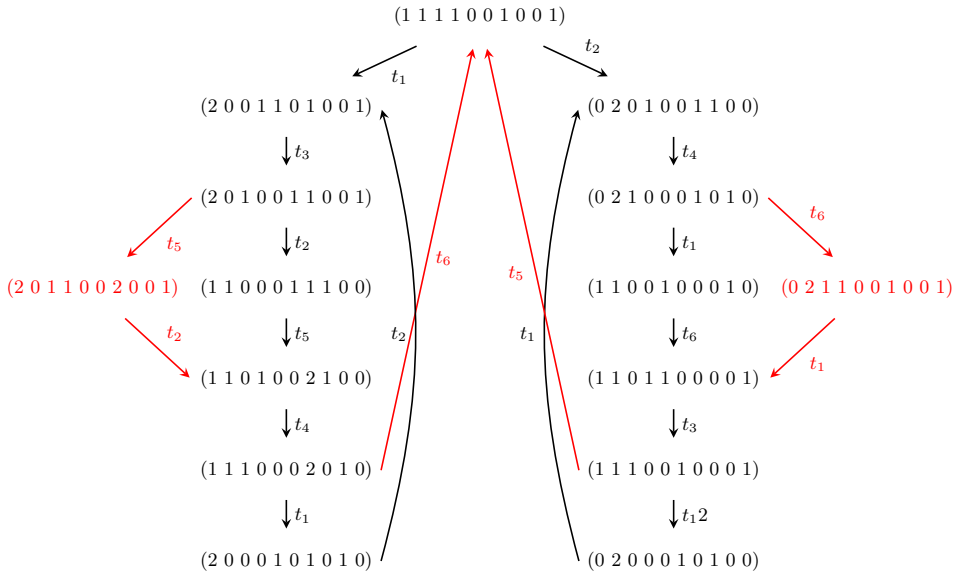
beschäftigt, sei der interessierte Leser für weitergehende Informationen zur quantitativen Analyse auf [7], [35] und [36] verwiesen.

Für die qualitative Analyse eines GSPNs lässt sich ähnlich wie beim S/T-Netz ein Erreichbarkeitsgraph verwenden. Dabei muss allerdings die höhere Priorität von zeitlosen gegenüber zeitbehafteten Transitionen berücksichtigt werden. Dies kann erreicht werden, indem Algorithmus 2.2 in Verbindung mit der Definition für aktivierte Transitionen im GSPN aus Gleichung 2.6 verwendet wird [35].

Beispiel 2.4.8. Abbildung 2.14 zeigt ein GSPN (a) und seinen Erreichbarkeitsgraphen (b). Im Erreichbarkeitsgraphen sind die Teile, die durch Berücksichtigung von zeitlosen gegenüber zeitbehafteten Transitionen wegfallen, rot markiert. Bei Interpretation des Netzes als S/T-Netz wäre das Netz beschränkt, lebendig und alle Markierungen wären Heimatzustände (gesamter Erreichbarkeitsgraph ist eine SZK). Bei Berücksichtigung der Priorität von zeitlosen gegenüber zeitbehafteten Transitionen zerfällt der Erreichbarkeitsgraph in zwei terminale SZKs. Das GSPN ist ebenfalls beschränkt und lebendig, jedoch besitzt es aufgrund der beiden terminalen SZKs keine Heimatzustände mehr.



(a) GSPN



(b) Erreichbarkeitsgraph ohne Berücksichtigung der Priorität von zeitlosen gegenüber zeitbehafteten Transitionen (rot)

Abbildung 2.14: Ein GSPN und sein Erreichbarkeitsgraph (vgl. [7, S. 159])

Kapitel 3

Queueing-Petri-Netze

Warteschlangennetze (*queueing networks*) (QN) sind ein sehr gern genutztes Mittel zur Systemleistungsanalyse. Die Modellierung mit QNs ist relativ einfach und benötigt nur wenige Parameter. Hinzu kommt, dass sehr effiziente Methoden zur Analyse bereitstehen. Jedoch lässt sich mit QNs keine Synchronisation darstellen, die in vielen nebenläufigen Systemen notwendig ist.[48]

Petri-Netze hingegen bieten die Möglichkeit, nebenläufige Prozesse und deren Synchronisation zu modellieren. Sie werden genutzt, um die Korrektheit von System zu zeigen. Mit GSPNs (siehe Abschnitt 2.4.2) wurden Petri-Netze um zeitliche Aspekte erweitert, sodass auch einige quantitative Eigenschaften von Systemen untersucht werden können. Jedoch kann Scheduling in GSPNs nur umständlich modelliert werden.

Um die Vorteile beider Modellarten zu nutzen, wurden Queueing-Petri-Netze (QPN) [5] entwickelt. Sie kombinieren Warteschlangennetze und gefärbte GSPNs, indem in Stellen Warteschlangensysteme integriert werden können.

3.1 Warteschlangensysteme

Ein *Warteschlangensystem* (*queuing system*) besteht aus einem Wartebereich und einer oder mehreren Bedienstationen (siehe Abbildung 3.1). Entitäten, dies können z. B. Personen oder andere Dinge wie beispielsweise Prozesse oder Pakete sein, die von einem solchen System „bedient“ werden wollen, werden *Kunden* (*customers*) genannt. Betritt ein Kunde ein Warteschlangensystem, wird er sofort bedient, falls eine Bedienstation frei ist, andernfalls muss er im Wartebereich warten. Die *Bedienzeit* eines Kunden beschreibt die Zeit, die eine Bedienstation benötigt, um den Kunden zu bedienen. Ist der Kunde fertig bedient worden, verlässt er das Warteschlangensystem wieder. Welcher Kunde als nächstes bedient wird, wird durch die sogenannte *Scheduling-Strategie* festgelegt. Zur einfachen Beschreibung eines War-

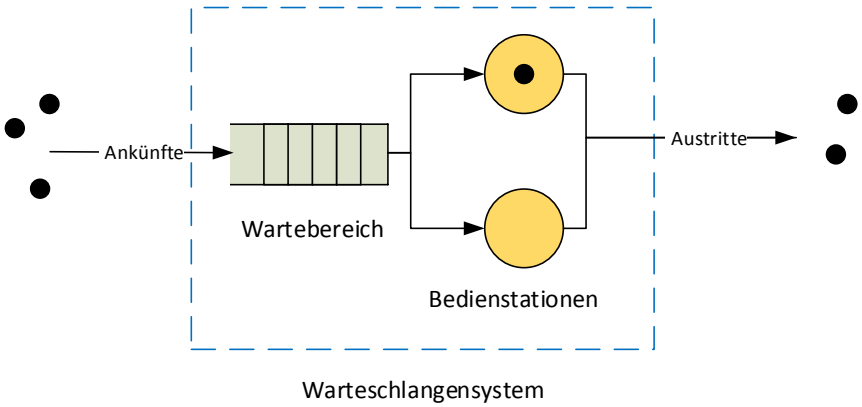


Abbildung 3.1: Aufbau eines Warteschlangensystems

teschlangensystems wird häufig die Kendall'sche Notation genutzt:

$$A/B/m/K/Z/Sched \tag{3.1}$$

Dabei gibt

- A die Verteilung der Zwischenankunftszeiten,
- B die Verteilung der Bedienzeiten,
- m die Anzahl der Bedienstationen,
- K die Kapazität der Warteschlange, d.h. wieviele Kunden sich maximal im Wartebereich aufhalten können,
- Z die Gesamtkundenpopulation, d.h. die Gesamtzahl der Kunden, die potenziell ins System eintreten können, und
- *Sched* die Scheduling Strategie

an. Häufig werden die Warteschlangenkapazität und die Kundenpopulation weggelassen, wenn diese unendlich sind.

Für die Angabe der Zwischenankunftszeiten und Bedienzeiten werden u. a. folgende Abkürzungen verwendet:

M Exponentialverteilung,

D deterministische Verteilung,

$\mathbf{C_k}$ k-phasige Cox-Verteilung

G allgemeine Verteilung

Bei einer deterministischen Verteilung sind die Ankunftszeiten konstant und im Voraus bekannt. Eine allgemeine Verteilung bedeutet, dass die genaue Verteilung nicht bekannt ist.

Einige der typischen Scheduling Strategien sind:

- *FCFS (First-Come-First-Served)*: Kunden werden in der Reihenfolge bedient, in der sie ankommen.
- *LCFS (Last-Come-First-Served)*: Der letzte ankommende Kunde wird als erster bedient.
- *PS (Processor-Sharing)*: Alle Kunden werden gleichzeitig bedient. Die Geschwindigkeit der Bedienstation wird gleichmäßig zwischen den einzelnen Kunden aufgeteilt.
- *IS (Infinite Server)*: Es gibt unendlich viele Bedienstationen.
- *PRIO (Priority-Service)*: Bedienung nach Prioritätsregeln.
- *RANDOM*: Kunden werden in zufälliger Reihenfolge bedient.

Beispiel 3.1.1. Es wird ein Router in einem paketvermittelnden Rechnernetz betrachtet (siehe Abbildung 3.2). Dieser hat einen Eingangspuffer von 32 MB. Jedes Paket hat eine Größe von 1 MB. Pakete kommen zu exponentialverteilten Zeiten an und werden in der selben Reihenfolge weitergeleitet, in der sie ankommen. Es wird angenommen, dass die Zeit zur Auswertung der Routinginformation konstant ist. Dann kann der Router als Warteschlangensystem $M/D/1/32/\infty/FCFS$ modelliert werden.

3.2 QPNs

Wie auch bei GSPNs wird bei QPNs zwischen zwei Arten von Transitionen unterschieden, den zeitlosen und den zeitbehafteten Transitionen. Zusätzlich können Stellen Warteschlangensysteme enthalten. Eine Stelle, die ein Warteschlangensystem enthält, wird *Queueing-Stelle* bezeichnet.

Definition 3.2.1 (Queueing-Petri-Netz [7]). Ein *Queueing-Petri-Netz* ist ein Tripel $\mathcal{QPN} = (\mathcal{CPN}, Q, W)$, wobei

- $CPN = (P, T, C, I^-, I^+, m_0)$ das zugrundeliegende gefärbte Petri-Netz ist,
- $Q = (\tilde{Q_1}, \tilde{Q_2}, (q_1, \ldots, q_{|P|}))$ mit
  - einer Menge von Queueing-Stellen $\tilde{Q_1} \subseteq P$,

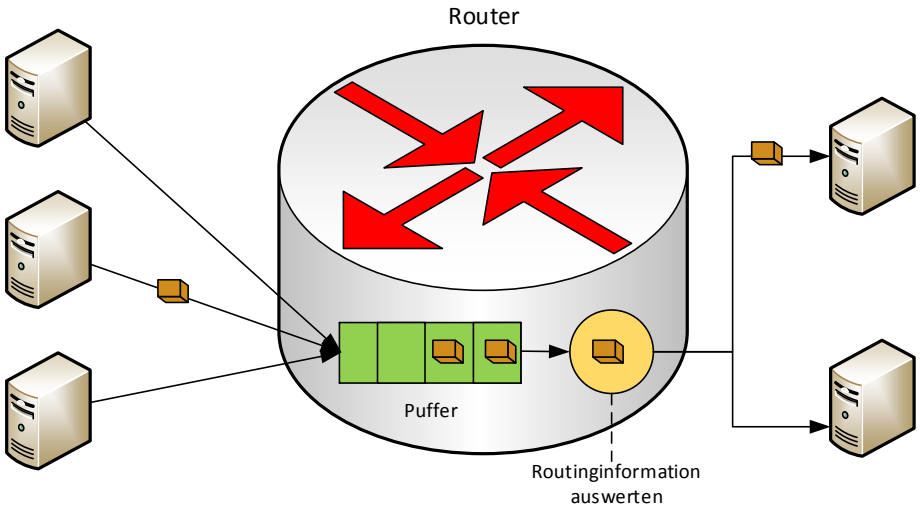


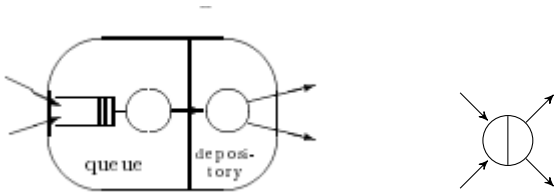
Abbildung 3.2: Router als Warteschlangensystem

  - einer Menge von gewöhnlichen Stellen $\tilde{Q}_2 \subseteq P$, wobei $\tilde{Q}_1 \cap \tilde{Q}_2 = \emptyset$ und $\tilde{Q}_1 \cup \tilde{Q}_2 = P$ und
  - der Beschreibung des Warteschlangennetzes q_i von Stelle p_i, wenn $p_i \in \tilde{Q}_1$, und *null*, wenn $q_i \in \tilde{Q}_2$, und

- $W = (\tilde{W}_1, \tilde{W}_2, (w_1, \ldots, w_{|T|}))$ mit
  - einer Menge von zeitbehafteten Transitionen $\tilde{W}_1 \subseteq T$,
  - einer Menge von zeitlosen Transitionen $\tilde{W}_2 \subseteq T$,
  - $(w_1, \ldots, w_{|T|})$ ist ein Array, dessen Eintrag $w_i \in [C(t_i) \to \mathbb{R}^+]$
    * die Feuerrate bezüglich einer Farbe $c \in C(t_i)$ angibt, wenn t_i eine zeitbehaftete Transition ist,
    * ein Feuergewicht bezüglich einer Farbe $c \in C(t_i)$ angibt, wenn t_i eine zeitlose Transition ist.[8]

In [5] wird zusätzlich zwischen zeitlosen und zeitbehafteten Queueing-Stellen unterschieden. Zeitbehaftete Queueing-Stellen entsprechen den hier definierten Queueing-Stellen. Zeitlose Queueing-Stellen entsprechen Warteschlangensystemen mit einer Bedienzeit von null und dienen damit nur dem Scheduling von Tokens. Der Einfachheit halber werden in dieser Arbeit nur Netze ohne zeitlose Queueing-Stellen betrachtet.

[8] vgl. GSPN



(a) (b)

Abbildung 3.3: Queuing-Stelle ([5, S. 4])

Queueing-Stellen bestehen aus zwei Teilen: dem Warteschlangensystem selbst und einem sogenannten Depository, in dem Tokens nach ihrer Bedienung abgelegt werden (siehe Abbildung 3.3(a)). Zu jeder Queueing-Stelle wird eine Beschreibung des Warteschlangensystems angegeben (beispielsweise in der Kendall'schen Notation). Queueing-Stellen werden grafisch durch einen in der Mitte geteilten Kreis dargestellt (siehe Abbildung 3.3(b)).

Das Schalten von zeitlosen Transitionen hat auch in QPNs Priorität gegenüber dem Schalten von zeitbehafteten Transitionen. Außerdem hat das Schalten von zeitlosen Transitionen auch Priorität gegenüber dem Bedienen in Queueing-Stellen, d.h. damit Tokens die Bedienstation ins Depository verlassen können, dürfen keine zeitlosen Transitionen aktiviert sein.

Definition 3.2.2 (Markierung in einem QPN). Eine *Markierung* in einem QPN $\mathcal{QPN} = (\mathcal{CPN}, Q, Q)$ ist ein Tupel $\mathcal{M} = (m, n)$ mit

- einer Funktion $m : P \to c(p)_{MS}$ und
- einer Funktion $n : \tilde{Q}_1 \to Zustand$.

Eine Markierung in einem QPN setzt sich aus zwei Teilen zusammen. m gibt wie bei CPNs (siehe Abschnitt 2.4.1) die Zahl der Tokens pro Stelle an. Bei Queueing-Stellen wird die Zahl der Tokens im Depository angegeben.

n gibt für jede Queueing-Stelle den Zustand des Warteschlangensystems an. Der Zustand kann z. B. bei FCFS-Queues ein String mit der Ankunftsreihenfolge der Tokenfarben sein. Bei PS-Queues kann der Zustand beispielsweise eine Multimenge der aktuell bearbeiteten Tokens sein.

Beispiel 3.2.3. Das Queueing-Petri-Netz aus Abbildung 3.4(a) wird formal beschrieben durch das gefärbte Petri-Netz $\mathcal{CPN} = (P, T, C, I^-, I^+)$ mit der Stellenmenge $P = \{p_1, p_2, p_3\}$, der Transitionsmenge $T = \{t_1, t_2, t_3\}$ und der Farbfunktion

$$\begin{aligned} C(p_1) &= \{b\} & C(t_1) &= \{b\} \\ C(p_2) &= \{a, b\} & C(t_2) &= \{a\} \\ C(p_3) &= \{a, b\} & C(t_3) &= \{a, b\} \end{aligned}$$

Die Inzidenzfunktionen sind:

$$\begin{aligned} I^-(p_1, t_1)(b)(b) &= 1 & I^+(p_2, t_1)(b)(b) &= 1 \\ I^-(p_2, t_3)(a)(a) &= 1 & I^+(p_2, t_2)(a)(a) &= 1 \\ I^-(p_2, t_3)(b)(b) &= 1 & I^+(p_3, t_3)(a)(a) &= 1 \\ & & I^+(p_2, t_3)(b)(b) &= 1 \end{aligned}$$

Ansonsten gilt für alle Stellen p, Transitionen t, Tokenfarben c und Schaltmodi c':

$$I^-(p, t)(c)(c') = 0 \text{ und } I^+(p, t)(c)(c') = 0$$

Die genaue Festlegung der Stellen ist:

$$Q = (\{q_2\}, \{q_1, q_3\}, (null, -/M/1/FCFS, null))$$

Es gibt also eine Queueing-Stelle p_2, die ein Warteschlangensytem mit einer Bedienstation, einer FCFS-Scheduling-Strategie und einer exponential-verteilten Bedienzeit enthält. Die Transitionen werden wie folgt festgelegt:

$$W = (\{t_3\}, \{t_1, t_2\}, (w_1, w_2, w_3))$$

Dabei sind alle Transitionen gleichgewichtet mit $w_1(b) = w_2(a) = w_3(a) = w_3(b) = 1$.

Die Startmarkierung ist $\mathcal{M}_0 = (m_0, n_0)$ mit $m_0(p_1) = \{2b\}$ und $m_0(p) = 0$ für alle anderen Plätze $p \in P$. Da das Warteschlangensystem zu Beginn immer leer ist, ist $n(p_2) = -$.

3.3 Analyse von QPNs

Wie auch GSPNs (siehe Abschnitt 2.4.2) beschreiben QPNs stochastische Prozesse und können durch die Bestimmung der stationären Verteilung der eingebetteten

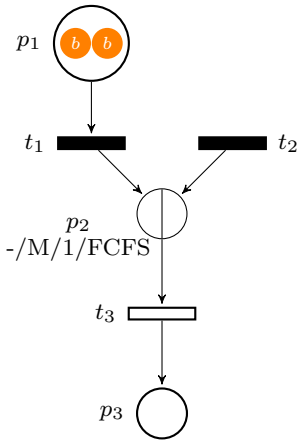


Abbildung 3.4: Ein Queueing-Petri-Netz

Markov-Kette untersucht werden. Der Zustandsraum kann in zwei Arten von Zuständen unterteilt werden:

- *tangible states*: In diesem Zustand treten nur zeitbehaftete Aktionen (z.B. feuern einer zeitbehafteten Transition oder Änderung des Warteschlangenzustands) auf.

- *vanishing states*: In diesem Zustand ist eine zeitlose Transition aktiviert, der Zustand wird in Nullzeit wieder verlassen.

Für die quantitative Analyse von QPNs ist es notwendig, dass der Zustandsraum endlich ist, d.h. das QPN beschränkt ist. Außerdem ist es wichtig, dass niemals ein Zustand erreicht werden kann, auf den nur noch vanishing states folgen, bzw. in allen Folgezustände nur noch zeitlose Transitionen gefeuert werden. Dieses Problem wird als *Nullzeit-Falle* (*timeless trap*) bezeichnet. Ein beschänktes und lebendiges QPN hat keine Nullzeit-Fallen. Für die Existenz einer stationären Verteilung des eingebetten Markov-Prozesses ist es notwendig, dass nur eine irreduzible Teilmenge von Zuständen im Zustandsraum existiert. Dies entspricht genau der Existenz von Heimatzuständen.[4]

Diese qualitativen Eigenschaften lassen sich beispielsweise mit dem Überdeckungsgraphen untersuchen. Dabei müssen allerdings mehrere Aspekte berücksichtigt werden. Zum Einen sind jetzt nicht nur Transitionen aktive Komponenten des Netzes. Die in Queueing-Stellen integrierten Warteschlangensysteme können die Bedienung eines Tokens beenden und damit den Systemzustand durch den Übergang der Tokens ins Depository ändern. Zum Anderen bestehen Markierungen in QPNs aus zwei Teilen. Überdeckende Markierungen müssen entsprechend neu definiert werden, um Unbeschränktheit finden zu können. Es gilt $\mathcal{M}' = (m', n') > \mathcal{M} = (m, n)$,

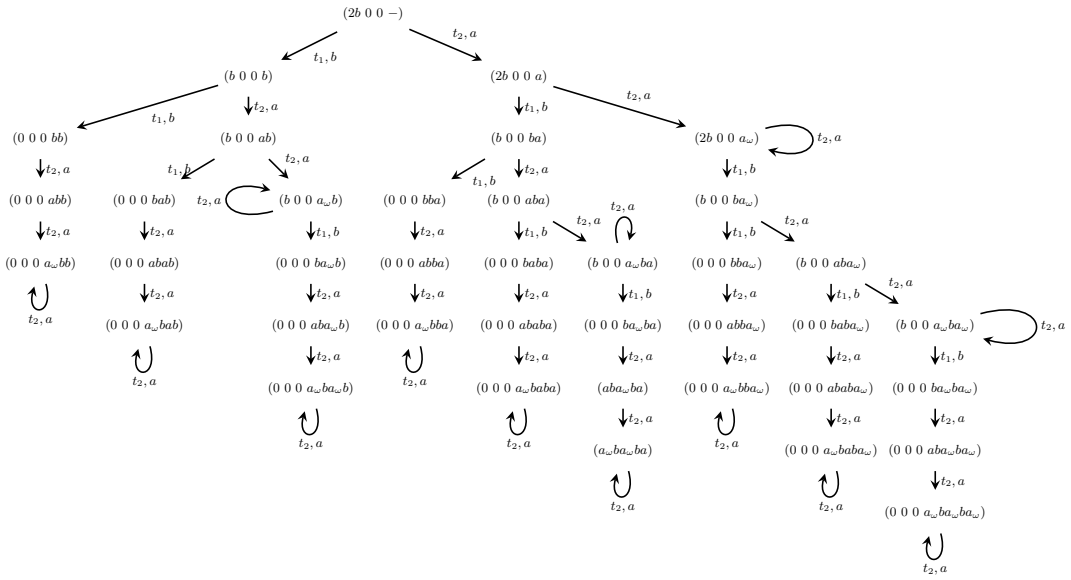


Abbildung 3.5: Überdeckungsgraph zum QPN aus Beispiel 3.2.3

wenn $m' > m \wedge n' \geq n$ oder $m' \geq m \wedge n' > n$. Dabei ist $n' > n$ von der Zustandsbeschreibung abhängig. Für FCFS-Warteschlangensysteme, deren Zustand als String beschrieben wird, ist $n' > n$, wenn n ein Teilstring von n' ist und beide im letzten Zeichen übereinstimmen. Beispielsweise überdeckt aab den Zustand ab. Bei überdeckenden Zuständen wird das letzte Zeichen mit ω gekennzeichnet, im Beispiel zuvor wäre der Zustand also $a_\omega b$. Für PS-Warteschlangensysteme mit einer Multimenge als Zustandsbeschreibung wäre $n' > n$, wenn $n \subset n'$. Algorithmus 3.1 zeigt die Generierung eines Überdeckungsgraphen für ein QPN. Dabei wird Algorithmus 2.1 darum erweitert, dass auch Warteschlangenzustände überdeckt werden können und entsprechende ω-Zustände gesetzt werden müssen (Zeilen 10-12). Ferner werden Zustandsübergänge durch ein Bedienende in einer Queueing-Stelle berücksichtigt, die aufgrund der Prioritäten nur eintreten, wenn keine zeitlose Transition aktiviert ist (Zeilen 18-32). Die Funktion *next* eines Queueing-Zustands q gibt dabei an, welche Token in dem Warteschlangensystem die Bedienstation potentiell als nächstes verlassen können.

Beispiel 3.3.1. Abbildung 3.5 zeigt den Überdeckungsgraphen zum QPN aus Beispiel 3.2.3. Aus dem Überdeckungsgraphen lässt sich ableiten, dass das QPN nicht beschränkt ist und keine Heimatzustände besitzt. Außerdem ist das QPN nicht quasilebendig, da Transition t_3 tot ist. Somit ist das Netz auch nicht lebendig. Es ist jedoch pseudo-lebendig.

Beispiel 3.3.1 deutet das Problem der Zustandsraumexplosion in QPNs durch Berücksichtigung der Warteschlangenzustände im Zustandsraum an. Während der Über-

Algorithmus 3.1 Generierung des Überdeckungsgraphen für ein QPN

Eingabe: Petri-Netz $\mathcal{QPN} = (\mathcal{CPN}, Q, W, \mathcal{M}_0)$
Ausgabe: Gerichteter Graph $CG(\mathcal{QPN}) = (V, E)$

1: Initialisiere $RG(\mathcal{QPN}) = (\{\mathcal{M}_0\}, \emptyset)$, m_0 ist nicht markiert
2: **while** es gibt noch unmarkierte Knoten in V **do**
3: wähle unmarkierten Knoten $\mathcal{M} = (m, n) \in V$ und markiere ihn
4: **for** jede in $\mathcal{M}$ aktivierte Transition t bezüglich Farbe c **do**
5: bestimme $\mathcal{M}' = (m', n')$, sodass $\mathcal{M} \xrightarrow{t,c} \mathcal{M}'$
6: **if** es gibt $\mathcal{M}'' = (m'', n'') \in V$ mit $\mathcal{M}'' \xrightarrow{\sigma} \mathcal{M}'$ und $\mathcal{M}' > \mathcal{M}''$ **then**
7: **for** jede Stelle p mit $m'(p) > m''(p)$ **do**
8: $m'(p) := \omega$
9: **end for**
10: **for** jede Queuing-Stelle q mit $n'(q) > n''(q)$ **do**
11: setze ω-Status für $n'(q)$
12: **end for**
13: **else if** es gibt kein $\mathcal{M}'' \in V$ mit $\mathcal{M}'' = \mathcal{M}'$ **then**
14: $V := V \cup \{\mathcal{M}'\}$
15: **end if**
16: $E := E \cup \{\mathcal{M}, (t, c), \mathcal{M}'\}$
17: **end for**
18: **if** es gibt keine in $\mathcal{M}$ aktivierte zeitlose Transition **then**
19: **for** jede Queuing-Stelle q **do**
20: **for** jede Farbe c in $q.next$ **do**
21: bestimme $\mathcal{M}' = (m', n')$, sodass $\mathcal{M} \xrightarrow{q,c} \mathcal{M}'$
22: **if** es gibt $\mathcal{M}'' = (m'', n'') \in V$ mit $\mathcal{M}'' \xrightarrow{\sigma} \mathcal{M}'$ und $\mathcal{M}' > \mathcal{M}''$ **then**
23: **for** jede Stelle p mit $m'(p) > m''(p)$ **do**
24: $m'(p) := \omega$
25: **end for**
26: **else if** es gibt kein $\mathcal{M}'' \in V$ mit $\mathcal{M}'' = \mathcal{M}'$ **then**
27: $V := V \cup \{\mathcal{M}'\}$
28: **end if**
29: $E := E \cup \{\mathcal{M}, t, \mathcal{M}'\}$
30: **end for**
31: **end for**
32: **end if**
33: **end while**
34: **return** $CG(\mathcal{QPN})$

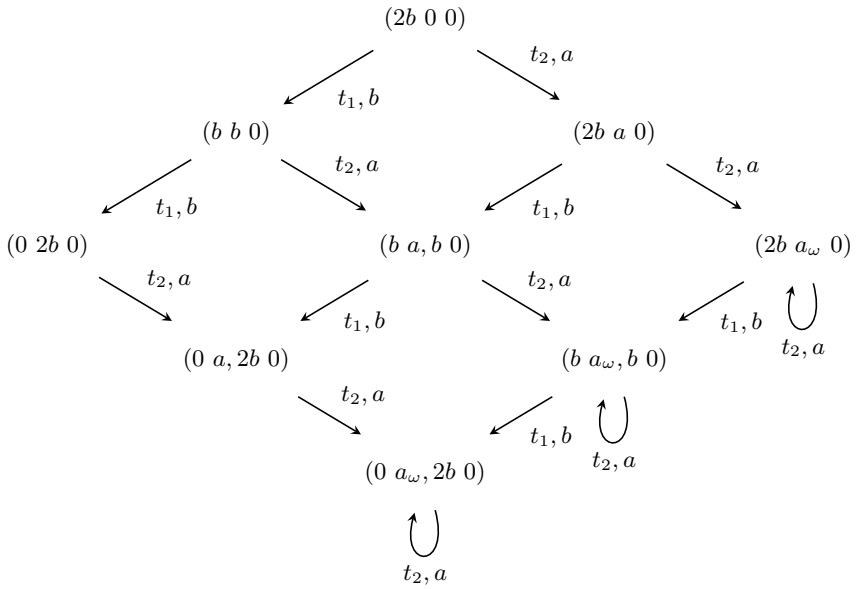


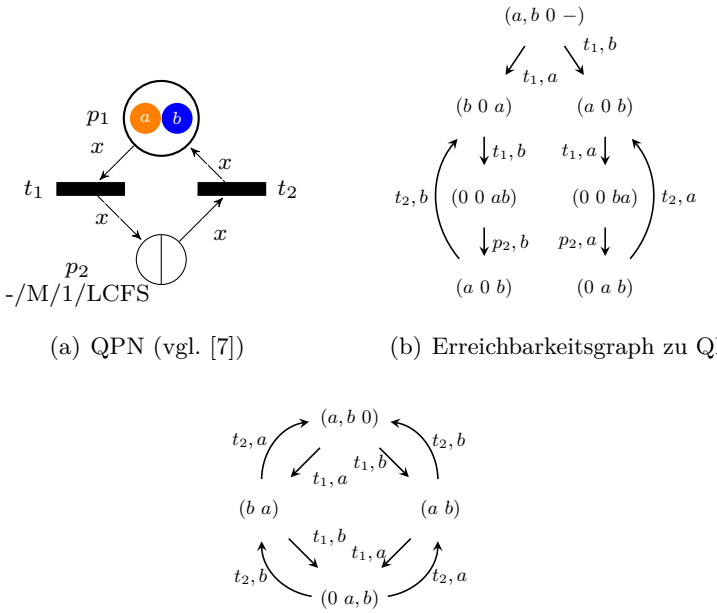
Abbildung 3.6: Überdeckungsgraph zum dem QPN aus Beispiel 3.2.3 zugrundeliegenden CPN unter Berücksichtigung der Priorität von zeitlosen gegenüber zeitbehafteten Transitionen

deckungsgraph zum QPN 35 Zuständen besitzt, besteht der Überdeckungsgraph des zugrundeliegenden gefärbten Generelised-Stochastic-Petri-Netzes (GSPN) nur aus 9 Zuständen (siehe Abbildung 3.6). Es wäre daher nützlich, wenn aus der Untersuchung der Eigenschaften des zugrundeliegenden Netzes auf Eigenschaften des QPNs geschlossen werden könnte. Während aus der Beschränkung des zugrundeliegenden CPNs die Beschränkung des QPNs folgt[9] [4], ist dies für andere Eigenschaften allgemein jedoch nicht der Fall.

Beispiel 3.3.2. Das QPN aus Abbildung 3.7(a) ist beschränkt, nicht lebendig und besitzt keine Heimatzustände (siehe Erreichbarkeitsgraph in Abbildung 3.7(b)), obwohl das zugrundeliegende CGSPN lebendig ist und Heimatzustände besitzt (siehe Erreichbarkeitsgraph in Abbildung 3.7(c)).

Es müssen also Bedingungen gefunden werden, unter denen die Eigenschaften im zugrundeliegenden Netz beibehalten. Da diese Bedingungen sehr einfach nachzuprüfen sein sollten und dies für allgemeine Petri-Netze schwierig ist, wurde sich in [6] auf EFC-Netze beschränkt.

[9] Voraussetzung ist, dass in den Warteschlanensystemen keine neuen Tokens erzeugt werden. Dies ist für alle in der Masterarbeit betrachteten Warteschlangensysteme der Fall.



(a) QPN (vgl. [7])

(b) Erreichbarkeitsgraph zu QPN

(c) Erreichbarkeitsgraph zu zugrundeliegendem CGSPN

Abbildung 3.7: Ein nicht-lebendiges Queuing-Petri-Netz

Definition 3.3.3 (EFC-QPN [6]). Ein QPN ist ein *Extended-Free-Choice-QPN* (EFC-QPN), wenn das dem QPN zugrundeliegende, entfaltete CPN ein EFC-Netz ist.

Eine wichtige Bedingung für die Beibehaltung bestimmter Eigenschaften ist, dass Transitionen, die miteinander in Konflikt stehen, vom selben Typ sein sollen.

Definition 3.3.4 (EQUAL-Conflict [4]). Ein QPN erfüllt die Bedingung EQUAL-Conflict genau dann, wenn für alle $t, t' \in T$, $p \in P$, $c \in C(p)$, $c' \in C(t)$ und $c'' \in C(t')$ gilt: wenn $I^-(p,t)(c')(c) \times I^-(p,t')(c'')(c) > 0$, dann $\{t, t'\} \subseteq \tilde{W}_1$ oder $\{t, t'\} \subseteq \tilde{W}_2$.

Ist das dem EFC-QPN zugrundeliegende CPN beschränkt und lebendig und erfüllt das EFC-QPN die Bedingung EQUAL-Conflict, so enthält es keine Nullzeit-Fallen [7]. Für die Übertragung der Lebendigkeit und Existenz von Heimatzuständen auf EFC-QPNs ist es wichtig, dass in den Warteschlangensystemen alle Tokens gleichzeitig bedient werden und dadurch Situationen wie in dem QPN aus Abbildung 3.7(a), in denen Tokens nicht bedient werden, nicht auftreten.

Definition 3.3.5 (EQUAL-Service [7]). Ein QPN erfüllt Bedingung EQUAL-Service genau dann, wenn für alle Queuing-Stellen $p \in \tilde{Q}_1$ gilt: die Verteilung der

Bedienzeiten ist eine Cox-Verteilung und wenn $|C(p) > 1|$, dann ist die Scheduling-Strategie PS oder IS.

Ein EFC-QPN, das Bedingung EQUAL-Service erfüllt, und dessen zugrundeliegendes CPN beschränkt und lebendig ist, ist genau dann lebendig, wenn es Bedingung EQUAL-Conflict erfüllt. EQUAL-Conflict ist in so einem EFC-QPN auch hinreichende Bedingung für die Existenz von Heimatzuständen.[6]

Kapitel 4

Reduktion von Petri-Netzen

Dieses Kapitel beschäftigt sich mit der Reduktion von Petri-Netzen. Eine Reduktionsregel beschreibt dabei eine Transformation eines Petri-Netzes $\mathcal{P}$ in ein in der Regel kleineres Petri-Netz $\mathcal{P}'$ unter Beibehaltung einer Menge von Eigenschaften Π. Eine Regel wird durch drei Dinge charakterisiert:

- die Voraussetzungen, die nötig sind, um die Regel anwenden zu können,
- die Anwendung der Regel, d.h. Anweisungen, wie aus einem Netz $\mathcal{P}$ das reduzierte Netz $\mathcal{P}'$ gewonnen werden kann, und
- die Eigenschaften, die unter der Reduktion beibehalten werden.

Eine Eigenschaft π wird unter der Reduktion beibehalten, wenn das reduzierte Netz $\mathcal{P}'$ die Eigenschaft π genau dann erfüllt, wenn das ursprüngliche Netz $\mathcal{P}$ auch π erfüllt.

Ist ein Satz von Reduktionsregeln gegeben, so wird die Reduktion durchgeführt, indem die Regeln iterativ angewendet werden, bis keine Regel mehr angewendet werden kann oder das Netz nur noch aus einer Stelle und einer Transition besteht. Ziel der Reduktion ist, dass das Netz nach der Reduktion einfacher analysiert werden kann. Dabei sollte die Überprüfung von Voraussetzungen der Regeln effizient sein, da ansonsten keine Minderung der Berechnungskomplexität gewonnen wird.

In diesem Kapitel wird zunächst ein Satz bekannter Reduktionsregeln für Stellen/Transitions-Netze vorgestellt. Anschließend wird ein Teil der Regeln für die Anwendung in einem Queueing-Petri-Netz angepasst.

4.1 Regeln zur Reduktion von S/T-Netzen

Bei der Analyse von Queuing-Petri-Netzen (siehe Kapitel 3) spielen die Eigenschaften Beschränktheit, Lebendigkeit und die Existenz von Heimatzuständen eine wich-

tige Rolle. Daher werden nachfolgend nur Reduktionsregeln vorgestellt, die alle drei Eigenschaften beibehalten.

In [9] werden zehn Transformationsregeln vorgestellt, davon werden im Folgenden vier Regeln präsentiert. Die nicht aufgeführten Regeln erschienen als ungeeignet, da sie bestimmte Eigenschaften nicht beibehalten, die Überprüfung der Voraussetzungen zu komplex ist oder sie ein Petri-Netz erweitern und nicht reduzieren.

Regel 1 (Vereinfachung von redundanten Stellen). Betrachtet wird eine Stelle p.
Voraussetzungen: Es gibt eine Teilmenge $Q \subseteq P \setminus \{p\}$ und eine Gewichtsfunktion $v : Q \cup \{p\} \to \mathbb{N}$, sodass

(a) es eine Zahl $b \in \mathbb{N}_0$ gibt, für die gilt: $v(p)m_0(p) - \sum_{q \in Q} v(q)m_0(q) = b$,

(b) für alle $t \in T$ gilt: $v(p)I^-(p,t) - \sum_{q \in Q} v(q)I^-(q,t) \leq b$ und

(c) es für alle $t \in T$ eine Zahl $c_t \in \mathbb{N}_0$ gibt, sodass
$v(p)(I^+(p,t) - I^-(p,t)) - \sum_{q \in Q} v(q)(I^+(q,t) - I^-(q,t)) = c_t$.

Anwendung:

1. Entferne alle ein- und ausgehenden Kanten von p.
2. Füge für jede Transition $t \in T$ eine Kante (t, c_t, p) hinzu. Ein Kantengewicht mit $c_t = 0$ bedeutet, dass keine Kante eingefügt werden muss. Gilt für alle Transitionen $c_t = 0$, so ist der Knoten isoliert und kann entfernt werden.

Redundante Stellen sind Stellen, die niemals alleine verhindern, dass eine nachfolgende Transition nicht aktiviert ist, und können daher entfernt werden. Diese Regel ändert zwar nicht die Größe des Erreichbarkeitsgraphen, führt aber sehr häufig dazu, dass andere Regeln anwendbar werden [18]. Für die Überprüfung der Voraussetzungen können ähnliche Methoden wie für die Berechnung von Stellen-Invarianten verwendet werden.

Beispiel 4.1.1 (vgl. [8]). In dem Petri-Netz aus Abbildung 4.1(a) ist die Stelle p_3 redundant für $Q = \{p_2, p_5\}$ und $v(p_3) = v(p_2) = v(p_5) = 1$. Da $c_t = 0$ für alle Transitionen t, kann p_3 samt aller ein- und ausgehenden Kanten entfernt werden (b). Die Regel kann erneut angewendet werden, da die Stelle p_4 redundant ist für $Q = \emptyset$ und $v(p_4) = 1$. Auch hier ist wieder $c_t = 0$ für alle Transitionen t, sodass p_4 entfernt werden kann (c).

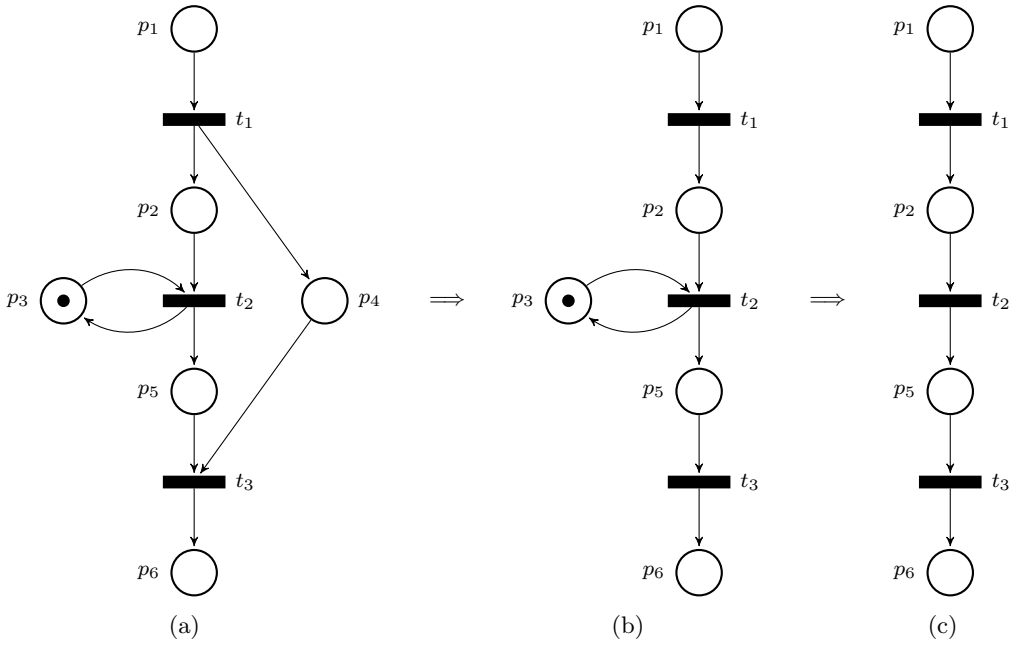


Abbildung 4.1: Petri-Netz nach zweifacher Anwendung der Regel *Vereinfachung von redundanten Stellen*

Regel 2 (Post-Fusion von Transitionen). Betrachtet wird eine Menge von Transitionen $F \subseteq T$ und eine Menge von Transitionen $H \subseteq T$.
Voraussetzungen: Es gibt eine Zahl $b \in \mathbb{N}$ und eine Stelle $p \in P$, sodass

(a) für alle Transitionen $f \in F$ gilt: ${}^\bullet f = \{p\}$, $I^-(p, f) = b$ und $p \notin f^\bullet$,

(b) es eine Transition $f \in F$ gibt mit $f^\bullet \neq \emptyset$,

(c) für alle Transitionen $h \in H$ gilt: $p \notin {}^\bullet h$ und es existiert eine Zahl $k_h \in \mathbb{N}$ mit $I^+(p, h) = k_h b$,

(d) für alle Transitionen $t \in T \setminus (F \cup H)$ gilt: $t \notin {}^\bullet p$ und $t \notin p^\bullet$ und

(e) $m_0(p) \leq b$.

Anwendung:

1. Entferne alle Transitionen $f \in F$, alle Transitionen $h \in H$ und die Stelle p samt aller ein- und ausgehenden Kanten.

2. Füge neue Transitionen $T' = \{t_{h,f} \mid h \in H, f \in F\}$ ein.

3. Füge für jede Transition $t_{h,f} \in T'$ für jede Stelle $q \in {}^\bullet h \setminus \{p\}$ eine Kante $(q, t_{h,f})$ mit $I^-(q, t_{h,f}) = I^-(q, h)$ ein.

4. Füge für jede Transition $t_{h,f} \in T'$ für jede Stelle $q \in f^\bullet \cup h^\bullet \setminus \{p\}$ eine Kante $(t_{h,f}, q)$ mit $I^+(q, t_{h,f}) = I^+(q, h) + I^+(q, f)$ ein.

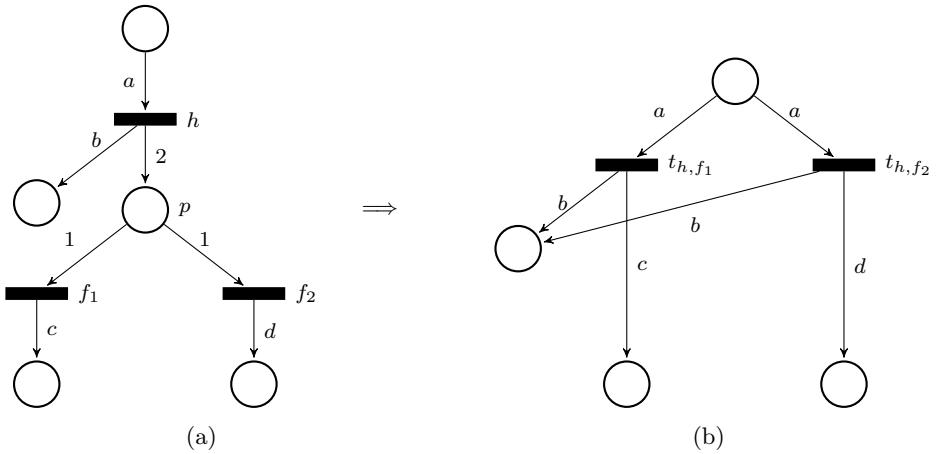


Abbildung 4.2: Petri-Netz nach Anwendung der Regel *Post-Fusion von Transitionen* (vgl. [9])

Bedingung (a) besagt, dass die Stelle p die einzige Vorstelle jeder Transition aus F ist und nicht zu deren Nachstellen gehört. Alle Bögen von p zu den Transitionen aus F müssen das gleiche Gewicht haben. Bedingung (b) fordert, dass mindestens eine Transition auf F eine Nachstelle hat. Bedingung (c) verlangt, dass p keine Vorstelle der Transitionen aus H ist, aber zu deren Nachstellen gehört. Dabei soll das Gewicht der Bögen von den Transitionen aus H zu p ein Vielfaches des Bogengewichts von p zu den Transitionen aus F sein. Bedingung (d) besagt, dass außer den Transitionen aus F und H keine weiteren Transitionen mit p verbunden sein dürfen. Die letzte Bedingung verlangt, dass die initiale Markierung von p kleiner ist als das Gewicht der Bögen von p zu einer Transition aus F. Diese Voraussetzungen stellen sicher, dass alle Transitionen aus F genau dann aktiviert werden, wenn eine Transition aus H gefeuert hat. Durch die Fusion der beiden Mengen wird der Zwischenschritt über die Stelle p eingespart.

Beispiel 4.1.2. Das Petri-Netz aus Abbildung 4.2(a) erfüllt für $F = \{f_1, f_2\}$, $H = \{h\}$ und $p = p$ die Voraussetzungen für die Post-Fusion. Abbildung 4.2(b) zeigt das aus der Fusion resultierende Netz.

Regel 3 (Pre-Fusion von Transitionen). Betrachtet wird eine Teilmenge von Transitionen $F \subseteq T$ und eine Transition $h \in T$.
Voraussetzungen: Es gibt eine Stelle $p \in P$ und eine Zahl $b \in \mathbb{N}$, sodass

(a) $h^\bullet = \{p\}$, $I + (p, h) = b$ und $p \notin {}^\bullet h$,

(b) ${}^\bullet h \neq \emptyset$,

(c) für alle Transitionen $f \in F$ gilt: $I^-(p, f) = b$ und $p \notin f^\bullet$,

(d) für alle Transitionen $t \in T \setminus (F \cup \{h\})$ gilt: $t \notin {}^\bullet p$ und $t \notin p^\bullet$,

(e) für alle Transitionen $t \in T$ mir $t \neq h$ gilt ${}^\bullet h \cap {}^\bullet t = \emptyset$ und

(f) $m_0(p) < b$.

Anwendung:

1. Entferne alle Transitionen $f \in F$, die Transition h und die Stelle p samt aller ein- und ausgehenden Kanten.
2. Füge neue Transitionen $T' = \{t_{h,f} \mid f \in F\}$ ein.
3. Füge für jede Transition $t_{h,f} \in T'$ für jede Stelle $q \in {}^\bullet h$ eine Kante $(q, t_{h,f})$ mit $I^-(q, t_{h,f}) = I^-(q, h)$ ein.
4. Füge für jede Transition $t_{h,f} \in T'$ für jede Stelle $q \in {}^\bullet f$ eine Kante $(q, t_{h,f})$ mit $I^-(q, t_{h,f}) = I^-(q, f)$ ein.
5. Füge für jede Transition $t_{h,f} \in T'$ für jede Stelle $q \in f^\bullet$ eine Kante $(t_{h,f}, q)$ mit $I^+(q, t_{h,f}) = I^+(q, f)$ ein.

Die Bedingungen besagen, dass p die einzige Nachstelle und keine Vorstelle von h ist (a), h mindestens eine Vorstelle hat (b) und jede Transition aus F die Stelle p nicht als Nachstelle hat (c). Ferner dürfen außer den Transitionen aus F und h keine weiteren Transitionen mit p verbunden sein (d) und die Vielfachheit aller ein- und ausgehenden Bögen von p muss gleich sein (a)+(c). Die Transition h darf keine Vorstellen mit anderen Transitionen teilen (f) und die initiale Markierung von p darf keine Tokens enthalten (e). Die Voraussetzungen ähneln denen aus der Post-Fusion, nur darf die Transition h jetzt nur ungeteilte Vorstellen und nur p als Nachstelle haben. Dafür können die Transitionen aus F mehrere Vorstellen haben. Daraus ergibt sich, dass h feuern muss, bevor die Transitionen aus F aktiviert werden. Wie auch bei der Post-Fusion wird der Zwischenschritt über die Stelle p nach der Reduktion ausgelassen.

Beispiel 4.1.3. Abbildung 4.3 zeigt die Anwendung der Regel *Pre-Fusion von Transitionen*. Dabei ist $F = \{f_1, f_2\}$.

Regel 4 (Quer-Fusion von Transitionen). Betrachtet werden zwei Transitionen $t_l, t_r \in T$.
Voraussetzungen: Es gibt eine Transition $t_c \in T$ und zwei Stellen $p_l, p_r \in P$, sodass

(a) $I^+(p_l, t_c) = I^+(p_r, t_c) = 1$ und ${}^\bullet p_l = {}^\bullet p_r = \{t_c\}$,

(b) ${}^\bullet t_l \neq \emptyset$, $t_l{}^\bullet \neq \emptyset$, ${}^\bullet t_r \neq \emptyset$ und $t_r{}^\bullet \neq \emptyset$,

(c) $p_l{}^\bullet = \{t_l\}$,

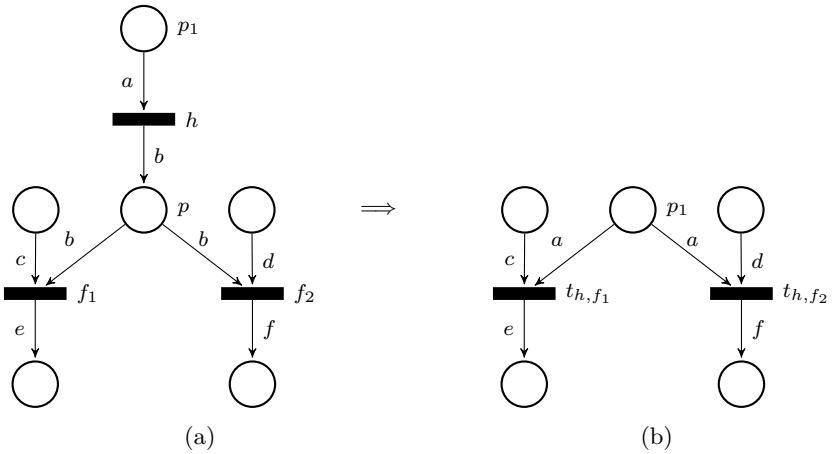


Abbildung 4.3: Petri-Netz nach Anwendung der Regel *Pre-Fusion von Transitionen* (vgl. [9])

(d) ${p_r}^\bullet = \{t_r\}$,

(e) wenn ${t_l}^\bullet \neq \emptyset$, dann ${}^\bullet t_r = \{p_r\}$,

(f) wenn ${t_r}^\bullet \neq \emptyset$, dann ${}^\bullet t_l = \{p_l\}$,

(g) für alle Transitionen $t \in T \setminus \{t_l\}$ gilt: ${}^\bullet t \cap {}^\bullet t_l = \emptyset$,

(h) für alle Transitionen $t \in T \setminus \{t_r\}$ gilt: ${}^\bullet t \cap {}^\bullet t_r = \emptyset$ und

(i) $m_0(p_l) = m_0(p_r)$.

Anwendung:

1. Entferne die Transitionen t_l und t_r.
2. Füge eine neue Transition $t_{l,r}$ ein.
3. Füge für jede Stelle $p \in {}^\bullet t_l$ eine Kante $(p, t_{l,r})$ mit $I^-(p, t_{l,r}) = I^-(p, t_l)$ ein.
4. Füge für jede Stelle $p \in {}^\bullet t_r$ eine Kante $(p, t_{l,r})$ mit $I^-(p, t_{l,r}) = I^-(p, t_r)$ ein.
5. Füge für jede Stelle $p \in {t_l}^\bullet$ eine Kante $(t_{l,r}, p)$ mit $I^+(p, t_{l,r}) = I^+(p, t_l)$ ein.
6. Füge für jede Stelle $p \in {t_r}^\bullet$ eine Kante $(t_{l,r}, p)$ mit $I^+(p, t_{l,r}) = I^+(p, t_r)$ ein.

Die erste Bedingung fordert, dass die Stellen p_l und p_r nur t_c als Vortransition haben und das Bogengewicht von t_c zu den beiden Stellen 1 ist. Bedingung (b) fordert, dass t_l und t_r beide Vor- und Nachstellen haben. Durch Bedingungen (c) und (d) wird beschrieben, dass t_l die Nachtransition von p_l und t_r die Nachtransition von p_r sein soll. Bedingungen (e) und (f) verlangen, dass wenn eine der beiden Transitionen t_l und t_r Nachstellen besitzt, die andere Transition nur eine Vorstelle

hat, nämlich p_l bzw. p_r. Bedingung (i) fordert die gleiche initiale Markierung von p_l und p_r. Bedingungen (g) und (h) besagen, dass t_l und t_r ihre Vorstellen nicht mit anderen Transitionen teilen. Daraus lässt sich folgern, dass wenn sowohl t_l und t_r Nachstellen haben, sie nur p_l bzw. p_r als Vorstellen haben. Dann ergibt sich aus den restlichen Bedingungen, dass in dem Fall t_l und t_r immer gleichzeitig aktiviert sind und somit miteinander verschmolzen werden können. In dem Fall, dass eine der beiden Transitionen eine weitere Vorstelle hat, hat die andere keine Nachstellen. Daher ist es unter den gegebenen Bedingungen für den Zustandsraum in Bezug auf Lebendigkeit, Beschränktheit und die Existenz von Heimatzuständen irrelevant, wenn diese Transition bei der Fusion eine weitere Vorbedingung (im Sinne einer weiteren Vorstelle) erhält.

Regel 5 (Quer-Fusion von Transitionen 2). Betrachtet werden zwei Transitionen $t_l, t_r \in T$.
Voraussetzungen: Es gibt eine Transition $t_c \in T$ und zwei Stellen $p_l, p_r \in P$, sodass

(a) $I^-(p_l, t_c) = I^-(p_r, t_c) = 1$ und $p_l{}^\bullet = p_r{}^\bullet = \{t_c\}$,

(b) ${}^\bullet t_l \neq \emptyset$, $t_l{}^\bullet \neq \emptyset$, ${}^\bullet t_r \neq \emptyset$ und $t_r{}^\bullet \neq \emptyset$,

(c) ${}^\bullet p_l = \{t_l\}$,

(d) ${}^\bullet p_r = \{t_r\}$,

(e) wenn ${}^\bullet t_l \neq \emptyset$, dann $t_r{}^\bullet = \{p_r\}$,

(f) wenn ${}^\bullet t_r \neq \emptyset$, dann $t_l{}^\bullet = \{p_l\}$,

(g) für alle Transitionen $t \in T \setminus \{t_l\}$ gilt: $t^\bullet \cap t_l{}^\bullet = \emptyset$,

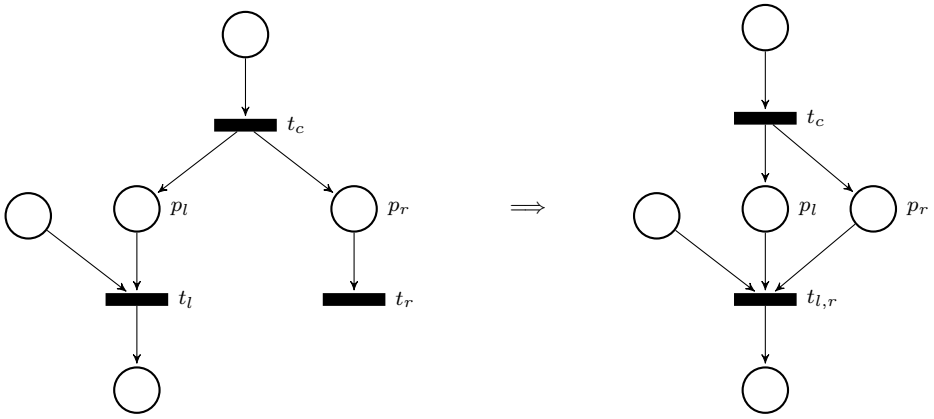
(h) für alle Transitionen $t \in T \setminus \{t_r\}$ gilt: $t^\bullet \cap t_r{}^\bullet = \emptyset$ und

(i) $m_0(p_l) = m_0(p_r)$.

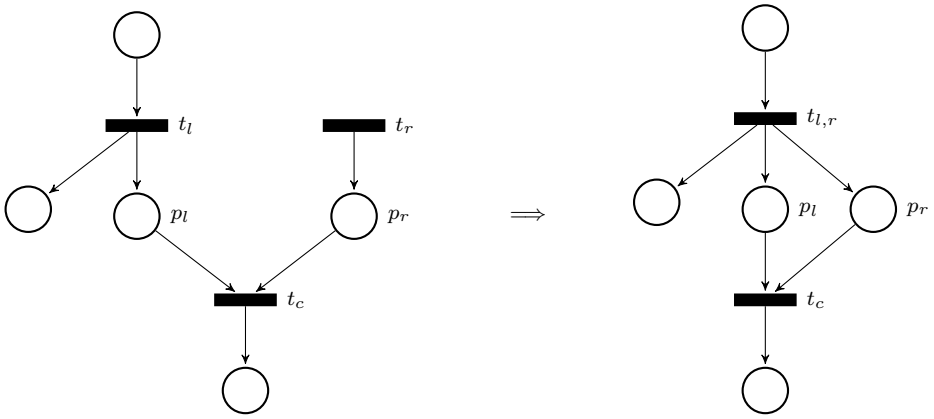
Anwendung:

1. Entferne die Transitionen t_l und t_r.
2. Füge eine neue Transition $t_{l,r}$ ein.
3. Füge für jede Stelle $p \in {}^\bullet t_l$ eine Kante $(p, t_{l,r})$ mit $I^-(p, t_{l,r}) = I^-(p, t_l)$ ein.
4. Füge für jede Stelle $p \in {}^\bullet t_r$ eine Kante $(p, t_{l,r})$ mit $I^-(p, t_{l,r}) = I^-(p, t_r)$ ein.
5. Füge für jede Stelle $p \in t_l{}^\bullet$ eine Kante $(t_{l,r}, p)$ mit $I^+(p, t_{l,r}) = I^+(p, t_l)$ ein.
6. Füge für jede Stelle $p \in t_r{}^\bullet$ eine Kante $(t_{l,r}, p)$ mit $I^+(p, t_{l,r}) = I^+(p, t_r)$ ein.

Diese Regel entspricht im Wesentlichen Regel 5, es werden lediglich überall Vorbereich und Nachbereich getauscht.



(a) Quer-Fusion von Transitionen 1



(b) Quer-Fusion von Transitionen 2

Abbildung 4.4: Petri-Netz nach Anwendung der Regel *Quer-Fusion von Transitionen*

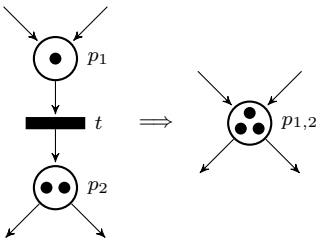
Beispiel 4.1.4. Abbildung 4.4 zeigt die Anwendung der beiden Regeln *Quer-Fusion von Transitionen 1*) (a) und *Quer-Fusion von Transitionen 2* (b). An den Abbildungen lässt sich die Symmetrie der beiden Regeln gut erkennen.

Als Nächstes wird ein Satz sehr einfacher Regeln vorgestellt. Diese Regeln sind aus [47] entnommen und entsprechen im Wesentlichen Spezialfällen der bereits vorgestellten Regeln. Abbildung 4.5 fasst die Funktionsweise der einzelnen Regeln graphisch zusammen.

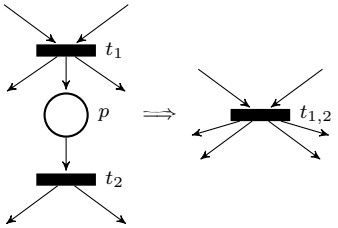
Regel 6 (Fusion von seriellen Stellen). Betrachtet werden zwei Stellen p_1 und p_2.
Voraussetzungen: Es gibt eine Transition t, sodass

(a) ${}^\bullet t = \{p_1\}$ und $t^\bullet = \{p_2\}$,

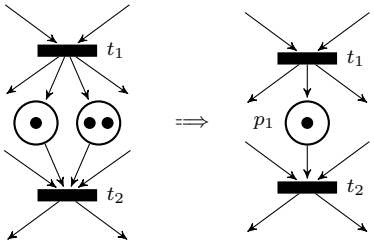
(b) $p_1{}^\bullet = {}^\bullet p_2 = \{t\}$ und



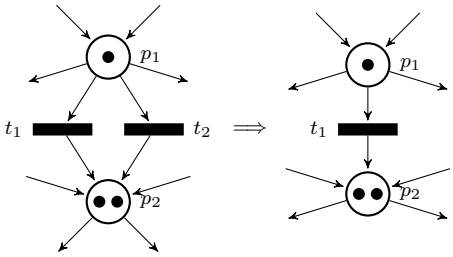
(a) Fusion von seriellen Stellen



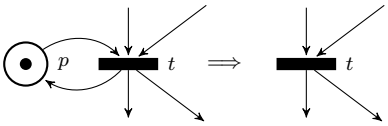
(b) Fusion von seriellen Transitionen



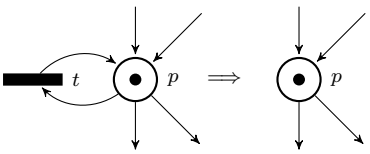
(c) Eliminierung von identischen Stellen



(d) Eliminierung von identischen Transitionen



(e) Eliminierung von Self-Loop-Stellen



(f) Eliminierung von Self-Loop-Transitionen

Abbildung 4.5: Ein Satz einfacher Reduktionsregeln (vgl. [47])

(c) $I^-(p_1, t) = I^+(p_2, t) = 1$

Anwendung:

1. Entferne die Transition t samt aller ein- und ausgehenden Kanten und die Stellen p_1 und p_2.
2. Füge eine Stelle $p_{1,2}$ hinzu.
3. Füge für alle Transitionen $h \in {}^\bullet p_1$ eine Kante $(h, p_{1,2})$ mit $I^+(p_{1,2}, h) = I^+(p_1, h)$ hinzu.
4. Füge für alle Transitionen $h \in p_2{}^\bullet$ eine Kante $(p_{1,2}, h)$ mit $I^-(p_{1,2}, h) = I^-(p_2, h)$ hinzu.
5. Setze $m_0(p_{1,2}) = m_0(p_1) + m_0(p_2)$.

Zwei serielle Plätze p_1 und p_2 können miteinander verschmolzen werden (siehe Abbildung 4.5(a)), wenn es eine Transition t gibt, die einzige Nachtransition von p_1 und einzige Vortransition von p_2 ist (Bedingung (b)), t keine weiteren Vor- und Nachstellen hat (Bedungung (b)) und die ein- und ausgehenden Bögen von t ein Gewicht von 1 haben (Bedingung (c)). Diese Regel ist ein Spezialfall der Macroplace-Regel von Silva [41].

Regel 7 (Fusion von seriellen Transitionen). Betrachtet werden zwei Transitionen t_1 und t_2.
Voraussetzungen: Es gibt eine Zahl $b \in \mathbb{N}$ und eine Stelle $p \in P$, sodass

(a) ${}^\bullet t_2 = \{p\}$, $t_2{}^\bullet \neq \emptyset$ und $I^-(p, t_2) = b$,

(b) es existiert eine Zahl $k_h \in \mathbb{N}$ mit $I^+(p, t_1) = k_h b$,

(c) ${}^\bullet p = \{t_1\}$ und $p^\bullet = \{t_2\}$ und

(d) $m_0(p) \leq b$.

Anwendung:

1. Entferne Transitionen t_1 und t_2 und die Stelle p samt aller ein- und ausgehenden Kanten.
2. Füge eine neue Transition $t_{1,2}$ ein.
3. Füge für jede Stelle $q \in {}^\bullet t_1$ eine Kante $(q, t_{1,2})$ mit $I^-(q, t_{1,2}) = I^-(q, t_1)$ ein.
4. Füge für jede Stelle $q \in t_1{}^\bullet \cup t_2{}^\bullet \setminus \{p\}$ eine Kante $(t_{1,2}, q)$ mit $I^+(q, t_{1,2}) = I^+(q, t_1) + I^+(q, t_2)$ ein.

Regel 7 ist ein Spezialfall von Regel 2, bei der die Mengen H und F jeweils nur aus einer Transition bestehen.

Regel 8 (Eliminierung von identischen Stellen). Betrachtet wird eine Menge von Stellen $Q = \{p_1, p_2, \ldots, p_n\} \subseteq P$.
Voraussetzungen:

(a) Für alle Transitionen $t \in T$ gilt: $I^-(p_1, t) = I^-(p_2, t) = \ldots = I^-(p_n, t)$ und $I^+(p_1, t) = I^+(p_2, t) = \ldots = I^+(p_n, t)$.

Anwendung:

1. Bestimme Stelle $p = \underset{p_i \in Q}{\operatorname{argmin}}\, m_0(p_i)$
2. Entferne alle Stellen $p_i \in Q \setminus \{p\}$ samt aller ein- und ausgehenden Kanten.

Hat eine Menge von Stellen die selben Vor- und Nachtransitionen und ist Bogengewicht von/zu je einer Transition bei allen Stellen gleich, so wird die Markenzahl in allen Stellen bei Schalten einer Vor- bzw. Nachtransition immer um den gleichen Betrag erhöht bzw. reduziert. Dadurch genügt es, nur die Stelle mit der geringsten Markenzahl zu betrachten, um zu bestimmen, welche Nachtransitionen aktiviert sind. Diese Regel ist ein Spezialfall von Regel 1.

Regel 9 (Eliminierung von identischen Transitionen). Betrachtet wird eine Menge von Transitionen $F = \{t_1, t_2, \ldots, t_n\} \subseteq T$.
Voraussetzungen:

(a) Für alle Stellen $p \in P$ gilt: $I^-(p, t_1) = I^-(p, t_2) = \ldots = I^-(p, t_n)$ und $I^+(p, t_1) = I^+(p, t_2) = \ldots = I^+(p, t_n)$.

Anwendung:

1. Entferne alle Transitionen $t_i \in F \setminus \{t_1\}$ samt aller ein- und ausgehenden Kanten.

Regel 9 ist dual zu Regel 8. Hat eine Menge von Transitionen die gleichen Vor- und Nachstellen und ist das Bogengewicht von/zu jeder Stelle jeweils gleich, so sind alle Transitionen aus dieser Menge zeitgleich aktiviert. Die Markierung nach dem Feuern einer Transition ist für all diese Transitionen gleich, sodass für den Erreichbarkeitsgraphen irrelevant ist, welche der Transitionen feuert. Daher können alle Transitionen aus dieser Menge bis auf eine entfernt werden.

Regel 10 (Eliminierung von Self-Loop-Stellen). Betrachtet wird eine Stelle p.
Voraussetzungen:

(a) ${}^\bullet p = p^\bullet$,

(b) für alle Transitionen t gilt: $I^+(p,t) = I^-(p,t)$ und

(c) $m_0 \geq \max_{t \in p^\bullet} I^-(p,t)$.

Anwendung:

1. Entferne die Stelle p samt aller ihrer ein- und ausgehenden Kante.

Entspricht die Menge der Vortransitionen einer Stellen der Menge deren Nachtransitionen und entspricht das Bogengewicht zu einer Transition jeweils dem Bogengewicht von der Transition zur Stelle, so bleibt die Zahl der Marken in der Stelle in jeder erreichbaren Markierung konstant. Bedingung (c) verhindert, dass durch das Entfernen der Stelle eine Transition, die vor der Reduktion tot war, es hinterher nicht mehr ist, und somit die Lebendigkeit unter der Reduktion beibehalten wird. Ist die Bedingung nicht erfüllt, so ist das Petri-Netz nicht lebendig, da es mindestens eine tote Transition gibt. Diese Regel entspricht einem Spezialfall von Regel 1

Regel 11 (Eliminierung von Self-Loop-Transitionen). Betrachtet wird eine Transition t.
Voraussetzungen:

(a) ${}^\bullet t = t^\bullet$,

(b) für alle Stellen p gilt: $I^+(p,t) = I^-(p,t)$

Anwendung:

1. Entferne die Transition t samt aller ihrer ein- und ausgehenden Kante.

Regel 11 ist dual zu Regel 10. Dadurch dass Vor- und Nachstellen der Transition mit entsprechendem Bogengewicht gleich sind, ändert das Feuern der Transition nicht den Systemzustand. Daher kann die Transition entfernt werden.

Beispiel 4.1.5 (vgl. [25]). Es wird das Petri-Netz aus Abbildung 4.6(a) betrachtet. Zunächst lassen sich mit Regel 6 das Teilnetz $op_1 - t_3 - wait\_dep$ (rot) zu einer Stelle p_3 und das Teilnetz $op_2 - t_6 - wait\_free$ (blau) zu einer Stelle p_6 zusammenfassen (siehe Abbildung 4.6(b)). Durch Anwendung von Regel 7 lassen sich anschließend die Teilnetze $t_1 - load - t_2$ (grün) zur Transition $t_{1,2}$, $t_4 - deposit - t_5$ (magenta) zur Transition $t_{4,5}$, $t_7 - unload - t_8$ (orange) zur Transition $t_{7,8}$ und $t_9 - withdrawal - t_{10}$ (violett) zur Transition $t_{9,10}$ reduzieren. In dem daraus resultierenden Netz (siehe Abbildung 4.6(c)) erfüllt die Stelle R (blau) nun die Voraussetzungen für die Anwendung von Regel 10 und kann entfernt werden. Die Teilnetze $wait\_raw - t_{1,2} - p_3$ (grün) und $p_6 - t_{7,8} - wait\_with$ (orange) aus Abbildung 4.6(d) können mit Regel 6 zu Stellen p_{12} bzw. p_{78} zusammengefasst werden (siehe Abbildung 4.6(e)). Anschließend können durch Anwendung von Regel 10 die Stellen p_{12}(grün) und p_{78} (orange)

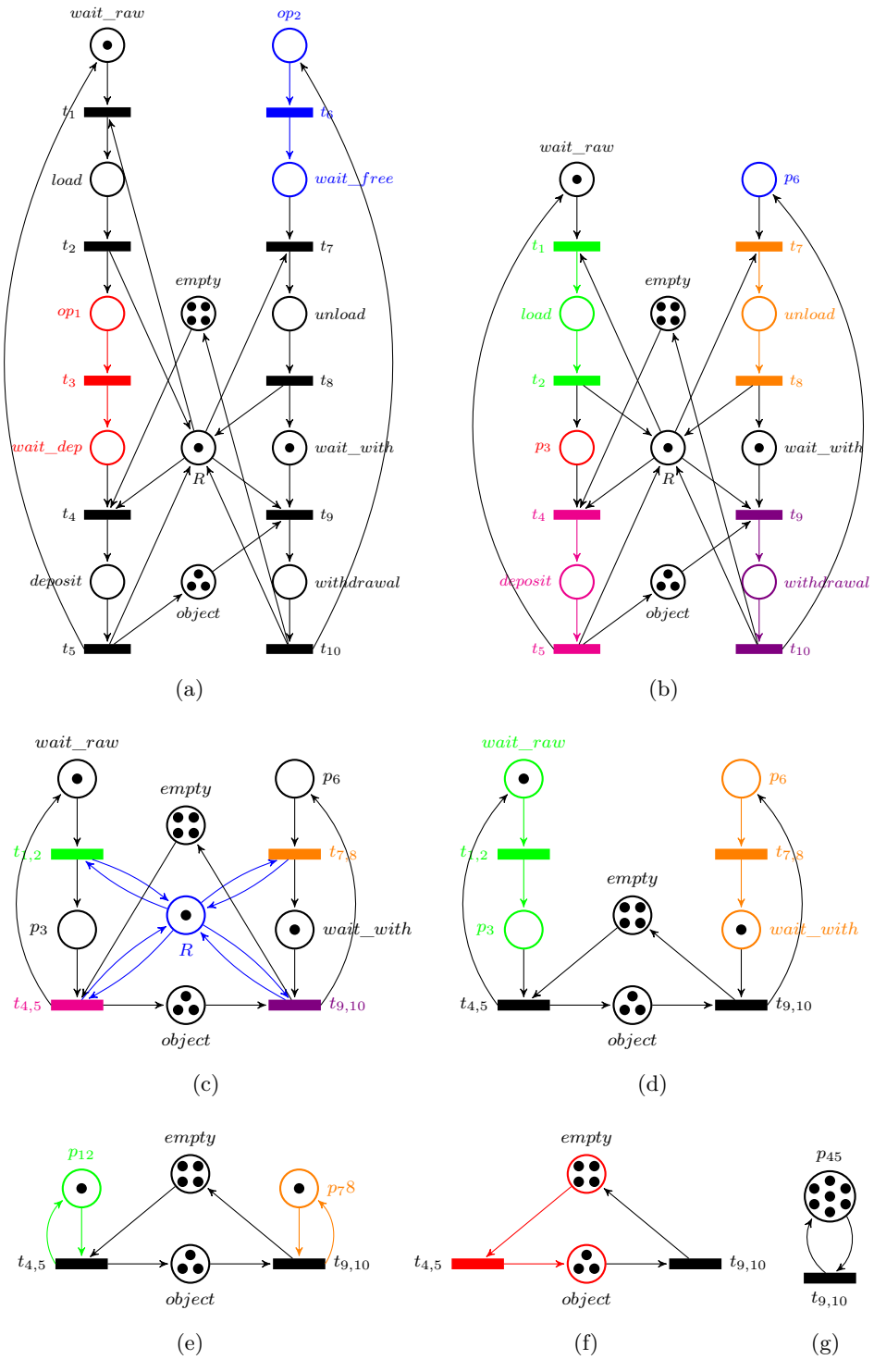


Abbildung 4.6: Schrittweise Reduktion eines Petri-Netzes (vgl. [25, S. 71,281])

Abbildung 4.7: Atomares Netz nach Reduktion eines lebendigen und beschränkten FC-Netzes

entfernt werden. Im Netz aus Abbildung 4.6(f) kann Regel 6 angewendet werden und das Teilnetz $empty - t_{45} - object$ (rot) zu einer Stelle p_{45} verschmolzen werden. Das daraus resultierende Netz (siehe Abbildung 4.6(g)) besteht nur noch aus einer Stelle und einer Transition, sodass der Reduktionsvorgang beendet werden kann. Es lässt sich nun sehr einfach feststellen, dass das reduzierte – und damit auch das ursprüngliche – Petri-Netz beschränkt, lebendig und reversibel ist.

Für allgemeine Petri-Netze gibt es noch keinen Satz von Regeln, der vollständig wäre. Vollständig bedeutet in dem Zusammenhang, das alle Netze die bestimmte Eigenschaften erfüllen, wie z.B. Beschränktheit und Lebendigkeit, immer auf das gleiche Netz reduziert werden können. Gäbe es so einen Satz von Regeln, so würde es reichen, nur die Form des durch die Reduktion resultierenden Netzes zu betrachten, um daraus die Eigenschaften des Netzes ableiten zu können. [9]

Ein solcher Satz von Regeln existiert jedoch für Free-Choice-Netze (siehe Abschnitt 2.3.3). In [21] wurden vier Regeln vorgestellt, die es ermöglichen, lebendige und beschränkte Free-Choice-Netze – und nur solche Netze – auf ein Netz bestehend aus einem Kreis aus einer Stelle und einer Transition (siehe Abbildung 4.7) zu reduzieren [21]. Da diese Regeln größtenteils bereits durch die hier dargestellten Regeln abgedeckt werden, werden sie hier nicht noch einmal aufgeführt.

4.2 Erweiterung der Regeln zur Reduktion von QPNs

Die Reduktionsregeln aus dem vorherigen Abschnitt können für QPNs leider nicht einfach so übernommen werden, da sich durch die Integration der Zeit der Zustandsraum ändern kann. Bei der Betrachtung der Regeln für den Fall von QPNs wird sich auf die letzten sechs, einfachen Regeln beschränkt. Ferner wird von einem bereits entfalteten QPN $\mathcal{QPN} = (P, T, I^-, I^+, Q, W, m_0)$ ausgegangen.

Für von der Reduktion betroffene Teilnetze, die nur aus zeitlosen Transitionen und gewöhnlichen Stellen bestehen, sollte ein Großteil der Regeln komplett übernommen werden können, da die Teilnetze S/T-Netzen entsprechen. Für zeitbehaftete

Transitionen muss jedoch genau geprüft werden, wie sie in die Reduktionsregeln eingebunden werden können.

QPN-Regel 1 (Fusion von seriellen Stellen). Betrachtet werden zwei gewöhnliche Stellen $p_1, p_2 \in \tilde{Q}_2$.
Voraussetzungen: Es gibt eine Transition $t \in \tilde{W}_2$, sodass

(a) ${}^\bullet t = \{p_1\}$ und $t^\bullet = \{p_2\}$,

(b) $p_1{}^\bullet = {}^\bullet p_2 = \{t\}$ und

(c) $I^-(p_1, t) = I^+(p_2, t) = 1$

Anwendung:

1. Entferne die Transition t samt aller ein- und ausgehenden Kanten und die Stellen p_1 und p_2.
2. Füge eine gewöhnliche Stelle $p_{1,2}$ hinzu.
3. Füge für alle Transitionen $h \in {}^\bullet p_1$ eine Kante $(h, p_{1,2})$ mit $I^+(p_{1,2}, h) = I^+(p_1, h)$ hinzu.
4. Füge für alle Transitionen $h \in p_2{}^\bullet$ eine Kante $(p_{1,2}, h)$ mit $I^-(p_{1,2}, h) = I^-(p_2, h)$ hinzu.
5. Setze $m_0(p_{1,2}) = m_0(p_1) + m_0(p_2)$.

Bei Regel 1 darf die Transition nur zeitlos sein. Durch die Priorität von zeitlosen gegenüber zeitbehafteten Transitionen könnte ein Netz, das ursprünglich nicht lebendig war, durch das Entfernen der zeitbehafteten Transition lebendig werden.

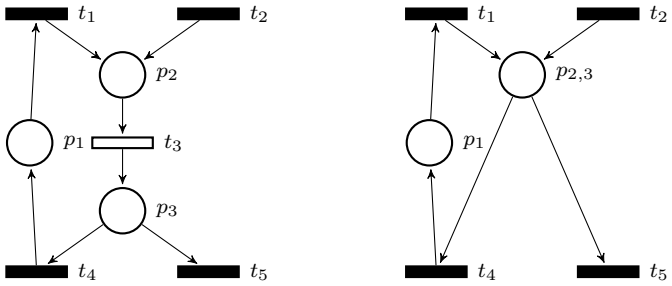
Beispiel 4.2.1. In dem QPN aus Abbildung 4.8(a) bildet die Transition t_2 eine Nullzeit-Falle. Dadurch wird die zeitbehaftete Transition t_3 niemals gefeuert und das Netz ist nicht lebendig. Wird dennoch QPN-Regel 1 angewendet und die Stellen p_2, p_3 und die Transition t_3 miteinander verschmolzen, so ist das daraus resultierende Netz (b) lebendig.

QPN-Regel 2 (Fusion serieller Transitionen). Betrachtet werden zwei Transitionen $t_1 \in T$ und $t_2 \in \tilde{W}_2$.
Voraussetzungen: Es gibt eine Zahl $b \in \mathbb{N}$ und eine gewöhnlichen Stelle $p \in \tilde{Q}_2$, sodass

(a) ${}^\bullet t_2 = \{p\}$, $t_2{}^\bullet \neq \emptyset$ und $I^-(p, t_2) = b$,

(b) es existiert eine Zahl $k_h \in \mathbb{N}$ mit $I^+(p, t_1) = k_h b$,

(c) ${}^\bullet p = \{t_1\}$ und $p^\bullet = \{t_2\}$ und



(a) Ein nicht-lebendiges QPN (b) Fusion von Stellen p_2 und p_3

Abbildung 4.8: Falsche Anwendung von QPN-Regel 1

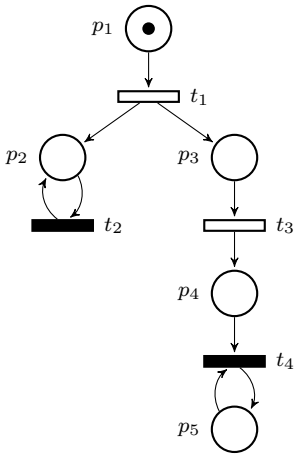
(d) $m_0(p) \leq b$.

Anwendung:

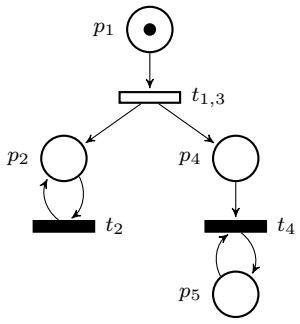
1. Entferne Transitionen t_1 und t_2 und die Stelle p samt aller ein- und ausgehenden Kanten.
2. Füge eine neue Transition $t_{1,2}$ ein. Die Transition ist zeitbehaftet, wenn $t_1 \in \tilde{W}_1$ und zeitlos, wenn $t_1 \in \tilde{W}_2$.
3. Füge für jede Stelle $q \in {}^\bullet t_1$ eine Kante $(q, t_{1,2})$ mit $I^-(q, t_{1,2}) = I^-(q, t_1)$ ein.
4. Füge für jede Stelle $q \in t_1{}^\bullet \cup t_2{}^\bullet \setminus \{p\}$ eine Kante $(t_{1,2}, q)$ mit $I^+(q, t_{1,2}) = I^+(q, t_1) + I^+(q, t_2)$ ein.

Da Transition t_2 eine zeitlose Transition ist, kann sie nach dem Feuern von t_2 auf jeden Fall feuern und die gleichen Stellen belegen wie im reduzierten Netz nach dem Feuern von Transition $t_{1,2}$ und damit die gleiche Markierung erreichen. Transition $t_{1,2}$ muss auf jeden Fall zeitbehaftet sein, wenn t_1 zeitbehaftet ist, da es sein kann, dass ein Zustand erreicht wird, ab dem die Transition niemals mehr aktiviert werden kann. In dem Fall wird sie auch in dem reduzierten Netz niemals aktivierbar sein und damit die Lebendigkeit bzw. Nicht-Lebendigkeit beibehalten.

Für den Fall, dass t_1 zeitlos und t_2 zeitbehaftet oder beide Transitionen zeitbehaftet sind, muss die Regel etwas angepasst werden, da durch das Feuern von t_1 ein Zustand erreicht werden kann, ab dem Transition t_2 niemals feuern kann. Ersetzt man in ersterem Fall die Transitionen durch eine zeitlose Transitionen, erhalten die Nachstellen von t_2 eventuell Marken, die sie so niemals erhalten hätten. Ersetzt man die Transitionen hingegen durch eine zeitbehaftete Transition, könnte ein Zustand erreicht werden, ab dem die Transition niemals feuern kann und die Nachstellen von t_1 erhalten eventuell keine Marken, die sie sonst erhalten hätten. Im Fall, dass beide Transitionen zeitbehaftet sind, kann das zweite Problem ebenfalls auftreten, falls



(a) Ein beschränktes QPN



(b) Nicht beschränktes QPN nach Fusion von Transitionen t_1 und t_3

Abbildung 4.9: Falsche Anwendung von QPN-Regel 2

durch das Feuern von t_1 ein Zustand erreicht wird, ab dem t_2 niemals aktivierbar ist. Um dieses Problem zu umgehen, muss die Einschränkung hinzugefügt werden, dass t_1 keine Nachstellen außer p hat.

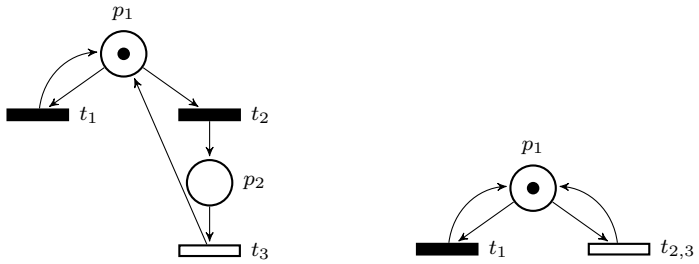
Beispiel 4.2.2. Das Petri-Netz aus Abbildung 4.9(a) ist beschränkt. Fusioniert man Transition t_1 mit Transition t_3, so erhält Stelle p_4 ein Token, das es im ursprünglichen Netz nie erhalten hätte. Dadurch ist jetzt Transition t_4 aktiviert und kann aufgrund der Schleife zu p_4 unendlich oft feuern, sodass Stelle p_5 unendlich viele Tokens erhalten kann. Damit ist das reduzierte Netz nicht mehr beschränkt. Wäre Transition t_3 im ursprünglichen Netz hingegen zeitlos, so wäre auch das ursprüngliche Netz unbeschränkt, sodass es bei er Fusion zu keinen Problemen käme.

Eine weitere Beschränkung muss für den Fall, dass t_1 zeitlos, t_2 zeitbehaftet und die Vorstellen von t_1 weitere Nachtransitionen haben, eingeführt werden. Ist unter den anderen Nachtransitionen eine zeitlose Transition, so kann es passieren, dass die fusionierte Transition niemals feuert, obwohl t_2 es vorher konnte (siehe Abbildung 4.10). Im Fall das die anderen Transitionen zeitbehaftet sind, könnten diese feuern, obwohl sie es vorher nicht konnten (siehe Abbildung 4.11).

QPN-Regel 2* (Fusion serieller Transitionen). Betrachtet werden zwei Transitionen $t_1 \in T$ und $t_2 \in T$.
Voraussetzungen: Es gibt eine Zahl $b \in \mathbb{N}$ und eine gewöhnlichen Stelle $p \in \tilde{Q}_2$, sodass

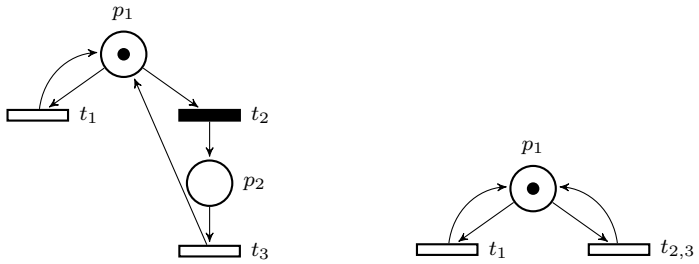
(a) ${t_1}^\bullet = \{p\}$,

(b) ${}^\bullet t_2 = \{p\}$, ${t_2}^\bullet \neq \emptyset$ und $I^-(p, t_2) = b$,



(a) Ein lebendiges QPN (b) Ein nicht-lebendiges QPN nach Fusion von Transitionen t_2 und t_3

Abbildung 4.10: Falsche Anwendung von QPN-Regel 2*



(a) Ein nicht-lebendiges QPN (b) Ein lebendiges QPN nach Fusion von Transitionen t_2 und t_3

Abbildung 4.11: Falsche Anwendung von QPN-Regel 2*

(c) es existiert eine Zahl $k_h \in \mathbb{N}$ mit $I^+(p, t_1) = k_h b$,

(d) ${}^\bullet p = \{t_1\}$ und $p^\bullet = \{t_2\}$,

(e) wenn $t_1 \in \tilde{W}_2$ und $t_2 \in \tilde{W}_1$, dann gilt $p_i{}^\bullet = \{t_1\}$ für alle $p_i \in {}^\bullet t_1$ und

(f) $m_0(p) \leq b$.

Anwendung:

1. Entferne Transitionen t_1 und t_2 und die Stelle p samt aller ein- und ausgehenden Kanten.
2. Füge eine neue Transition $t_{1,2}$ ein. Die Transition ist zeitbehaftet, wenn $t_1 \in \tilde{W}_1$ oder $t_2 \in \tilde{W}_1$ und sonst zeitlos.
3. Füge für jede Stelle $q \in {}^\bullet t_1$ eine Kante $(q, t_{1,2})$ mit $I^-(q, t_{1,2}) = I^-(q, t_1)$ ein.
4. Füge für jede Stelle $q \in t_1{}^\bullet \cup t_2{}^\bullet \setminus \{p\}$ eine Kante $(t_{1,2}, q)$ mit $I^+(q, t_{1,2}) = I^+(q, t_1) + I^+(q, t_2)$ ein.

Die nächste Regel kann problemlos für QPNs übernommen werden, da es für die Fusion von mehreren parallelen Stellen irrelevant ist, von welchem Typ ihre Vor- und Nachtransitionen sind. In der Markierung des Erreichbarkeitsgraphen fällt lediglich eine Stelle weg, ansonsten ändert sich der Graph nicht, da weiterhin die gleichen Transitionen aktiviert sind.

QPN-Regel 3 (Eliminierung von identischen Stellen). Betrachtet wird eine Menge von gewöhnlichen Stellen $Q = \{p_1, p_2, \ldots, p_n\} \subseteq \tilde{Q}_2$.
Voraussetzungen:

(a) Für alle Transitionen $t \in T$ gilt: $I^-(p_1, t) = I^-(p_2, t) = \ldots = I^-(p_n, t)$ und $I^+(p_1, t) = I^+(p_2, t) = \ldots = I^+(p_n, t)$.

Anwendung:

1. Bestimme Stelle $p = \underset{p_i \in Q}{\operatorname{argmin}}\, m_0(p_i)$
2. Entferne alle Stellen $p_i \in Q \setminus \{p\}$ samt aller ein- und ausgehenden Kanten.

Besteht die Menge der parallelen Transitionen aus zeitbehafteten und zeitlosen Transitionen, so sind alle zeitbehafteten Transitionen aufgrund der Priorität der zeitlosen Transitionen tot. Bei der Reduktion wird nur eine zeitlose Transition beibehalten. Aufgrund ihrer Priorität ändert sich nichts am Erreichbarkeitsgraphen. Es fallen lediglich die Kantenbeschriftungen der eliminierten Transitionen weg. Eigenschaften wie Beschränktheit und die Existenz von Heimatzuständen werden beibehalten. Lediglich die Nicht-Lebendigkeit kann nicht mehr erhalten werden. Dies ist für die

Analyse jedoch nicht relevant, da das Netz bereits als nicht lebendig klassifiziert werden kann. Besteht die Menge jedoch nur aus zeitbehafteten Transitionen, so wird bei der Reduktion nur eine der Transition beibehalten und die übrigen entfernt.

QPN-Regel 4 (Eliminierung von identischen Transitionen). Betrachtet wird eine Menge von Transitionen $F = \{t_1, t_2, \ldots, t_n\} \subseteq T$.
Voraussetzungen:

(a) Für alle Stellen $p \in P$ gilt: $I^-(p, t_1) = I^-(p, t_2) = \ldots = I^-(p, t_n)$ und $I^+(p, t_1) = I^+(p, t_2) = \ldots = I^+(p, t_n)$.

Anwendung:

1. Ist $F \cap \tilde{W}_2 \neq \emptyset$, so ist $t_i \in F \cap \tilde{W}_2$, sonst ist $t_i \in F$. Entferne alle Transitionen $t \in F \setminus \{t_i\}$ samt aller ein- und ausgehenden Kanten.

Hinweis: Ist $F \cap \tilde{W}_1 \neq \emptyset$ und $F \cap \tilde{W}_2 \neq \emptyset$, dann ist das Netz nicht lebendig.

Bei der nächsten Regel geht es um die Eliminierung von Self-Loop-Stellen. Die Stelle behält unabhängig davon, ob eine zeitlose oder eine zeitbehaftete Transition feuert, immer ihren Zustand bei. Daher kann Regel 10 ohne Differenzierung der Transitionstypen übernommen werden.

QPN-Regel 5 (Eliminierung von Self-Loop-Stellen). Betrachtet wird eine gewöhnliche Stelle $p \in \tilde{Q}_2$.
Voraussetzungen:

(a) ${}^\bullet p = p^\bullet$,

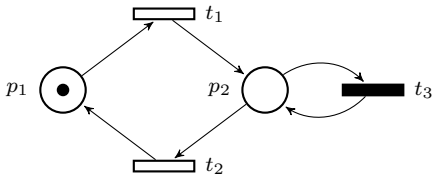
(b) für alle Transitionen $t \in T$ gilt: $I^+(p, t) = I^-(p, t)$ und

(c) $m_0 \geq \max_{t \in p^\bullet} I^-(p, t)$.

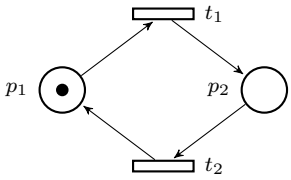
Anwendung:

1. Entferne die Stelle p samt aller ihrer ein- und ausgehenden Kante.

Für zeitbehaftete Self-Loop-Transitionen gilt wieder, dass sie im Falle, dass das Netz Nullzeit-Fallen hat, dafür sorgen, dass das Netz nicht lebendig ist. Durch Entfernen der Transition könnte es jedoch lebendig werden. Auch bei zeitlosen Self-Loop-Transitionen ist Vorsicht geboten, da sie eine Nullzeitfalle bilden können, wenn die Vor-/Nachstellen nur zeitbehaftete Nachtransitionen haben. Durch Entfernen der Self-Loop-Transition könnte die Nullzeit-Falle verschwinden und ein nicht-lebendiges Netz lebendig werden (siehe Abbildung 4.12).



(a) Ein nicht-lebendiges QPN



(b) Lebendiges QPN nach Eliminierung von Transition t_3

Abbildung 4.12: Falsche Anwendung von QPN-Regel 6

QPN-Regel 6 (Eliminierung von Self-Loop-Transitionen). Betrachtet wird eine zeitlose Transition $t \in \tilde{W}_2$.
Voraussetzungen:

(a) ${}^\bullet t = t^\bullet$ und $t^\bullet \in \tilde{Q}_2$

(b) für alle Stellen p gilt: $I^+(p,t) = I^-(p,t)$

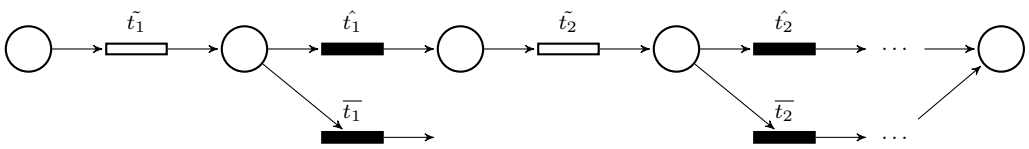
(c) für alle Stellen $p \in t^\bullet$ gilt: $p^\bullet \cap \tilde{W}_2 \setminus \{t\} \neq \emptyset$

Anwendung:

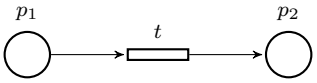
1. Entferne die Transition t samt aller ihrer ein- und ausgehenden Kante.

Da QPNs eine echte Obermenge von GSPNs sind [3], können die hier entwickelten Reduktionsregeln auch auf GSPNs angewendet werden.

Eine Queueing-Stelle, die Bedingung EQUAL-Service (siehe S. 43) erfüllt, kann für jede darin bediente Tokenfarbe durch das Teilnetz aus Abbildung 4.13(a) dargestellt werden [7]. Anhand des Erreichbarkeitsgraphen für das teilnetz kann gesehen werden, dass die letzte Stelle ein Heimatzustand ist. Das bedeutet, er wird unabhängig davon, welcher Pfad gewählt wird, immer erreicht, vorausgesetzt $\tilde{t_1}$ wird irgendwann aktivtiert. Dadurch lässt sich die Queuing-Stelle auf zwei Stellen, die über eine zeitbehaftete Transition miteinander verbunden sind, reduziert werden (siehe Abbildung 4.13). Dabei repräsentiert die Stelle p_1 die Tokens, die sich im Warteschlangensystem befinden, die Transition t die Bedienzeit und p_2 den Depository-Platz. Diese Ersetzung bewirkt zwar keine Änderung im Zustandsraum, kann aber dazu führen, dass bestimmte Reduktionsregeln anwendbar werden.



(a) GSPN-Darstellung einer Cox-Verteilung [7]



(b) reduzierte Darstellung einer Queuing-Stelle mit Cox-Verteilung

Abbildung 4.13: Substitut für eine Queuing-Stelle, die Bedingung EQUAL-Service erfüllt

Kapitel 5

Implementierung

Dieses Kapitel beschäftigt sich mit der Implementierung der qualitativen Analyse von Queueing-Petri-Netzen. Dabei wird das *Queueing Petri Net Modeling Environment* (QPME) [33], ein Softwarepaket zur Modellierung und Leistungsanalyse von Queueing-Petri-Netzen (siehe siehe Kapitel 3), als Basis verwendet und um die Funktion der qualitativen Analyse erweitert. Zunächst wird das QPME-Tool an sich und die grafische Oberfläche vorgestellt. Anschließend wird der Aufbau der Analysefunktionalität und der Ablauf der Analyse erläutert.

5.1 QPME und die grafische Oberfläche

QPME ist ein von der Descartes Research Group[10] entwickeltes Softwarepaket für den Umgang mit Queueing-Petri-Netzen und besteht aus zwei Komponenten. Die erste Komponente, QPE (*QPN Editor*), ist ein auf dem Eclipse/GEF-Framework[11] basierender grafischer Editor zur einfachen Modellierung von QPNs. Die zweite Komponente, SimQPN, ist hingegen ein effizienter Simulator für die Leistungsanalyse von QPNs.

Abbildung 5.1 zeigt das Hauptfenster von QPME und den darin eingebetteten QPN Editor. Die einzelnen Komponenten eines QPNs können in der Palette rechts ausgewählt und mit einem Klick in das Editorfenster einfach hinzugefügt werden. Es stehen dabei folgende QPN-Komponenten zur Verfügung:

- gewöhnliche Stellen,
- Queueing-Stellen,
- Subnetz-Stellen,
- zeitlose Transitionen,

[10] `http://www.descartes-research.net`
[11] `http://www.eclipse.org/gef/`

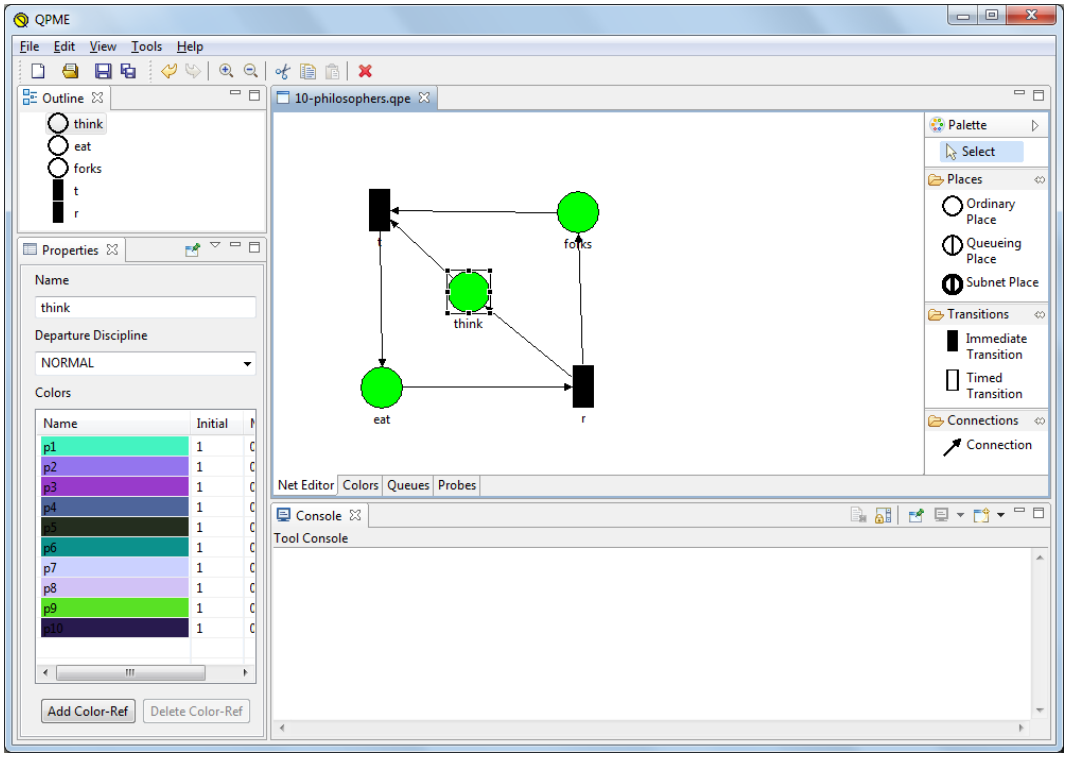


Abbildung 5.1: QPME-Hauptfenster

- zeitbehaftete Transitionen und
- Verbindungen zwischen den einzelnen Knoten.

Subnetz-Stellen sind dabei Stellen, die wiederum selbst ein komplettes QPN enthalten können und somit einen hierarchischen Aufbau von QPNs ermöglichen. QPNs mit Subnetz-Stellen werden hierarchische Queueing-Petri-Netze [2] genannt. In QPME werden die Tokenfarben und die Warteschlangennetze global für das gesamte Netz unter den Reitern *Colors* und *Queues* festgelegt. Für jedes Warteschlangensystem muss dazu die Zahl der Bedienstationen und die Scheduling-Strategie angegeben werden. Momentan werden von QPME die Scheduling-Strategien *FCFS*, *PS*, *IS*, *Priority-Scheduling* und *zufällig* unterstützt. Die globale Definition von Warteschlangensystemen ermöglicht es, ein Warteschlangensystem für mehrere Queueing-Stellen zu verwenden.

Durch Klick auf eine Komponente im *Net Editor* können im *Properties*-Fenster (links) die näheren Eigenschaften der Komponente definiert werden, dazu gehören bei Stellen beispielsweise die unterstütztemn Tokenfarben und deren initiale Markierung und bei den Transitionen die einzelnen Feuermodi. Für Queueing-Stellen lässt sich dort auch das enthaltene Warteschlangennetz auswählen. Durch Doppelklick auf eine Transition öffnet sich der Editor für die Inzidenzfunktion, wo durch einfaches

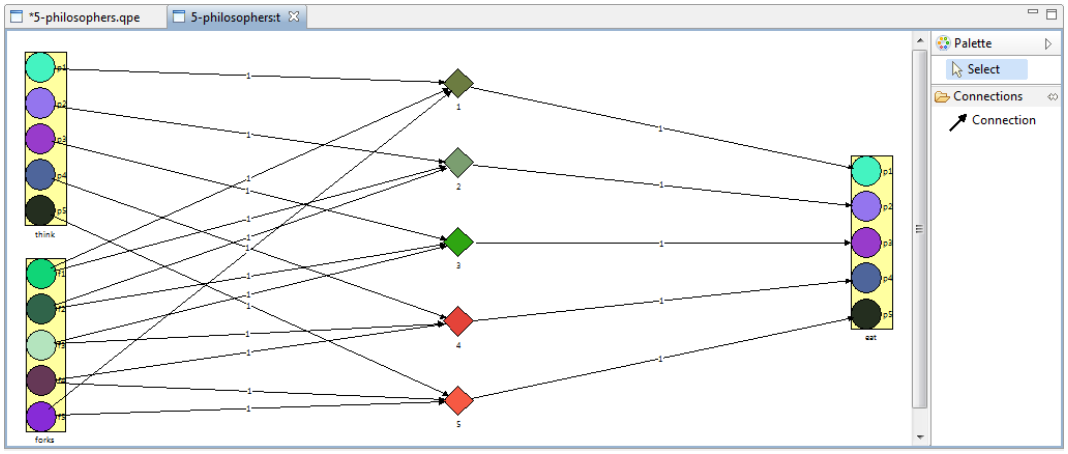


Abbildung 5.2: Inzidenzfunktions-Editor für die Transition t aus dem Philosophenproblem (siehe S. 5) mit fünf Philosophen

Verbinden von entfalteten Stellen[12] und den Feuermodi die Inzidenzfunktion definiert werden kann (siehe Abbildung 5.2). Für eine genauere Anleitung zu QPME mit Auflistung und Erklärung aller Attribute der einzelnen Komponenten sei der Leser auf [32] verwiesen.

5.2 Werkzeuge und Bibliotheken

In dieser Arbeit dient QPME als Ausgangspunkt und wird um die Möglichkeit der qualitativen Analyse der darin entwickelten Modelle erweitert. Als Entwicklungsumgebung kommt daher *Eclipse Helios*[13] zum Einsatz.

Da QPME über normale Queueing-Petri-Netze hinausgehende Features bietet, wurden für die qualitative Analyse folgende Einschränkungen gemacht:

- In QPMEs wird bei Stellen zwischen zwei sogenannten *Departure Disciplines* zu unterscheiden: NORMAL und FIFO. Dabei meint FIFO, dass Tokens die Stelle in der Reihenfolge wieder verlassen, wie sie angekommen sind. Gewöhnliche Stellen mit der FIFO-Departure-Discipline entsprechen dann den zeitlosen Queueing-Stellen aus der QPN-Definition in [5]. Der Einfachheit halber wurde sich in dieser Arbeit und der Implementierung nur auf QPNs ohne zeitlose Queuing-Stellen beschränkt. Enthält das zu untersuchende Netz eine Stelle mit FIFO-Departure Discipline, so wird sie bei der qualitativen Analyse wie eine gewöhnliche Stelle behandelt.

[12] also eine Stellen-Tokenfarbe-Kombination
[13] `http://www.eclipse.org`

- QPME bietet für Stellen die Möglichkeit, Kapazitätsgrenzen für die Zahl der Tokens einer Farbe festzulegen. Da die Reduktionsregeln jedoch nur für Netze ohne Kapazitätsbeschränkungen definiert sind, werden sämtliche Kapazitätsgrenzen bei der qualitativen Analyse ignoriert.
- Da der Umgang mit ω-Markierungen bei Prioritätswarteschlangen Probleme bereitet, werden Netze die Warteschlangensysteme mit prioritätsbasiertem Scheduling enthalten, nicht von der qualitativen Analyse unterstützt.
- Für den Fall, dass Transitionen in einem Feuermodus Tokens verschiedener Farben auf die selbe Queueing-Stelle mit integriertem FCFS-Warteschlangensystem feuern, ist nicht klar definiert, welches dieser Tokens als erstes eingereiht wird. Daher werden Netze, in denen solche Situationen auftreten können, nicht von der qualitativen Analyse unterstützt.
- Wie bereits erwähnt, können mit QPME hierarchische QPNs modelliert werden. Die qualitative Analyse für HQPNs wird derzeit noch noch nicht unterstützt, sodass beim Start der Analyse von Netzen mit Subnetzen ein Hinweis ausgegeben wird.

Die Graphen, also das QPN und der Erreichbarkeitsgraphen, werden mithilfe des *Java Universal Network/Graph Framework* (JUNG)[14] modelliert und im Falle des Erreichbarkeitsgraphen auch visualisiert. Bei der Modellierung bringt das den Vorteil, dass keine graphtypischen Methoden, wie beispielsweise das Hinzufügen von Kanten oder Knoten oder das Verwalten von Adjazenzlisten, noch einmal selbst implementiert werden müssen. Für die Visualisierung enthält JUNG Algorithmen, die die automatische Anordnung von Knoten und Kanten übernehmen. Dies ist im Falle des Erreichbarkeitsgraphen wichtig, da im Voraus nicht bekannt ist, wie der Graph aussehen wird.

5.3 Aufbau und Ablauf der funktionalen Analyse

Die qualitative Analyse lässt sich über den Menüpunkt *Tools→Analysis* starten. Dabei öffnet sich zunächst ein Fenster (siehe Abbildung 5.3), in dem folgende Einstellungen vorgenommen werden können:

- *Create a file for unfolded net*: Das Netz, das nach Entfaltung des QPNs entsteht, in eine Datei speichern.
- *Use reduction techniques*: Netz vor der Konstruktion des Erreichbarkeitsgraphen reduzieren.

[14] `http://jung.sourceforge.net`

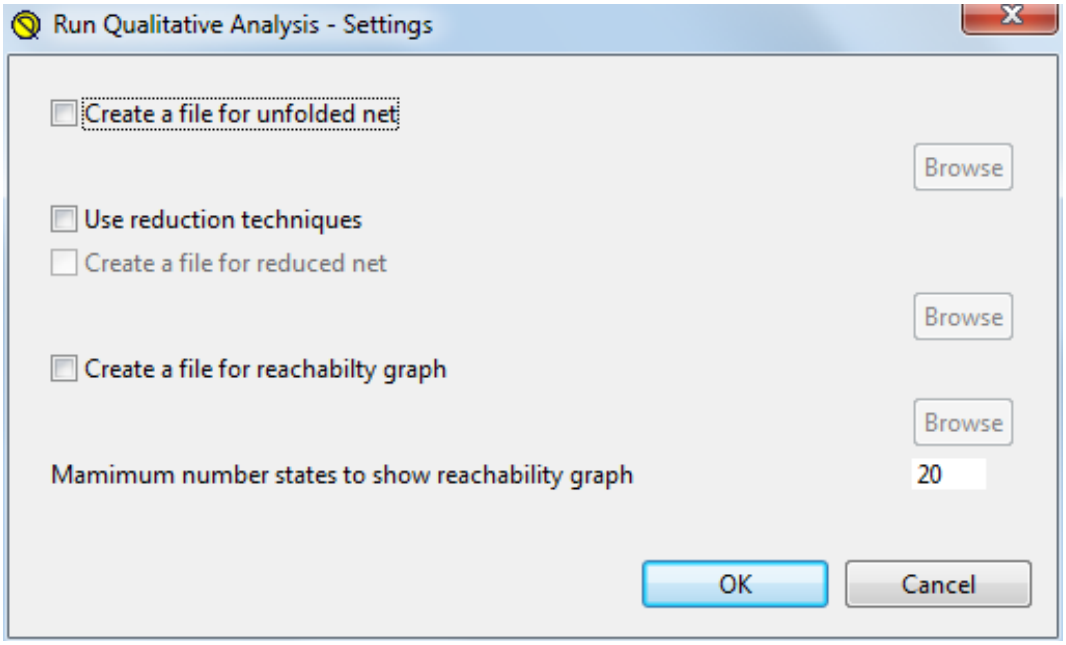


Abbildung 5.3: Einstellungs-Fenster für die qualitative Analyse

- *Create a file for reduced net*: Das reduzierte Netz in eine Datei speichern.
- *Create a file for reachability graph*: Den Erreichbarkeitsgraphen in eine Datei speichern.
- *Maximum number states to show reachability graph*: Maximale Anzahl der Knoten in einem Erreichbarkeitsgraphen, für die der Erreichbarkeitsgraph noch graphisch angezeigt werden soll. Erfahrungswerte zeigen, dass der Erreichbarkeitsgraph schon ab 20 Knoten sehr unübersichtlich werden kann.

Die Graphen können jeweils als .csv-Datei gespeichert werden. Diese Dateien enthalten alle Stellen und Transitionen bzw. Markierungen und die Inzidenzfunktionen als Tabellenform. Während für den Erreichbarkeitsgraphen eine grafische Darstellung verfügbar ist, wurde für das entfaltete und reduzierte Petri-Netz darauf verzichtet. Dies hat den Grund, dass im Voraus nicht bekannt ist, wie die Graphen aussehen werden, und der Net Editor im QPME keine Möglichkeit bietet, die Knoten automatisch anzuordnen. Um die Konsistenz in der grafischen Darstellung zu wahren, wurde auf den Einsatz von JUNG für die Visualisierung des reduzierten Netzes verzichtet.

Abbildung 5.4 zeigt ein Aktivitätsdiagramm, das den Ablauf der qualitativen Analyse zusammenfasst. Zunächst wird das QPN-Modell aus dem zugrundeliegenden XML-Dokument extrahiert. Anschließend wird, falls der Benutzer dies gewünscht hat, das Netz mit den Regeln aus Kapitel 4.2 reduziert. Danach wird die Struktur des Netzes analysiert und das Netz einer oder mehreren Netzklassen zugeordnet.

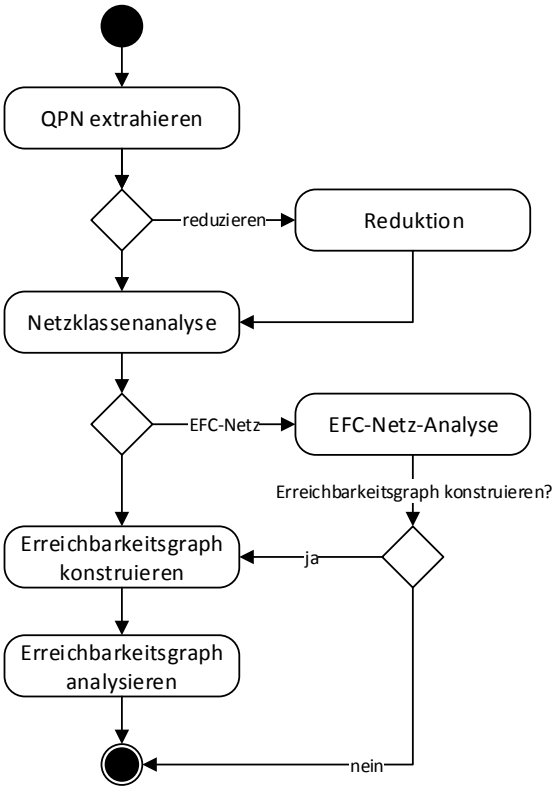


Abbildung 5.4: Ablauf der Analyse

Falls das Netz ein EFC-Netz ist, so wird es gesondert mit speziellen Methoden für EFC-Netze untersucht. Der Benutzer erhält das Ergebnis der Analyse und kann dann entscheiden, ob ihm das Ergebnis genügt oder noch zusätzlich der Erreichbarkeitsgraph konstruiert und analysiert werden soll. Falls das Netz kein EFC-Netz ist, so wird direkt mit der Konstruktion des Erreichbarkeitsgraphen fortgefahren. Nachdem der Erreichbarkeitsgraph konstruiert wurde, kann er analysiert werden, um so festzustellen, ob das Netz beschränkt und lebendig ist und Heimatzustände besitzt. Im Folgenden werden die einzelnen Stationen des Analyseprozesses näher erläutert.

5.3.1 Extraktion des QPNs

QPME nutzt für die Verwaltung des QPNs keine eigenen Klassen, sondern basiert auf einer *Document Object Model*-Implementierung (DOM)[15]. Das Netzmodell wird also als *Document* im XML-Format gehandhabt, bei dem auf einzelne Elemente per

[15] `http://www.w3.org/DOM/`

XPath[16] zugegriffen werden kann. Da im QPME lediglich auf einzelne Elemente zugegriffen wird und nur einzelne feste Elemente oder Attribute verändert, gelöscht oder hinzugefügt werden, mag dies in dem Fall sinnvoll erscheinen. Für die Analyse von QPNs, bei denen schnell über ganze Mengen von Elementen iteriert werden soll und auf denen komplexe Operationen durchgeführt werden müssen, ist das ineffizient und unpraktisch. Daher wird zu Beginn der Analyse das QPN aus dem Dokument extrahiert und die einzelnen Bestandteile als Java-Objekte angelegt. Da QPNs ein zugrundeliegendes gefärbtes Petri-Netz haben, wird das Netz schon bei der Extraktion entfaltet. Die Entfaltung erfolgt analog zu dem in Kapitel 2.4.1 beschriebenen Verfahren. Dabei werden zeitlose Transition zu zeitlosen und zeitbehaftete wieder zu zeitbehafteten Transitionen entfaltet. Queueing-Stellen werden wie gewöhnliche Stellen gemäß der Farben in ihrem Depository entfaltet, behalten aber alle das gleiche Warteschlangensystem.

Abbildung 5.5 zeigt das Klassendiagramm zu den Klassen, die für die Repräsentation des QPNs verwendet werden. Die Klasse `QPN` erbt von der JUNG-Klasse `DirectedSparseGraph` und erhält somit alle notwendigen Methoden für das Verwalten von Knoten und Kanten. Für diese Klasse muss der Knoten- und Kantentyp angegeben werden. Knoten können in einem QPN sowohl Stellen als auch Transitionen sein und werden daher durch die abstrakte Klasse `QPNVertex` repräsentiert. Die Klasse `QPNEdge` repräsentiert hingegen die Bögen im QPN und hat als Parameter das Bogengewicht. Der Zustand eines Warteschlangensystems wird durch die abstrakte Klasse `StateQueue` repräsentiert. Diese Klasse wird je nach Scheduling Strategie durch die Klassen `StateQFCFS` für FCFS-Warteschlangensysteme oder `StateQOther` für alle übrigen Warteschlangensysteme realistiert. Der Zustand für ein FCFS-Warteschlangensystem besteht dabei aus einer verketteten Liste für die Tokenfarben. Das Token, das als nächstes bedient wird, steht dabei ganz vorne in der Liste. Für die anderen Scheduling Strategien besteht der Zustand aus einer *Map*, die jeder Tokenfarbe die Zahl der Tokens innerhalb des Warteschlangensystems zuordnet.

5.3.2 Reduktion

Für die Reduktion des QPNs ist die Klasse `ReductionController` zuständig. Dabei werden der Reihe nach die Regeln aus Kapitel 4.2 angewendet. Es erschien sinnvoll, zunächst mit der Eliminierung von Self-Loop-Stellen und Self-Loop-Transitionen zu beginnen, da dadurch weitere Regeln anwendbar gemacht werden könnten. Anschließend werden die Regeln zur Fusion von identischen Stellen und Transitionen genutzt. Dadurch entstehen in der Regel seriell fusionierbare Stellen und Transitionen, die

[16] `http://www.w3.org/TR/xpath-30/`

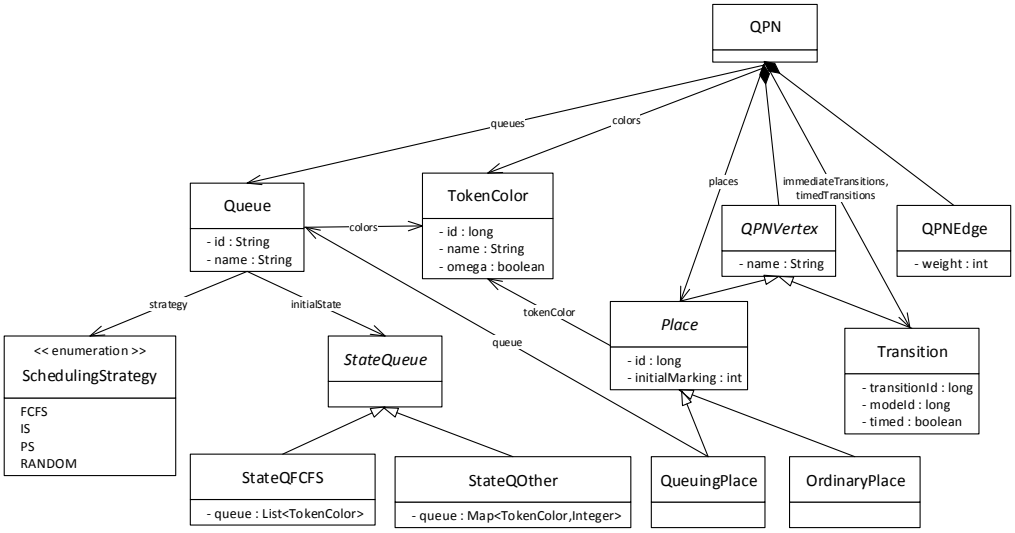


Abbildung 5.5: Klassendiagramm zur Modellierung des QPNs

im nächsten Schritt der Reduktion durch Anwendung der Regeln QPN-1, QPN-2 und QPN-2* zusammengeführt werden könnten. Die Überprüfung und Anwendung der Regeln erfolgt in polynomieller Zeit. Die Komplexität für Regel QPN-1 ist im Worst-Case $\mathcal{O}(|P| \cdot |E|)$, für QPN-2 und QPN-2* beträgt sie $\mathcal{O}(|T| \cdot |E|)$. Dabei beschreibt $|T|$ die Zahl der Transitionen, $|P|$ die Zahl der Stellen und $|E|$ die Zahl der Bögen/Kanten im QPN. Regeln QPN-3 und QPN-4 haben hingegen eine Worst-Case-Laufzeit von $\mathcal{O}(|P|^2 \cdot |E|)$ bzw. $\mathcal{O}(|T|^2 \cdot |E|)$, da jede Stelle (resp. Transition) mit jeder anderen Stelle (resp. Transition) verglichen werden muss, um identische Stellen (resp. Transitionen) zu finden. Im Schnitt kann die Komplexität jedoch jeweils um den Faktor $|E|$ gekürzt werden, da nicht alle Stellen oder Transitionen mit allen Bögen verbunden sein können. Regeln QPN-5 und QPN-6 können mit der Komplexität $\mathcal{O}(|P| \cdot |T|)$ angewendet werden. Auch hier kann im Schnitt jeweils einer der Faktoren wegfallen, da selten alle Stellen alle Transitionen als Vortransitionen haben und umgekehrt. Die Listings zu den einzelnen Regeln können im Anhang gefunden werden.

5.3.3 Netzklassenbestimmung und Analyse des EFC-Netzes

Für QPNs mit EFC-Netz-Struktur kann unter bestimmten Bedingungen Eigenschaften des zugrundeliegenden zeitlosen Netzes auf Eigenschaften des QPNs geschlossen werden (siehe Kapitel 3). Daher macht es Sinn vor der Konstruktion des Erreichbarkeitsgraphen zu überprüfen, ob das Netz eine EFC-Netz-Struktur besitzt. Die Ansiedlung der Reduktion vor der Netzklassenbestimmung hatte den Zweck, dass

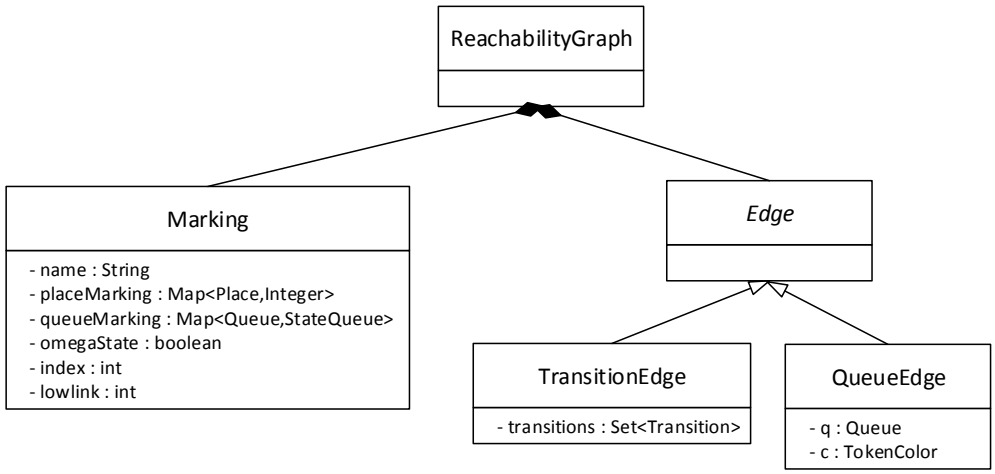


Abbildung 5.6: Klassendiagramm zur Modellierung des Erreichbarkeitsgraphen

Netze, die vorher keine EFC-Netz-Struktur hatten, diese eventuell nach der Reduktion erhalten. Auch die mögliche Zugehörigkeit zu anderen Netzklassen wird in diesem Schritt ermittelt. Die Bestimmung der Netzklassen erfolgt ganz einfach durch sequentielles Überprüfen der Definitionen aus Kapitel 2.3. Falls das Netz eine EFC-Struktur besitzt, wird das zugrundeliegende Netz erst reduziert und anschließend mittels Erreichbarkeitsgraph analysiert.

5.3.4 Konstruktion und Analyse des Erreichbarkeitsgraphen

Abbildung 5.6 zeigt ein Klassendiagramm, dass die Klassen zur Repräsentation des Erreichbarkeitsgraphen veranschaulicht. Die Klasse `ReachabilityGraph` erbt von der JUNG-Klasse `DirectedSparseGraph`, um ebenfalls alle für die Arbeit auf Graphen benötigten Methoden zu erhalten. Die Knoten des Graphen sind Markierungen, repräsentiert durch die Klasse `Marking`. Beim Erreichbarkeitsgraphen für QPNs gibt es zwei Arten von Kanten. Zum einen Kanten, die das Feuern einer Transition darstellen, vertreten durch die Klasse `TransitionEdge`, und zum anderen Kanten, die das Bedienende in einem Warteschlangensystem anzeigen, repräsentiert durch die Klasse `QueueEdge`.

Die Konstruktion des Erreichbarkeitsgraphen erfolgt nach Algorithmus 3.1. Dabei gab es drei Stellen, die noch genauer festgelegt werden mussten. Zum einen müssen die Tokens, die potenziell ins Depository übergehen dürfen, mit der Methode `getNext()` bestimmt werden. Für die Scheduling-Strategien IS, PS und RANDOM können das alle Tokens sein, die aktuell im Warteschlangensystem vorhanden sind. Bei FCFS ist es normalerweise immer das Token, das als erstes in der Liste steht. Da

```
1 public void changeToOmegaState(StateQueue q) {
2   if (!queue.isEmpty()){
3     TokenColor c = queue.removeLast();
4     queue.removeLast();
5     TokenColor cOmega = c.clone();
6     cOmega.setOmega(true);
7     queue.add(cOmega);
8   }
9 }
```

Listing 5.1: Die Methode `changeToOmegaState` für die Klasse `StateQFCFS`

ω zwar eine beliebige Zahl, aber niemals tatsächlich unendlich sein kann, kann das Token, das als zweites in der Liste steht, ebenfalls irgendwann bedient werden. Ist das erste Token also ein ω-Token, so wird auch das zweite Token von der Methode `getNext` zurückgegeben.

Die zweite Überlegung, die getroffen werden muss, ist die Berechnung des ω-Zustandes für die Warteschlangennetze. Ein Warteschlangenzustand, der durch einen anderen Warteschlangenzustand überdeckt wird, wird durch den Aufruf der Methode `changeToOmegaState(StateQueue q)` in einen ω-Zustand überführt. Dabei wird der überdeckte Warteschlangenzustand übergeben. Die Erzeugung des ω-Zustandes ist abhängig von der Scheduling-Strategie. Listing 5.1 zeigt die Methode für FCFS-Warteschlangensysteme. Dabei wird das letzte Token in der Liste kopiert und als ω-Token markiert, was bedeutet, dass das Token beliebig oft hintereinander auftauchen kann. Anschließend werden die letzten beiden Tokens aus der Liste entfernt und das neue Token an letzter Stelle hinzugefügt, so wird beispielsweise aus einem $a-b-b-$ ein $a-b_\omega-$. Der Zustand von Warteschlangenystemen mit der Bedienstrategie PS, IS oder RANDOM wird durch eine Map repräsentiert, die jeder Tokenfarbe die Anzahl der in der Warteschlange vorhandenen Tokens zuordnet. Der ω-Zustand für solche Warteschlangensysteme wird berechnet, indem für jede Tokenfarbe die Zahl der Tokens mit der Zahl der Tokens in dem übergebenen überdeckten Zustand verglichen wird. Ist die Zahl der Tokens größer, so wird als Anzahl für die Tokens ω bzw. in der Implementierung `Integer.MAX_VALUE` eingetragen. Dies kann in seltenen Fällen dazu führen, dass ein Netz als unbeschränkt klassifiziert wird, wenn seine Schranke größer als `Integer.MAX_VALUE` ist.

Als Letztes muss noch festgelegt werden, wann ein Warteschlangensystemzustand einen anderen Zustand überdeckt. Dazu wird eine Methode `bigger(StateQOther` in den Klassen `StateQFCFS` und `StateQOther` implementiert. Im Falle von `StateQother` reicht es zu überprüfen, ob die Zahl der Tokens für jede Farbe größer oder gleich und für mindestens eine Farbe größer der Zahl der Tokens in dem übergebenen Zustand ist. Falls ja, so überdeckt der Zustand den übergebenen Zustand, andernfalls nicht. Listing 5.2 zeigt die Methode `bigger` für die Klasse `StateQFCFS`. Dabei wird

```
1  public boolean bigger(StateQueue queueMarking) {
2    if (this.equals(queueMarking)) return false;
3    if (queueMarking instanceof StateQFCFS){
4      StateQFCFS sq = (StateQFCFS) queueMarking;
5      if (sq.getQueue().isEmpty()) return false;
6      if (queue.size() <= sq.getQueue().size()) return false;
7      for (int i=0; i<sq.getQueue().size();i++){
8        if (!queue.get(i).equals(sq.getQueue().get(i))) return false;
9      }
10     return (queue.getLast().equals(sq.getQueue().getLast()));
11   }
12   else return false;
13 }
```

Listing 5.2: Die Methode `bigger` für die Klasse `StateQFCFS`

zunächst überprüft, ob die Liste länger ist, als die des übergebenen Zustandes, andernfalls wird `false` zurückgegeben. Anschließend wird geschaut, ob die Liste des übergebenen Zustandes mit den ersten $n-1$, wobei n die Zahl der Elemente in der eigenen Liste ist, Elementen der eigenen Liste übereinstimmt. Damit der betrachtete Zustand tatsächlich größer als der übergebene Zustand ist, muss das letzte Element der eigenen Liste mit dem letzten Element der Liste des übergebenen Zustands übereinstimmen.

Die Analyse des Erreichbarkeitsgraphen erfolgt durch die Bestimmung der starken Zusammenhangskomponenten. Für die Berechnung der starken Zusammenhangskomponenten wird der Algorithmus von Tarjan [44] genutzt. Die SZK werden dabei von der Klasse `SCC` repräsentiert. Diese Klasse enthält neben allen Knoten, die zu der SZK gehören, auch alle zugehörigen Kanten. Ferner ist immer angegeben, ob die SZK terminal ist oder nicht. Dadurch ist es sehr einfach zu überprüfen, ob das Netz lebendig ist, indem über alle SCCs iteriert wird. Bei terminalen SCCs werden alle Transitionskanten durchlaufen und die zugehörigen Transitionen in einer Menge `transitions` gesammelt. Anschließend wird überprüft, ob diese Menge alle Transitionen des QPNs enthält. Falls ja, so ist das Netz lebendig, andernfalls nicht. Für die Überpüfung der Existenz von Heimatzuständen werden einfach die terminalen SZKs gezählt. Wenn die Zahl größer als eins ist, so gibt es keine Heimatzustände. Für die Überprüfung der Beschränktheit wurden bereits bei der Konstruktion des Erreichbarkeitsgraphen Markierungen, die ω-Zustände enthalten, gekennzeichnet. Es muss lediglich geprüft werden, ob eine mit ω gekennnzeichnete Markierung existiert. Das Ergebnis der Analyse wird in der Konsole ausgegeben, Listing 5.3 zeigt ein Beispiel dafür.

```
1  --------------------------------
2  Start qualitative analysis...
3  --------------------------------
4  Build QPN Model... 18ms
5
6  Number of places: 15
7  Number of transitions: 10
8  --------------------------------
9  Analyse netclass... 2ms
10
11 Ordinary net: true
12 State machine: false
13 Marked Graph: false
14 Free choice net: false
15 Extended free choice net: false
16 Simple net: false
17 Extended simple net: false
18 EQUAL-Conflict: true
19 EQUAL-Service: true
20 --------------------------------
21 Reduce net... 2ms
22
23 Number of places: 5
24 Number of transitions: 5
25 --------------------------------
26 Build reachability graph... 12ms
27
28 Number of states: 101
29 Number of arcs: 175
30 --------------------------------
31 Analyse reachability graph... 0ms
32
33 Bounded: true
34 Live: true
35 Has home states: true
36 Home states: m52 m14 m13 m48 m30 m98 m74 m37 m4 m42 m62 m20 m83 m38 m28 m89 m3 m87 m17 m5 m78
       m49 m25 m88 m35 m56 m10 m57 m43 m0 m81 m68 m97 m34 m16 m19 m64 m2 m15 m29 m100 m1 m96
     m36 m7 m84 m95 m91 m8 m80 m94 m86 m6 m41 m75 m58 m92 m51 m12 m82 m60 m65 m55 m54 m33 m71
     m69 m9 m93 m50 m26 m23 m70 m85 m47 m27 m21 m67 m39 m63 m53 m99 m11 m40 m46 m66 m77 m76
     m59 m72 m32 m22 m24 m61 m18 m79 m73 m45 m90 m31 m44
```

Listing 5.3: Beispiel für die Ergebnisausgabe einer qualitativen Analyse

Kapitel 6

Auswertung

Dieses Kapitel behandelt an Beispielen, inwieweit sich der Aufwand für die Konstruktion und Analyse des Erreichbarkeitsgraphen mithilfe des in dieser Arbeit entwickelten und in QPME implementierten Reduktionsverfahrens reduzieren lässt. Dazu wird zunächst die Testumgebung beschrieben und anschließend die Ergebnisse vorgestellt.

6.1 Testumgebung

Die Aufwandsreduzierung wird gemessen, indem die Zeit für die Konstruktion und Analyse des Erreichbarkeitsgraphen jeweils einmal mit und einmal ohne Verwendung der Reduktion gemessen wird. Die Zeitmessung erfolgt durch Verwendung der Methode `System.nanoTime()` (siehe Listing 6.1). Die Methode ist zwar relativ ungenau, da z. B. der Garbage-Collector die Messzeit beeinflussen kann, reicht für die Arbeit jedoch völlig aus, da eher das Verhältnis der Zeiten relevant ist und nicht die absoluten Werte.

Ausgeführt wurde QPME auf einem handelüblichen Laptop mit einer Intel Core i17-3612QPM CPU, 2,1 GHz, 8 GB RAM und Windows 7. Um Speicherüberläufen vorzubeugen wurden der *Java Virtual Machine* folgende Parameter übergeben:

```
-Xms1024m -Xmx1024m -Xss8m
```

Getestet wurden je zwei Beispiele für S/T-Netze, GSPNs und QPNs:

- `philosophers(20)`: Das Philosophenproblem aus Beispiel 2.1.4 für 20 Philosophen.
- `production cell(5)`: Produktionszelle, in der fünf Metallplatten gleichzeitig verarbeitet werden [27].
- `fms cell(15)`: Ein flexibles Fertigungssystem (*flexible manufacturing system*) mit 15 unbeladenen Paletten [15].

```
1 long before = System.nanoTime();
2 // Zu messende Methode
3 long after = System.nanoTime();
4 long runningTimeMs = (after-before)/1000000;
```

Listing 6.1: Messung der Ausführungsdauer

- `gLite(10,10,10,3)`: Angepasste Version von [14] einer Grid-Site (am Beispiel von gLite[17]) mit 10 einzelnen Jobs, 10 MPI-Jobs und 10 Worker Nodes. Die MPI-Jobs benötigen drei CPUs.
- `ispass03(15,15,15,50)`: Modell der *SPECjAppServer2001*[18] *OrderEntry Application* mit 15 WLS-Threads, 15 JDBC-Verbindungen zum Datenbanksystem, 15 Oracle-Prozessen und 50 Clients [31].
- `specJ2004(1,0,0,0)`: SPECjAppServer2004 mit einem Browse-Client [30].

Die grafischen Darstellungen der QPME-Modelle zu den Beispielen lassen sich im Anhang finden.

6.2 Ergebnisse

Tabelle 6.1 listet die Ergebnisse für die einzelnen Beispiele auf. Dabei wird zuerst das Resultat der Analyse des normalen Netzes aufgeführt und darunter die des reduzierten Netzes. $|P|$, $|T|$ und $|Q|$ kennzeichnen dabei die Anzahl der Stellen, Transitionen und Warteschlangen im Netz, $|S|$ und $|A|$ stehen hingegen für die Anzahl der Markierungen bzw. Anzahl der Kanten im Erreichbarkeitsgraphen. t_{red}, t_{RG}, t_A und t_{ges} benennen jeweils die Ausführungsdauern für die Reduktion, die Konstruktion des Erreichbarkeitsgraphen, die Analyse des Erreichbarkeitsgraphen und die Gesamtdauer der Analyse.

Die Ergebnisse zeigen, dass bei allen Beispielen mit Ausnahme von `fms cell` eine deutliche Reduktion um mindestens den Faktor 200 in der Ausführungsdauer und den Faktor 17 in der Zahl der Zustände erreicht werden konnte. Insbesondere ließen sich die beiden Beispiele `philosophers` und `isPass03` komplett reduzieren. Beispiel `gLite` verdeutlicht auf anschauliche Weise die Umkehrung der Zustandsraumexplosion. Obwohl die Zahl der Stellen und Transitionen nicht mal halbiert wurde, ist die Zahl der Zustände um den Faktor 17 kleiner geworden. Erfreulich ist auch, dass die Dauer der Reduktion immer im Bereich weniger Millisekunden liegt.

[17] `http://glite.cern.ch`
[18] `http://www.spec.org`

Fall	|P|	|T|	|Q|	|S|	|A|	t_{red}	t_{RG}	t_A	t_{ges}
philosophers(20)	60	40	0	15127	167240		6,6 min	10 s	6,8 min
	1	1	0	1	1	4 ms	1 ms	2 ms	7 ms
philosophers(25)	75	50	0	–[a]	–[a]		–[a]	–[a]	–[a]
	1	1	0	1	1	3 ms	<1 ms	<1 ms	4-5 ms
production cell(5)	59	44	0	25352	98172		10,3 min	4,2 s	10,4 min
	28	14	0	802	2616	6 ms	282 ms	16 ms	304 ms
fms cell(15)	16	25	0	47514	105886	<1 ms[b]	23,5 min	9,3 s	23,7 min
gLite(10,10,10,3)	17	18	0	64498	261796		45,7 min	387 ms	45,7 min
	11	12	0	3680	9402	<1 ms	3,1 ms	10 ms	3,2 s
isPass03(15,15,15,50)	8	5	4	30487	45215		6,7 min	1,8 s	6,8 min[c]
	1	1	0	1	1	<1 ms	<1 ms	<1 ms	1-2 s
specJ2004(1,0,0,0)	66	101	13	6359	9016		1,7 min	16 ms	1,7 min
	64	99	1	15	22	26 ms	6 ms	2 ms	34 ms

Tabelle 6.1: Evaluationsergebnisse ([a]da nach zwei Stunden immer noch kein Ergebnis vorlag, wurde die Berechnung abgebrochen [b]Anwendung der Reduktionsregeln führte nicht zur Reduktion des Netzes [c]Analyse des zugrundeliegenden Netzes möglich (Dauer <1 ms)

Natürlich ist das Ergebnis immer sehr stark abhängig von der Struktur des Netzes. Es fällt auf, dass die Reduktionsregeln bei GSPNs weniger Erfolg haben als bei den anderen Netztypen. Dies liegt vermutlich daran, dass Regel QPN-2* sehr stark eingeschränkt ist und dadurch bei Netzen, in denen sich zeitlose und zeitbehaftete Transitionen häufig abwechseln, selten Anwendung findet. Dieses Problem lässt sich genauso auf QPNs übertragen, obwohl die QPNs mehr Erfolg zu haben scheinen. Dies liegt daran, dass die hier betrachteten Netze viele Warteschlangensysteme enthalten, die die Bedingung EQUAL-Service (siehe Seite 43) erfüllen und von einer zeitlosen Transition gefolgt werden. Durch Substitution der Queueing-Stelle und anschließender Anwendung von Regel QPN-2 kann das Netz um eine Warteschlange reduziert werden, ohne die Menge der Stellen und Transitionen zu erhöhen.

Zusammengefasst kann gesagt werden, dass die Reduktion bei den hier untersuchten Beispielen einen großen Nutzen hatte und den Analyseaufwand in den meisten Fällen stark reduziert hat. Da der Aufwand für die Reduktion verschwindend gering ist, sollte diese bei jeder Analyse angewendet werden. Es sollte angemerkt werden, dass die hier implementierte Konstruktion des Erreichbarkeitsgraphen noch weit vom Optimum entfernt liegt und lediglich zur Untersuchung der Wirksamkeit der Reduktionsregeln dienen sollte. Es gibt weitaus effizientere Tools zur Zustandsraumanalyse [22; 23].

Kapitel 7

Zusammenfassung und Ausblick

Zum Abschluss werden an dieser Stelle nochmal die wichtigsten Punkte der Arbeit zusammengefasst und Erweiterungsmöglichkeiten betrachtet.

7.1 Zusammenfassung

Diese Arbeit beschäftigte sich mit der funktionalen Analyse von Queueing-Petri-Netzen durch Reduktion der Netzstruktur. Queueing-Petri-Netze sind eine Kombination aus zeitbehafteten Petri-Netzen und Warteschlangennetzen. Damit können nicht nur Synchronisationsvorgänge, sondern auch Wartesituation modelliert werden. Dadurch wird es ermöglicht mit nur einem Modell sowohl funktionale Eigenschaften, wie z. B. Lebendigkeit, als auch quantitative Eigenschaften, wie z. B. Antwortzeiten, zu untersuchen.

Dazu wurden zunächst Petri-Netze und einige der wichtigsten Eigenschaften dieser vorgestellt. Zu diesen Eigenschaften zählen Beschränktheit, Lebendigkeit und die Existenz von Heimatzuständen, die insbesondere eine notwendige Bedingung für die quantitative Analyse von QPNs sind. Da diese Eigenschaften auf dem Zustandsraum definiert sind, ist das gängigste Mittel zu ihrer Analyse die Konstruktion und Untersuchung des Erreichbarkeits- bzw. Überdeckungsgraphen. Für größere Netze kann der Zustandsraum bzw. Erreichbarkeitsgraph so groß werden, dass er aufgrund von Zeit- oder Speicherschranken nicht mehr untersucht werden kann. Eine Möglichkeit, mit diesem Problem umzugehen, ist die Reduktion der Netzstruktur. Dabei werden sogenannte Reduktionsregeln angewendet, bei denen bestimmte Eigenschaften beibehalten werden, um das Netz zu verkleinern. Davon wird erhofft, dass der Zustandsraum des reduzierten Netzes um einiges kleiner ist als der des ursprünglichen Netzes. In dieser Arbeit wurden einige Reduktionsregeln betrachtet, die vor Allem Beschränktheit, Lebendigkeit und die Existenz von Heimatzuständen bei der Reduktion beibehalten. Diese Regeln sind auf S/T-Netzen definiert. Durch die Zeitaspekte

in QPNs können die meisten Regeln jedoch nicht einfach so für QPNs übernommen werden. Daher wurde untersucht, inwieweit die Regeln für die Verwendung in QPNs angepasst werden müssen. Durch die Priorität von zeitlosen gegenüber zeitbehafteten Transitionen musste besonders bei Regeln, bei denen Transitionen bei der Reduktion entfernt wurden, aufgepasst werden, da ansonsten Eigenschaften wie z. B. Lebendigkeit nicht mehr beibehalten werden könnten. Außerdem wurde gezeigt, dass sich Queueing-Stellen, deren integrierte Warteschlangensysteme die Bedingung EQUAL-Service erfüllen, durch ein Subnetz aus zwei Stellen und einer zeitbehafteten Transition ersetzt werden können. Dadurch werden in der Regel andere Reduktionsregeln anwendbar.

Anschließend wurde die funktionale Analyse von Queueing-Petri-Netzen zusammen mit den angepassten Reduktionsregeln in das Open-Source-Tool QPME integriert und einige Beispiele analysiert. Dabei zeigte sich, dass in den meisten Fällen die Zeit zur Berechnung des Erreichbarkeitsgraphen erheblich reduziert werden konnte. Während für die Analyse ohne Reduktion mehrere Minuten gebraucht wurden, lag die Zeit für die Analyse von reduzierten Netzen oft nur im Sekunden- bis Millisekunden-Bereich. Auch die Reduktion selbst dauerte nur wenige Millisekunden. Dies zeigt, dass die Reduktion für die funktionale Analyse von QPNs sehr großen Nutzen hat. Aufgrund der kurzen Reduktionszeitdauer lohnt sich der kleine Mehraufwand in jedem Fall.

7.2 Ausblick

In dieser Arbeit wurden nur sehr einfache Regeln auf Queueing-Petri-Netze übertragen. Es würde sich anbieten auch die komplexeren Regeln aus [9] für zeitbehaftete Petri-Netze zu übertragen. Ferner sind alle Regeln nur für gewöhnliche Stellen definiert. Es könnte auch untersucht werden, unter welchen Umständen Queueing-Stellen beispielsweise fusioniert oder eliminiert werden könnten. Auch die Untersuchung von Regeln zur Reduktion von Stellen mit Kapazitätsgrenzen wäre denkbar. In [26] werden die Reduktionsregeln aus [9] auf gefärbte Petri-Netze übertragen. Bisher muss das Netz für die in dieser Arbeit erweiterten Reduktionsregeln entfaltet werden. Es wäre schön, wenn die Regeln dahingehend weiterentwickelt werden würden, dass sie direkt auf gefärbten Petri-Netzen arbeiten können.

Bei der Implementierung in QPME wurden einige Einschränkungen getroffen (siehe S. 69f). Als Weiterführung der Arbeit könnten Methoden entwickelt werden, um die Einschränkungen aufzuheben, und die fehlenden Funktionalitäten zu ergänzen. Dazu zählen beispielsweise die Konstruktion eines Erreichbarkeitsgraphen, der auch

zeitlose Queueing-Stellen berücksichtigt oder die Analyse von Netzen, die Queueing-Stellen mit Prioritäts-Warteschlangen enthalten.

Bisher wird das zugrundeliegende Netz mit den gleichen Methoden wie das QPN untersucht. Da das zugrundliegende Netz ein EFC-Netz sein muss, lässt es sich in der Regel auf ein sehr kleines Netz reduzieren, insbesondere wenn es lebendig und beschränkt ist. Es gibt für EFC-Netze jedoch wesentlich effizientere Verfahren, um zu bestimmen, ob das Netz lebendig und beschränkt ist. Dazu gehören beispielsweise die Reduktion aus [21] oder die Verwendung von Siphons, Fallen und P-Komponenten [34].

Nicht immer führt die Reduktion zu einem Netz mit kleinem Zustandsraum. Die Berechnung des Erreichbarkeitsgraphen ist bei der Implementierung in dieser Arbeit nicht auf Effizienz bedacht. Es gibt viele Implementierungen, die um einiges schneller sind [22; 23]. Es könnte daher an der Optimierung der Berechnungszeit gearbeitet werden. Dafür würde sich z. B. die Parallelisierung der Zustandsraumexploration anbieten oder die Verwendung von Methoden, die nicht den gesamten Zustandsraum untersuchen [46].

Anhang A

Listings zu den Reduktionsregeln

```
1  private boolean useRule5(){
2    Set<Place> toRemove = new HashSet<Place>();
3    for (Place p : qpn.getOrdinaryPlaces()){
4      if (qpn.getPlaces().size() - toRemove.size() > 1){
5        // Condition (a)
6        if (qpn.getPredecessors(p).containsAll(qpn.getSuccessors(p))&& qpn.getSuccessors(p).
             containsAll(qpn.getPredecessors(p))){
7          boolean identicWeights = true;
8          int maxWeight = 0;
9          for (QPNVertex t : qpn.getPredecessors(p)){
10           int weight = qpn.findEdge(p, t).getWeight();
11           if (weight != qpn.findEdge(t, p).getWeight()) identicWeights = false;
12           if (weight > maxWeight) maxWeight = weight;
13         }
14         // Condition (b) and (c)
15         if (identicWeights && p.getInititalMarking() >= maxWeight) toRemove.add(p);
16       }
17     }
18   }
19   /* Reduce net */
20   if (toRemove.isEmpty()) return false;
21   else{
22     for(Place p : toRemove){
23       qpn.removePlace(p);
24     }
25     return true;
26   }
27 }
```

Listing A.1: Regel QPN-5 – Eliminierung von Self-Loop-Stellen

```
1  private boolean useRule6(){
2    Set<Transition> toRemove = new HashSet<Transition>();
3    for (Transition t : qpn.getImmediateTransitions()){
4      if (qpn.getTransitions().size() - toRemove.size() > 1){
5        // Condition (a)
6        if (qpn.getPredecessors(t).containsAll(qpn.getSuccessors(t))&& qpn.getSuccessors(t).
             containsAll(qpn.getPredecessors(t))){
7          boolean identicWeights = true;
8          boolean containsImmediate = true;
9          for (QPNVertex p : qpn.getSuccessors(t)){
10           if (containsImmediate){
11             if (qpn.findEdge(p, t).getWeight() != qpn.findEdge(t, p).getWeight())
                   identicWeights = false;
12             Set<QPNVertex> postP = new HashSet<QPNVertex>(qpn.getSuccessors(p));
13             postP.remove(t);
14             containsImmediate = false;
15             for(Transition t2 : qpn.getImmediateTransitions()){
16               if (postP.contains(t2)) containsImmediate = true;
17             }
18           }
19         }
20         // Condition (b) and (c)
21         if (identicWeights && containsImmediate) toRemove.add(t);
22       }
23     }
24   }
25   /* Reduce net */
26   if (toRemove.isEmpty()) return false;
27   else{
28     for(Transition t : toRemove){
29       qpn.removeTransition(t);
30     }
31     return true;
32   }
33 }
```

Listing A.2: Regel QPN-6 – Eliminierung von Self-Loop-Transitionen

```
1  private boolean useRule1() {
2    Set<Transition> toRemove = new HashSet<Transition>();
3    for (Transition t : qpn.getImmediateTransitions()) {
4      if (qpn.getPlaces().size() > 1  && (qpn.getTransitions().size()-toRemove.size()>1)) {
5        // Condition (a)
6        if ((qpn.getPredecessorCount(t) == 1) && (qpn.getSuccessorCount(t) == 1)) {
7          Iterator<QPNVertex> i = qpn.getPredecessors(t).iterator();
8          Place p1 = (Place) i.next();
9          Iterator<QPNVertex> j = qpn.getSuccessors(t).iterator();
10         Place p2 = (Place) j.next();
11
12         // Condition (b)
13         if (!p1.isQueuingPlace() && !p2.isQueuingPlace() & (qpn.getSuccessorCount(p1) == 1)
               && (qpn.getPredecessorCount(p2) == 1)) {
14           boolean isOrdinary = true;
15           for (QPNEdge e : qpn.getInEdges(t)) {
16             if (e.getWeight() > 1)
17               isOrdinary = false;
18           }
19           for (QPNEdge e : qpn.getOutEdges(t)) {
20             if (e.getWeight() > 1)
21               isOrdinary = false;
22           }
23
24           // Condition (c)
25           if (isOrdinary) {
26             /* Reduce net */
27             Place newPlace = new OrdinaryPlace(p1.getId(),p1.getName() + "+" + p2.getName(),
                   p1.getInititalMarking()+ p2.getInititalMarking(),p2.getTokenColor());
28             qpn.addPlace(newPlace);
29             for (QPNEdge e : qpn.getInEdges(p1)) {
30               QPNEdge newEdge = new QPNEdge(e.getWeight());
31               qpn.addEdge(newEdge, qpn.getSource(e), newPlace);
32             }
33             for (QPNEdge e : qpn.getOutEdges(p2)) {
34               QPNEdge newEdge = new QPNEdge(e.getWeight());
35               qpn.addEdge(newEdge, newPlace, qpn.getDest(e));
36             }
37             qpn.removePlace(p1);
38             qpn.removePlace(p2);
39             toRemove.add(t);
40           }
41         }
42       }
43     }
44   }
45   if (toRemove.isEmpty())
46     return false;
47   else
48     for (Transition t : toRemove) {
49       qpn.removeTransition(t);
50     }
51   return true;
52 }
```

Listing A.3: Regel QPN-1 – Fusion von seriellen Stellen

```
1  private boolean useRule2(){
2    Set<Place> toRemove = new HashSet<Place>();
3    for (Place p : qpn.getOrdinaryPlaces()) {
4      if (qpn.getPlaces().size()-toRemove.size() > 1 && (qpn.getTransitions().size() > 1)) {
5        // Condition (a)
6        if ((qpn.getPredecessorCount(p) == 1) && (qpn.getSuccessorCount(p) == 1)) {
7          Transition t1 = (Transition) qpn.getPredecessors(p).iterator().next();
8          Transition t2 = (Transition) qpn.getSuccessors(p).iterator().next();
9
10         if (!t2.isTimed()){
11           int b = qpn.findEdge(p, t2).getWeight();
12           // Condition (a), (b), (c) and (d)
13           if ((qpn.findEdge(t1, p).getWeight() % b == 0) && (qpn.getSuccessorCount(t2) > 0)
                 && (qpn.getPredecessorCount(t2) == 1) && p.getInititalMarking()<b){
14             /* Reduce net */
15             Transition newTransition = new Transition(t1.getTransitionId(),t1.getModeId(), t1
                     .getName()+"+"+t2.getName(), t1.isTimed());
16             qpn.addTransition(newTransition);
17             for (QPNEdge e : qpn.getInEdges(t1)) {
18               QPNEdge newEdge = new QPNEdge(e.getWeight());
19               qpn.addEdge(newEdge, qpn.getSource(e), newTransition);
20             }
21             for (QPNEdge e : qpn.getOutEdges(t1)) {
22               QPNEdge newEdge = new QPNEdge(e.getWeight());
23               qpn.addEdge(newEdge, newTransition, qpn.getDest(e));
24             }
25             for (QPNEdge e : qpn.getOutEdges(t2)) {
26               QPNEdge newEdge = new QPNEdge(e.getWeight());
27               QPNEdge sEdge = qpn.findEdge(t1, qpn.getDest(e));
28               if (sEdge != null){
29                 newEdge = new QPNEdge(sEdge.getWeight()+e.getWeight());
30                 qpn.removeEdge(qpn.findEdge(newTransition, qpn.getDest(e)));
31               }
32               qpn.addEdge(newEdge, newTransition, qpn.getDest(e));
33             }
34             qpn.removeTransition(t1);
35             qpn.removeTransition(t2);
36             toRemove.add(p);
37           }
38         }
39       }
40     }
41   }
42   if (toRemove.isEmpty())
43     return false;
44   else
45     for (Place p : toRemove) {
46       qpn.removePlace(p);
47     }
48   return true;
49 }
```

Listing A.4: Regel QPN-2 – Fusion von seriellen Transitionen

```
1  private boolean useRule2a(){
2    Set<Place> toRemove = new HashSet<Place>();
3    for (Place p : qpn.getOrdinaryPlaces()) {
4      if (qpn.getPlaces().size()-toRemove.size() > 1 && (qpn.getTransitions().size() > 1)) {
5        // Condition (d)
6        if ((qpn.getPredecessorCount(p) == 1) && (qpn.getSuccessorCount(p) == 1)) {
7          Transition t1 = (Transition) qpn.getPredecessors(p).iterator().next();
8          Transition t2 = (Transition) qpn.getSuccessors(p).iterator().next();
9          //Condition (e)
10         boolean noPred = true;
11         if (!t1.isTimed() && t2.isTimed()){
12           for (QPNVertex pi : qpn.getPredecessors(t1)) {
13             if (qpn.getSuccessorCount(pi)>1) noPred =false;
14           }
15         }
16         if (noPred){
17           int b = qpn.findEdge(p, t2).getWeight();
18           // Condition (c), (b), (a) and (e)
19           if ((qpn.findEdge(t1, p).getWeight() % b == 0) && (qpn.getSuccessorCount(t2) > 0)
                 && (qpn.getPredecessorCount(t2) == 1) && (qpn.getSuccessorCount(t1)==1) && p.
                 getInititalMarking()<b){
20             /* Reduce net */
21             Transition newTransition = new Transition(t1.getTransitionId(), t1.getModeId(),
                     t1.getName()+"+"+t2.getName(), t1.isTimed()||t2.isTimed());
22             qpn.addTransition(newTransition);
23             for (QPNEdge e : qpn.getInEdges(t1)) {
24               QPNEdge newEdge = new QPNEdge(e.getWeight());
25               qpn.addEdge(newEdge, qpn.getSource(e), newTransition);
26             }
27             for (QPNEdge e : qpn.getOutEdges(t1)) {
28               QPNEdge newEdge = new QPNEdge(e.getWeight());
29               qpn.addEdge(newEdge, newTransition, qpn.getDest(e));
30             }
31             for (QPNEdge e : qpn.getOutEdges(t2)) {
32               QPNEdge newEdge = new QPNEdge(e.getWeight());
33               QPNEdge sEdge = qpn.findEdge(t1, qpn.getDest(e));
34               if (sEdge != null){
35                 newEdge = new QPNEdge(sEdge.getWeight()+e.getWeight());
36                 qpn.removeEdge(qpn.findEdge(newTransition, qpn.getDest(e)));
37               }
38               qpn.addEdge(newEdge, newTransition, qpn.getDest(e));
39             }
40             qpn.removeTransition(t1);
41             qpn.removeTransition(t2);
42             toRemove.add(p);
43           }
44         }
45       }
46     }
47   }
48
49   if (toRemove.isEmpty())
50     return false;
51   else
52     for (Place p : toRemove) {
53       qpn.removePlace(p);
54     }
55   return true;
56 }
```

Listing A.5: Regel QPN-2* – Fusion von seriellen Transitionen 2

```
1  private boolean useRule3(){
2    Set<Place> toRemove = new HashSet<Place>();
3    for (Place p : qpn.getOrdinaryPlaces()){
4      Set<Place> identicalPlaces = new HashSet<Place>();
5      if (!toRemove.contains(p)){
6        for (Place p2 : qpn.getOrdinaryPlaces()){
7          if (qpn.getPredecessors(p).containsAll(qpn.getPredecessors(p2)) && qpn.
               getPredecessors(p2).containsAll(qpn.getPredecessors(p))&& qpn.getSuccessors(p).
               containsAll(qpn.getSuccessors(p2))&& qpn.getSuccessors(p2).containsAll(qpn.
               getSuccessors(p))){
8            /* Check if same weight */
9            boolean identical = true;
10
11           // Condition (a)
12           for (QPNEdge e : qpn.getInEdges(p)){
13             QPNEdge e2 = qpn.findEdge(qpn.getSource(e), p2);
14             if (e.getWeight() != e2.getWeight()) identical=false;
15           }
16           for (QPNEdge e : qpn.getOutEdges(p)){
17             QPNEdge e2 = qpn.findEdge(p2, qpn.getDest(e));
18             if (e.getWeight() != e2.getWeight()) identical=false;
19           }
20           if (identical) identicalPlaces.add(p2);
21         }
22       }
23     }
24     /* Reduce net */
25     int minMarking = p.getInititalMarking();
26     Place min = p;
27     for (Place p2 : identicalPlaces){
28       if (p2.getInititalMarking() < minMarking){
29         minMarking = p.getInititalMarking();
30         min = p2;
31       }
32       toRemove.add(p2);
33     }
34     toRemove.remove(min);
35   }
36   if (toRemove.isEmpty()) return false;
37   else{
38     for (Place p : toRemove){
39       qpn.removePlace(p);
40     }
41     return true;
42   }
43 }
```

Listing A.6: Regel QPN-3 – Eliminierung von identischen Stellen

```
1  private boolean useRule4(){
2    Set<Transition> toRemove = new HashSet<Transition>();
3    for (Transition t : qpn.getTransitions()){
4      Set<Transition> identicalTransitions = new HashSet<Transition>();
5      if (!toRemove.contains(t)){
6        for (Transition t2 : qpn.getTransitions()){
7          if (qpn.getPredecessors(t).containsAll(qpn.getPredecessors(t2)) && qpn.
               getPredecessors(t2).containsAll(qpn.getPredecessors(t))&& qpn.getSuccessors(t).
               containsAll(qpn.getSuccessors(t2))&& qpn.getSuccessors(t2).containsAll(qpn.
               getSuccessors(t))){
8            /* Check if same weight */
9            boolean identical = true;
10
11           // Condition (a)
12           for (QPNEdge e : qpn.getInEdges(t)){
13             QPNEdge e2 = qpn.findEdge(qpn.getSource(e), t2);
14             if (e.getWeight() != e2.getWeight()) identical=false;
15           }
16           for (QPNEdge e : qpn.getOutEdges(t)){
17             QPNEdge e2 = qpn.findEdge(t2, qpn.getDest(e));
18             if (e.getWeight() != e2.getWeight()) identical=false;
19           }
20           if (identical) identicalTransitions.add(t2);
21         }
22       }
23     }
24     /* Reduce net */
25     Transition timed = null;
26     Transition immediate = null;
27     for (Transition t2 : identicalTransitions){
28       if (t2.isTimed()) timed = t2;
29       else immediate = t2;
30       toRemove.add(t2);
31     }
32     if (timed == null) toRemove.remove(immediate); // only immediate transitions
33     else if (immediate == null) toRemove.remove(timed); // only timed transitions
34     else{
35       toRemove.remove(immediate);
36       Result.setLive(false);
37     }
38   }
39   if (toRemove.isEmpty()) return false;
40   else{
41     for (Transition t : toRemove){
42       qpn.removeTransition(t);
43     }
44     return true;
45   }
46 }
```

Listing A.7: Regel QPN-4 – Eliminierung von identischen Transitionen

Anhang B

QPME Modelle zu den untersuchten Beispielen

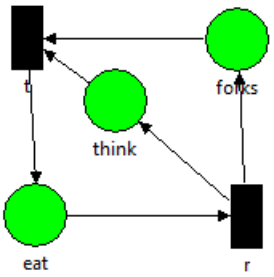


Abbildung B.1: `philosophers`

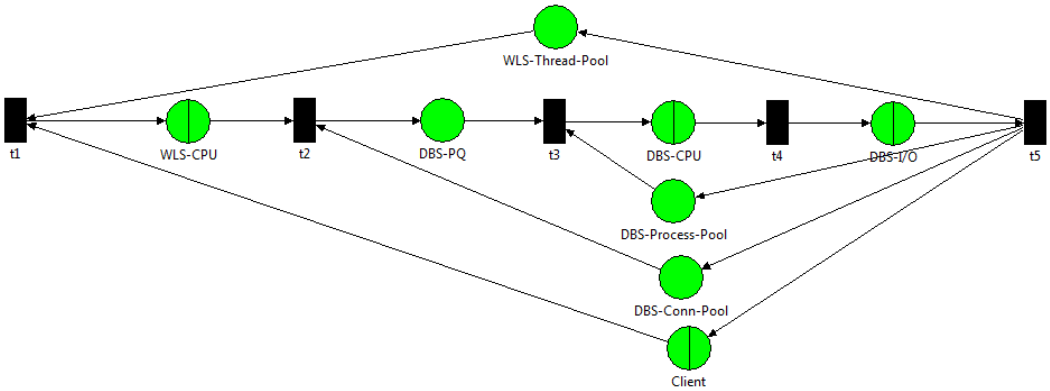


Abbildung B.2: `ispass03`

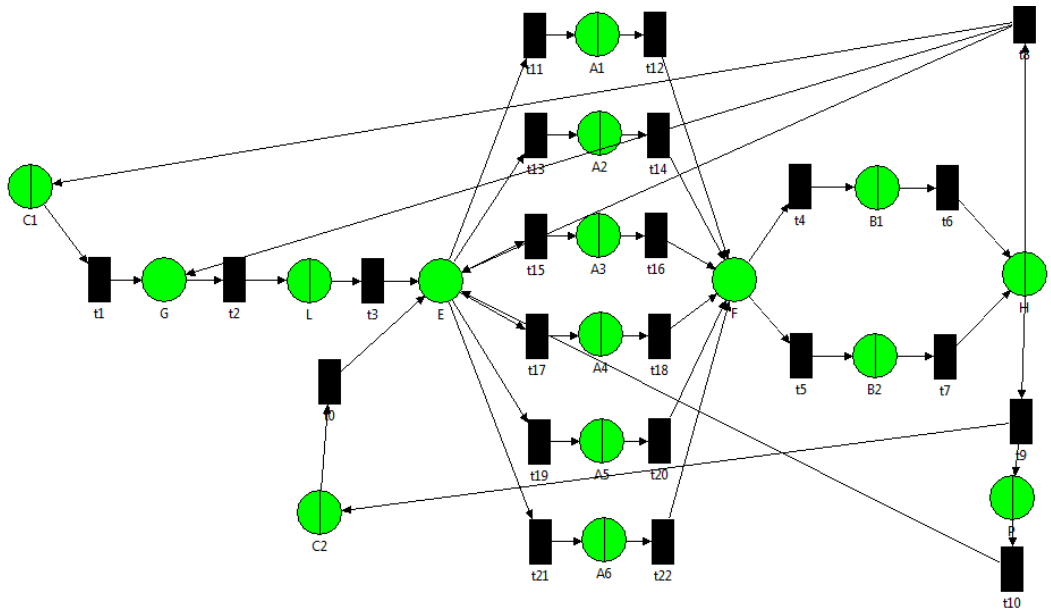


Abbildung B.3: `specJ2004`

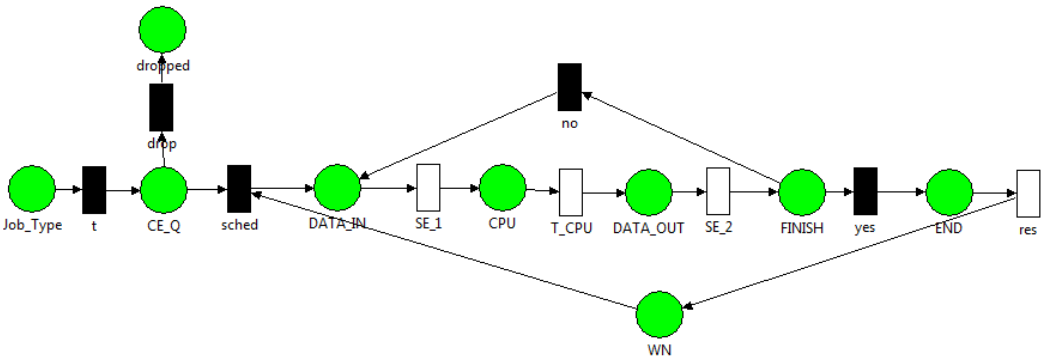


Abbildung B.4: `gLite`

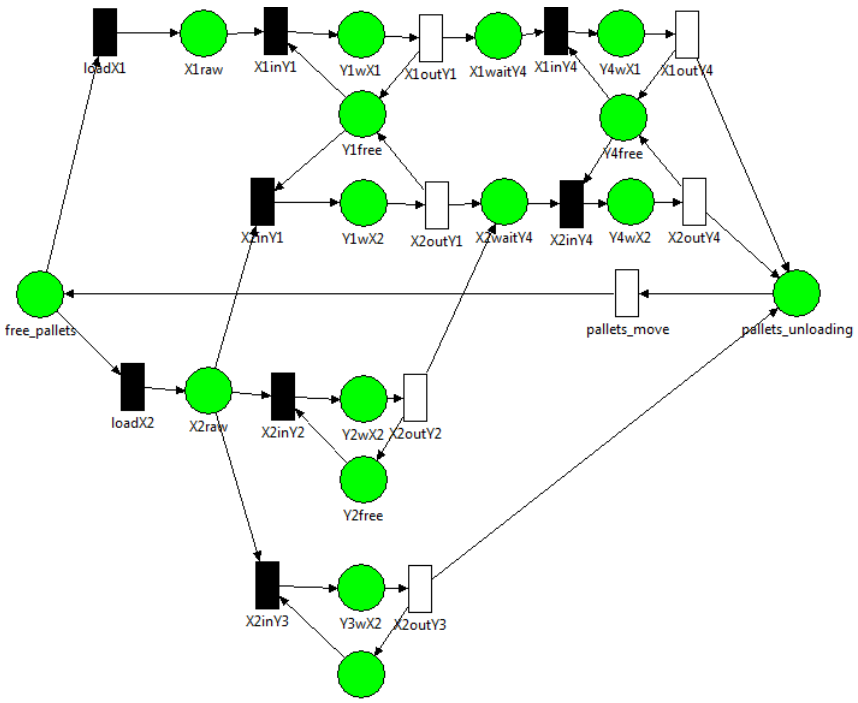


Abbildung B.5: `fms cell`

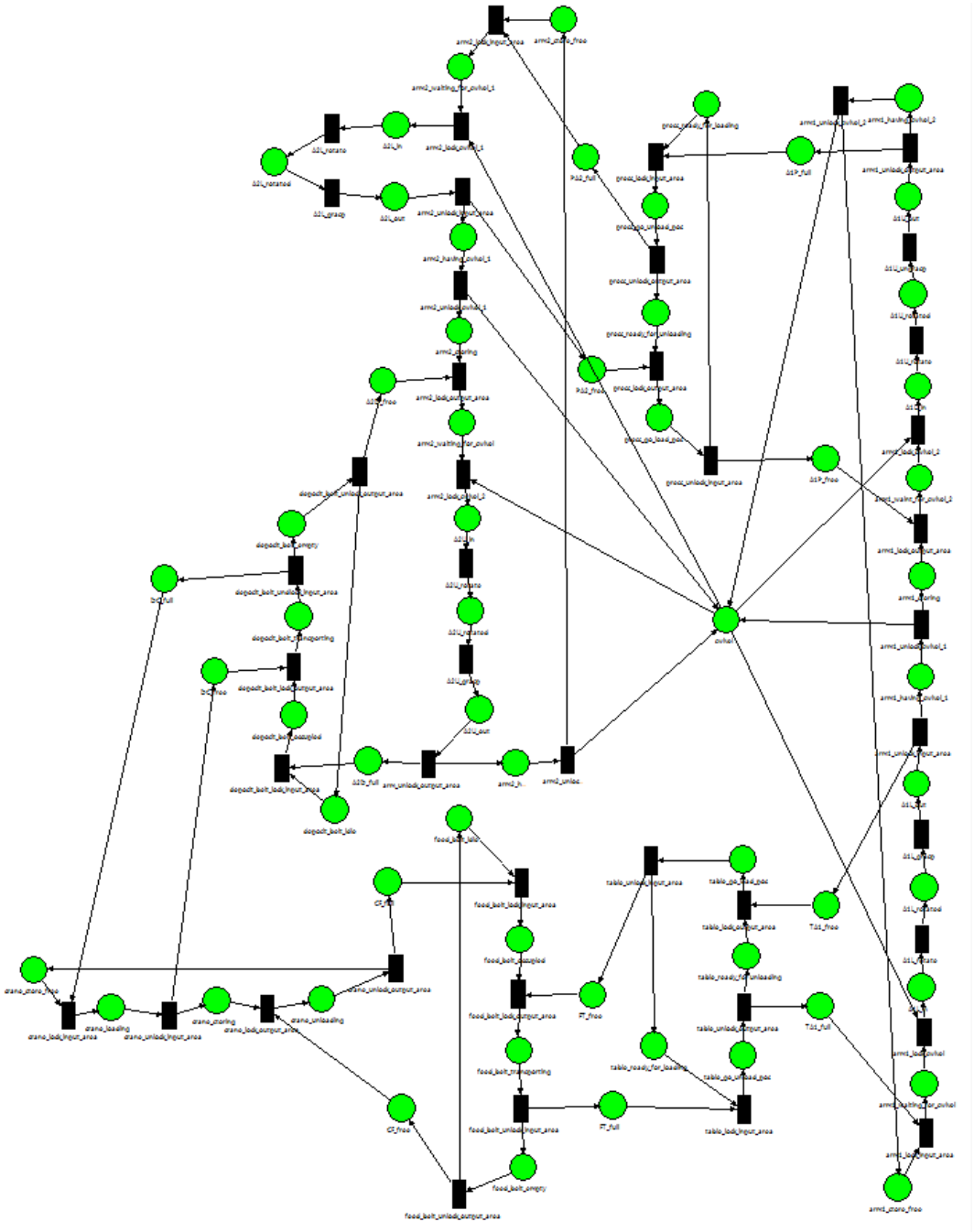


Abbildung B.6: `production cell`

Anhang C

Inhalt der beigefügten CD-ROM

Der Arbeit ist eine CD-ROM beigefügt, die die Erweiterung von QPME sowie die zugehörigen Quelldateien enthält. Es liegen ausführbare Dateien für Windows und Linux jeweils in 32-Bit und 64-Bit vor. Zur Ausführung wird mindestens JRE 6 benötigt. Falls nicht vorhanden, kann unter `https://www.java.com/de/download/` eine heruntergeladen werden. Die Argumente für die Virtual Machine lassen sich in der Datei *QPME.ini* anpassen.

Außerdem enthält die CD-ROM alle in Kapitel 6 betrachteten Beispiele als QPME-Modell und den *User's Guide* zu QPME.

Abbildungsverzeichnis

Literaturverzeichnis

1 Aalst, Wil M. P. van der: *The Application of Petri Nets to Workflow Management.* Journal of Circuits, Systems, and Computers, 8(1):21–66, 1998. 2

2 Bause, F., P. Buchholz und P. Kemper: *QPN-Tool for the Specification and Analysis of Hierarchically Combined Queueing Petri Nets.* In: Beilner, H. und F. Bause (Herausgeber): *Quantitative Evaluation of Computing and Communication Systems*, Band 977 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 1995. 5.1

3 Bause, Falko: *"QN + PN = QPN Combining Queueing Networks and Petri Nets*, 1993. 4.2

4 Bause, Falko: *Queueing Petri Nets - A Formalism for the Combined Qualitative and Quantitative Analysis of Systems.* In: *In Proceedings of the 5th International Workshop on Petri nets and Performance Models. IEEE*, Seiten 14–23. IEEE Computer Society, 1993. 3.3, 3.3, 3.3.4

5 Bause, Falko und Heinz Beilner: *Eine Modellwelt zur Integration von Warteschlangen- und Petri-Netz-Modellen.* In: Stiege, Günther und J. S. Lie (Herausgeber): *MMB*, Band 218 der Reihe *Informatik-Fachberichte*, Seiten 190–204. Springer, 1989. 1.1, 3, 3.2, 3.3, 5.2

6 Bause, Falko und Heinz Beilner: *Analysis of a Combined Queueing-Petri-Network World*, 1991. 3.3, 3.3.3, 3.3

7 Bause, Falko und Pieter S. Kritzinger: *Stochastic Petri Nets.* Vieweg-Teubner Verlag, 2nd Edition Auflage, 2002. 1.2, 2, 2.2.2, 2.7, 2.3, 2.3.2, 2.8(a), 2.4.7, 2.4.2, 2.14, 3.2.1, 3.7(a), 3.3, 3.3.5, 4.2, 4.13(a)

8 Berthelot, G.: *Transformations and Decompositions of Nets.* In: *Advances in Petri Nets 1986, Part I on Petri Nets: Central Models and Their Properties*, Seiten 359–376, London, UK, UK, 1987. Springer-Verlag. 1.2, 4.1.1

9 Berthelot, Gérard: *Checking properties of nets using transformation.* In: Rozenberg, Grzegorz (Herausgeber): *Applications and Theory in Petri Nets*, Band 222 der Reihe *Lecture Notes in Computer Science*, Seiten 19–40. Springer, 1985. 1.1, 1.2, 4.1, 4.2, 4.3, 4.1, 7.2

10 Berthelot, Gérard und Richard Terrat: *Petri Nets Theory for the Correctness of Protocols.* In: *DDSS*, Seiten 23–43, 1981. 2

11 Best, Eike: *Structure Theory of Petri Nets: the Free Choice Hiatus.* In: Brauer, Wilfried, Wolfgang Reisig und Grzegorz Rozenberg (Herausgeber): *Advances in Petri Nets*, Band 254 der Reihe *Lecture Notes in Computer Science*, Seiten 168–205. Springer, 1986. 2.3.4

12 Best, Eike, Jörg Desel und Javier Esparza: *Traps characterize home states in free choice systems.* Theoretical Computer Science, 101(2):161 – 176, 1992. 2.3.3

13 Best, Eike und Pa Thiagarajan: *Some Classes of Live and Safe Petri Nets.* In: *Concurrency and Nets*, Advances in Petri Nets, Seiten 71–94. Springer-Verlag, 1987. 2.3.3

14 Bruneo, Dario, Marco Scarpa und Antonio Puliafito: *A GSPN Model to Analyze Performance Parameters in gLite Grids.* In: *WETICE*, Seiten 198–203. IEEE Computer Society, 2008. 6.1

15 Chiola, Giovanni, Marco Ajmone Marsan, Senior Member, Gianfranco Balbo und Gianni Conte: *Generalized stochastic Petri nets: A definition at the net level and its implications.* IEEE Transactions on Software Engineering, 19:89–107, 1993. 6.1

16 Commoner, F., A.W. Holt, S. Even und A. Pnueli: *Marked directed graphs.* Journal of Computer and System Sciences, 5(5):511 – 523, 1971. 2.3.2, 2.3.2

17 David, René und Hassane Alla: *Discrete, Continuous and Hybrid Petri Nets.* Springer-Verlag, 2005.

18 Diaz, Michel (Herausgeber): *Petri Nets: Fundamental Models, Verification and Applications.* ISTE, 2009. 1.2, 2.2.2, 2.2.6, 2.2.7, 2.2.1, 2.2.9, 2.2.2, 2.2.2, 4.1

19 Diaz, Michel und Pierre Azéma: *Petri net based models for the specification and validation of protocols.* In: Rozenberg, Grzegorz, Hartmann J. Genrich und Gérard Roucairol (Herausgeber): *European Workshop on Applications and Theory in Petri Nets*, Band 188 der Reihe *Lecture Notes in Computer Science*, Seiten 101–121. Springer, 1984. 2

20 Dijkstra, Edsger W.: *Hierarchical Ordering of Sequential Processes.* Acta Informatica, Seiten 29–52, 1971. 2.1.4

21 Esparza, J.: *Reduction and Synthesis of Live and Bounded Free Choice Petri Nets.* Information and Computation, 114(1):50 – 87, 1994. 4.1, 7.2

22 Evangelista, Sami: *High Level Petri Nets Analysis with Helena.* In: Ciardo, Gianfranco und Philippe Darondeau (Herausgeber): *ICATPN*, Band 3536 der Reihe *Lecture Notes in Computer Science*, Seiten 455–464. Springer, 2005. 6.2, 7.2

23 Fronc, Łukasz und Franck Pommereau: *Optimising the compilation of Petri net models.* In: *Proceedings of CompoNet'11 and SUMO'11*, Band 726 der Reihe *Workshop Proceedings*, Seiten 49–63. CEUR, 06 2011. 6.2, 7.2

24 Genrich, Hartmann J. und Kurt Lautenbach: *Synchronisationsgraphen.* Acta Inf., 2:143–161, 1973. 2.3.1

25 Girault, Claude und Rüdiger Valk (Herausgeber): *Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications.* Springer-Verlag, 2003. 2.1, 4.1.5, 4.6

26 Haddad, Serge: *A Reduction Theory for Coloured Petri Nets.* In: Rozenberg, Grzegorz (Herausgeber): *Advances in Petri Nets 1989, Papers from the 9th International Conference on Applications and Theory of Petri Nets (APN'88)*, Band 424 der Reihe *Lecture Notes in Computer Science*, Seiten 209–235, Venice, Italy, 1990. Springer-Verlag. 7.2

27 Heiner, Monika und Peter Deussen: *Petri Net Based Qualitative Analysis - a Case Study.* Technischer Bericht, BTU Cottbus, Dep. of CS, Techn. Report I-08/1995, 1995. 6.1

28 Jensen, Kurt: *Coloured petri nets and the invariant-method.* Theoretical Computer Science, 14(3):317 – 336, 1981. 1.1, 2.4.1

29 Karp, Richard M. und Raymond E. Miller: *Parallel program schemata.* Journal of Computer and System Sciences, 3(2):147 – 195, 1969. 2.2.2

30 Kounev, Samuel: *Performance Modeling and Evaluation of Distributed Component-Based Systems using Queueing Petri Nets.* IEEE Transactions on Software Engineering, 32(7):486–502, July 2006. 6.1

31 Kounev, Samuel und Alejandro Buchmann: *Performance Modeling of Distributed E-Business Applications using Queueing Petri Nets.* In: *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2003), Austin, Texas, USA, March 6-8, 2003*, Seiten 143–155, Washington, DC, USA, 2003. IEEE Computer Society. Best-Paper-Award. 6.1

32 Kounev, Samuel und Simon Spinner: *QPME 2.0 User's Guide.* Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany, Mai 2011. 5.1

33 Kounev, Samuel, Simon Spinner und Philipp Meier: *QPME 2.0 - A Tool for Stochastic Modeling and Analysis Using Queueing Petri Nets.* In: Sachs, Kai, Ilia Petrov und Pablo Guerrero (Herausgeber): *From Active Data Management to Event-Based Systems and More*, Band 6462 der Reihe *Lecture Notes in Computer Science*, Seiten 293–311. Springer, 2010. 1.2, 5

34 Kovalyov, A.V.: *An $O(|S| \times |T|)$-algorithm to verify liveness and boundedness in extended free choice nets.* In: *Intelligent Control, 1995., Proceedings of the 1995 IEEE International Symposium on*, Seiten 597–601, Aug 1995. 7.2

35 Marsan, M. Ajmone, Gianfranco Balbo, Gianni Conte, Susanna Donatelli und Giuliana Franceschinis: *Modelling with Generalised Stochastic Petri Nets.* John Wiley & Sons, Inc., 1994. 2.4.2

36 Marsan, Marco Ajmone, Gianni Conte und Gia Balbo: *A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems.* ACM Transactions on Computer Systems (TOCS), 2(2):93–122, 1984. 1.1, 2.4.2, 2.4.2

37 Murata, T.: *Petri nets: Properties, Analysis and Applications.* 77(4):541–580, 1989. 1.2, 2.5, 2.6, 2.2.2, 2.2.17, 2.3, 2.9, 2.10, 2.11, 2.12, 2.3.14

38 Peterson, James Lyle: *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

39 Petri, Carl Adam: *Kommunikation mit Automaten.* Doktorarbeit, Technische Hochschule Darmstadt, 1962. 1.1, 2

40 Salimifard, Khodakaram und Mike Wright: *Petri net-based modelling of workflow systems: An overview.* European Journal of Operational Research, 134(3):664 – 676, 2001. 2

41 Silva, M.: *Sur le concept de macroplace et son utilisation pour lanalyse des réseaux de Petri.* RAIRO-Automatique, 15:335–345, 1981. 4.1

42 Silva, Manuel und Enrique Teruel: *Petri Nets for the Design and Operation of Manufacturing Systems.* European Journal of Control, 1997. 2

43 Starke, Peter H.: *Analyse von Petri-Netz-Modellen.* Teubner, 1990. 2.1, 2.2.2

44 Tarjan, Robert: *Depth first search and linear graph algorithms.* SIAM Journal on Computing, 1972. 5.3.4

45 Valmari, Antti: *The State Explosion Problem.* In: Reisig, Wolfgang und Grzegorz Rozenberg (Herausgeber): *Petri Nets*, Band 1491 der Reihe *Lecture Notes in Computer Science*, Seiten 429–528. Springer-Verlag. 1.1, 2.2.2

46 Valmari, Antti: *Stubborn Sets for Reduced State Space Generation.* In: *Proceedings of the Tenth International Conference on Application and Theory of Petri Nets*, Seiten 1–22, 1989. 1.1, 2.2.2, 7.2

47 Vernadat, F., F. Dicesare, G. Harhalakis, J.M. Proth und M. Silva: *Practice of Petri Nets in Manufacturing.* Chapman and Hall, 1993. 4.1, 4.5

48 Vernon, Mary K., Edward D. Lazowska und John Zahorjan: *A comparison of performance Petri nets and queueing network models.* Technischer Bericht TR 0669, University of Wisconsin (Madison, WI US), 1986. 3

Tabellenverzeichnis

Verzeichnis der Listings