



FAKULTÄT II  
DEPARTMENT FÜR INFORMATIK  
Abteilung Parallele Systeme

Simulation von Generalized Stochastic Petri Nets

26. Juli 2006

**Bearbeitet von:** Tim Strazny  
**Erstgutachter:** Prof. Dr. Eike Best  
**Zweitgutachter:** Dipl.-Inform. Christian Stehno  
**Anschrift:** Tim Strazny, Bloherfelder Straße 109, 26129 Oldenburg  
tim.strazny@informatik.uni-oldenburg.de



## Zusammenfassung

Petrinetze mit ihrer eindeutigen Semantik und dem zugrunde liegenden mathematischen Modell bewähren sich vermehrt bei Entwurf, Analyse und Steuerung komplexer, asynchroner und verteilter Systeme. Insbesondere zeitbehaftete stochastische gefärbte Petrinetze bieten Möglichkeiten komplizierte Sachverhalte strukturiert und kompakt ausdrücken, jedoch ist die Simulation solcher höheren Petrinetze um ein Vielfaches aufwändiger. Mit Geschwindigkeit und Vielfältigkeit als Hauptziel wurden mit P-UMLaut/Sim ein Simulator für solche mehrfach erweiterten Petrinetze entwickelt.

Petri nets with their unambiguous semantics and their algebraic background have proven themselves valuable for the design, analysis and control of complex, asynchronous and distributed systems. Particularly timed stochastic coloured Petri nets provide good ways to express complex behaviour in a structured and compact manner. However simulation of these high level Petri nets is more complex than simulation of low level Petri nets. With performance and flexibility as the main design goals, the high level Petri net simulator P-UMLaut/Sim has been developed.

## Danksagung

Ich möchte mich ganz herzlich bei meinen Eltern Frauke und Klaus für die nicht enden wollende Unterstützung bedanken.

Auch der Antrieb, der Rat und die Hilfestellungen meines Betreuers Christian Stehno und meines Mentors Henning Dierks sollen hier nicht unerwähnt bleiben.

Danke.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Grundlagen</b>	<b>2</b>
2.1	P/T-Netze . . . . .	2
2.2	Weitere Kantentypen . . . . .	4
2.2.1	Inhibitorkanten . . . . .	4
2.2.2	Lesekanten . . . . .	5
2.2.3	Resetkanten . . . . .	5
2.3	Stellen mit Kapazitäten . . . . .	6
2.4	Netze mit Prioritäten . . . . .	6
2.5	Netze mit Zeit . . . . .	7
2.5.1	Memory Policies . . . . .	8
2.5.2	Server-Semantiken . . . . .	9
2.6	Stochastische Netze . . . . .	10
2.7	Verallgemeinerte stochastische Netze . . . . .	10
2.8	Gefärbte Netze . . . . .	11
2.9	Simulation von Petrinetzen . . . . .	13
<b>3</b>	<b>Konzept</b>	<b>15</b>
3.1	Ursprüngliche Anforderungen . . . . .	15
3.2	Idee und Methodik . . . . .	16
3.3	Schnittstellen . . . . .	17
<b>4</b>	<b>Implementierung der Basisfunktionalität</b>	<b>18</b>
4.1	Grundlegendes . . . . .	18
4.2	Datenstrukturen . . . . .	19
4.2.1	Wertetypen . . . . .	19

4.2.2	Variablenbelegung . . . . .	19
4.2.3	Syntaxbaumrechner . . . . .	20
4.3	Grammatik logischer/arithmetischer Ausdrücke . . . . .	20
4.4	Behandlung logischer/arithmetischer Ausdrücke . . . . .	20
4.5	Gefärbte Petrinetze . . . . .	24
4.6	Berechnung aktivierender Variablenbelegungen . . . . .	28
4.7	Tupel . . . . .	31
4.8	Funktionen . . . . .	32
<b>5</b>	<b>Integration der Erweiterungen</b>	<b>36</b>
5.1	Kapazitäten . . . . .	36
5.2	Ungetypte Stellen . . . . .	38
5.3	Prioritäten . . . . .	38
5.4	Weitere Kantentypen . . . . .	39
5.4.1	Resetkanten . . . . .	39
5.4.2	Inhibitorkanten . . . . .	40
5.4.3	Lesekanten . . . . .	40
5.5	Zeit . . . . .	40
5.5.1	Age Memory . . . . .	42
5.5.2	Server-Semantiken . . . . .	44
5.6	Stochastische Elemente . . . . .	48
5.7	Paralleles Feuern . . . . .	49
5.8	Entfaltung . . . . .	50
5.9	Ausführung . . . . .	51
5.10	Performance Tuning . . . . .	52
<b>6</b>	<b>Vergleich mit anderen Werkzeugen</b>	<b>54</b>
6.1	PEP-NetSim . . . . .	54
6.2	cgpetri . . . . .	55
<b>7</b>	<b>Zusammenfassung und Ausblick</b>	<b>57</b>
7.1	Ausblick . . . . .	59
7.1.1	Funktionalität . . . . .	59
7.1.2	Performance Tuning . . . . .	60

<b>Anhang</b>	<b>61</b>
<b>Abbildungsverzeichnis</b>	<b>61</b>
<b>Literaturverzeichnis</b>	<b>62</b>
<b>Glossar</b>	<b>65</b>
<b>Index</b>	<b>73</b>

# 1 Einleitung

„Chef ist nicht der, der etwas tut, sondern  
der das Verlangen weckt, etwas zu tun.“

Edgar Pisani

Seit ihrer Einführung wurden Petrinetze [Pet62] (siehe auch [PNW]) um zusätzliche Konzepte wie Zeit, Wahrscheinlichkeiten und Prioritäten erweitert. In gefärbten (*high-level*) Petrinetzen können mehrere gleichartige Netzelemente zusammengefasst werden, was unter anderem die Modellierung mit Petrinetzen erleichtert. Solche farbigen Petrinetze können zu semantisch äquivalenten (*low-level*) Petrinetzen entfaltet werden, für die es formale Analysemethoden gibt.

Insbesondere während der Modellierung, aber auch während der Test- und Analysephase bietet die Simulation der Netze einen intuitiven Zugang zum modellierten Verhalten. Über eine geeignete Anbindung an weitere Algorithmen kann die Simulation auch zur Generierung weiterer Datenstrukturen zur Untersuchung des Systems, z.B. Erreichbarkeitsgraphen genutzt werden.

Im Rahmen des Projekts P-UMLaut [PUM] wurde der Petrinetzsimulator P-UMLaut/Sim als ereignisgesteuerte Ausführungskomponente entwickelt und in das Framework integriert. Er steuert eine interaktive 3D-Visualisierungskomponente für Prozessmodelle, ist aber auch für die allgemeine Petrinetzsimulation der unterstützten Petrinetzklassen nutzbar.

Ziel dieser Arbeit war es, den P-UMLaut/Sim so zu erweitern, dass Generalized Stochastic Petri Nets [MBC<sup>+</sup>95] [Bal01] simuliert werden können.

## 2 Grundlagen

„Arbeit ist eine Sucht, die wie eine Notwendigkeit aussieht.“

Peter Altenberg

In den 60er Jahren stellte Prof. Dr. Carl Adam Petri in seiner Dissertation *Kommunikation mit Automaten* [Pet62] die Petrinetze als neues Konzept zur Darstellung nebenläufiger Prozesse vor und verallgemeinerte damit die Automatentheorie. Seit ihrer Definition wurden Petrinetze mit ihren Eigenschaften weiter erforscht und unter anderem formale Verifikation ermöglicht.

Im Folgenden wird ein Überblick über die notwendigen Grundlagen gegeben, insbesondere werden die von Carl Adam Petri erdachten Netze und einige darauf aufbauende Erweiterungen vorgestellt. Ab Kapitel 4 *Implementierung der Grundlagen* ab Seite 18 wird genauer auf die einzelnen Punkte und ihre Umsetzung im P-UMLaut/Sim eingegangen.

Um einen guten Einstieg in das Thema der Petrinetze zu erhalten und ihre zugrunde liegendes mathematisches Modell kennen zu lernen, kann beispielsweise in [PW02] und [Smi98] nachgeschlagen werden. Die wesentlichen Definitionen werden im Weiteren vorausgesetzt.

### 2.1 P/T-Netze

Die einfachste Form von Petrinetzen sind solche, bei denen Stellen, Transitionen und Kanten unbeschriftet sind und nur eine Sorte Marken im Netz vorkommt. Zwischen einer Transition und einer Stelle können eine beliebige Anzahl gerichteter Kanten existieren. Eine Transition  $t$  ist genau dann aktiviert, wenn jede Stelle  $s$  im Vorbereich  $\bullet t$  von  $t$  mit

mindestens einer Marke für jede Kante von  $s$  nach  $t$  belegt ist. Nach dem Feuern von  $t$  werden auf jeder Stelle  $s'$  in  $t^\bullet$  so viele Marken erzeugt, wie es Kanten von  $t$  nach  $s'$  gibt.

Eine Stelle  $s$  ist genau dann im Vorbereich  ${}^\bullet t$  einer Transition  $t$ , wenn es eine Kante von  $s$  nach  $t$  gibt. Genau dann, wenn es eine Kante von Transition  $t$  nach Stelle  $s$  gibt, ist  $s$  im Nachbereich  $t^\bullet$ .

Ein solches Netz ist ein Tupel  $(P, T, \mathbb{F}, \mathbb{B})$ , wobei  $P = \{p_1, \dots, p_{|P|}\}$  eine endliche Menge von Stellen,  $T = \{t_1, \dots, t_{|T|}\}$  eine endliche Menge von Transitionen mit  $P \cap T = \emptyset$  und  $\mathbb{F}$  und  $\mathbb{B}$  beides  $|P| \times |T|$ -Matrizen über  $\mathbb{N}$  sind. Die *Forward*-Matrix  $\mathbb{F}$  beschreibt an Stelle  $\mathbb{F}_{i,j}$  wie viele Kanten von Stelle  $s_i$  zu Transition  $t_j$  führen. Analog beschreibt ein Eintrag  $\mathbb{B}_{i,j}$  in der *Backward*-Matrix, wie viele Kanten von Transition  $t_j$  zu Stelle  $s_i$  führen. Eine Transition  $t_j$  ist genau dann unter einer Markierung  $M \in \mathbb{N}^P$  aktiviert, wenn gilt  $M \geq \mathbb{F}_{\bullet,j}$ , wobei  $\mathbb{F}_{\bullet,j}$  gerade die  $j$ -te Spalte von  $\mathbb{F}$  beschreibt. Ist Transition  $t_j$  in  $M \in \mathbb{N}^P$  aktiviert, so ist der Nachfolgezustand  $M' = M - \mathbb{F}_{\bullet,j} + \mathbb{B}_{\bullet,j} \in \mathbb{N}^P$ . Die Schreibweise für das Feuereiner Transition  $t_j$  ist  $M[t_j]M'$  oder  $M \xrightarrow{t_j} M'$ . [PW02]

Als Kurzschreibweise für mehrere Marken auf einer Stelle kann die Anzahl der Marken geschrieben werden (Multiplizität).

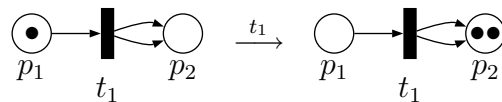


Abbildung 2.1: P/T-Netz und Feuereiner Transition

In Abbildung 2.1 ist ein Zustandsübergang eines sehr einfachen P/T-Netzes dargestellt. Als verkürzende Notation werden meist alle Kanten mit denselben Quellen und Zielen zu einer einzelnen zusammengefasst, die dann mit der ursprünglichen Anzahl beschriftet wird. Im Falle des Beispiels ging eine mit 2 beschriftete Kante von Transition  $t_1$  zu Stelle  $p_2$ .

Aus der Menge der aktivierten Transitionen wird nichtdeterministisch eine zu feuernde ausgewählt. Das Feuereiner Transitionen wird interleaved betrieben, zwei Transitionen feuern niemals gleichzeitig.

Zwei Transitionen  $t_1, t_2$  stehen miteinander in Konflikt, wenn beide aktiviert sind und das Feuereiner der beiden die andere deaktiviert. Das ist dann der Fall, wenn sich  $t_1$

und  $t_2$  mindestens eine Stelle  $s$  im Vorbereich teilen, also  $s \in \bullet t_1 \cap \bullet t_2$  gilt.

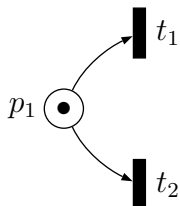


Abbildung 2.2: Ein direkter Konflikt

Abbildung 2.2 zeigt eine einfache Konfliktsituation.

Ist die Anzahl der Marken die auf einer Stelle  $s$  liegen können nach oben durch ein  $k \in \mathbb{N}$  beschränkt, so ist die Stelle  $k$ -beschränkt. Genau dann, wenn eine Stelle für kein  $k \in \mathbb{N}$  beschränkt ist, die Anzahl der Marken auf der Stelle also prinzipiell beliebig zunehmen kann, ist die Stelle *unbeschränkt*. Sind alle Stellen eines Netzes beschränkt, so ist das Petrinetz selbst beschränkt. Sind alle Stellen eines Netzes 1-beschränkt, so wird das Petrinetz als sicher bezeichnet.

## 2.2 Weitere Kantentypen

Neben den *normalen* Kanten, wie sie im vorangegangenen Abschnitt 2.1 ab Seite 2 vorgestellt wurden, gibt es weitere, die allesamt von Stellen zu Transitionen führen. Manche dieser Kanten erweitern die Mächtigkeit der Petrinetze nicht.

### 2.2.1 Inhibitorkanten

Eine mit  $k$  beschriftete Inhibitorkante von Stelle  $s$  zu Transition  $t$  erlaubt genau dann die Aktivierung von  $t$ , wenn  $s$  weniger als  $k$  Marken enthält. Wird die Beschriftung ausgelassen, so wird die Kante wie mit 1 beschriftet behandelt und die Stelle muss leer sein. Entgegen einer normalen Kante entfernt sie keine Marken von der Stelle. Bei der grafischen Notation erhält die Kante einen leeren Kreis an Stelle der Pfeilspitze.

In Abbildung 2.3 ist Transition  $t_1$  aktiviert, da  $p_3$  wie verlangt mit mindestens einer Marke belegt ist und  $p_2$ , wie von der Inhibitorkante verlangt, leer ist. Mit dem Feuern

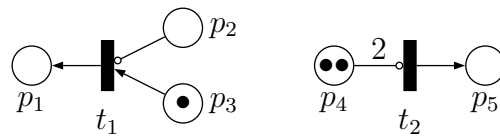


Abbildung 2.3: Inhibitorkanten

von  $t_1$  würde die Marke von  $p_3$  gelöscht und eine Marke auf  $p_1$  erzeugt werden. Transition  $t_2$  hingegen ist nicht aktiviert, da die mit 2 beschriftete Inhibitorkante von Stelle  $p_4$  verlangt, dass weniger als 2 Marken auf  $p_4$  liegen.

Während Inhibitorkanten, die von beschränkten Stellen wegführen durch weitere Stellen und normale Kanten simuliert werden können, führen Inhibitorkanten von unbeschränkten Stellen Turing-Vollständigkeit ein.

### 2.2.2 Lesekanten

Eine mit  $k$  beschriftete Lesekante *testet*, ob auf der entsprechenden Stelle mindestens  $k$  Marken liegen – durch die Lesekante werden die Marken nicht von der Stelle entfernt.

Im Grunde stellt eine mit  $k$  beschriftete Lesekante eine einfache Schlinge zwischen einer Transition  $t$  und einer Stelle  $s$  dar, wobei. Sie kann durch zwei mit  $k$  beschriftete Kanten, eine von  $t$  nach  $s$  und eine von  $s$  nach  $t$ , simuliert werden.

Unterschiede im Verhalten eines Netzes mit zwei normalen Kanten und einem mit Lesekanten ergeben sich nur, wenn man paralleles Feuern von Transitionen erlaubt.

Grafisch wird eine Lesekante als Kante ohne Pfeilspitze notiert.

### 2.2.3 Resetkanten

Um eine Stelle mit dem Feuern leer zu räumen, wenn die tatsächliche Anzahl Marken nicht interessiert, können Resetkanten verwendet werden. Sie löschen die verbleibenden Marken von der entsprechenden Stelle und haben selbst keinen Einfluss auf das Aktiviertsein einer Transition.

In dieser Arbeit werden Resetkanten durch doppelte Pfeilspitzen dargestellt.

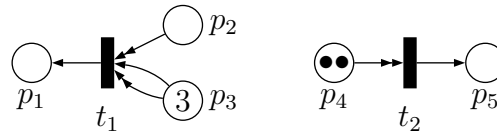


Abbildung 2.4: Resetkanten

Im Netz in Abbildung 2.4 kann Transition  $t_1$  einmal feuern: da  $p_2$  leer ist, hat die Resetkante von dort keine Arbeit zu verrichten, die normale Kante entfernt eine Marke von  $p_3$  und die Resetkante die restlichen beiden. Auf  $p_1$  wird eine Marke erzeugt. Transition  $t_2$  kann beliebig oft feuern: beim ersten Mal löscht die Resetkante beide Marken von  $p_4$ .

## 2.3 Stellen mit Kapazitäten

Eine Erweiterung von Petri-Netzen ist es, die Stellen, die ja unbegrenzt viele Token enthalten dürfen, künstlich zu beschränken. Stattet man eine Stelle mit einer Kapazität  $k$  aus, so können nur Token auf der Stelle erzeugt werden, wenn nach dem Erzeugen höchstens  $k$ -viele Token auf der Stelle liegen. Ist das nicht der Fall, so kann die entsprechende Transition nicht feuern.

## 2.4 Netze mit Prioritäten

Durch das Hinzufügen von Prioritäten kann die Menge der Transitionen in Prioritätsklassen zerlegt und vorher bestehende Konflikte deterministisch aufgelöst werden. Innerhalb einer Prioritätsklasse wird die zu feuernde Transition aber wie bei Petri-Netzen ohne Prioritäten nichtdeterministisch ausgewählt. Prioritäten können als globale oder als lokale Eigenschaften aufgefasst werden. Bei globalen Prioritäten wird eine Transition mit Priorität  $k$  nur dann gefeuert, wenn keine Transition mit Priorität echt größer als  $k$  hat aktiviert ist. Bei lokalen Prioritäten wird eine Transition mit Priorität  $k$  nur dann gefeuert, wenn keine Transition mit der sie in einem Konflikt steht und die eine Priorität echt größer als  $k$  aktiviert ist.

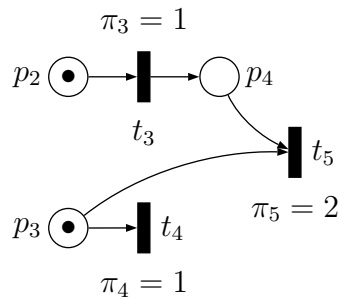


Abbildung 2.5: Ein indirekter Konflikt

In Petrinetzen mit Prioritäten kann es zu sogenannten indirekten Konflikten zwischen Transitionen kommen. Abbildung 2.5 zeigt einen indirekten Konflikt zwischen Transitionen  $t_3, t_4$ . Im Petrietz stehen  $t_3$  und  $t_4$  in Konflikt, da durch das Feuere von  $t_3$  Transition  $t_5$  aktiviert und gefeuert wird, die eine höhere Priorität  $\pi_5 = 2$  als  $t_4$  mit  $\pi_4 = 1$  hat. Durch das Schalten von  $t_3$  wird also eine Serie von Transitionen mit höheren Prioritäten gefeuert, die dazu führen, dass  $t_4$  nicht mehr aktiviert ist. [Bal01] [MBC<sup>+</sup>95]

Auf indirekte Konflikte wird hier nicht weiter eingegangen.

## 2.5 Netze mit Zeit

Bezogen auf Petrinetze gibt es mehrere Zeitkonzepte: zeitbehaftete Stellen, zeitbehaftete Marken, zeitbehaftete Kanten und zeitbehaftete Transitionen. Bei allen diesen Konzepten wird darauf geachtet, die grundlegende Idee und das grundlegende Verhalten der Petrinetze beizubehalten. [Bal01]

In dieser Arbeit wird nur das gängigste Zeitkonzept, die zeitbehafteten Transitionen mit atomarem Feuere betrachtet – im Gegensatz zum dreiphasigen Feuere: 1. Marken konsumieren, 2. Zeit verstreichen lassen, 3. Marken erzeugen. Netze mit zeitbehafteten Transitionen werden *Timed Transition Petri Nets* (TTPN) genannt und erlauben ähnlich den Prioritäten (siehe Abschnitt 2.4 *Netze mit Prioritäten* ab Seite 6) das deterministische Auflösen von Konflikten.

Zeitbehafteten Transitionen verfügen über eine Uhr, die bei der Aktivierung der Transition auf 0 gesetzt wird und mit einer bestimmten Geschwindigkeit erhöht wird. Ist

der bestimmte Wert  $\tau$  erreicht, so feuert die Transition und ihre Uhr wird zurück auf 0 gesetzt. Die Transition muss vor dem Feuern also  $\tau$  Zeiteinheiten lang *warten* bevor sie feuern kann.

Ein allgemeineres Zeitkonzept für Transitionen entsteht, wenn statt eines Wertes  $\tau$  ein Intervall  $[\tau_1, \tau_2]$  verwendet wird: die Transition *muss* irgendwann nach verstreichen einer zufälligen Zeit zwischen  $\tau_1$  und  $\tau_2$  Zeiteinheiten feuern (striktes Zeitkonzept). Bei einem schwachen Zeitkonzept *kann* die Transition ihr Zeitfenster, in dem sie feuern kann auch verstreichen lassen, kann dann jedoch nicht mehr schalten. Setzt man  $\tau_1 = \tau_2$ , so erhält man das erstgenannte Zeitkonzept mit einem einzigen Wert  $\tau$ .

Zeitfreie Transitionen können durch das Intervall  $[0, \infty]$  durch zeitbehaftete modelliert werden.

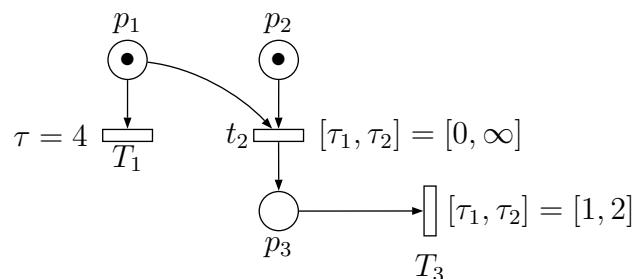


Abbildung 2.6: Ein TTPN

In Abbildung 2.6 könnte die zeitbehaftete Transition  $T_1$  feuern, nachdem sie 4 Zeiteinheiten lang aktiviert war, wäre da nicht die Transition  $t_2$ , die schon zum Zeitpunkt 0 feuerbar ist. Sie konsumiert die Marken von Stellen  $p_1$  und  $p_2$  und erzeugt eine Marke auf Stelle  $p_3$ . Damit sind  $T_1$ ,  $t_2$  deaktiviert und  $T_3$  aktiviert. Da in diesem Beispiel das strikte Zeitkonzept gelte, feuert Transition  $T_3$  irgendwann zwischen 1 und 2 Zeiteinheiten im aktivierten Zustand. Aus dem Anfangszustand könnte die Uhr auch bis zum Zeitpunkt 4 fortschreiten, zu dem Transition  $T_1$  gefeuert werden könnte. Danach können keine weiteren Schaltvorgänge im Netz stattfinden.

### 2.5.1 Memory Policies

Dafür, wann und wie eine Transition Zeit verbraucht, werden in [Bal01] und [MBC<sup>+</sup>95] drei Strategien vorgestellt:

**Resampling:** Nach dem Feuern einer Transition werden die Uhren aller Transitionen zurückgesetzt. Da diese Strategie normalerweise von weniger praktischen Nutzen ist, wird auf sie in dieser Arbeit nicht weiter eingegangen.

**Enabling Memory:** Wenn eine Transition deaktiviert wird, also ihre Aktivierung verliert, so wird ihre Uhr zurückgesetzt. Die Uhr misst dann die Dauer der längsten durchgehenden Aktivierung.

**Age Memory:** Im Gegensatz zur Enabling-Memory-Strategie *vergisst* eine Uhr nicht, wenn bereits Zeit verstrichen ist. Stattdessen läuft die Uhr immer dann weiter, wenn die Transition aktiviert ist und wird erst beim Feuern zurückgesetzt. Die Uhr misst dann die kumulierte Dauer der Aktivierungen.

## 2.5.2 Server-Semantiken

Im Falle, dass eine Transition mehr als ein Mal feuern kann, also mehrfach aktiviert ist, muss bei TTPN ein gewünschtes Verhalten für die Transition ausgewählt werden.

In [Bal01] und [MBC<sup>+</sup>95] werden folgende drei Semantiken unterschieden:

**Single-Server-Semantik:** Mehrfache Aktivierungen werden *nacheinander* behandelt. Die Transition hat eine einzige Uhr, die für die erste Aktivierung läuft, bis sie gefeuert wird. Danach wird die Uhr für die nächste mögliche Aktivierung zurück gesetzt.

**Infinite-Server-Semantik:** Die Transition behandelt alle Aktivierungen *gleichzeitig*, beziehungsweise sobald eine neue Aktivierung entsteht. Für jede Aktivierung wird eine Uhr initialisiert, die dann parallel zu allen weiteren Uhren läuft.

**Multiple-Server-Semantik:** Bei einer  $k$ -Server-Semantik werden höchstens  $k$  Aktivierungen *gleichzeitig* behandelt. Es stehen  $k$  Uhren bereit, die parallel für höchstens  $k$ -viele Aktivierungen laufen. Wenn alle Uhren in Verwendung sind, warten überzählige Aktivierungen, bis die Transition feuert und eine Uhr frei wird.

## 2.6 Stochastische Netze

Wird das Zeitintervall, in dem eine Transition feuern kann durch eine Zufallsvariable bestimmt, oder werden die Prioritäten durch Zufallsvariablen beeinflusst, so spricht man von stochastischen Petrinetzen (SPN).

Besonders von Interesse ist hier die Exponentialverteilung  $\lambda \cdot e^{-\lambda x}$ , wobei  $\lambda = \frac{1}{\mu}$  und  $\mu$  der Erwartungswert ist. Ihr Vorteil ist die Gedächtnislosigkeit: Zustandswechsel und das Verstreichen von Zeit beeinflussen die Verteilung nicht.

Einer zeitbehafteten Transition  $T_i$  wird eine feste Rate  $\mu_i$  zugeordnet. Die Verzögerung  $\tau_i$  der Transition  $T_i$  wird dann als exponentialverteilte Zufallsvariable mit Mittelwert  $\mu_i$  aufgefasst. Immer wenn die Uhr der Transition zurückgesetzt wird, wird auch ein entsprechender Zufallswert für die Verzögerung bestimmt.

Typischerweise gibt es in SPN nur zeitbehaftete Transitionen und keine zeitfreien.

Da die exponentialverteilte Zufallsvariable gedächtnislos ist, kann die in Abschnitt 2.5 *Netze mit Zeit* ab Seite 7 beschriebene Unterscheidung zwischen Resampling, Enabling Memory und Age Memory vernachlässigt werden.

## 2.7 Verallgemeinerte stochastische Netze

Da bei den SPN alle Transitionen Zeit verbrauchen, taucht das Problem auf, das sich der *Kontrollfluss* kaum voraussehen lässt. Durch die Einführung von zeitfreien Transitionen mit Prioritäten verallgemeinert man das Konzept der SPN zu den Generalized Stochastic Petri Nets (GSPN). Zeitfreie Transitionen haben eine höhere Präzedenz als zeitbehaftete Transitionen. Um diese beiden Typen unterscheiden zu können werden zeitbehaftete Transitionen als Rechtecke dargestellt.

In Abbildung 2.7 könnte die zeitbehaftete Transition  $T_1$  feuern, nachdem sie für eine zufällige Dauer (exponentialverteilte Zufallsgröße mit Erwartungswert  $\mu_{T_1} = 4$ ) aktiviert war, wäre da nicht die zeitfreie Transition  $t_2$ , die in jedem Fall vor der zeitbehafteten schaltet. Ihr Gewicht und ihre Priorität wurden in diesem Beispiel zwar festgelegt, sind aber unerheblich, da sie die einzige zeitfreie Transition ist. Sie konsumiert die Marken von Stellen  $p_1$  und  $p_2$  und erzeugt eine Marke auf Stelle  $p_3$ . Damit sind  $T_1, t_2$  deaktiviert

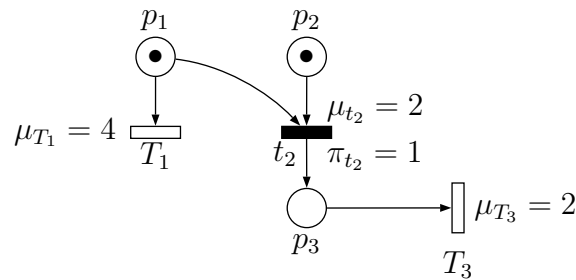


Abbildung 2.7: Ein GSPN

und  $T_3$  aktiviert. Wie  $T_1$  feuert Transition  $T_3$  nach einer zufälligen Dauer im aktivierten Zustand, nur dass ihre Rate  $\mu_{T_3} = 2$  ist.

Markierungen, die mit dem Schalten einer zeitfreien Transition wieder verlassen werden (ohne das Zeit verstrichen ist), werden *verschwindende* (vanishing) Markierungen genannt. Solche, in denen Zeit verstreicht, werden *greifbare* oder *konkrete* (tangible) Markierungen genannt.

Zeitfreie Transitionen werden neben Prioritäten noch mit einem Pendant zu den Raten zeitbehafteter Transitionen – den Gewichten – ausgestattet. Stehen zwei zeitfreie Transitionen gleicher Prioritätsklasse im Konflikt, so wird für beide ein Zufallswert ihrer Gewichte bestimmt und die Transition mit höherem Wert ausgewählt. Die Gewichte sind genau wie die Raten Mittelwerte für exponentialverteilte Zufallsvariablen.

Da die exponentialverteilte Zufallsvariable gedächtnislos ist, kann wie bei den SPN die in Abschnitt 2.5 *Netze mit Zeit* ab Seite 7 beschriebene Unterscheidung zwischen Resampling, Enabling Memory und Age Memory vernachlässigt werden.

## 2.8 Gefärbte Netze

Mit steigender Komplexität der modellierten Systeme können die Petrinetze größer und ebenfalls komplexer werden. Gefärbte Petrinetze (auch im weiten Begriff der High-Level-Petrinetze enthalten) können dem entgegenwirken, indem sie bestimmte gleichartige Netzelemente zusammenfassen.

Das Zusammenfassen macht es notwendig, dass Marken unterschieden werden können. Es können an Stelle von  $\bullet$  zum Beispiel die natürlichen Zahlen als Token benutzt wer-

den (M-Netze), prinzipiell sind aber alle Typen von Token möglich (gängig sind unter anderem Teilmengen von  $\{\bullet, \text{true}, \text{false}\} \cup \mathbb{N}$  und Tupel über solchen Mengen).

Kanten, die von oder zu einer Stelle vom Typ  $V$  führen, haben keine Multiplizitäten (siehe Abschnitt 2.1 *P/T-Netze* ab Seite 2), können aber dafür mit Multimengen (bei Resetkanten: Mengen) von Werten in  $V$  und Variablen beschriftet sein. An einer Inhibitor-kante die von einer Stelle  $s$  mit Typ  $\{A, B, C\}$  wegführt, bedeutet eine Beschriftung  $A, A, B$ , dass  $s$  weniger als zwei  $A$ s und kein  $B$  enthalten darf, über  $C$  wird keine Aussage gemacht.

Ist  $V$  eine Multimenge, so sei  $|V \cap \{v\}|$  die Anzahl der Vorkommen von  $v$  in  $V$ .

Transitionen können nun mit verschiedenen Variablenbelegungen (Bindings) aktiviert sein. Beispielsweise ist Transition  $t_1$  in Abbildung 2.8 unter der Variablenbelegung  $x = \bullet$ , wie auch unter  $x = 5$  aktiviert. Beim Schalten von  $t_1$  würde der jeweilige Wert von Stelle  $p_1$  entfernt und auf  $p_2$  erzeugt. Im Beispiel feuert  $t_1$  erst mit  $x = 5$  und dann mit  $x = \bullet$ .

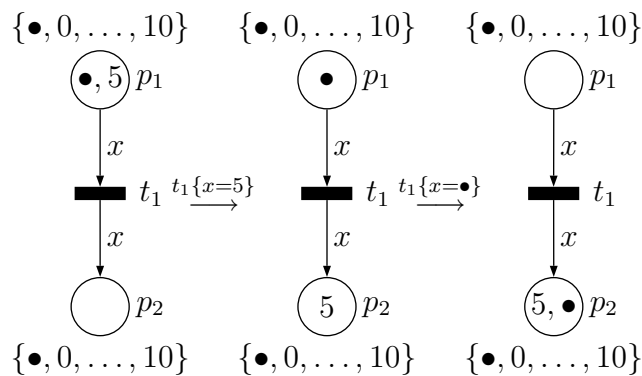


Abbildung 2.8: Verschiedene Variablenbelegungen

In High-Level-Petrinetzen können Transitionen mit Ausdrücken ausgestattet werden, die alle Variablen aus dem Kontext der Transition enthalten können:

**logische Ausdrücke:** Wächter: Der Wächter ist ein boolescher Ausdruck, in dem prinzipiell beliebige Berechnungen vorgenommen werden können. Kann er für eine gültige Variablenbelegung zu *wahr* ausgewertet, so ist die Transition unter der Belegung aktiviert. Wird der Ausdruck zu *falsch* ausgewertet, kann die Transition nicht schalten.

In einem Wächter können weitere Variablen gebunden werden – das heißt ihre

Werte werden berechnet. Abbildung 2.9 stellt ein Beispiel dar, bei dem nach Feuern von  $t_1$  mit Binding  $\{x = 4, y = 9\}$  und mit Binding  $\{x = 5, y = 11\}$  Stelle  $p_1$  leer ist und  $p_2$  mit 9, 11 belegt ist.

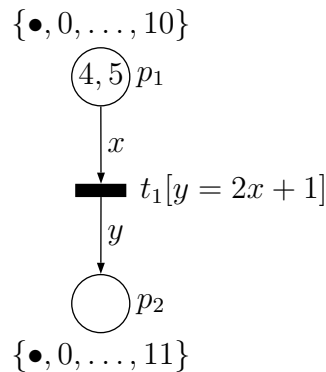


Abbildung 2.9: Wächter

**arithmetische Ausdrücke:** Priorität und Gewicht bei zeitfreier, beziehungsweise Intervallgrenzen oder Rate bei zeitbehafteter Transition: Nach der Auswertung des Wächters stehen alle Variablenbelegungen fest und die numerischen Ausdrücke können vollständig ausgewertet werden.

Gefärbte Petrinetze sind ebenso mächtig wie P/T-Netze. Allerdings wird die Modellierung von komplexen Sachverhalten deutlich erleichtert und die Netze werden übersichtlicher. Im Abschnitt 5.8 *Entfaltung* ab Seite 50 wird dies durch Algorithmen, mit dem High-Level-Petrinetze in semantisch äquivalente Low-Level-Petrinetze übersetzt werden können, verdeutlicht.

## 2.9 Simulation von Petrinetzen

Auf bestimmte Typen von Petrinetzen kann Verifikation betrieben werden. So kann man beschränkte P/T-Netze gegen die Linear Temporal Logic (LTL) und gegen die Computational Tree Logic (CTL) model-checken. Auch linear-algebraische und stochastische Methoden existieren für diese einfachen Netze. Führt man allerdings bestimmte Erweiterungen ein, zum Beispiel Prioritäten, Inhibitorkanten, Resetkanten oder Zeit ein, so wird die betrachtete Netzklasse Turing-vollständig und lässt kein allgemeines Model-Checking mehr zu.

In Situationen, in denen eine Analyse des Petrinetzes schlecht oder gar nicht durchführbar ist, bietet sich die Simulation des Verhaltens des Netzes an. Mit steigender Komplexität der zu simulierenden Petrinetzklasse werden die Berechnungen des Simulators üblicherweise aufwändiger. Dennoch liegt der Speicher- und Zeitbedarf bei der Simulation meist unter dem der Analyse, da nicht alle möglichen Zustände des Petrinetzes, sondern nur der aktuelle, betrachtet werden müssen. [Smi98] [?]

Die Simulation eines Petrinetzes kann automatisch oder interaktiv geschehen. Bei der automatischen Simulation kann geprüft werden, ob ein bestimmter Zustand des Netzes in dem bestimmten Simulationslauf zufällig erreicht wird. Es ist keine erschöpfende Analyse, aber bietet Anhaltspunkte für Eigenschaften des Petrinetzes. Bei der interaktiven Simulation kann dem Benutzer vor jedem Feuern die Menge der aktivierten Transitionen (eventuell mit Variablenbelegungen) ausgegeben werden, aus welcher er dann eine auswählen kann, die schalten soll. Anfragen über den aktuellen Zustand des Petrinetzes werden entsprechend beantwortet. Beispielsweise wird im Werkzeug der Projektgruppe P-UMLaut [PUM] ein Petrinetz-Simulator zur interaktiven Steuerung von Ereignissen in einer 3D-Visualisierung eingesetzt.

# 3 Konzept

„Alles Große in unserer Welt geschieht nur, weil jemand mehr tut, als er muss.“

Hermann Gmeiner

Konzipiert wurde der P-UMLaut/Sim im direkten Zusammenhang mit P-UMLaut, da dort ein ausreichend schneller Simulator mit klaren Anforderungen an die Funktionalität benötigt wurde. Der P-UMLaut/Sim sollte den Simulator des PEP-Tools, den PEP-NetSim, ablösen (siehe dazu auch Abschnitt 6.1 *PEP-NetSim* ab Seite 54).

## 3.1 Ursprüngliche Anforderungen

Für P-UMLaut wurde ein Simulator mit folgenden Eigenschaften gefordert:

**Geschwindigkeit:** Der vorher angebundene, projektexterne PEP-NetSim verursachte deutliche Verzögerungen in der interaktiven Visualisierungskomponente des Werkzeuges. Ein schnellerer, in Java programmierter, Simulator sollte angebunden werden.

**Petrinetzklassen:** Es musste eine um Wahrheitswerte und schwarze Token erweiterte Klasse der M-Netze simuliert werden können.

**Kanten:** Lediglich normale Kanten mussten unterstützt werden.

**Transitionen:** Wächter-Terme mussten ausgewertet werden können. Weder Zeit, noch Prioritäten, noch stochastische Elemente wurden benötigt.

**Stellen:** Es mussten keine Erweiterungen unterstützt werden.

**Schnittstellen:** Der Simulator sollte eine Benutzerinteraktion zulassen, so dass es Operationen geben musste, über die die Menge der aktivierten Transitionen und ihrer Variablenbelegungen ausgegeben und solche, mit denen die Simulation gesteuert werden konnte – also eingegeben werden konnte, welche Transition mit welcher Variablenbelegung feuern sollte.

## 3.2 Idee und Methodik

Der ursprüngliche Simulator war in Java 1.4 implementiert und bot nicht viel mehr Funktionalität als die Anforderungen verlangten. Um ihn zur Simulation von GSPN einsetzen zu können, sollen die Erweiterungen und die Eigenschaften der GSPN schrittweise implementiert werden. Die Zerlegung in einzelne Erweiterungen findet sich in der Struktur von Kapitel 5 *Integration der Erweiterungen* ab Seite 36 wieder:

### Stellen

- Kapazitäten\*
- Ungetypte Stellen\*

### Transitionen

- Zeit
- Memory Policies
- Server-Semantiken
- Raten
- Gewichte

### Kanten

- Inhibitorkanten
- Lesekanten\*
- Resetkanten\*

---

\*Diese Erweiterungen sind nicht zur Simulation von GSPN notwendig, erleichtern aber die Modellierung bestimmter Systeme.

Zudem sollte von den mit Java 1.5 eingeführten Konzepten der *enums* (aufzählbare endliche Typen) und der *Generics* (generische Typen – *Platzhalter*) Gebrauch gemacht werden, um mögliche (Typ-) Fehler bei der Programmierung früher finden und beheben zu können.

### 3.3 Schnittstellen

Die Schnittstellen der ursprünglichen Version sollten erhalten bleiben, so dass die Funktionalität von P-UMLaut nicht eingeschränkt wird.

Die Eingabe der Petrinetze sollte mittels der `PEPReader`- und `PNMLReader`-Klassen erfolgen, die verändert wurden, so dass sie alle Erweiterungen der Petrinetze einlesen können. Analog dazu sollte die Ausgabe von Petrinetzen mittels der `PEPWriter`- und `PNMLWriter`-Klassen erfolgen, die verändert wurden, so dass sie alle Erweiterungen der Petrinetze schreiben können. [EFH<sup>+</sup>05]

# 4 Implementierung der Basisfunktionalität

„Holzhacken ist deshalb so beliebt, weil man bei dieser Tätigkeit den Erfolg sofort sieht.“

Albert Einstein

In diesem Abschnitt wird auf Implementierung und Verhalten des Simulators eingegangen. Es werden die Funktionalität und die Details der Behandlung von High-Level-Petri-Netzen, sowie die Grammatik der vom Simulator verstandenen mathematischen Ausdrücke vorgestellt.

Grundlegende Algorithmen zur Vereinfachung der bei einer Simulation anfallenden Berechnungen werden ebenfalls eingeführt und im folgenden Kapitel 5 *Integration der Erweiterungen* ab Seite 36 genauer betrachtet.

Teile dieses Kapitels sind vom Autor während der Arbeit an der Projektgruppe P-UMLaut [EFH<sup>+</sup>05] erstellt worden und liegen hier in aktualisierter Form vor.

## 4.1 Grundlegendes

Mit dem im Rahmen der Projektgruppe P-UMLaut entwickelten `PNModel`-Paket steht dem Simulator ein Backend zur Verfügung, über das High-Level- wie auch Low-Level-Petrinetze eingelesen und im Speicher aufgebaut werden können. Low-Level-Petrinetze werden allerdings als High-Level-Petrinetze aufgefasst und geladen, so dass P-UMLaut/Sim nicht zwischen diesen Klassen unterscheiden können muss.

Das `PNModel`-Paket bietet viele Methoden zum direkten und möglichst schnellen Zugriff auf alle Elemente von Petri-Netzen, die im Simulator verwendet werden. In Unter-

abschnitt 7.1.2 *Performance Tuning* ab Seite 60 werden mögliche Verbesserungen der Datenstrukturen angesprochen.

Eine Einführung in die Funktionsweise des Simulators ohne Erweiterungen und in das angebundene `PNModel`-Paket ist auch in [EFH<sup>+</sup>05] zu finden. Im Folgenden wird genauer auf die verwendeten Algorithmen eingegangen, die später in Bezug auf die neuen Erweiterungen verfeinert werden.

## 4.2 Datenstrukturen

Aus Gründen der Effizienz sind die vom P-UMLaut/Sim benutzten Datenstrukturen mit denen aus dem `PNModel`-Paket eng verzahnt. Die hier beschriebenen Datenstrukturen werden zur Funktion des `PNModel`-Paketes nicht wirklich gebraucht, sie wurden zur Erleichterung der Simulation eingefügt. Im Rahmen der Umstellung von Java 1.4 auf Java 5 wurden alle Datenstrukturen überarbeitet.

### 4.2.1 Wertetypen

In der Werteklasse werden die Typen *Zahl*, *Zeichenkette*, *ungetypte Variable*, *Wahrheitswert*, *schwarzes Token* und *Tupel von Werten* zusammengefasst. Beim Erzeugen einer neuen Instanz entscheidet die Klasse selbst, welcher Wertetyp vorliegt. Instanzen dieser Werteklasse werden auf den Stellen als Belegung in Form einer Multimenge gehalten und vom Syntaxbaumrechner verarbeitet und als Ergebnis der Auswertung von Termen zurückgeliefert (siehe Unterabschnitt 4.2.3 *Syntaxbaumrechner* ab Seite 20).

### 4.2.2 Variablenbelegung

Im Grunde ist eine Variablenbelegung nicht mehr als eine Menge von Assoziationen zwischen Variablenbezeichnern und Werten. Erweitert man diese Definition, so dass die Ergebnisse der Auswertungen der zur Transition gehörenden Terme mit gespeichert werden, so wird die mehrfache Auswertung ein und desselben Ausdruckes unnötig. Zusätzlich erhält die `Binding`-Klasse die in Abschnitt 4.4 *Behandlung logischer/arithmetischer Ausdrücke* ab Seite 20 genannten Assoziationen zwischen einer Variablen und einer Menge

von möglichen Werten, die die Variable annehmen könnte.

### 4.2.3 Syntaxbaumrechner

Der Syntaxbaumrechner kann aus einem Ausdruck in umgekehrter polnischer Notation, also einem Ausdruck in Postfix-Schreibweise, einen semantisch äquivalenten Syntaxbaum aufbauen. Die entstandenen Syntaxbäume können vom Rechner vereinfacht werden, so dass sie mit weniger durchzuführenden Operationen auszuwerten sind.

## 4.3 Grammatik logischer/arithmetischer Ausdrücke

Die Grammatik der vom Simulator verstandenen Ausdrücke ist recht flexibel. Variablen sind ungetypt. Als Literale können Fließkommazahlen, `tt` für *wahr*, `ff` für *falsch* und in " eingefasste Zeichenketten (`\`" und `\\` sind die Escape-Sequenzen für " und `\`) verwendet werden. In Tabelle 4.1 werden die verfügbaren Operatoren aufgelistet.

Tupel werden wie gewöhnlich gebildet: `(A, B, 3, (23, C))`). Mit den eckigen Klammern, zum Beispiel im Ausdruck `X[i]`, wird eine Kurzschreibweise der Funktion `proj(X, i)` angeboten, wie sie in typischen Programmiersprachen zur Indizierung von Feldern vorkommt. Alle aufgelisteten Operatoren können in Präfix-Schreibweise verwendet werden, beispielsweise entspricht `+(4, 3, 2, 1)` dem Ausdruck `4 + 3 + 2 + 1`.

Da diese Grammatik nicht eindeutig ist, wurde statt eines automatischen Parsergenerators, wie ANTLR [ANT] oder dem mächtigeren CUP [CUP], ein eigener Parser geschrieben, der einen Term erst in die umgekehrte polnische Notation und weiter in einen Syntaxbaum übersetzt. Die so erzeugten Syntaxbäume müssen dann zwar nicht eindeutig sein, aber zur Laufzeit, wenn die Terme ausgewertet werden, entscheidet der Syntaxbaumrechner, wie mit auftretenden Mehrdeutigkeiten umzugehen ist.

## 4.4 Behandlung logischer/arithmetischer Ausdrücke

Im hier betrachteten Simulator werden verschiedene Auswertungen und Optimierungen an den Termen der Transitionen eines Netzes vorgenommen. Durch eine Extraktion von

Symbol	Operanden	Assoziativität	Präzedenz	Bedeutung
-	1	R	8	unäres Minus
!	1	R	8	logisches Nicht
~	1	R	8	bitweises Nicht
**	2	L	7	Potenz
*	2	L	6	Multiplikation
/	2	L	6	Division
%	2	L	6	Modulo
+	2	L	5	Addition
+	2	L	5	String-Konkatenation
-	2	L	5	Subtraktion
<	2	L	4	Kleiner
>	2	L	4	Größer
<=	2	L	4	Kleiner gleich
>=	2	L	4	Größer gleich
=	2	L	3	Gleichheit
#	2	L	3	Ungleichheit
&	2	L	2	logisches Und
&	2	L	2	bitweises Und
	2	L	1	logisches Oder
	2	L	1	bitweises Oder
^	2	L	1	logisches exklusives Oder
^	2	L	1	bitweises exklusives Oder

Tabelle 4.1: Unterstützte Operatoren

Einschränkungen der Wertemengen von Variablen und die Vereinfachung der Terme sollen redundante Informationen vermieden und ein schnelles Ausrechnen der Terme ermöglicht werden.

Wie das PEP-Tool, erlaubt es der P-UMLaut/Sim Terme um Einschränkungen von Variablen zu erweitern. Die Syntax dazu in BNF: [`<VARIABLE><STELLENTYP> {'&'} <VARIABLE><STELLENTYP> ':'`] `<TERM>`. Dabei beschreibt ein Stellentyp eine Wertemenge, etwa `{dot,true,false,2..10}`.

Beispielsweise sollen Terme folgendermaßen vereinfacht werden:

- $A\{0..10\} \& B\{5..15\} \& X\{\text{true},\text{false},0..2\} \& Y\{2..5\} : (A=B \& X) \mid (A=B \& X=Y)$   
zu  $A\{5..10\} \& B\{5..10\} \& X\{\text{true},2\} \& Y\{2..5\} : (A=B \& X) \mid (A=B \& X=Y)$
- $A\{0..10\} \& B\{5..15\} : A \leq 5 \& A=B$   
zu  $A\{5\} \& B\{5\} : \text{true}$
- $A\{0..10\} \& B\{5..15\} \& X\{\text{true},\text{false},0..2\} \& Y\{2..5\} : A=0 \& B=5 \& X=Y \& Y \neq 5$   
zu  $A\{0\} \& B\{5\} \& X\{2\} \& Y\{2\} : \text{true}$

Wie beim letzten Beispielterm zu sehen ist, können bei günstigen Wertemengen manche Terme vollständig abgebaut werden und die erlaubten Variablenbelegungen ganz aus dem Term ausgelesen werden. Um diese Optimierungen umzusetzen, werden drei Schritte wiederholt ausgeführt, bis keine Änderungen mehr vorgenommen werden:

### 1. Einschränken der Wertemengen der vorkommenden Variablen

Um die Wertemengen stärkstmöglich einzuschränken, wird der Term für jede Variable einmal durchlaufen. Als Eingabe erhält der Algorithmus den Syntaxbaum des Terms, den Variablennamen und die Wertemengen der Variablen. Rückgabe ist dann eine Teilmenge der Wertemenge der Variablen.

Der Algorithmus traversiert rekursiv durch den Syntaxbaum und unterscheidet anhand des aktuellen Knoteninhalts, welche der unten aufgeführten Regeln anzuwenden ist. Da es einige Sonderfälle gibt, wird der Algorithmus hier nur kurz umrissen. Der Simulator könnte hier um weitere Optimierungen ergänzt werden.

**Hat der Knoten weitere Kinder:** Rufe den Algorithmus rekursiv für die Kinder auf und merke die Ergebnismengen der Aufrufe.

**Ist dieser Knoten ein |:** Liefere die Vereinigung der Ergebnismengen zurück.

**Ist dieser Knoten ein &:** Liefere den Schnitt der Ergebnismengen zurück.

**Ist dieser Knoten ein !:** Liefere das Komplement der Ergebnismenge bezüglich der ursprünglichen Wertemenge zurück.

**Ist dieser Knoten ein =:** Unterscheide die Fälle

**Der Teilterm ist nicht erfüllbar:** Liefere die leere Menge zurück.

**Der Teilterm ist erfüllt:** Liefere die alte Wertemenge der Variablen zurück.

**Der Teilterm entspricht  $[VAR1]=[VAR2]$ :** Wenn eine der beiden Variablen die gesuchte ist, liefere den Schnitt der alten Wertemenge und der Wertemenge der zweiten Variablen zurück.

**Der Teilterm entspricht  $[VAR]=[WERT]$ :** Wenn die Variable die gesuchte ist, liefere den Schnitt der alten Wertemenge und von  $\{[WERT]\}$  zurück.

**Ist dieser Knoten ein  $\neq$ :** Unterscheide die Fälle

**Der Teilterm ist nicht erfüllbar:** Liefere die leere Menge zurück.

**Der Teilterm ist erfüllt:** Liefere die alte Wertemenge der Variablen zurück.

**Der Teilterm entspricht  $[VAR1]\#[VAR2]$ :** Wenn eine der beiden Variablen die gesuchte ist, liefere die Differenz der alten Wertemenge und der Wertemenge der zweiten Variablen zurück.

**Der Teilterm entspricht  $[VAR]\#[WERT]$ :** Wenn die Variable die gesuchte ist, liefere die Differenz der alten Wertemenge und von  $\{[WERT]\}$  zurück.

**Ist dieser Knoten ein  $<$ :** Unterscheide die Fälle

**Der Teilterm ist nicht erfüllbar:** Liefere die leere Menge zurück.

**Der Teilterm ist erfüllt:** Liefere die alte Wertemenge der Variablen zurück.

**Der Teilterm entspricht  $[VAR1]<[VAR2]$ :** Wenn eine der beiden Variablen die gesuchte ist, liefere die alte Wertemenge zurück, aus der alle Werte entfernt wurden, die entweder keine Zahlen oder zu groß (zu klein) sind. Betrachte dazu die größte beziehungsweise die kleinste Zahl der Wertemenge der zweiten Variablen.

**Der Teilterm entspricht  $[VAR]<[WERT]$ :** Wenn die Variable die gesuchte ist, liefere die alte Wertemenge zurück, aus der alle Werte entfernt wurden, die entweder keine Zahlen oder zu groß (zu klein) sind.

**Ist der Knoten eines aus  $>$ ,  $<=$ ,  $>=$ :** Analog zu  $<$ .

**Ist dieser Knoten anders:** Liefere die alte Wertemenge der Variablen zurück.

## 2. Ersetzen der Variablen bei eindeutiger Belegung:

Enthalten Wertemengen von Variablen nach dem Einschränken nur einen Wert, so werden alle Vorkommen dieser Variablen im Term der Transition durch den entsprechenden Wert ersetzt.

### 3. Sinnvolles Löschen von Teiltermen, die zu `true` ausgewertet werden

Um Terme zu vereinfachen, werden sie nach dem Einschränken der Variablen wiederholt durchlaufen und Teilterme, die in Bezug auf die Wertemengen der Variablen immer zu `true` ausgewertet werden, durch `true` ersetzt. In der Praxis ist es nicht praktikabel hier einen Brute-Force-Algorithmus anzuwenden. Stattdessen sollte ein Algorithmus benutzt werden, der Teilterme löscht, bei denen offensichtlich ist, dass sie nach dem Einschränken zu `true` ausgewertet werden. Ein solcher Algorithmus könnte beispielsweise alle Teilterme entfernen, die nicht innerhalb von Operanden eines *oder*-Operatoren auftreten.

Beispielsweise wird der Term  $A\{0..9\} : A \geq 5$  im ersten oben erläuterten Schritt zu  $A\{5..9\} : A \geq 5$  und beim Vereinfachen der Teilterme zu  $A\{5..9\} : \text{true}$ . Es werden also Einschränkungen aus dem Term in die Wertemengen der Variablen (die in jedem Fall aufgebaut werden) übernommen.

Damit diese Informationen nicht doppelt vorliegen, oder sogar mehrfach ausgewertet werden müssen, werden sie aus dem Term gestrichen.

### 4. Wenn eine Änderung des Terms erfolgte, wiederholen ab 1.

Dieser Algorithmus wird beim Laden des Petrinetzes für alle vorkommenden Wächter-Terme angewendet.

## 4.5 Gefärbte Petrinetze

Die grundlegende Funktionsweise des Simulators kann in folgende Schritte zerlegt werden, da, wie in Kapitel 4 *Implementierung der Basisfunktionalität* ab Seite 18 beschrieben, Low-Level-Petrinetze intern wie High-Level-Petrinetze behandelt werden.

- 1. Petrinetz laden:** Ein bestimmtes Petrinetz wird in den Simulator geladen.
- 2. Statische Optimierung:** Möglichst viele Informationen werden im Voraus ausgewertet. Bei diesem Simulator werden folgende Optimierungen an dem zu simulierenden Netz vorgenommen:
  1. Es wird geprüft, ob alle Werte an Kanten auf die entsprechenden Stellen gelegt werden können, beziehungsweise ob die Werte von den entsprechenden Stellen

heruntergenommen werden können. Ist eines von beidem nicht möglich, so wird der Wächter-Term der (toten) Transition auf den Wert `false` gesetzt.

2. Für jede im Kontext (hier: Ein-, Ausgangskanten und Wächter-Term der Transition) einer Transition vorkommende Variable  $v$  wird eine endliche Menge  $C_v$  aufgebaut, die alle von  $v$  annehmbaren Werte enthält. Während der Berechnung von aktivierenden Variablenbelegungen (siehe Abschnitt 4.6 *Berechnung aktivierender Variablenbelegungen* ab Seite 28) wird auf diese Mengen zugegriffen. Je größer eine solche Menge ist, desto aufwändiger ist es, aktivierende Variablenbelegungen zu finden, da eventuell viele Kombinationen geprüft werden müssen. Deswegen wird versucht, diese Mengen schon vor der Simulation einzuschränken und auch die Terme, die wiederholt ausgewertet werden, zu vereinfachen.

Zu Beginn wird die Menge  $C_v$  einer Variablen (für Variablen in Tupeln kann ähnlich vorgegangen werden)  $v$ , die als Beschriftung an einer Kante vorkommt, als Schnitt aller Wertemengen der Typen, der mit den Kanten verbundenen Stellen festgelegt. Teilweise kann die Grundmenge  $C_v$  auch aufgebaut werden, wenn  $v$  nicht an einer Kante vorkommt. Die verschiedenen Kantentypen müssen unterschieden und entsprechend behandelt werden.

Nach Bestimmung von  $C_v$  wird versucht, den Term der Transition zu vereinfachen und eventuell  $C_v$  weiter einzuschränken. Dieser Schritt wird in Abschnitt 4.4 *logische/arithmetische Ausdrücke* ab Seite 20 genauer beschrieben und bietet einen erheblichen Geschwindigkeitsgewinn für den Simulator.

3. Gibt es bei einer Transition eine Variable  $v$ , so dass  $C_v = \emptyset$  ist, so wird ihr Term auf `false` gesetzt.
4. Alle Transitionen, deren Term `false` ist, werden aus dem Netz entfernt.
5. Tote Teile des Netzes werden entfernt. Dies kann wie folgt in Pseudo-Code beschrieben werden: Sei  $S$  die Menge der Stellen,  $T$  die Menge der Transitionen des Netzes und sei  $M(s)$  die Markierung der Stelle  $s$  (eine Multimenge).  $P$  ist die Menge der zu prüfenden Stellen.

```

1  $P := S$ 
2 for each  $s \in P$  do
3     // Entferne  $s$  aus  $P$ . Stelle  $s$  wird nun geprüft.
```

```

4   P := P \ {s}
5   if •s = ∅ ∧ M(s) = ∅ then
6       // Stelle s ist leer und wird niemals markiert
7       // Lösche alle Transitionen in s•
8       // (sie können niemals feuern).
9       T := T \ s•
10      // Prüfe alle Stellen der Nachbereiche der
11      // gelöschten Transitionen in s•.
12      P := P ∪ (s•)•
13  else if s• = •s = ∅ then
14      // Lösche die isolierte Stelle.
15      S := S \ {s}
16  fi
17 od

```

Dieser Algorithmus kann noch für weitere Kantentypen verändert werden. Siehe dazu Abschnitt 5.4 *weitere Kantentypen* ab Seite 39.

**3. Berechnen der aktivierten Transitionen:** Für jede Transition werden alle aktivierenden Belegungen berechnet, mit denen sie unter der aktuellen Markierung aktiviert ist. Siehe Abschnitt 4.6 *Berechnung aktivierender Variablenbelegungen* ab Seite 28.

**4. Ist keine Transition aktiviert, bricht die Ausführung des Simulators ab.**

**5. Feuern einer Transition:** Aus der Menge der aktivierten Transitionen des Netzes wird zufällig eine ausgewählt und mit einer aktivierenden Variablenbelegung (Binding) ein Schaltvorgang auf dem Netz ausgeführt. Mit dem Hinzufügen von Zeit und Prioritäten muss der Algorithmus zur Auswahl der zufeuernden Transition verändert werden.

**6. Bestimmen der zu prüfenden Transitionen:** Je nach Änderung der Netzmarkierung sind nur bestimmte Transitionen auf Aktivierung zu prüfen.

Ein Algorithmus zum Bestimmen der Menge kann wie folgt in Pseudo-Code ausgedrückt werden: Sei  $t$  die gefeuerte Transition,  $A$  die Menge der vor dem Feuern von  $t$  aktivierten Transitionen,  $\overline{A} = T \setminus A$  die Menge der vor dem Feuern von  $t$

nicht aktivierten Transitionen und  $P$  die aufzubauende Menge der zu prüfenden Transitionen.  $M'(s)$  sei die Belegung der Stelle  $s$  nach dem Feuern von  $t$  und  $M(s)$  die Belegung vor dem Feuern.

```

1  $P := \emptyset$ 
2 for each  $s \in t^\bullet \cup {}^\bullet t$  do
3   // für jede Stelle  $s$  des Vor- und Nachbereiches von  $t$ 
4   if  $s \in t^\bullet \setminus {}^\bullet t$  then
5     // Stelle  $s$  ist nur im Nachbereich von  $t$ , also
6     //  $M(s) \subset M'(s)$ . Menge  $P$  wird um die Transitionen
7     // aus  $s^\bullet$  erweitert, da Bindings hinzugekommen
8     // sein könnten.
9      $P := P \cup s^\bullet$ 
10  else if  $s \in {}^\bullet t \setminus t^\bullet$  then
11    // Stelle  $s$  ist nur im Vorbereich von  $t$ , also
12    //  $M(s) \supset M'(s)$ . Menge  $P$  wird um die aktivierten
13    // Transitionen aus  $s^\bullet$  erweitert, da diese nun
14    // deaktiviert sein könnten.
15     $P := P \cup (s^\bullet \cap A)$ 
16  else if  $s \in t^\bullet \cap {}^\bullet t \wedge M(s) \neq M'(s)$  then
17    // Stelle  $s$  ist im Vor- und Nachbereich von  $t$  und
18    // ihre Belegung hat sich geändert. Menge  $P$  wird
19    // um alle Transitionen aus  $s^\bullet$  erweitert.
20     $P := P \cup s^\bullet$ 
21  fi
22 od

```

Dieser Algorithmus betrachtet nur *normale* Kanten, so dass in Abschnitt 5.4 *Weitere Kantentypen* ab Seite 39 erneut darauf eingegangen wird. Auch in Abschnitt 5.1 *Kapazitäten* ab Seite 36 wird der Algorithmus noch verändert werden.

**7. Berechnen der aktivierten Transitionen:** Für jede Transition der unter Punkt 6 bestimmten Menge wird berechnet, ob sie unter der aktuellen Markierung aktiviert ist. Siehe Abschnitt 4.6 *Berechnung aktivierender Variablenbelegungen* ab Seite 28.

**8. Wiederholen ab einschließlich Punkt 4.**

## 4.6 Berechnung aktivierender Variablenbelegungen

Eine Transition ist in einem High-Level-Petrinetz dann aktiviert, wenn es eine gültige Belegung aller Variablen im Kontext der Transition gibt. Das heißt, dass der Wächter-Term der Transition unter der Belegung zu `true` ausgewertet wird und die entsprechenden Marken auf den Vorbereichsstellen liegen und die Nachbereichsstellen die entsprechenden Marken enthalten dürfen. Eine solche Belegung der Variablen wird auch *enabling binding* genannt.

Ein Ansatz zur Berechnung von aktivierenden Variablenbelegungen wäre die Übersetzung von Teilen des Petrinetzes in eine geeignete Programmiersprache. Dies kann etwa die Benutzung einer logikbasierten Sprache sein, wenn die Berechnung als Unifikationsproblem aufgefasst wird. Mit der Verwendung einer weiteren Programmiersprache käme allerdings ein Mehraufwand für Übersetzung und Anbindung hinzu.

Der hier besprochene Simulator geht anders vor und berechnet die aktivierenden Variablenbelegungen selbständig. Dazu wird an einem zu berechnenden Binding für jede Variable  $v$  eine Kopie der Menge  $C_v$  angehängt, die im Verlauf der Berechnung möglichst stark eingeschränkt wird. Wird eine solche Menge zu irgendeinem Zeitpunkt der Berechnung die leere Menge, so kann die Berechnung abgebrochen werden, da mindestens eine Variable nicht sinnvoll gebunden werden kann. Grob kann die Berechnung in das Einschränken der Wertemengen der Variablen (das Binden von Variablen, bei denen nur ein Wert in Frage kommt) und die Brute-Force-Suche untergliedert werden.

Die Wertemengen werden anhand der Marken der Vorbereichsstellen der Transition und anhand des Wächter-Terms (siehe Abschnitt 4.4 *Logische/arithmetische Ausdrücke* ab Seite 20) verkleinert. Wenn immer auf die Belegung einer Stelle zugegriffen wird, betrachten die in diesem Abschnitt besprochenen Algorithmen eine Kopie der Markierung. Diese Kopie bezieht sich nur auf die Stellen im Vorbereich der betrachteten Transition, so dass der Speicheraufwand überschaubar bleibt.

Bevor auf den Wächter-Term eingegangen wird, werden alle Werte und Tupel ohne Variablen an entsprechenden Kanten aus der Kopie der Markierung konsumiert. Kann ein Wert oder Tupel nicht entfernt werden, da es nicht vorhanden ist, so ist die Transition nicht aktiviert und die Berechnung bricht ab.

Ein Algorithmus, der aus dem Wächter-Term die Vergleiche ausliest (diese können als

Zuweisungen aufgefasst werden), wird solange durchlaufen, wie noch nicht behandelte Zuweisungen gefunden wurden. Da in den Algorithmen die Terme von links nach rechts durchlaufen werden, wirkt sich die Reihenfolge auch auf die Anzahl der nötigen Durchläufe aus. Das folgende vereinfachte Beispiel verdeutlicht diesen Sachverhalt:

- Term  $A=B \ \& \ B=C \ \& \ C=5$ . Auf Anhieb könnte man sagen, dass  $A=B=C=5$  sind. Wegen der Durchlaufreihenfolge kann der einfache Algorithmus  $A=B$  und  $B=C$  zu Beginn nicht zuweisen, da  $B$  beziehungsweise  $C$  noch nicht zugewiesen sind. Der Algorithmus stellt nur die Belegung  $C=5$  fest.
- Im nächsten Schritt kann  $A=B$  noch nicht festgelegt werden, aber  $B=C$ . Der Algorithmus stellt nur die Belegung  $B=5$  fest.
- Im letzten Schritt kann  $A=B$  festgelegt werden. Der Algorithmus stellt nur die Belegung  $A=5$  fest.

Um das mehrmalige Durchlaufen eines Terms zu unterbinden, wird einmalig eine Sortierung von Operanden kommutativer Operatoren vorgenommen. Die Teilterme werden nach der Anzahl von ungebundenen Variablen aufsteigend sortiert. Hier wird davon ausgegangen, dass die Situationen, in der eine Transition aktiviert werden kann und ihr Term ausgewertet werden muss, über die ganze Simulation hinweg ähnlich zueinander sind.

Es wird beim ersten Berechnen der Aktivierung einer Transition der oben beschriebene Algorithmus so oft ausgeführt, bis keine Variablen mehr gebunden werden und nach jedem (und vor dem ersten) Ausführungsschritt wird eine (stabile) Sortierung des Terms durchgeführt. Der Betrachtete Term sähe dann so aus:

- Originalterm:  $A=B \ \& \ B=C \ \& \ C=5$
- Erstes Sortieren:  $5=C \ \& \ A=B \ \& \ B=C$
- Nach Schritt 1,  $C=5$ :  $5=C \ \& \ C=B \ \& \ A=B$
- Nach Schritt 2,  $B=5$ :  $5=C \ \& \ C=B \ \& \ B=A$
- Nach Schritt 3,  $A=5$ :  $5=C \ \& \ C=B \ \& \ B=A$

Durch die Sortierung wird die Reihenfolge der Teilterme (nach Möglichkeit) folglich so verändert, dass bei der nächsten Berechnung der Aktivierung nur ein einziger Durchlauf des Terms nötig ist. Sollten bestimmte Transitionen im Simulationsprozess nicht gefeuert werden, so werden nicht alle Terme des Netzes sortiert.

Es hat sich auch gezeigt, dass das Berechnen von aktivierenden Bindings durch Äquivalenzumformungen beschleunigt werden kann. Beispielsweise kann aus dem Term  $4*3 = 2*(3+X)$  die einzige Variable nicht direkt ausgelesen werden, da sie nicht alleine auf einer Seite der Gleichung steht. Durch einen Algorithmus wird ein solcher Term zu  $(4*3)/2=3+X$  und weiter zu  $(4*3)/2-3=X$  umgeformt. (Bei der statischen Optimierung vor Beginn der Simulation wird ein so umgeformter Term zu  $3=X$  vereinfacht.) Anstatt zum Beispiel mit  $X$  die Werte von 0 bis 5 zu durchlaufen und festzustellen, dass nur Wert 3 den Term erfüllt, kann nun der Wert für  $X$  direkt ausgerechnet werden. Diese Äquivalenzumformungen werden ebenso wie die Sortierung vor der Simulation und dann beim ersten Berechnen der Aktivierung einer Transition ausgeführt.

Im zweiten Schritt, der Brute-Force-Suche, werden noch ungebundene Variablen nacheinander mit jedem Wert ihrer aktuellen Wertemenge belegt und es wird geprüft, ob der Wert von den entsprechenden Vorbereichsstellen genommen werden kann. Dazu wird der oben eingeführten (eingeschränkte) Kopie der Netzmarkierung der jeweilige Wert entnommen, und im Falle des Backtrackings wieder zurück gelegt.

Wenn alle Variablen probeweise gebunden sind, wird der Wächter-Term ausgewertet. Ist das Ergebnis der Auswertung `true`, so wird das gefundene aktivierende Binding eingesammelt und mit allen weiteren zurückgeliefert.

Schlägt die Auswertung des Wächter-Terms zu `true` mit allen möglichen Kombinationen von Variablenbelegungen fehl, oder befinden sich nicht die richtigen Marken auf den Vorbereichsstellen, so ist die entsprechende Transition nicht aktiviert. [EFH<sup>+</sup>05]

## 4.7 Tupel

Um mit Tupeln umgehen zu können, wurde der Simulator um einen Operator ergänzt, der nur intern im Syntaxbaum eines Terms verwendet wird und ein Tupel aus seinen Operanden erzeugt.

Die einzigen fest eingebauten Vergleichsoperatoren für Tupel sind  $\neq$  und  $=$ , allerdings können über selbstdefinierte Funktionen auch andere Vergleiche benutzt werden. Beispielsweise kann ein elementweises Größer als

$$greater(T1, T2) = forall(i, 0, dim(T1) - 1, 1, proj(T1, i) > proj(T2, i))$$

geschrieben werden.

Die arithmetischen Operatoren werden auf Tupeln wie folgt umgesetzt: Ist ein Operand ein Skalar und der andere ein Tupel, so wird der Operator mit dem Skalar elementweise auf dem Tupel ausgeführt und das Ergebnis ist ein Tupel. Sind beide Operanden Tupel gleicher Dimension so wird der Operator elementweise auf beiden Tupeln ausgeführt und das Ergebnis ist ein Tupel. In allen anderen Fällen kann die Operation nicht durchgeführt werden. Zusätzlich stehen einige Funktionen zur Verfügung, die Tupel als Parameter akzeptieren (siehe dazu Abschnitt 4.8 *Funktionen* ab Seite 32).

Da Tupel Variablen enthalten und auf Stellen und an Kanten vorkommen können, ist es nötig, in den in Abschnitt 4.4 *logische/arithmetische Ausdrücke* ab Seite 20 genannten Verfahren Tupel genauer zu betrachten: Beispielsweise werden Vergleiche von Tupeln, die elementweise durchgeführt werden, wie eine Menge von Vergleichen optimiert. Der Ausdruck  $(A, (B, C, D), E) = (29, (3*4, 0, 0), dot)$  würde intern optimiert werden zu  $A = 29 \ \& \ (B, C, D) = (3*4, 0, 0) \ \& \ E = dot$  und weiter zu  $A = 29 \ \& \ B = 3*4 \ \& \ C = 0 \ \& \ D = 0 \ \& \ E = dot$ . Bei verschachtelten Tupeln wird das Verfahren also sukzessive fortgeführt.

Kommen Variablen in einem Tupel an einer Eingangskante einer Transition vor, so ist beim Berechnen, ob die Transition aktiviert ist, festzustellen, ob bestimmte Tupel auf einer Stelle liegen. Die Wertemengen der in Tupeln vorkommenden Variablen werden analog zum in Abschnitt 4.6 *Berechnung aktivierender Variablenbelegungen* ab Seite 28 beschriebenen Verfahren eingeschränkt. Grundsätzlich werden Tupel mit Variablen wie Variablen und Tupel ohne Variablen wie Werte behandelt.

## 4.8 Funktionen

Es wurde eine Funktionsauswertung implementiert, so dass der Simulator auch Netze verarbeiten kann, in denen Terme wie  $(\max(A, B) > 6 \mid \deg(C) = 180) \ \& \ \exp(A) = e() \ \& \ D = \pi()$  vorkommen. Zur Auswertung werden die Java-Funktionen der `java.lang.Math`-Klasse benutzt.

Eine besondere Behandlung wird von der `rand()`-Funktion gefordert, da diese nur ein einziges Mal ausgewertet werden darf. Würde `A=rand()` mehrfach ausgewertet werden, so könnte Folgendes passieren:

1. Da `A=floor(rand() * 10) & B=A` als Zuweisung eines zufälligen Wertes an `A` aufgefasst werden kann, wird der Term ausgewertet und `A` wird zum Beispiel mit 6 gebunden. Außerdem wäre auch `B` an 6 gebunden.
2. Gibt es noch ungebundene Variablen, so werden diese in der Brute-Force-Suche gebunden und der Term wiederholt ausgewertet.
3. Bei jeder dieser weiteren Auswertungen entscheidet der Zufall, ob `floor(rand() * 10)` gerade zu 6 ausgewertet wird oder nicht. Dies führt zu unvorhersehbarem Verhalten, da eventuell gültige Bindings verworfen werden, wenn der Term gerade zufällig nicht zu `A=6` ausgewertet wurde.

Als Lösung des Problems wird für Terme gespeichert, ob sie Funktionen enthalten, die nur ein einziges Mal ausgewertet werden dürfen. Darf der Term nur ein einziges Mal ausgewertet werden, so wird der Syntaxbaum kopiert und die Teilterme während der ersten Auswertung (im Beispiel Schritt 1) sinnvoll durch `true` ersetzt. Im Beispiel würde der Term nach der Festlegung von `A` und `B` auf 6 zu `true & true` und weiter zu `true` vereinfacht. Ein Problem mit der Zufallsfunktion ist die Entfaltung zu Low-Level-Netzen (siehe Abschnitt 5.8 *Entfaltung* ab Seite 50). Dort wird jeder `rand()`-Aufruf nur ein einziges Mal ausgeführt, so dass der Zufall im entfalteten Netz verloren geht. An Stelle der Zufallsfunktion kann eine freie Variable mit einer entsprechenden Wertemenge benutzt werden. Statt `A=floor(rand()*10) & B=A` könnte man `A{0..9} : B=A` schreiben.

Es werden auch benutzerdefinierte Funktionen unterstützt. Beispielsweise kann der Benutzer eine Funktion zur Berechnung der Fakultät einer Zahl rekursiv als `fak(X) = ifthen(X <= 1, 1, X*fak(X-1))` definieren oder aber iterativ als `fak(X) = ifthen(X`

`<= 1, 1, prod(create(i, 1, X, 1, i))`). Rekursive Funktionen werden allerdings langsamer berechnet als iterative. Fest eingebaute Funktionen können nicht überschrieben werden.

Bei benutzerdefinierten Funktionen wird auch das Überladen gestattet. Sind zwei Funktionen  $f(X) = 2*X$  und  $f(X,Y) = X*Y$  definiert, so wird bei  $f(4)$  die erste Funktion ausgeführt und bei  $f(3,4)$  die zweite Funktion. Benutzerdefinierte Funktionen können wie fest eingebaute Funktionen überall dort eingesetzt werden, wo ein mathematischer Ausdruck erlaubt ist.

Beim Aufruf einer benutzerdefinierten Funktion werden die formalen Parameter an die übergebenen Werte gebunden und die Funktion wird ausgewertet. Wird eine Funktion mehrfach zur gleichen Zeit ausgewertet, etwa bei einem rekursiven Aufruf, so wird eine Kopie des Syntaxbaumes der Funktion im Speicher angelegt. Da dies ein ineffizientes Verfahren ist, sollte die Benutzung rekursiver Funktionen vermieden werden.

In Tabelle 4.2 werden alle direkt unterstützten Funktionen mit der jeweiligen Anzahl an Parametern und einer Kurzbeschreibung aufgelistet.

Tabelle 4.2: Funktionen des P-UMLaut/Sim

Funktion	Parameter	Beschreibung
abs	1	Betragsfunktion
acos	1	Arcus Cosinus
acot	1	Arcus Cotangens
append	2	$append((a_0, \dots, a_n), X) = (a_0, \dots, a_n, X)$
asin	1	Arcus Sinus
asTuple	$\geq 0$	Erzeugt ein Tupel, das die Argumente enthält.
atan	1	Arcus Tangens
cb	1	Kubik
cbrt	1	Kubikwurzel
ceil	1	Aufrunden
concatenate	2	$concatenate((a_0, \dots, a_n), (b_0, \dots, b_m))$ $= (a_0, \dots, a_n, b_0, \dots, b_m)$
cos	1	Cosinus
cosh	1	hyperbolischer Cosinus
cot	1	Cotangens

coth	1	hyperbolischer Cotangens
create	3	$create(var, (t_0, \dots, t_n), ausdruck) = (a_0, \dots, a_n)$
create	5	$create(var, von, bis, schritt, ausdruck)$ $= (a_0, \dots, a_{\lfloor (bis-von)/schritt \rfloor})$
deg	1	Umrechnung von Bogenmaß zu Grad
e	0	die eulersche Zahl $e$
evaluate	2	$evaluate(„f“, (t_0, \dots, t_n)) = f(t_0, \dots, t_n)$
exists	3	$exists(var, (t_0, \dots, t_n), bedingung)$
exists	5	$exists(var, von, bis, schritt, bedingung)$
exists1	3	$exists1(var, (t_0, \dots, t_n), bedingung)$
exists1	5	$exists1(var, von, bis, schritt, bedingung)$
exp	1	Exponentialfunktion
factorial	1	Fakultät
floor	1	Abrunden
forall	3	$forall(var, (t_0, \dots, t_n), bedingung)$
forall	5	$forall(var, von, bis, schritt, bedingung)$
ifthen	> 2	$ifthen(if, then, \{elseif, then, \} else)$
isBoolean	1	<b>true</b> genau dann, wenn der übergebene Wert ein Wahrheitswert ist
isDot	1	<b>true</b> genau dann, wenn der übergebene Wert <b>dot</b> ist
isNumber	1	<b>true</b> genau dann, wenn der übergebene Wert eine Zahl ist
isString	1	<b>true</b> genau dann, wenn der übergebene Wert eine Zeichenkette ist
isTuple	1	<b>true</b> genau dann, wenn der übergebene Wert ein Tupel ist
log	2	$log(a, b) = log_a b$ Logarithmus von $b$ zur Basis $a$
log10	1	dekadischer Logarithmus
logn	1	natürlicher Logarithmus
map	2	$map(„f“, v) = f(v)$ , $map(„f“, (a_0, \dots, a_n))$ $= (f(a_0), \dots, f(a_n))$
max	> 0	Maximum der Parameter (oder des Tupels)
min	> 0	Minimum der Parameter (oder des Tupels)
pi	0	die Zahl $\pi$

pow	2	$pow(a, b) = a^b$
prod	1	$prod((a_0, \dots, a_n)) = a_0 \cdot \dots \cdot a_n$
proj	2	$proj((a_0, \dots, a_n), k) = a_k$ mit $0 \leq k \leq n$
property	1	property(„eigenschaft“) liefert den Wert der Eigenschaft, wenn verfügbar
rad	1	Umrechnung von Grad zu Bogenmaß
rand	0	eine Zufallszahl $x$ mit $0 \leq x < 1$
read	$\geq 0$	gibt die Parameter mit write(...) aus. Wenn dies erfolgreich war, wird darauf folgende Benutzereingabe zurückgeliefert
remove	2	$remove((a_0, \dots, a_n), i) = (a_0, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$
replace	3	$replace((a_0, \dots, a_n), i, X)$ $= (a_0, \dots, a_{i-1}, X, a_{i+1}, \dots, a_n)$
reverse	1	$reverse((a_0, \dots, a_n)) = (a_n, \dots, a_0)$
root	2	$root(a, b) = \sqrt[a]{b}$
round	1	Runden
sig	1	Vorzeichenfunktion
sin	1	Sinus
sinh	1	hyperbolischer Sinus
sqr	1	Quadrat
sqrt	1	Quadratwurzel
sum	1	$sum((a_0, \dots, a_n)) = a_0 + \dots + a_n$
tan	1	Tangens
tanh	1	hyperbolischer Tangens
write	$\geq 0$	gibt die übergebenen Parameter auf der Konsole aus

# 5 Integration der Erweiterungen

„Die stillsten Worte sind es, welche den Sturm bringen. Gedanken, die mit Taubenfüßen kommen, lenken die Welt.“

Friedrich Nietzsche

Im Folgenden werden die neu dazu gekommenen Erweiterungen und die notwendigen Änderungen am Simulator erläutert. Dabei wird schrittweise vorgegangen, so dass spätere Abschnitte die vorangegangenen voraussetzen. Dies wird auch durch Verweise deutlich gemacht.

## 5.1 Kapazitäten

Bei der Berechnung der aktivierenden Variablenbelegungen von Transitionen ist darauf zu achten, ob die Transition die gewünschten Token auf den Stellen ihres Nachbereichs erzeugen kann. Ist das nicht der Fall, etwa weil die Kapazität der Stelle überschritten würde, so ist die Transition nicht aktiviert.

Eine Folgerung daraus ist, dass eine Transition  $t$  aktiviert werden kann, indem eine andere Transition Marken von einer Stelle des Nachbereichs von  $t$  konsumiert. Analog dazu kann eine Transition  $t$  deaktiviert werden, wenn eine andere Transition Marken auf einer Stelle des Nachbereichs von  $t$  erzeugt.

Diese beiden Fälle sind in den Algorithmus zur Berechnung der zu prüfenden Transitionen in Abschnitt 4.5 *Gefärbte Petrinetze* ab Seite 24 zu integrieren. Seien die Bezeichner wie dort beschrieben gewählt und beschreibe  $C(s) \in \mathbb{N} \cup \{\infty\}$  die Kapazität einer Stelle  $s$ . Um zu verdeutlichen, dass in den Änderungen nur Stellen mit Kapazitäten betrachtet werden, wurde den Fällen die Bedingung  $C(s) \neq \infty$  vorangestellt.

```

1 // die Menge der zu prüfenden Transitionen ist anfänglich leer
2  $P := \emptyset$ 
3 for each  $s \in t^\bullet \cup {}^\bullet t$  do
4   // für jede Stelle  $s$  des Vor- und Nachbereiches von  $t$ 
5   if  $s \in t^\bullet \setminus {}^\bullet t$  then
6     // Stelle  $s$  ist nur im Nachbereich von  $t$ ,  $M(s) \subset M'(s)$ .
7     // Menge  $P$  wird um die Transitionen aus  $s^\bullet$  erweitert,
8     // da Bindings hinzugekommen sein könnten.
9      $P := P \cup s^\bullet$ 
10  >   if  $C(s) \neq \infty \wedge |M'(s)| = C(s)$  then
11  >     //  $s$  war nicht 'voll', ist es aber nach dem Feuern,
12  >     // folglich können aktivierte Transitionen des
13  >     // Vorbereichs von  $s$  deaktiviert worden sein
14  >      $P := P \cup ({}^\bullet s \cap A)$ 
15  >   fi
16  else if  $s \in {}^\bullet t \setminus t^\bullet$  then
17  >     // Stelle  $s$  ist nur im Vorbereich von  $t$ ,  $M(s) \supset M'(s)$ .
18  >     // Menge  $P$  wird um die aktivierten Transitionen aus  $s^\bullet$ 
19  >     // erweitert, da diese nun deaktiviert sein könnten.
20  >      $P := P \cup (s^\bullet \cap A)$ 
21  >   if  $C(s) \neq \infty \wedge |M(s)| = C(s)$  then
22  >     //  $s$  war 'voll' und ist nicht mehr, folglich
23  >     // können deaktivierte Transitionen des
24  >     // Vorbereichs von  $s$  aktiviert worden sein
25  >      $P := P \cup ({}^\bullet s \cap \overline{A})$ 
26  >   fi
27  else if  $s \in t^\bullet \cap {}^\bullet t \wedge M(s) \neq M'(s)$  then
28  >     // Stelle  $s$  ist im Vor- und Nachbereich von  $t$  und
29  >     // ihre Belegung hat sich geändert. Menge  $P$  wird
30  >     // um alle Transitionen aus  $s^\bullet$  erweitert.
31  >      $P := P \cup s^\bullet$ 
32  fi
33 od

```

In der aktuellen Implementierung gibt es eine Inkonsistenz (unter anderem in Bezug auf eine Entfaltung): Die Kapazität einer Stelle bezieht sich zur Zeit auf alle Token der Stelle. Würden die Kapazitäten konsistent zur Entfaltung behandelt werden, so müsste eine Stelle für jeden Wert ihres Wertebereichs eine Kapazität angeben, etwa *diese Stelle darf höchstens fünf schwarze Token und höchstens drei Vieren enthalten*, denn dann könnten diese Kapazitäten *mitentfaltet* werden. Siehe dazu auch Abschnitt 5.8 *Entfaltung* ab Seite 50 und Unterabschnitt 7.1.1 *Funktionalität* ab Seite 59.

## 5.2 Ungetypte Stellen

Diese Erweiterung erlaubt es, bei Stellen ihren Typ auszulassen. Vom Simulator wird dann angenommen, dass jeder Wert auf der Stelle liegen könnte. Stellen ohne Stellentyp lassen sich nicht entfalten, aber eine Simulation ist dennoch möglich.

Die Wertebereiche von Variablen an Kanten, die mit der Stelle verbunden sind, lassen sich dann allerdings nicht mehr so optimieren, wie es Punkt 2.1 in Abschnitt 4.5 *Gefärbte Petrinetze* ab Seite 24 beschrieben wurde. Das Optimieren der Wächter-Terme bleibt unbeeinflusst.

## 5.3 Prioritäten

Durch Prioritäten kann das Feuern bestimmter Transitionen anderen vorgezogen werden. Um die Transitionen mit den höchsten Prioritäten schnell ermitteln zu können, werden die aktivierenden Variablenbelegungen nach ihren Prioritäten sortiert in einer Liste gehalten. Kommt ein neues Binding hinzu oder muss eines entfernt werden, so kann die sortierte Liste effizient nach der Einfügestelle (Sortieren durch Einfügen mit binärer Suche der Einfügestelle), beziehungsweise der Position des zu löschenden Bindings (binäre Suche), durchsucht werden.

Da unter Umständen viele Transitionen mit derselben Prioritätsklasse aktiviert sind, wurde ein künstlicher zweiter Ordnungsschlüssel für Bindings eingeführt, über den die oben genannte Liste absteigend sortiert ist. Auch wenn in einem Petrinetz mit vielen Transitionen keine Prioritäten vergeben wurden, kann ein Binding so effizient in der Liste gefunden werden.

Je nachdem, ob *globale* oder *lokale* Prioritäten bei der Simulation verwendet werden sollen, wird die zufeuernde Transition auf zwei verschiedene Arten ermittelt:

**globale Priorität:** Aus der sortierten Liste der Bindings werden die mit der höchsten Priorität herausgesucht. Aus diesen wird eines zufällig ausgewählt. Das Heraussuchen der Bindings mit den höchsten Prioritäten kann ebenfalls durch eine binäre Suche realisiert werden.

**lokale Priorität:** Es wird zufällig eine Transition  $t$  ausgewählt und aus der Menge der aktivierten Transitionen, deren Vorbereiche sich mit dem von  $t$  überschneiden wird zufällig eine mit der höchsten Priorität ausgewählt. Die sortierte Liste der Bindings wird hier nicht benötigt.

## 5.4 Weitere Kantentypen

Mit der Erweiterung des Simulators kamen drei neue Kantentypen hinzu, deren Behandlung bei der Simulation im Folgenden beschrieben wird.

### 5.4.1 Resetkanten

Die Beschriftung einer Resetkante ist im Gegensatz zu denen anderer Kanten eine Menge. Wird eine Transition  $t$  gefeuert, zu der von Stelle  $s$  aus eine mit  $V$  beschriftete Resetkante führt, so werden mit dem Schalten von  $t$  alle Werte aus  $V$  von  $s$  durch die Kante konsumiert.

Bei der Berechnung aktivierender Variablenbelegungen können Resetkanten vernachlässigt werden, da sie keine Anforderungen an die zugehörige Eingangsstelle haben. Die Implementierung wird dadurch vereinfacht. Beim Feuern werden erst alle anderen Eingangskanten bearbeitet, bevor die gewünschten Marken durch Resetkanten konsumiert werden.

Zur Vereinfachung der Modellierung wird ein spezieller Wert  $*$  an Resetkanten erlaubt, der besagt, dass die entsprechende Stelle komplett leergeräumt werden soll. Damit ist  $*$  eine Kurzschreibweise für den Typ der jeweiligen Stelle.

## 5.4.2 Inhibitorkanten

Anders als bei Resetkanten können Inhibitorkanten beim Feuern vernachlässigt werden, müssen dafür aber beim Ermitteln von aktivierenden Variablenbelegungen beachtet werden.

Eine mit Multimenge  $V$  beschriftete Inhibitorkante von Stelle  $s$  zu einer Transition fordert, dass  $\forall v \in V : |M(s) \cap \{v\}| < |V \cap \{v\}|$  gilt ( $M(s)$  beschreibt die aktuelle Belegung von  $s$ ). Für Werte und Tupel ohne Variablen kann die Überprüfung der Bedingung dem Prüfen von anderen Kanten vorgezogen werden. Die konkreten Werte von Variablen stehen in dem in Abschnitt 4.6 *Berechnung aktivierender Variablenbelegungen* ab Seite 28 beschriebenen Algorithmus allerdings erst spät fest. Vorher könnten bereits Werte probeweise von Stellen der dort angesprochenen Kopie der Markierung entfernt worden sein. Die Forderungen der Inhibitorkanten sind auf der ursprünglichen Markierung zu prüfen, auf die auch während der zweiten Phase des Algorithmus zugegriffen werden kann.

Zur Vereinfachung der Modellierung wird hier ebenfalls ein spezieller Wert  $*$  an Inhibitorkanten erlaubt, der besagt, dass die entsprechende Stelle komplett leer sein muss. Damit ist  $*$  eine Kurzschreibweise für den Typ der jeweiligen Stelle: Jeder mögliche Wert muss weniger als ein Mal vorkommen, also gar nicht.

## 5.4.3 Lesekanten

Im Algorithmus zur Berechnung der Bindings können Lesekanten wie normale Kanten behandelt werden. Beim Feuern werden sie nicht weiter betrachtet.

Dieses Verhalten entspricht dem einer Schlinge, bei der das Konsumieren und Erzeugen von Token zu *keine Änderung* zusammengefasst wurde.

## 5.5 Zeit

Zeitfreie Transitionen haben beim P-UMLaut/Sim grundsätzlich eine höhere Präzedenz als zeitbehaftete. Eine Transition wird vom Simulator als zeitfrei angenommen, wenn beim Laden des Petrinetzes keine Zuweisung an einen Term geschieht, der etwas über das Zeitverhalten der Transition aussagt, das heißt keiner der Terme **Earliest**

Firing, Latest Firing, Rate oder Generalized Rate (auf die letzten beiden wird in Abschnitt 5.6 *Stochastische Elemente* ab Seite 48 eingegangen).

Transitionen können wie in Abschnitt 2.5 *Netze mit Zeit* ab Seite 7 beschrieben mit einem Zeitintervall  $[\tau_1, \tau_2]$ , das mit dem Binding berechnet wird, ausgestattet werden ( $\tau_1$  ist **Earliest Firing**,  $\tau_2$  ist **Latest Firing**). Da zu einem Zeitpunkt  $z$  nur die Transitionen gefeuert werden können, für die  $\tau_1 \leq z \leq \tau_2$  gilt, ist die Menge der feuerbaren Transitionen eine Untermenge der aktivierten Transitionen.

Es wird analog zu Abschnitt 5.3 *Prioritäten* ab Seite 38 vorgegangen, so dass ein neuer Sortierungsschlüssel  $\tau_1$  eingeführt wird, über den aufsteigend sortiert wird. Zusammen mit den Prioritäten wird dann folgendermaßen sortiert werden:

1. aufsteigende Sortierung über  $\tau_1$  (erster Zeitpunkt, zu dem gefeuert werden kann).
2. absteigende Sortierung über Priorität.
3. beliebige aber feste Sortierung über den künstlichen Schlüssel eines Bindings.

Mit der Auswahl der zu feuern Transition kann dann analog zur globalen Priorität verfahren werden.

Wird der Wert der globalen Uhr erhöht, so können Bindings gefeuert (strikes Zeitkonzept) oder deaktiviert (schwaches Zeitkonzept) werden, wenn mit dem neuen Zeitpunkt  $\tau_2 < z$  gilt und damit ihr Zeitintervall verlassen wurde. Um diese Menge der zu behandelnden Transitionen effizient ermitteln zu können, wird eine zweite sortierte Liste angelegt, in der die Bindings aufsteigend nach  $\tau_2$  und beliebig aber fest nach dem künstlichen Schlüssel sortiert sind.

Wird nach dem strikten Zeitkonzept verfahren, so wird aus der Menge der zu feuern Bindings solange eine ausgewählt und gefeuert, bis alle Bindings der Menge deaktiviert sind.

Im Verlauf der Simulation kann es zu Situationen kommen, in denen zwar Transitionen aktiv sind, aber keine von ihnen feuerbar. Beschreibt  $e$  das kleinste  $\tau_1$  und  $l$  das kleinste  $\tau_2$  der aktivierten Bindings, so wird die Uhr auf einen zufälligen Wert im Intervall  $[e, l]$  gesetzt, denn dann kann ein Binding gefeuert werden und es wurde nicht gegen das schwache/starke Zeitkonzept verstoßen.

### 5.5.1 Age Memory

Das Standardverhalten des Simulators ist es, zeitbehaftete Transitionen mit *Enabling Memory* auszustatten. Ein Binding mit Zeitintervall  $[\tau_1, \tau_2]$  muss mindestens für  $\tau_1$  Zeiteinheiten durchgehend aktiviert sein und kann dann gefeuert werden.

Um wahlweise für zeitbehaftete Transitionen den Einsatz der *Age Memory*-Strategie zu erlauben, wird jede Transition  $T_i$  mit einem assoziativen Speicher für die kumulierte Dauer des Aktiviertseins für jedes ihrer Bindings ausgestattet – hierbei ist die konkrete Variablenbelegung gemeint, im untenstehenden Algorithmus wird mit Binding eine bestimmte Instanz gemeint. Im Zusammenhang mit Unterabschnitt 5.5.2 *Server-Semantiken* ab Seite 44 wird genauer darauf eingegangen.

Im Folgenden ist mit  $A_{T_i}(b)$  dieser Wert für Binding  $b$  gemeint, initial ist  $A_{T_i}(b)$  für alle Bindings auf 0 gesetzt.

- Wird ein Binding  $b$  aktiviert, das die Age-Memory-Strategie verwenden soll, so wird der Zeitpunkt der Aktivierung als  $b.z_0$  gespeichert. Der aktuelle Zeitpunkt sei mit  $z$  bezeichnet. Die Berechnung des Zeitintervalls eines Bindings wird verändert:

$$[z + \max\{0, \tau_1 - A_{T_i}(b)\}, z + \max\{\tau_2 - \tau_1, \tau_2 - A_{T_i}(b)\}]$$

Mit der Änderung des Intervalls wird  $\min\{\tau_1, A_{T_i}(b)\}$  als  $b.a$  im Binding gespeichert und  $A_{T_i}(b) = \max\{0, A_{T_i}(b) - \tau_1\}$  gesetzt.

Eine Instanz  $b$  eines Bindings verbraucht nur soviel der kumulierten Zeit, wie es auch *benötigt*, den Rest lässt es in  $A_{T_i}(b)$  für andere Instanzen zurück.

- Auf das Feuern eines Bindings muss in Bezug auf die Age-Memory-Strategie nicht reagiert werden.
- Wird ein Binding  $b$  das zu zu Zeitpunkt  $b.z_0$  aktiviert wurde, deaktiviert und beschreibt  $z$  den aktuellen Zeitpunkt, so wird  $A_{T_i}(b) = A_{T_i}(b) + b.a + z - b.z_0$  gesetzt.

Im TTPN in Abbildung 5.1 benutze Transition  $T_1$  die Age-Memory-Strategie. Ohne Age Memory würde diese Transition bei einem strikten Zeitkonzept nie feuern können, da sie immer mit Transition  $T_2$  aktiviert ist und  $T_2$  immer vor  $T_1$  feuern würde. Tabelle 5.1 spielt

das Verhalten des Netzes einmal mit der oben eingeführten Behandlung von Age Memory durch. Anschließend wird der Ablauf in Bezug auf Age Memory genauer betrachtet und die Speicherinhalte werden verfolgt. Die Zeit wird wie in Abschnitt 5.5 *Zeit* ab Seite 40 beschrieben erhöht.

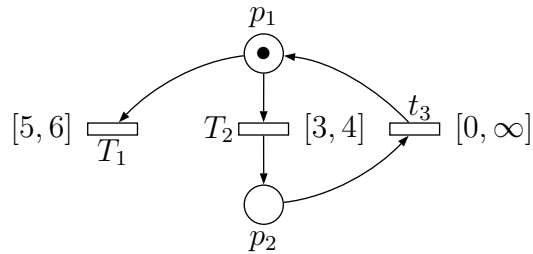


Abbildung 5.1: Behandlung von Age Memory

Tabelle 5.1: Möglicher Ablauf des Netzes in Abbildung 5.1

Schritt	Uhr	aktivierte Transitionen	feuerbare T.	Aktion
1	0	$\{ T_1[5, 6], T_2[3, 4] \}$	$\emptyset$	Uhr erhöhen
2	3.5	$\{ T_1[5, 6], T_2[3, 4] \}$	$\{ T_2 \}$	$T_2$ feuern
3	3.5	$\{ t_3[3.5, \infty] \}$	$\{ t_3 \}$	Uhr erhöhen
4	5	$\{ t_3[3.5, \infty] \}$	$\{ t_3 \}$	$t_3$ feuern
5	5	$\{ T_1[6.5, 7.5], T_2[8, 9] \}$	$\emptyset$	Uhr erhöhen
6	7	$\{ T_1[6.5, 7.5], T_2[8, 9] \}$	$\{ T_1 \}$	$T_1$ feuern
7	7	$\emptyset$	$\emptyset$	-

1. Die anfänglich aktivierten Transitionen werden bestimmt. Im Binding von  $T_1$  (als Instanz einer aktivierender Variablenbelegung) wird die kumulierte Dauer des Aktiviertseins gespeichert ( $\min\{5, A_{T_1}(\{\})\} = 0$ ) und  $A_{T_1}(\{\}) = \max\{0, A_{T_1}(\{\}) - 5\} = 0$  gesetzt. Da keine Transition feuerbar ist, wird die Uhr auf einen zufälligen Wert im Intervall  $[3, 4]$  gesetzt: 3.5
2. Nach dem Erhöhen der Uhr ist Transition  $T_2$  als einzige feuerbar. Durch ihr Feuern wird die Marke von  $p_1$  konsumiert und eine auf  $p_2$  erzeugt und die Instanz ihres Bindings verworfen. Transition  $T_1$  wird deaktiviert und es wird  $A_{T_1}(\{\}) = A_{T_1}(\{\}) + 0 + 3.5 - 0 = 3.5$  zugewiesen, da  $A_{T_1}(\{\}) = 0$  war und  $T_1$  zu Zeitpunkt 0 aktiviert wurde.

3. Nun ist  $t_3$  aktiviert und sofort feuerbar. Anstatt  $t_3$  zu feuern, wir allerdings die Uhr auf einen beliebigen Wert erhöht: 5
4. Nun wird  $t_3$  gefeuert und die Marke von  $p_2$  konsumiert und eine auf  $p_1$  erzeugt.
5. Da  $T_1, T_2$  aktiviert werden, sind ihre Zeitintervalle zu berechnen:  $5 + [5, 6] = [10, 11]$  für  $T_1$  und  $5 + [3, 4] = [8, 9]$  für  $T_2$ . Für  $T_1$  wird die Age-Memory-Strategie verwendet, so dass das Intervall auf  $5 + [\max\{0, 5 - A_{T_1}(\{\})\}, \max\{6 - 5, 6 - A_{T_1}(\{\})\}] = [6.5, 7.5]$  geändert, der Wert von  $\min\{\tau_1, A_{T_1}\} = \min\{5, 3.5\} = 3.5$  in der Instanz des Bindings gespeichert und  $A_{T_1}(\{\}) = \max\{0, A_{T_1}(\{\}) - 5\} = 0$  gesetzt wird.  
Zum aktuellen Zeitpunkt 5 ist keine Transition feuerbar, so dass die Uhr auf einen zufälligen Wert aus dem Intervall  $[6.5, 7.5]$  gesetzt wird: 7
6. Transition  $T_1$  ist als einzige feuerbar. Mit dem Feuern wird die Marke von  $p_1$  konsumiert und die Instanz ihres Bindings verworfen.
7. Es ist keine Transition mehr aktiviert. Das Netz befindet sich in einem Deadlock.

Durch die Einführung von Age Memory konnte Transition  $T_1$  trotz des strikten Zeitkonzepts feuern, da sie nicht durchgehend aktiviert sein musste um die geforderte Zeit verstreichen zu lassen. Die Dauer der Aktivierung des Bindings wurde kumuliert.

Im Folgenden Abschnitt wird vorgestellt, wie die Age-Memory-Strategie zusammen mit den verschiedenen Server-Semantiken funktioniert.

## 5.5.2 Server-Semantiken

Bei der Implementierung des Simulators wurde entschieden, im Bezug auf die Server-Semantiken verschiedene aktivierende Variablenbelegungen einer Transition im High-Level-Petrinetz als verschiedene Transitionen im Low-Level-Petrinetz zu behandeln. Für die Single-Server-Semantik bedeutet das, dass eine Transition gleichzeitig mehrfach aktiviert sein darf, solange die Bindings verschieden sind. Analoges gilt für die Multiple-Server-Semantik. Die Infinite-Server-Semantik bleibt unbeeinflusst.

Durch diese Entscheidung wurde die Single-Server-Semantik (wie die Enabling-Memory-Strategie) direkt als Standard-Semantik für Transitionen implementiert.

Zur Integration der beiden anderen Semantiken muss für ein Binding das sogenannte Enabling Degree ermittelt werden können. Das Enabling Degree ist eine natürliche Zahl oder  $\infty$ , die aussagt, wie oft ein Binding direkt hintereinander gefeuert werden könnte.

Im P-UMLaut/Sim wird das Enabling Degree für ein Binding  $b$  einer Transition  $t$  folgendermaßen berechnet:

Sei  $M(s)$  die aktuelle Belegung von  $s$ ,  $Arc(s,t)$  die Beschriftung (eine Multimenge) der normalen Kante von  $s$  nach  $t$ ,  $Read(s,t)$  die Beschriftung (eine Multimenge) der Lesekante und  $b(V)$  eine Multimenge, bei der alle in der Multimenge  $V$  vorkommenden Variablen – auch in Tupeln – durch ihre entsprechende Belegung ersetzt wurden.

```

1 degree =  $\infty$  // gehe von unbegrenztem Enabling Degree aus
2 // betrachte alle Kanten, die Marken verlangen
3 for each  $s \in \bullet t \cup {}^L t$  do
4   // für jede Kante die verlangten Werte bestimmen
5    $V = b(Arc(s,t) \cup Read(s,t))$ 
6   for each  $v \in V$  do
7     // Wert evtl. mehrfach in Multimenge  $V$ 
8     // Optimierung: jeden Wert nur einmal betrachten.
9     // für jeden Wert zählen, wie oft er verlangt wird
10    // und wie oft er auf  $s$  vorhanden ist
11     $v\_count = |V \cap \{v\}|$ 
12     $s\_count = |M(s) \cap \{v\}|$ 
13    //  $\lfloor s\_count / v\_count \rfloor$  besagt,
14    // wie oft  $t$  in Bezug auf  $v$  feuern kann
15    // Enabling Degree gegebenenfalls verringern
16     $degree = \min(degree, \lfloor s\_count / v\_count \rfloor)$ 
17  od
18 od
19 // degree ist das gesuchte Enabling Degree

```

Offensichtlich ist das Enabling Degree  $\infty$ , wenn  $\bullet t \cup {}^L t = \emptyset$ . Dieser Fall wird weder in [Bal01], noch in [MBC<sup>+</sup>95] betrachtet, ist für die Implementierung der Semantiken jedoch nicht zu vernachlässigen.

Im Falle, dass eine bestimmte Variablenbelegung im Kontext einer Transition die Infini-

te-Server-Semantik verwendet und das zugehörige Binding  $\infty$  als Enabling Degree hat, so wird eine Fehlermeldung ausgegeben und die Simulation abgebrochen. Die beiden anderen Semantiken *vertragen* sich mit einem unbegrenzten Enabling Degree, da nur eine endliche Anzahl Instanzen der Variablenbelegung als Binding erzeugt wird.

**Infinite-Server-Semantik:** Wird ein Binding  $b$  aktiviert, welches mit Infinite-Server-Semantik benutzt werden soll, so wird zuerst das Enabling Degree von  $b$  bestimmt. Ist es unbegrenzt, so wird wie oben besprochen eine Fehlermeldung ausgegeben. Andernfalls werden die bereits erzeugten aktivierten Instanzen von  $b$  gezählt und vom Enabling Degree abgezogen. Ist die Differenz positiv, werden entsprechend viele neue Instanzen als Kopien von  $b$  erzeugt.

**Multiple-Server-Semantik:** Wird ein Binding  $b$  aktiviert, das mit  $k$ -Server-Semantik benutzt werden soll, so wird zuerst das Enabling Degree  $e$  von  $b$  bestimmt. Es werden die bereits erzeugten aktivierten Instanzen von  $b$  gezählt und von  $\min\{e, k\}$  abgezogen. Ist die Differenz positiv, werden entsprechend viele neue Instanzen als Kopien von  $b$  erzeugt.

Betrachten wir nun, wie Age Memory (siehe Unterabschnitt 5.5.1 *Age Memory* ab Seite 42) und die Semantiken mit mehr als einem Server zusammenspielen: Sei  $b$  ein Binding mit Multiple- oder Infinite-Server-Semantik und Age Memory, dessen Zeitintervall  $[\tau_1, \tau_2]$  ist und sei bereits Zeit in  $A(b)$  mit  $2\tau_1 > A(b) > \tau_1$  gespeichert. Angenommen  $b$  würde aktiviert und es würde wie oben besprochen eine Kopie erzeugt werden (Differenz ist 1). Durch das Age Memory steht für die Instanzen von  $b$  noch Zeit zur Verfügung und es wären verschiedene Ansätze denkbar, wie sie auf die Instanzen verteilt werden könnte:

- Jede Instanz verbraucht so viel Zeit aus  $A(b)$  wie nötig ist.
- Alle Instanzen verbrauchen den gleichen Anteil der in  $A(b)$  verbliebenen Zeit, höchstens jedoch ihr jeweiliges  $\tau_1$ .

Im P-UMLaut/Sim wurde die erste Variante umgesetzt.

Das Beispiel in Abbildung 5.2, bei dem  $T_1$  eine Multiple- oder Infinite-Server-Semantik besitze, ist dem aus Unterabschnitt 5.5.1 *Age Memory* ab Seite 42 ähnlich.

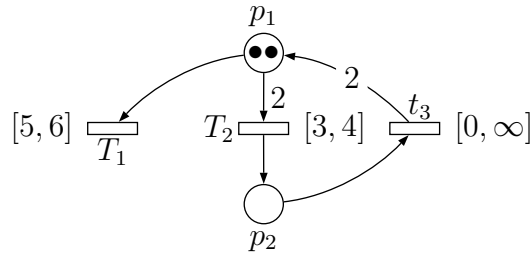


Abbildung 5.2: Multiple-/Infinite-Server-Semantik mit Age Memory

Tabelle 5.2: Möglicher Ablauf des Netzes in Abbildung 5.2

Schritt	Uhr	aktivierte Transitionen	feuerbare T.	Aktion
1	0	$\{ T_1[5, 6], T_1[5, 6], T_2[3, 4] \}$	$\emptyset$	Uhr erhöhen
2	3.5	$\{ T_1[5, 6], T_2[3, 4] \}$	$\{ T_2 \}$	$T_2$ feuern
3	3.5	$\{ t_3[3.5, \infty] \}$	$\{ t_3 \}$	Uhr erhöhen
4	5	$\{ t_3[3.5, \infty] \}$	$\{ t_3 \}$	$t_3$ feuern
5	5	$\{ T_1[5, 6], T_1[8, 9], T_2[8, 9] \}$	$\{ T_1[5, 6] \}$	$T_1$ feuern
6	5	$\{ T_1[8, 9] \}$	$\emptyset$	Uhr erhöhen
7	9	$\{ T_1[8, 9] \}$	$\{ T_1 \}$	$T_1$ feuern
8	9	$\emptyset$	$\emptyset$	-

- Die anfänglich aktivierten Transitionen werden bestimmt. Für beide Bindings von  $T_1$  (als Instanzen einer aktivierender Variablenbelegung) wird die kumulierte Dauer des Aktiviertseins gespeichert ( $\min\{6, A_{T_1}(\{\})\} = 0$ ) und es wird  $A_{T_1}(\{\}) = \max\{0, A_{T_1}(\{\}) - 5\} = 0$  gesetzt. Da keine Transition feuerbar ist, wird die Uhr auf einen zufälligen Wert im Intervall  $[3, 4]$  gesetzt: 3.5
- Nach dem Erhöhen der Uhr ist Transition  $T_2$  als einzige feuerbar. Durch ihr Feuern werden die Marken von  $p_1$  konsumiert und eine auf  $p_2$  erzeugt und die Instanz ihres Bindings verworfen. Transition  $T_1$  wird deaktiviert und es wird für das erste Binding  $A_{T_1}(\{\}) = A_{T_1}(\{\}) + 0 + 3.5 - 0 = 3.5$  zugewiesen, da  $A_{T_1}(\{\}) = 0$  war, sowie für das zweite Binding  $A_{T_1}(\{\}) = A_{T_1}(\{\}) + 0 + 3.5 - 0 = 7$ , da beide Bindings zu Zeitpunkt 0 aktiviert wurden. Zusammen waren beide Bindings also für  $A(b) = 7$  Zeiteinheiten aktiviert.
- Nun ist  $t_3$  aktiviert und sofort feuerbar. Anstatt  $t_3$  zu feuern, wird allerdings die Uhr auf einen beliebigen Wert erhöht: 5

4. Nun wird  $t_3$  gefeuert und die Marke von  $p_2$  konsumiert und zwei auf  $p_1$  erzeugt.
5. Da  $T_1$ ,  $T_2$  aktiviert werden, sind ihre Zeitintervalle zu berechnen:  $5 + [3, 4] = [8, 9]$  für  $T_2$ . Für  $T_1$  wird die Age-Memory-Strategie und Infinite-Server-Semantik verwendet. Das Enabling Degree von  $T_1$  ist 2, folglich werden zwei Instanzen des aktivierenden Bindings erzeugt. Für das erste wird das Intervall auf  $5 + [\max\{0, 5 - A_{T_1}(\{\}) = -2\}, \max\{6 - 5, 6 - A_{T_1}(\{\}) = -1\}] = [5, 6]$  gesetzt und weiterhin wird  $\min\{5, A_{T_1}(\{\})\} = 5$  in der Instanz gespeichert, sowie  $A_{T_1}(\{\}) = \max\{0, A_{T_1}(\{\}) - 5\} = 2$  gesetzt. Somit enthält  $A_{T_1}(\{\}) = 2$  nun die unverbrauchte Zeit für die zweite zu erzeugende Instanz. Für diese wird das Intervall auf  $5 + [\max\{0, 5 - A_{T_1}(\{\}) = 3\}, \max\{6 - 5, 6 - A_{T_1}(\{\}) = 4\}] = [8, 9]$  gesetzt und weiterhin wird  $\min\{5, A_{T_1}(\{\})\} = 2$  in der Instanz gespeichert, sowie  $A_{T_1}(\{\}) = \max\{0, A_{T_1}(\{\}) - 5\} = 0$  gesetzt. Die in  $A_{T_1}(\{\})$  angesammelte Zeit wurde auf Bindings verteilt.  
Zum aktuellen Zeitpunkt 5 ist ein Binding feuert:  $T_1[5, 6]$  Durch sein Feuern wird eine Marke von  $p_1$  konsumiert und die gefeuerte Binding-Instanz verworfen.
6. Durch das Konsumieren der Marke auf  $p_1$  wurde  $T_2$  deaktiviert. Es ist kein Binding feuert, so dass die Uhr auf einen zufälligen Wert aus  $[8, 9]$  gesetzt wird: 9.
7. Transition  $T_1$  ist als einzige feuert. Mit dem Feuern wird die letzte Marke von  $p_1$  konsumiert und die Instanz ihres Bindings verworfen.
8. Es ist keine Transition mehr aktiviert. Das Netz befindet sich in einem Deadlock.

## 5.6 Stochastische Elemente

Durch die Einführung von Raten und Gewichten kann der Petrinetz-Simulator GSPN ausführen. Da Rate und Gewicht exponentialverteilte Zufallsvariablen sind, wird der gängige Mechanismus eingesetzt, mit dem die normalverteilte Zufallswerte der `Random`-Klasse (beziehungsweise der `Math.random()`-Methode) von Java in exponentialverteilte umgerechnet werden. Gegeben eine normalverteilte Zufallsvariable  $x$  und einen Erwartungswert  $\mu$ , so genügt  $-\log(x) \cdot \mu$  der Exponentialverteilung.

Bei jeder Erzeugung/Instantiierung eines Bindings mit Gewicht oder Rate wird der Zufallszahlengenerator benutzt, um einen exponentialverteilten Zufallswert für das Binding

zu erzeugen. Dies ist insbesondere in Rückblick auf die verschiedenen Server-Semantiken zu beachten (siehe Unterabschnitt 5.5.2 *Server-Semantiken* ab Seite 44), bei denen Bindings vervielfältigt werden konnten.

Der konkrete Zufallswert  $w$  einer Rate wird dann als Zeitintervall  $[w, w]$  der Transition verwendet und so behandelt.

Für Gewichte wird die in Abschnitt 5.5 *Zeit* ab Seite 40 beschriebene Ordnung von Bindings auf folgende erweitert:

1. aufsteigende Sortierung über  $\tau_1$  (erster Zeitpunkt, zu dem gefeuert werden kann).
2. absteigende Sortierung über Priorität.
3. absteigende Sortierung über aktuelles Gewicht.
4. beliebige aber feste Sortierung über den künstlichen Schlüssel eines Bindings.

Zum Feuern wird dann unter Beachtung von lokaler oder globaler Priorität eine Transition  $t_1$  ausgewählt. Hat diese Transition ein assoziiertes Gewicht, so wird ähnlich der lokalen Priorität diejenige Transition  $t_2$  in Konflikt mit  $t_1$  zum Feuern ausgewählt, deren Priorität und Gewicht maximal ist.

Der Simulator bietet weiterhin Verallgemeinerungen von Rate und Gewicht an. Dabei wird die Zufallsfunktion vom Benutzer unter Verwendung der normalverteilten Zufallsfunktion `rand()`-Funktion und `getProperty("time")` eingegeben. Auch diese verallgemeinerten Raten und Gewichte werden bei jeder Erzeugung eines Bindings ausgewertet.

## 5.7 Paralleles Feuern

Als Experiment wurde eine Option implementiert, die das zeitgleiche Feuern von Transitionen erlaubt. Dazu wurde das Feuern einer Transition in zwei Teile zerlegt: das Entfernen der Marken von Vorbereichsstellen und das Ablegen von Marken auf den Nachbereichsstellen. Diese beiden Schritte sind atomar.

Durch das Konzept des parallelen Feuerns können Situationen entstehen, bei denen eine Transition, die mehrfach aktiviert ist, wiederholt den ersten Schritt schaltet und

Marken entfernt, bevor sie das erste Mal Marken ablegt. Insbesondere entsteht hierdurch ein vermehrter Arbeitsaufwand, da nach einem teilweisen Schaltvorgang erheblich mehr Transitionen auf Aktiviert sein geprüft werden müssen. In Abschnitt 4.5 *Gefärbte Petrinetze* ab Seite 24 beschriebenen Algorithmus entfällt der letzte Fall [...] **else if**  $s \in t^\bullet \cap \bullet t \wedge s' \neq s$  **then** [...]. Das bedeutet, dass, wenn eine Transition eine Art Doppelkante zu einer Stelle hat und sich durch ein normales Feuern der Transition die Belegung der Stelle nicht ändert, sie sich beim parallelen Feuern zweimal ändert. Wie beschrieben führt das dazu, dass öfter Bindings von mehr Transitionen berechnet werden müssen.

Der Simulator muss weiterhin um eine Menge erweitert werden, die die Transitionen zusammen mit den Variablenbelegungen enthält, deren erster Schritt gefeuert wurde und deren zweiter Schritt noch aussteht.

## 5.8 Entfaltung

Da der Aufwand, einen Petrinetzsimulator so zu erweitern, dass er auch High-Level-Petrinetze in Low-Level-Petrinetze entfalten kann, nicht groß ist, wurde der P-UMLaut/Sim entsprechend angepasst. Es stellte sich heraus, dass sich die vorgenommenen Optimierungen an den Termen auch günstig auf den Aufwand des Entfaltens auswirken. Netze, die mit einem nicht optimierten Entfalter auch nach mehreren Tagen Laufzeit nicht zu entfalten waren, ließen sich nun in unter einer Sekunde verarbeiten.

Beim Entfalten werden die Wertemengen der Variablen nicht durch die Marken eingeschränkt, die auf den Vorbereichsstellen liegen. Im Algorithmus wird für jede Stelle des ursprünglichen Netzes eine Menge von Stellen erzeugt: für jede mögliche Marke der ursprünglichen Stelle eine einzelne Stelle. Es werden für jede Transition alle aktivierenden Variablenbelegungen bestimmt. Für jede dieser Belegungen werden dann die Ein- und Ausgangskanten mit dem gleichen Kantentyp, je nach Marke und Vorkommen derselben (Anzahl Vorkommen als natürliche Zahl), zu den neuen Stellen erzeugt.

Alle Terme an Transitionen werden mit dem entsprechenden Binding ausgerechnet und die Ergebnisse werden an Stelle der Terme an die Transitionen im Low-Level-Petrinetz geschrieben. Das entfaltete Netz ist dann ein Low-Level-Petrinetz: Die Stellen können nur mit schwarzen Marken belegt werden und nur solche Marken können über Kanten

transportiert werden. Um endliche Netze zu erhalten, sollten die Typen der Stellen, wie auch die Wertemengen für Variablen, endlich sein. [EFH<sup>+</sup>05]

## 5.9 Ausführung

Wird der Simulator direkt in einer Kommandozeile aufgerufen, so können die in Tabelle 5.3 erläuterten Argumente übergeben werden. Der Aufruf erfolgt im entsprechenden Verzeichnis mit `java pnmodel.simulator.Simulator [Argumente]`.

Tabelle 5.3: Kommandozeilenparameter des P-UMLaut/Sim

Schalter	Bedeutung
-a	Gibt nach jedem Feuern die aktiven Transitionen aus.
-bC [ANZAHL]	Beendet den Simulator, nachdem Anzahl-viele Transitionen gefeuert wurden.
-bM [STELLE] : [MARK]	Beendet den Simulator, sobald die angegebene Stelle wie angegeben markiert wurde.
-bT [TRANSITION]	Beendet den Simulator, wenn die angegebene Transition gefeuert wurde.
-h	Gibt einen kurzen Hilfstext aus und beendet den Simulator.
-f [DATEI]	Lädt die angegebene Datei im PEP-Format.
-F [ANZAHL]	Nachdem Anzahl-viele Transitionen gefeuert wurden, wird ausgegeben, welche Transitionen noch nicht gefeuert wurden.
-m	Gibt nach jedem Feuern den aktuellen Zustand des Netzes aus.
-m [STELLE]	Gibt nach jedem Feuern die aktuelle Belegung der Stelle aus.
-o [DATEI]	Schreibt das minimierte geladene Netz im PEP-HLNet-Format in die angegebene Datei und beendet den Simulator.
-p	Aktiviert das parallele Feuern.
-r	Setzt das geladene Netz in den Anfangszustand zurück.
-r [DATEI]	Schreibt eine <code>.h1_11_ref</code> -Datei im PEP-Format.
-R	Erweitert die <code>.h1_11_ref</code> -Datei um Informationen zu Werten und Bindings.

-s [LÄNGE]	Gibt nach Beenden des Simulationsprozesses die Feuersequenz aus. Bei zu vielen Schaltvorgängen werden nur die letzten Länge-viele ausgegeben.
-s [SEQUENZ]	Versucht die Transitionssequenz zu feuern.
-t [ANZAHL]	Misst die Feuergeschwindigkeit in pro Sekunde und gibt diese aus, wenn Anzahl-viele Transitionen gefeuert wurden.
-tt [WAHRSCH]	Simuliert das Netz als Timed Transition Petri Net mit strikter Semantik. Mit der angegebenen Wahrscheinlichkeit wird an Stelle eines Feuerns die Uhr erhöht. (etwa -tT0.4)
-tT [WAHRSCH]	Simuliert das Netz als Timed Transition Petri Net mit schwacher Semantik. Mit der angegebenen Wahrscheinlichkeit wird an Stelle eines Feuerns die Uhr erhöht. (etwa -tT0.4)
-u [DATEI]	Entfaltet das geladene Netz in ein Low-Level-Netz und speichert es im PEP-PTNet-Format in der angegebenen Datei und beendet den Simulator.
-v	Gibt aus, welche Transitionen mit welchen Variablenbelegungen gefeuert werden.
-v [REGEX]	Gibt detailliert aus, was beim Ermitteln des enabling bindings passiert, wenn der Transitionsname den (Java-) regulären Ausdruck REGEX erfüllt.
-vv	Gibt aus, wie die Terme der Transitionen vereinfacht wurden.

## 5.10 Performance Tuning

Grundsätzlich wurde P-UMLaut/Sim mit der Annahme implementiert, dass die zu simulierenden Petrinetze *nicht allzu groß* werden und Stellen *nicht allzu viele* Marken enthalten. Damit ist gemeint, dass zum Beispiel eher eine Liste von Bindings zweimal, verschieden sortiert, angelegt wird (siehe Abschnitt 5.5 *Zeit* ab Seite 40), als dass eine Liste aufwändig durchsucht werden muss. Die Laufzeit verbessert sich durch solche Optimierungen bei Petrinetzen ab einer bestimmten Größe, jedoch steigt damit auch der Speicherbedarf des Programms. Für Petrinetze die *nicht allzu groß* sind ist der Geschwindigkeitsgewinn deutlich höher als der Zuwachs des Speicherverbrauchs. Bei der Simulation des in [EFH<sup>+</sup>05] beschriebenen sicheren Petrinetzes `autoelevator.hl_net`

verbraucht der Simulator beim Feuern von einer Million Transitionen etwa 12,7 MB Hauptspeicher (inklusive Java-Virtual Machine) und 21 Sekunden Laufzeit, was etwa 47000 gefeuerten Transitionen pro Sekunde entspricht. Erfahrungsgemäß wirkt sich die Anzahl der Variablen sehr stark auf die Laufzeit aus. Bei der Simulation von Low-Level-Petrinetzen wird eine Geschwindigkeit von etwa 100000 gefeuerten Transitionen pro Sekunde erreicht. Die Maschine, auf der die Tests durchgeführt wurden, ist ein AMD Athlon 64 3200+ mit 2.0 Ghz und einem Gigabyte Hauptspeicher, das Betriebssystem war Windows XP, eingesetzt wurde Java 6 JDK 1.6.0 Beta 2.

Eine Optimierung des Speicherverbrauchs und der Laufzeit stellt das sogenannte Object Pooling (auch bekannt als Object Reuse oder Free List) dar. In den früheren Versionen von Java war der Garbage Collector, wie auch das Erzeugen eines neuen Objekts mittels `new`-Operator langsam und einige Autoren empfahlen Object Pooling einzusetzen [Shi00]. Mittlerweile hat sich das Bild geändert und `new`, sowie der Garbage Collector können in vielen Situationen bedeutend schneller ausgeführt werden. In P-UMLaut/Sim gibt es allerdings einige Typen von Objekten, bei denen durch das Pooling eine Laufzeitverbesserung von bis zu 30% erzielt werden kann. Listen, assoziative Speicher (`Maps`) und Bindings, die fast überall im Simulator verwendet werden, haben oft einen sehr kurzen Lebenszyklus. Die Initialisierung dieser Objekte braucht jedoch vergleichsweise viel Zeit und bei Listen kommt noch das Wachstum hinzu: Aus Effizienzgründen werden `ArrayLists` verwendet, die dynamisch wachsen können. Dazu wird jedes Mal ein neues Feld erzeugt, der Inhalt hinein kopiert und das alte verworfen. Durch das Object Pooling werden größere Listen im Speicher gehalten und es müssen kaum neue Felder, die zusammenhängenden Speicher darstellen, erzeugt werden.

Experimentiert wurde auch mit der Parallelisierung der Berechnung aktivierender Variablenbelegungen während der Simulation, wobei jedem Prozess ein eigener Object Pool und eine Kopie des Petrinetzes zugeteilt wurde. Schaltvorgänge wurden dann an alle Prozesse weitergereicht, die diese parallel auf ihren Netzinstanzen durchführten. Leider konnte keine Geschwindigkeitsverbesserung erzielt werden. In der aktuellen Ausprägung ist P-UMLaut/Sim ein sequentieller Simulator.

Durch das Performance Tuning konnte die Laufzeit des ursprünglichen Simulators übertroffen werden. Bei der Umstellung von Java 2 auf Java 5 büßte der Simulator einiges an Geschwindigkeit ein, was vermutlich an der dynamischen Typüberprüfung von Java 5 lag. Die Geschwindigkeit liegt dennoch über der des alten Simulators.

## 6 Vergleich mit anderen Werkzeugen

„Donner ist gut und eindrucksvoll,  
aber die Arbeit leistet der Blitz.“

Mark Twain

Im Rahmen dieser Arbeit wurde der P-UMLaut/Sim mit weiteren Simulatoren verglichen. Die hier verwendete Hardware ist in Abschnitt 5.10 *Performance Tuning* ab Seite 52 beschrieben.

### 6.1 PEP-NetSim

Der PEP-NetSim ist ein High-Level-Petrinetz-Simulator, dessen Hauptunterschiede zum P-UMLaut/Sim die Menge der unterstützten Petrinetz-Erweiterungen und der Umgang mit Termen von Transitionen sind. Beim PEP-NetSim werden Terme in umgekehrter polnischer Notation (UPN) verwaltet, während sie beim hier betrachteten Simulator als Syntaxbäume gespeichert werden. Zwar sind Terme in UPN im Grunde schneller auszurechnen als Syntaxbäume, jedoch erlauben letztere einfache Termumformungen, da eine Operator-Operanden-Beziehung hier sehr leicht festzustellen und zu verändern ist.

Auch eine Kurzschlussauswertung (steht der Wahrheitswert eines Ausdruckes mit logischem *und* oder logischem *oder* fest, bevor alle Operanden ausgewertet wurden, so kann die Auswertung abgebrochen werden) der logischen Operatoren *und* und *oder* lässt sich in Syntaxbäumen einfach umsetzen, während die UPN diese aufgrund der festen Auswertungsreihenfolge strikt verbietet (an einem Syntaxbaum kann die feste Auswertungsreihenfolge der UPN lax als umgekehrter Breitendurchlauf beschrieben werden, bei dem immer erst die *tiefsten* Blätter ausgewertet werden – bei einem Syntaxbaum hingegen lässt sich die Auswertungsreihenfolge zur Laufzeit frei wählen). [EFH<sup>+</sup>05]

Messungen ergaben eine etwa zehnfache Ausführungsgeschwindigkeit von P-UMLaut/Sim gegenüber dem PEP-NetSim.

## 6.2 *cgpetri*

Der parallele Simulator *cgpetri* ist ein Low-Level-Petrinetz-Simulator, bei dem der Ansatz verfolgt wird, die Berechnung der aktivierten Transitionen auf einem vorhandenen Grafikprozessor durchzuführen. Im vorliegenden Paper [GHSW05] (Herbst 2005) wurde der Simulator auf einem Rechner mit zwei 2.0-Ghz-AMD-Opteron-Prozessoren, zwei Gigabytes Hauptspeicher und einer Nvidia 6800 Ultra mit 256 MB Speicher ausgeführt, das Betriebssystem war Linux 2.6.9.

Die Autoren betrachteten zwei Netze: das wohlbekannte Philosophenproblem und ein Netz zur Anwendung der Lattice-Boltzmann-Methode. Verglichen wurde *cgpetri* mit *xpetri*, und *spnp* die bei den folgenden Betrachtungen überlineares Laufzeitverhalten aufwiesen. Lediglich für sehr kleine Netze (weniger als 60 Philosophen, beziehungsweise weniger als 100 Stellen beim Lattice-Boltzmann-Netz) konnten sie die gestellten Aufgaben schneller als *cgpetri* bewältigen.

Die angegebenen Beispielnetze wurden nachgebaut, so dass die Laufzeiten von *cgpetri* und P-UMLaut/Sim verglichen werden konnten. Für die Simulation von 100000 gefeuerten Transitionen in einem Netz mit 400 Philosophen auf der genannten Hardware benötigte *cgpetri* etwa 42 Sekunden. Auf der anderen, vergleichbaren Maschine, benötigte P-UMLaut/Sim für die gleiche Simulation circa 2.1 Sekunden. Das Laden des Netzes benötigte etwa 750 Millisekunden.

Aufgrund der Struktur des Netzes zum Philosophenproblem sind nach jedem Feuern genau zwei Transitionen auf Aktiviertsein zu prüfen. Während die Laufzeit von *cgpetri* im Bereich von etwa 19 Sekunden bei 25 Philosophen und etwa 42 Sekunden bei 400 Philosophen langsam aber deutlich steigt (linear), nimmt die Laufzeit bei P-UMLaut/Sim nur sehr langsam bis etwa 5 Sekunden für 100000 gefeuerten Transitionen bei 8000 Philosophen linear zu.

Lediglich das Laden des Petrinetzes wirkt sich deutlich auf die Laufzeit aus: bei 8000 Philosophen betrug die Dateigröße im PEP-Format rund 1.8 Megabytes und das Netz brauchte 75 Sekunden um geladen zu werden.

---

Bei den angegebenen Lattice-Boltzmann-Netzen erreichte *cgpetri* eine nahezu konstante Laufzeit von etwa 17 Sekunden für Petrinetze mit 12 bis 400 Stellen und 100000 gefeuerten Transitionen. Für Eingaben zwischen 12 und 17400 Stellen stieg die Laufzeit bei der Simulation durch P-UMLaut/Sim linear von rund 800 Millisekunden auf etwa 2.5 Sekunden. Wieder benötigte das Laden der Netze ab einer gewissen Größe deutlich mehr Zeit als die Simulation. Das Lattice-Boltzmann-Netz mit 17400 Stellen hatte eine Größe von fast einem Megabyte im PEP-Format.

## 7 Zusammenfassung und Ausblick

„Es ist gefährlich, einen extrem fleißigen  
Bürokollegen einzustellen, weil die anderen  
Mitarbeiter ihm dann dauernd zuschauen.“  
Henry Ford

Mit dieser Arbeit wurde der P-UMLaut/Sim so erweitert, dass er GSPN behandeln kann. Dazu wurde nach der Betrachtung der Grundlagen in Kapitel 2 mit Kapitel 3 *Konzept* ab Seite 15 ein Einblick in die vorzunehmenden Erweiterungen geliefert. Kapitel 4 *Implementierung der Basisfunktionalität* ab Seite 18 ging auf die Basis der Arbeit ein und fügte einige Details hinzu. Dort wurde auch die Geschwindigkeit des Simulators angesprochen, die in Abschnitt 5.10 *Performance Tuning* ab Seite 52 genauer betrachtet wurde.

Mit Kapitel 5 *Integration der Erweiterungen* ab Seite 36 wurden die neuen Konzepte und ihre Einbindung in P-UMLaut/Sim besprochen.

Abschließend wurde der Simulator mit seiner Funktionalität und Performance mit zwei anderen Simulatoren verglichen.

Neben Geschwindigkeit war eine Anforderung an P-UMLaut/Sim, dass Petrinetze in den vom Petrinetzwerkzeug PEP [Ste] unterstützten Formaten, unter anderem der Petri Net Markup Language (PNML) [PNM], gelesen und geschrieben werden können.

Bei der Simulation war eine hohe Ausführungsgeschwindigkeit ein wesentliches Designziel. Insbesondere die Wahl geeigneter Datenstrukturen und die Optimierung der Schaltvorgänge der Petrinetze bieten viel Spielraum für Performancegewinne.

Für Low-Level-Netze kann mit wenig Rechenaufwand die Menge der aktivierten Transitionen bestimmt werden. Beim Start der Simulation sind alle Transitionen einmalig mit geringem Aufwand zu prüfen. Nach dem Feuern einer Transition werden nur diejenigen

nichtaktivierten Transitionen auf Aktiviertsein geprüft, die sich im Nachbereich einer Stelle befinden, auf der die Anzahl der Marken erhöht wurde. Analog dazu werden nur die aktivierten Transitionen auf Aktiviertsein geprüft, die sich im Nachbereich einer Stelle befinden, der beim Feuern Marken entnommen wurden. Für die eigentliche Prüfung einer Transition sind nur Vergleiche von Ganzzahlen erforderlich, etwa wie viele Stellen im Vorbereich der Transition sind. Sind auch Lesekanten, Inhibitoranten oder Stellen mit Kapazitäten beteiligt, so wird die Simulation aufwändiger, die Vergleiche bleiben aber prinzipiell gleich.

Bei farbigen Petrinetzen kann ähnlich vorgegangen werden. Statt der Ganzzahlvergleiche sind dann allerdings aufwändige Berechnungen nötig. Variablen mit endlichen Wertebereichen müssen gebunden und logische oder arithmetische Ausdrücke ausgewertet werden. Eine Transition kann in farbigen Petrinetzen mehrfach, mit verschiedenen Variablenbindungen, aktiviert sein.

Erschwert wird die Simulation von Petrinetzen zusätzlich durch das Hinzufügen von partiell globalen Eigenschaften wie Zeit und Prioritäten, für die verschiedene Semantiken in P-UMLaut/Sim angeboten werden. Bei dem *schwachen* Zeitkonzept können Transitionen in einem bestimmten Zeitfenster feuern, bei einem *strikten* müssen sie es. Im Gegensatz zu globalen Prioritäten liegt die Semantik der erweiterten Petrinetze näher an der ursprünglichen Semantik, wenn Prioritäten lediglich lokal, im Kontext eines direkten Konflikts betrachtet werden.

Durch Aufspalten des Feuerns einer Transition in die zwei Phasen des Markenkonsumierens und des Markenausgebens lässt sich die Interleaving-Simulation von Petrinetzen zur quasi-parallelen Simulation erweitern, so dass nebenläufige Transitionen virtuell zum gleichen Zeitpunkt schalten können.

Für die Simulation von High-Level-Netzen, bei der die Geschwindigkeit im Vordergrund steht, bieten sich zudem geeignet gewählte Datenstrukturen, Vorsortierung von Variablen in Ausdrücken, ein möglichst frühes Einschränken der Wertebereiche der Variablen im Kontext einer Transition und eine partielle Auswertung der zu berechnenden Ausdrücke als weitere Optimierungen an.

Durch die Kombination der genannten Petrinetzklassen, Simulationsalgorithmen und Optimierungen ist mit P-UMLaut/Sim ein Petrinetz-Simulator für eine sehr große Anzahl von gefärbten Petrinetz-Typen entwickelt worden. Trotz der komplexen Modelle

weist der Simulator eine sehr gute Geschwindigkeit im Vergleich mit anderen Simulatoren auf. Zudem ermöglicht die Java-basierte Implementierung deutlich einfachere Erweiterungen des Tools und verbesserte Einbindungen des Simulators in Fremdanwendungen.

## 7.1 Ausblick

Derzeit findet P-UMLaut/Sim lediglich Verwendung in P-UMLaut [EFH<sup>+</sup>05], eine Anbindung an das PEP-Tool [Ste] wäre aufgrund der vom Simulator bereitgestellten Schnittstellen denkbar und mit geringem Aufwand umzusetzen.

Neben der Erweiterung um neue Konzepte wie selbstmodifizierende Petrinetze oder Stellen, die sich wie FIFO-Puffer verhalten, kann der P-UMLaut/Sim unter anderem an folgenden Punkten fortentwickelt werden.

### 7.1.1 Funktionalität

In dieser Arbeit wurden einige Alternativen zu Designentscheidungen angesprochen. Ein Punkt an dem die Arbeit an P-UMLaut/Sim fortgeführt werden sollte, wäre das Einbauen von alternativen Vorgehensweisen und das Hinzufügen von Schaltern, über die der Benutzer festlegen kann, welches Verhalten er simuliert haben möchte. Einige Alternativen wären:

- In Abschnitt 5.1 *Kapazitäten* ab Seite 36 wurde die Inkonsistenz im Umgang mit Kapazitäten von Stellen beschrieben. Es sollten wahlweise Kapazitäten eingeführt werden, die kompatibel mit der Entfaltung zu Low-Level-Petrinetzen sind – ihre Semantik folglich erhalten bleibt.
- In Unterabschnitt 5.5.2 *Server-Semantiken* ab Seite 44 wird ähnliches wie für Kapazitäten gesagt. Hier wird allerdings in jedem Fall kompatibel zur Entfaltung vorgegangen. Dem Benutzer sollte hier die Option gegeben werden, die Server-Semantiken auf Ebene des High-Level-Petrinetzes zu behandeln. Bindings einer Transition würden dann nicht nach Variablenbelegungen unterschieden, sondern alle zusammen gezählt werden.

- Unterabschnitt 5.5.1 *Age Memory* ab Seite 42 spricht unter anderem eine Designentscheidung an, die festlegt, wie viel Zeit für ein Binding aus dem Age-Memory-Speicher entnommen werden soll. Andere Überlegungen wären optional zu implementieren.
- In Unterabschnitt 5.5.2 *Server-Semantiken* ab Seite 44 wird auf das Zusammenspiel von Age Memory und Multiple- oder Infinite-Server-Semantik eingegangen. Auch hier können andere Verteilungen der gesammelten Zeit auf die aktivierten Bindings eingebunden werden.

### 7.1.2 Performance Tuning

Zur Erhöhung der Geschwindigkeit bieten sich folgende Punkte zur Weiterarbeit an:

- Wie in Kapitel 4 *Implementierung* ab Seite 18 bereits erwähnt wurde, werden Low-Level-Petrinetze bei der Simulation in High-Level-Petrinetze umgewandelt. Bei Low-Level-Netzen ist der Aufwand der Berechnung, ob eine Transition aktiviert ist oder nicht, allerdings sehr viel geringer als bei High-Level-Netzen. Deswegen läge es nahe, den P-UMLaut/Sim so zu erweitern, dass er Low-Level-Petrinetze speziell behandelt und stärker optimiert simuliert - unter anderem wären Typüberprüfungen herauszunehmen.
- Die Rekursion von benutzerdefinierten Funktionen (siehe Abschnitt 4.8 *Funktionen* ab Seite 32) ist derzeit durch Kopieren der Syntaxbäume während der Auswertung gelöst. Es wäre zu überlegen, ob solche Funktionen effizienter auszuwerten sind.
- Da Stellenbelegungen in der aktuellen Implementierung als Listen gespeichert werden, ist jede Prüfung auf Enthaltensein eines Wertes eine lineare Suche. Insbesondere, wenn große Mengen oder Multimengen vorliegen bietet es sich viel eher an, Werte sortiert oder in einem möglichst ausgeglichenen Baum zu speichern. Es wäre eine Ordnung auf Werten, Tupeln und Variablen einzuführen und eine entsprechende Datenstruktur zu wählen.

# Abbildungsverzeichnis

2.1	P/T-Netz und Feuern . . . . .	3
2.2	Ein direkter Konflikt . . . . .	4
2.3	Inhibitorkanten . . . . .	5
2.4	Resetkanten . . . . .	6
2.5	Ein indirekter Konflikt . . . . .	7
2.6	Ein TTPN . . . . .	8
2.7	Ein GSPN . . . . .	11
2.8	Verschiedene Variablenbelegungen . . . . .	12
2.9	Wächter . . . . .	13
5.1	Behandlung von Age Memory . . . . .	43
5.2	Multiple-/Infinite-Server-Semantik mit Age Memory . . . . .	47

# Literaturverzeichnis

- [ANT] Homepage des LL(\*)-Parsergenerators ANTLR (Another Tool for Language Recognition). <http://www.antlr.org/>, letzter Zugriff 20.07.2006.
- [Bal01] Gianfranco Balbo. Introduction to Stochastic Petri Nets. In: Ed Brinksma, Holger Hermanns und Joost-Pieter Katoen, Herausgeber, *Lectures on Formal Methods and Performance Analysis*, Band 2090 von *Lecture Notes in Computer Science*, S. 84–155. Springer-Verlag, 2001.
- [CUP] Homepage des LALR-Parsergenerators CUP. <http://www2.cs.tum.edu/projects/cup/>, letzter Zugriff 20.07.2006.
- [EFH<sup>+</sup>05] Christoph Eichner, Eike Frost, Martin Hilscher, André Kaiser, Roland Meyer, Mark Ross, Casjen Schnars, Ulrik Schrimpf und Tim Strazny. Projektgruppe P-UMLaut. Endbericht, *Carl von Ossietzky Universität Oldenburg*, 2005.
- [EFM<sup>+</sup>05] Christoph Eichner, Hans Fleischhack, Roland Meyer, Ulrik Schrimpf und Christian Stehno. Compositional Semantics for UML 2.0 Sequence Diagrams Using Petri Nets. In: Andreas Prinz, Rick Reed und Jeanne Reed, Herausgeber, *SDL 2005*, Band 3530 von *Lecture Notes in Computer Science*, S. 133–148. Springer-Verlag, 2005.
- [FG97] Hans Fleischhack und Bernd Grahlmann. A Petri Net Semantics for B(PN)<sup>2</sup> with Procedures. In: Gul Agha und Stefano Russo, Herausgeber, *Parallel and Distributed Software Engineering*, S. 15–27. IEEE Computer Society, Mai 1997.
- [FG98] Hans Fleischhack und Bernd Grahlmann. A Compositional Petri Net Semantics for SDL. In: J. Desel und M. Silva, Herausgeber, *ICATPN*, Band 1420 von *Lecture Notes in Computer Science*, S. 144–164. Springer-Verlag, 1998.

- [Fla05] David Flanagan. *Java in a Nutshell*. O'Reilly, Sebastopol, 2005. ISBN: 0-596-007-73-6.
- [GHSW05] Robert Geist, Jacob Hicks, Mark Smotherman und James Westall. Parallel Simulation of Petri Nets On Desktop PC Hardware. In: M. E. Kuhl, N. M. Steiger, F.B. Armstrong und J. A. Joines, Herausgeber, *2005 Winter Simulation Conference*, Band WSC 05, Orlando, FL, 2005.
- [MBC<sup>+</sup>95] M. Ajome Marsan, G. Balbo, G. Conte, S. Donatelli und G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley, 1995.
- [McL02] Bret McLaughlin. *Java & XML*. O'Reilly, Köln, 2002. ISBN: 3-897-213-37-0.
- [Pet62] C. A. Petri. Kommunikation mit Automaten. Technischer Bericht RADCTR-65-377, Griffiss Air Force Base, 1962.
- [PNM] Petri Net Markup Language. <http://www2.informatik.hu-berlin.de/top/pnml/>, letzter Zugriff 13.07.2006.
- [PNW] Homepage der Petri Nets World. <http://www.informatik.uni-hamburg.de/TGI/PetriNets/>, letzter Zugriff 21.07.2006.
- [PUM] Project P-UMLaut. <http://www.p-umlaut.de/>, letzter Zugriff 13.07.2006.
- [PW02] Lutz Priese und Harro Wimmel. *Theoretische Informatik: Petri-Netze*. Springer-Verlag, 2002. ISBN: 3-540-44289-8.
- [Rob04] Robert Simmons, Jr. *Hardcore Java*. O'Reilly, Sebastopol, 2004. ISBN: 0-596-005-78-7.
- [Sed03] Robert Sedgewick. *Algorithms in Java, Third Edition, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*. Addison-Wesley, 2003. ISBN: 0-201-36120-5.
- [Sed04] Robert Sedgewick. *Algorithms in Java, Third Edition, Parts 5: Graph Algorithms*. Addison-Wesley, 2004. ISBN: 0-201-36120-3.
- [Shi00] Jack Shirazi. *Java Performance Tuning*. O'Reilly, Sebastopol, 2000. ISBN: 0-596-000-15-4.

- [Smi98] Einar Smith. Principles of High-Level Net Theory. In: Wolfgang Reisig und Grzegorz Rozenberg, Herausgeber, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, Band 1491 von *Lecture Notes in Computer Science*, S. 174–210. Springer-Verlag, 1998.
- [Ste] Christian Stehno. Homepage des PEP-Tools (Programming Environment based on Petri Nets). <http://theoretica.informatik.uni-oldenburg.de/~pep/>, letzter Zugriff 13.07.2006.

# Glossar

## A

**Age Memory** Im Gegensatz zur Enabling-Memory-Strategie *vergisst* eine Uhr nicht, wenn bereits Zeit verstrichen ist. Stattdessen läuft die Uhr immer dann weiter, wenn die Transition aktiviert ist und wird erst beim Feuern zurückgesetzt. Die Uhr misst dann die kumulierte Dauer der Aktivierungen.

**Aktivierung** Eine Transition wird in einem Petrinetz *aktiviert*, wenn alle durch Ein- und Ausgangskanten gestellten Bedingungen erfüllt sind und der Wächter der Transition zu *wahr* ausgewertet werden kann.

**Arc** siehe Kante

## B

**beschränkt** Kann eine Stelle aufgrund der Struktur des Petrinetzes mit höchstens  $k$  Token gleichzeitig belegt werden, so ist die Stelle *k-beschränkt*. Gibt es kein  $k$  für das die Stelle beschränkt ist, so ist sie unbeschränkt. Sind alle Stellen eines Netzes beschränkt, so ist das Netz beschränkt.

**Beschriftung** Einem Objekt in textueller Form hinzugefügte Information.

**Binding** Eine spezielle Variablenbelegung aller im Kontext einer Transition existierenden Variablen unter der eine Transition feuern kann. Unter welchen Bindings eine Transition feuern kann ist vom aktuellen Zustand des Petrinetzes abhängig.

**Brute-Force** Brute-Force- oder auch Rohe-Gewalt-Prinzip ist eine Lösungsmethode, die in der Informatik und der Spieltheorie für die Lösung schwerer Probleme genutzt wird. Brute Force probiert (nahezu) alle Varianten zur Lösung eines Problems, bis das Problem gelöst ist.

## C

**Computational Tree Logic** Die Computational Tree Logic (CTL) ist eine temporale Logik, die zum Beispiel zur Beschreibung von Eigenschaften im Rahmen der Verifikation von bestimmten Petrinetzen eingesetzt werden kann. Die CTL ist eine Obermenge der LTL.

**CTL** siehe Computational Tree Logic

## E

**Enabling Degree** Das Enabling Degree ist eine natürliche Zahl oder  $\infty$ , die aussagt, wie oft ein Binding direkt hintereinander feuerngefeuert werden könnte.

**Enabling Memory** Wenn eine Transition deaktiviert wird, also ihre Aktivierung verliert, so wird ihre Uhr zurückgesetzt. Die Uhr misst dann die Dauer der längsten durchgehenden Aktivierung.

**Entfaltung** Im Bezug auf Petrinetze meint eine Entfaltung eines High-Level-Netzes das Erstellen eines semantisch äquivalenten Low-Level-Netzes: High-Level-Netze können als Kurzschreibweisen für komplexe Vorgänge bezeichnet werden, die sich auch durch die einfacheren Low-Level-Netze ausdrücken lassen. Eine Entfaltung nimmt diese Vereinfachung vor.

**Erreichbarkeitsgraph** Nimmt man für ein Petrinetz alle vom Startzustand aus erreichbaren Zustände als Knoten und fügt zwischen zwei Knoten  $M, M'$  genau dann eine mit  $t$  beschriftete Kante ein, wenn Transition  $t$  in  $M$  aktiviert ist und in Zustand  $M'$  führt.

**Extensible Markup Language** Die Extensible Markup Language (XML) ist eine 1998 vom W3C standardisierte Sprache auf der Basis von SGML, mit der sich Auszeichnungssprachen definieren lassen. Ursprünglich als Format für die elektronische Publikation erstellt, spielt XML heute eine immer wichtigere Rolle beim Austausch von Daten in der Welt.

## F

**Feuern** siehe Schaltvorgang

## G

**Generalized Stochastic Petri Net** Die verallgemeinerten stochastischen Petrinetze bilden eine Oberklasse der SPN. Hier können auch zeitfreie Transitionen mit Prioritäten und Zufallsvariablen, den Gewichten, vorkommen.

**Gewicht** In GSPN können zeitfreie Transitionen mit einer Zufallsvariablen ausgestattet werden. Bei gleicher Priorität wird über das Gewicht entschieden, welche Transition bevorzugt geschaltet wird. Siehe auch Rate.

**GSPN** siehe Generalized Stochastic Petri Net

## H

**High-Level-Petrinetz** Ein komplexer Typ von Petrinetzen, bei dem Transitionen mit Wächter-Termen, Kanten mit Beschriftungen (Datenwerte, Variablen, Terme) und Stellen mit komplexen Typen vorkommen können. Durch High-Level-Netze können komplexere Systeme meist einfacher modelliert werden als über die Low-Level-Netze. Eine Überführung von High-Level-Netze in semantisch äquivalente Low-Level-Petrinetze stellt die Entfaltung dar.

## I

**Infinite-Server-Semantik** Die Transition behandelt alle Aktivierungen *gleichzeitig*, beziehungsweise sobald eine neue Aktivierung entsteht. Für jede Aktivierung wird eine Uhr initialisiert, die dann parallel zu allen weiteren Uhren läuft.

**Inhibitorkante** Eine Kante, die von einer Stelle  $s$  zu einer Transition führt und sie nur dann aktiviert, wenn  $s$  eine bestimmte Anzahl bestimmter Marken nicht enthält – zum Beispiel leer ist.

**Interleaving** Im Gegensatz zur Parallelität treten bestimmte Ereignisse beim Interleaving quasiparallel, d.h. in beliebiger Reihenfolge aber streng sequenziell auf. In Bezug auf Petrinetze: das Feuern einer Transition wird als atomar angenommen und zwei Transitionen schalten nie gleichzeitig.

## J

**Java** Objektorientierte Programmiersprache, bei der der Quelltext vor der Ausführung in einen Zwischencode übersetzt wird, der dann interpretiert wird. Seit Version 1.5 (Java 5) werden sogenannte *Generics* (generische Typen) angeboten.

## K

**Kante** Im Bezug auf Petrinetze verbindet eine Kante entweder eine Transition mit einer Stelle oder eine Stelle mit einer Transition. Kanten tragen als Beschriftungen, welche Marken sie bewegen, wobei hier Variablen oder auch Terme zugelassen sind.

**Kapazität** Bekommt eine Stelle eine Kapazität  $k$  zugewiesen, so kann sie mit höchstens  $k$ -viele Token belegt werden. Würde eine Transition diese Begrenzung bei der Erzeugung von Token überschreiten, so ist sie nicht aktiviert.

**Konflikt** Zwei Transitionen stehen in Konflikt, wenn sie sich eine oder mehrere Stellen teilen, von denen sie durch Eingangskanten abhängen.

## L

**Lesekante** Eine Kante, die von einer Stelle  $s$  zu einer Transition führt und sie nur dann aktiviert, wenn  $s$  die wie bei normalen Kanten geforderten Marken enthält. Im Gegensatz zu normalen Kanten werden die Marken auf  $s$  nicht konsumiert. Lesekanten stellen eine Art Test oder zusammengefasste Doppelkante dar. Wahrnehmbar wird dieses andere Verhalten beim parallelen Feuern.

**Linear Temporal Logic** Die Linear Temporal Logic (LTL) ist eine temporale Logik, die zum Beispiel zur Beschreibung von Eigenschaften im Rahmen der Verifikation von bestimmten Petrinetzen eingesetzt werden kann. Die LTL ist eine Untermenge der CTL.

**Low-Level-Petrinetz** Ein einfacher Typ von Petrinetzen, bei dem Transitionen ohne Terme, Kanten ohne Beschriftungen und Stellen nur mit Typ **dot** vorkommen.

**LTL** siehe Linear Temporal Logic

## M

**M-Netz** Ein High-Level-Petrinetz, in dem farbige Token, d.h. natürliche Zahlen, als Datenwerte erlaubt sind.

**Marke** Im Bezug auf Petrinetze stellen Marken Datenwerte (wie **wahr**, **falsch**, 38, -92, 0) dar, die auf den Stellen gehalten werden können und durch Transitionen bewegt werden können. Variablen können mit solchen Datenwerten belegt werden.

**Markierung** Belegung einer Petrinetz-Stelle mit Token. Oder, im Bezug auf ein ganzes Petrinetz: Die aktuelle Belegung aller Stellen (auch Zustand genannt).

**Multimenge** Die Besonderheit von Multimengen gegenüber dem gewöhnlichen Mengenbegriff besteht darin, dass die Elemente einer Multimenge mehrfach vorkommen können.

**Multiple-Server-Semantik** Bei einer  $k$ -Server-Semantik werden höchstens  $k$  Aktivierungen *gleichzeitig* behandelt. Es stehen  $k$  Uhren bereit, die parallel für höchstens  $k$ -viele Aktivierungen laufen. Wenn alle Uhren in Verwendung sind, warten überzählige Aktivierungen, bis die Transition feuert und eine Uhr frei wird.

**MultiSet** siehe Multimenge

## N

**Nachbereich** Im Bezug auf eine Transition bezeichnet der Nachbereich alle die Stellen, zu denen Kanten von dieser Transition existieren. Für eine Stelle bezeichnet der Nachbereich alle die Transitionen, zu denen Kanten von dieser Stelle existieren.

## P

**P-UMLaut** P-UMLaut war eine Projektgruppe im Wintersemester 2004/2005 und Sommersemester 2005 an der *Carl von Ossietzky Universität Oldenburg*, Department für Informatik, Abteilung Parallele Systeme, in der ein Werkzeug zur Übersetzung von UML-2.0-Sequenzdiagrammen in High-Level-Petrinetze und dreidimensionalen Visualisierung des Verhaltens entwickelt wurde. Siehe dazu [EFH<sup>+</sup>05].

**P/T-Netz** siehe Low-Level-Petrinetz

**PDF** siehe Probability Density Function

**PEP** Das an der *Carl von Ossietzky Universität Oldenburg* entwickelte Programm *Programming Environment based on Petri Nets* bietet eine Vielzahl von Werkzeugen zur Modellierung, Kompilierung, Simulation und Verifikation sowie eine grafische Oberfläche an.

**PEP-NetSim** Ein in das PEP-Tool eingebauter Simulator für Petrinetze.

**Petri Net Markup Language** Ein XML-basiertes Datenaustauschformat für Petrinetze mit Wurzeln an der Humboldt Universität Berlin entwickelt.

**Petrinetz** Ein Petrinetz ist ein mathematisches Modell von nebenläufigen Systemen. Sie verallgemeinern wegen der Fähigkeit, nebenläufige Ereignisse darzustellen, die Automatentheorie. Ein Petrinetz ist ein bipartiter und gerichteter Graph. Er besteht aus Stellen (Places) und Transitionen (Transitions). Stellen und Transitionen sind durch gerichtete Kanten verbunden. Es gibt keine direkten Verbindungen zwischen zwei Stellen oder zwei Transitionen.

**Place** siehe Stelle

**PNML** siehe Petri Net Markup Language

**Priorität** Sind mehrere Transitionen eines Petrinetzes gleichzeitig aktiviert, so werden die mit höherer Priorität beim Feuern bevorzugt.

**Probability Density Function** Die (stetige) Dichtefunktion einer Zufallsvariablen.

## R

**Rate** In SPN werden zeitbehaftete Transitionen mit einer Zufallsvariablen ausgestattet. Die von der Transition zu verbrauchende Zeit wird über diese Zufallsvariable ermittelt. Siehe auch Gewicht.

**regulärer Ausdruck** Gängige und kurze Beschreibung von (Unter-)Mengen von Zeichenketten.

**Resetkante** Eine Kante, die von einer Stelle  $s$  zu einer Transition  $t$  führt und  $s$  mit dem Feuern von  $t$  leer räumt. Für die Aktivierung von  $t$  ist es unerheblich wie viele Token auf  $s$  liegen.

**Reverse Polish Notation** siehe umgekehrte polnische Notation

**RPN** siehe Reverse Polish Notation

## S

**Schaltvorgang** (auch Feuern) Im Bezug auf Petrinetze beschreibt ein Schaltvorgang einer Transition (unter einem bestimmten Binding) das Entfernen der entsprechenden Token von den Vorbereichsstellen der Transition und Ablegen von entsprechenden Token auf den Nachbereichsstellen der Transition.

**sicher** Ein Petrinetz ist *sicher*, wenn alle seine Stellen 1-beschränkt sind, also jeweils nicht mehr als eine Marke enthalten können.

**Simulation** Im Bezug auf Petrinetze bestimmt ein Simulator die aktivierenden Variablenbelegungen (Bindings) für die Transitionen des Netzes und kann auch Schaltvorgänge durchführen.

**Simulator** siehe Simulation

**Single-Server-Semantik** Mehrfache Aktivierungen werden *nacheinander* behandelt. Die Transition hat eine einzige Uhr, die für die erste Aktivierung läuft, bis sie gefeuert wird. Danach wird die Uhr für die nächste mögliche Aktivierung zurück gesetzt.

**SPN** siehe Stochastic Petri Net

**Stelle** Als passive Elemente in Petrinetzen können Stellen mit Token markiert/belegt werden. Grafische Darstellung: Kreis

**Stellentyp** Petrinetz-Stellen sind typisiert. Ihr Typ gibt an, welche Token auf die Stelle gelegt werden können.

**Stochastic Petri Net** Ein Petrinetz mit zeitbehafteten Transitionen, deren Zeitpunkte der Schaltvorgänge durch Zufallsvariablen bestimmt werden (Rate).

**String** Zeichenkette

**Syntaxbaum** Darstellungsweise von Termen. Operatoren werden als innere Knoten mit ihren Operanden als Kinderknoten dargestellt. Konkrete Werte oder Variablen bilden die Blätter des Baumes.

## T

**Term** Mathematischer Ausdruck

**Timed Transition Petri Net** Petrinetze bei denen Transitionen reelle oder ganzzahlige Intervalle als Zeitinschriften tragen. Nach der Aktivierung kann/muss eine Transition in dem durch die Inschrift festgelegten Zeitintervall feuern.

**Token** siehe Marke

**Transition** Aktives Element eines Petrinetzes. Beim Feuern einer Transition wird die Markierung von Stellen des Vor- und Nachbereichs geändert. Transitionen haben ein- und ausgehende Kanten und einen Term, der für das Feuern erfüllt sein muss. Transitionen können mit weiteren Beschriftungen versehen werden. Grafische Darstellung: Rechteck

**TTPN** siehe Timed Transition Petri Net

**Tupel** Geordnete Zusammenstellung von Objekten, etwa  $(3, 6, X, -1.8)$ .

**Turing-vollständig** Eine Programmiersprache oder eine abstrakte Maschine ist Turing-vollständig, wenn sie die gleichen Berechnungen wie eine Turingmaschine durchführen kann, also ebenso mächtig ist.

## U

**umgekehrte polnische Notation** (englisch: Reverse Polish Notation, RPN) Notation für die Anwendung von Operationen. Bei der umgekehrten polnischen Notation werden zunächst die Operanden eingegeben und danach der darauf anzuwendende Operator.

**unbeschränkt** Ist eine Stelle nicht beschränkt, so ist sie *unbeschränkt*. Analog gilt dies für Petrinetze

**UPN** siehe umgekehrte polnische Notation

## V

**Variablenbelegung** Ersetzungsvorschrift für eine Menge von Variablen durch konkrete Werte.

**Verifikation** Hier: der (unter Umständen automatisch durchgeführte) mathematische Beweis, dass ein System eine Eigenschaft erfüllt oder nicht erfüllt.

**Vorbereich** Im Bezug auf eine Transition bezeichnet der Vorbereich alle die Stellen, von denen Kanten zu dieser Transition existieren. Für eine Stelle bezeichnet der Vorbereich alle die Transitionen, von denen Kanten zu dieser Stelle existieren.

## X

**XML** siehe Extensible Markup Language

## Z

**Zustand** Im Bezug auf Petrinetze: Die aktuelle Belegung aller Stellen mit Token.

## **Eigenständigkeitserklärung**

Ich erkläre hiermit, dass ich diese Arbeit selbständig angefertigt habe und alle Teile, die wörtlich oder inhaltlich anderen Quellen entstammen, als solche kenntlich gemacht und in das Literaturverzeichnis aufgenommen habe. Diese Arbeit wurde weder in dieser noch einer ähnlichen Form einer anderen Prüfungsbehörde vorgelegt.

Tim Strazny

Oldenburg, Juli 2006