Simulation zeitdiskreter Modelle mit Referenznetzen

Diplomarbeit

Universität Hamburg

Fachbereich Informatik

Arbeitsbereich Theoretische Grundlagen der Informatik

November 2003

Frauke Strümpel
EMail: 7struemp@informatik.uni-hamburg.de

Erstbetreuer: Dr. Daniel Moldt
Zweitbetreuer: Prof. Dr. Ing. Bernd Page
Fachbetreuer: Michael Köhler

Inhaltsverzeichnis

1 Einleitung

Die Computersimulation ist eine der ältesten Anwendungen der Informatik. Sie ist ein geeignetes und wichtiges Mittel zur Modellierung und Analyse komplexer Systeme. In den letzten Jahren hat die Simulation immer mehr Einzug in verschiedene Fachgebiete erhalten, da es immer wichtiger wird, beispielsweise aus Kostengründen oder zur Korrektur falscher Einschätzungen, komplexe Szenarien vor einer Umsetzung zu simulieren.

Es existieren verschiedene Arten der Simulation: U.A. die kontinuierliche, die quasikontinuierliche und die diskrete Simulation. Die diskrete Simulation kann in die zustandsdiskrete und die zeitdiskrete Simulation unterteilt werden. Als Beispiel für die zeitdiskrete Simulation sind insbesondere Bedien-/Wartesysteme zu nennen, die einen großen Bereich der Systeme abdecken, die in der Arbeitswelt auftreten und deshalb von großem Interesse sind. In dieser Arbeit wird deshalb speziell auf die zeitdiskrete Simulation eingegangen, da sie sich besonders gut für die Analyse von Prozessen eignet, bei denen es sich um eine Abfolge von Ereignissen zu unterschiedlichen Zeitpunkten handelt.

Es gibt eine Vielzahl von Programmiersprachen, die speziell für diese Anwendungen entwickelt wurden und die je nach Einsatzgebiet besser oder schlechter zur Modellierung eines jeweiligen Systems geeignet sind. Ein neuerer Ansatz ist die Simulation mit Petrinetzen. Petrinetze bieten aufgrund ihrer grafischen Darstellung und ihres mathematischen Hintergrundes eine sehr gute Möglichkeit, komplexe Sachverhalte schnell zu erfassen, zu modellieren und darzustellen.

Es sollen in dieser Arbeit deshalb die drei Bereiche *Modellierung*, *Simulation* und *Petrinetze* vorgestellt werden und deren Zusammenhang erarbeitet werden. Es wurde bereits 1990 in [Käm90] auf dieses Thema eingegangen. In der Arbeit wurde gezeigt, wie mit Petrinetzen (speziell mit S/T-Netzen und gefärbten Netzen) Prozesse modelliert und Simulationsmodelle erstellt werden können. Mittlerweile existieren aber neuere, geeignetere Formalismen für höhere Petrinetze (beispielsweise Referenznetze) und zur Simulation einsetzbare Petrinetzeditoren und -simulatoren (beispielsweise Renew), die einen Wiederaufgriff dieses Themas rechtfertigen.

Das Ziel dieser Arbeit ist es, Petrinetze (speziell Referenznetze) zum Modellieren von Szenarien und zur Simulation dieser Szenarien einzusetzen. Dazu werden die in Abbildung 1.1 dargestellten Begriffe Modellierung, Simulation und Petrinetze jeweils für sich und im Zusammenhang zueinander erklärt. Die Abbildung 1.1 erklärt anhand einer Grafik auf einen Blick worum es in der Arbeit im Wesentlichen geht. Jeder der Bereiche (Modellierung, Simulation und Petrinetze) wird an einer Ecke dargestellt und beschreibt ein Thema der Arbeit. Die Kanten des Dreiecks beschreiben die Zusammenhänge der jeweiligen Bereiche. Es lässt sich an der Abbildung ablesen, dass demzufolge alle drei Bereiche in einem Zusammenhang zueinander stehen. Diese Zusammenhänge sollen ebenfalls

in den folgenden Kapiteln ausführlich erläutert werden.

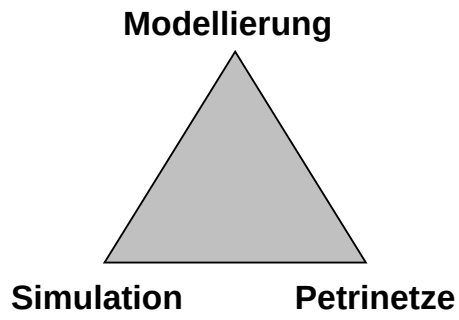

Abbildung 1.1: Modellierung, Simulation, Petrinetze

Es stellt sich die Frage, wie Petrinetze so eingesetzt werden können, dass man mit ihnen Simulationsexperimente durchführen kann. Wie bereits erwähnt, eignen sich Petrinetze aufgrund ihrer Struktur und ihres eindeutigen Formalismus sehr gut zur Modellierung von Systemen. Darüber hinaus gibt es zusätzlich eine Reihe verschiedener Petrinetzwerkzeuge, die man für die Simulation einsetzen kann, da sie bereits über Netzsimulatoren verfügen. Diese Netzsimulatoren stellen i.A. erweiterte Petrinetzformalismen zur Verfügung, die über den Formalismus des allgemein bekannten S/T-Netzes hinausgehen und als höhere Petrinetze bezeichnet werden. In dieser Arbeit wird als Werkzeug speziell der Petrinetzeditor und -simulator Renew benutzt. D.h., es werden höhere Petrinetzformalismen vorgestellt, die von Renew unterstützt werden und es werden nur Beispiele gezeigt, die mit Renew entwickelt wurden.

Weiter stellen sich die Fragen, welche Art von Ergebnissen bei der Simulation ausgewertet werden soll und wie man Netzsimulatoren einsetzen bzw. erweitern kann, um Simulationsergebnisse zu extrahieren. Es soll dazu für den Petrinetzeditor und -simulator Renew eine Auswertungskomponente entwickelt werden.

Um diese Fragen beantworten zu können, werden zuerst in Kapitel 2 die diskrete Simulation mit ihren verschiedenen Sichtweisen und die Modellbildung vorgestellt. Zu den verschiedenen Sichtweisen innerhalb der diskreten Simulation gehören die *ereignisorientierte*, die *transaktionsorientierte*, die *prozessorientierte* und die *aktivitätsorientierte* Sichtweise. Man kann zusammenfassend auch von der material- (ereignisorientiert und transaktionsorientiert) und der maschinenorientierten (prozessorientiert und aktivitätsorientiert) Sichtweise sprechen. Es wird zusätzlich gezeigt, was bei der Auswertung von Simulationsexperimenten bzw. -ergebnissen beachtet werden muss, was bei der Auswertung zum Erkenntnisgewinn relevant ist und was nicht.

Nach einer allgemeinen Einführung in Petrinetze in Kapitel 3 wird gezeigt, wie sich diese Sichten auf eine Modellierung mit Petrinetzen übertragen lassen. Es wird anhand von Beispielen beschrieben, wie Petrinetze zur Modellierung von Szenarien eingesetzt werden können. Danach werden die Vor- und Nachteile der Simulation mit Petrinetzen gegenüber imperativen Ansätzen diskutiert.

Über die S/T-Netze hinaus gibt es zusätzlich die Möglichkeit, mit höheren

Petrinetzen zu modellieren und zu simulieren. Zu den höheren Petrinetzen gehören u.a. die *gefärbten Netze*, die *Objekt-Petrinetze*, die *Netze in Netzen* und die *Referenznetze*. Es werden zu jedem Netztyp in Kapitel 4 jeweils eine Einführung und ein Beispiel gegeben. Bei der Durchführung von Simulationsexperimenten spielen Zeitverbrauch und stochastische Verteilungen eine große Rolle. Damit der Zeitverbrauch auch bei Petrinetzsimulationen berücksichtigt wird, wird zusätzlich eine Einführung in zeitbehaftete Petrinetze gegeben. Es werden die verschiedenen Formen der zeitbehafteten Petrinetze diskutiert und es wird der Zusammenhang zwischen Petrinetz, Erreichbarkeitsgraph und Markovkette hergestellt.

Um mit Referenznetzen zu modellieren und zu simulieren, wird das Petrinetz-Werkzeug Renew verwendet, das in Kapitel 5 vorgestellt wird. Renew ist ein von Olaf Kummer und Frank Wienberg an der Universität Hamburg im Fachbereich Informatik entwickelter Petrinetzeditor und -simulator, der Referenznetze unterstützt. Nach einer Einführung in die Benutzung von Renew wird gezeigt, wie Referenznetze in Renew entwickelt werden. Referenznetze verfügen über noch weiter reichende Eigenschaften als die anderen erwähnten höheren Netzformalismen und sind deshalb für den Einsatz in der Simulation besonders gut geeignet. Es wird gezeigt, wie Java-Klassen bzw. Methoden in Renew benutzt werden können und wie die integrierte Zeitkomponente in Renew aussieht.

Ein wichtiges Instrument zur Strukturierung der Simulationsmodelle ist die Objektorientierung. Es wird in Kapitel 6 gezeigt, warum die Objektorientierung für die Simulation so geeignet und deshalb so wichtig ist und warum herkömmliche imperative Sprachen weniger geeignet sind. Dazu wird ein kurzer Überblick über die Objektorientierung im Allgemeinen gegeben, um dann auf objektorientierte Programmiersprachen zu kommen. Auch Petrinetzmodelle existieren in objektorientierten Varianten. Welchen Bezug Objektorientierung und Petrinetze zueinander haben, wird daher ebenfalls diskutiert. Danach wird auf die Sichtenintegration eingegangen. D.h., es wird gezeigt, wie sich die maschinen- und die materialorientierte Sichtweise der diskreten Simulation auf Petrinetze übertragen lässt. Zum Schluss dieses Kapitels sollen die diskutierten Eigenschaften an einem Beispielszenario gezeigt werden.

Über die Objektorientierung hinaus kann man noch einen Schritt weiter gehen und Agenten, in diesem Fall Petrinetzagenten, zur Simulation einsetzen. In Kapitel 7 werden daher die konzeptionellen Grundlagen der Agentenorientierung dargestellt und auf die Simulation bezogen. Zur Veranschaulichung wird ein Beispielszenario mit Hilfe der Mulan-Architektur implementiert. Die Mulan-Architektur ist eine Entwicklungsumgebung für Petrinetz Multiagentensysteme, die ebenfalls an der Universität Hamburg entwickelt wurde und auf Renew aufsetzt.

In Kapitel 8 wird auf die Analyse von Simulationsexperimenten eingegangen. Es soll bei der qualitativen Analyse auf den Bezug zu Verklemmungsfreiheit, Lebendigkeit und Fairness eingegangen werden. Für die quantitative Analyse werden zunächst elementare Wartesysteme, Wartenetze und hierarchische Bediensysteme vorgestellt. Danach wird auf die mit den Wartesystemen in Zusammenhang stehenden Größen Zugang, Verweilzeit, Durchsatz und Füllung eingegangen. Es werden die Analyse von elementaren Wartesystemen mit Mar-

kovketten und die Analyse von Wartenetzen gezeigt. Darüber hinaus werden die für die Simulation wichtigen Begriffe Einschwingphase, Abbruchkriterium, Konfidenzintervall und Varianz diskutiert. Abschließend soll überlegt werden, wie sich Simulationsergebnisse für die Analyse geeignet darstellen lassen.

Im letzten Kapitel, Kapitel 9, wird besprochen, welche Möglichkeiten es gibt, Ergebnisse aus dem Petrinetzsimulator Renew zu extrahieren. Danach werden die konkret eingesetzten Auswertungsklassen vorgestellt. Darüber hinaus soll zusätzlich diskutiert werden, wie man die verwendeten Klassen zur Auswertung evtl. in die Benutzungsoberfläche von Renew integrieren kann. Zum Abschluss dieses Kapitels wird das objektorientierte Beispielszenario zur Demonstration simuliert und ausgewertet.

Zusammenfassend gesagt, soll in dieser Arbeit gezeigt werden, dass Petrinetze, speziell Referenznetze, ein geeignetes Mittel sind, um Szenarien als Simulationsmodell darzustellen. Des Weiteren soll mit dem Petrinetzeditor und -simulator Renew gezeigt werden, dass sich Petrinetzsimulationsmodelle auch zum konkreten Einsatz verwenden lassen. Dazu wird für Renew eine Auswertungskomponente entwickelt, die Simulationsergebnisse liefern und damit eine Analyse möglich machen soll.

2 Grundlagen der Simulation

Die Simulation ist eine der ältesten Anwendungen der Informatik. Mit der Entwicklung schneller Rechner in den frühen fünfziger Jahren erlangte sie große Bedeutung dadurch, dass man von nun an die Möglichkeit hatte, numerische Modelle bzw. Probleme von Computern berechnen zu lassen, die analytisch nicht zu lösen waren, wie beispielsweise komplexe Bedien-/Wartesysteme.

Ein entscheidender Grund für den Einsatz der Simulation ist der Wunsch, über Wissen zu verfügen, das die Zukunft betrifft, bzw. der Wunsch, zukünftige Entwicklungen voraussagen und beeinflussen zu können (aus [Pag92]).

Die Simulation ist eine sehr vielseitige Disziplin. Sie umfasst u.a. die drei für die Informatik wichtigen Bereiche Modellentwurf, Implementierung und Systemanalyse (vgl. [Pag92, S.1]). Darüber hinaus ist die Simulation nicht ausschließlich auf den Bereich der Informatik oder der Mathematik beschränkt, sondern auch in anderen Fachgebieten einsetzbar. Zu nennen sind hier insbesondere die Ingenieurswissenschaften, die Physik, die Geographie, die Biologie und die Wirtschaftswissenschaften. In den Ingenieurswissenschaften wird die Simulation häufig zur Planung von z. B. Maschinen oder Gebäuden eingesetzt, da in diesem Fall kein reales Modell existiert. In den Geowissenschaften dauern Abläufe häufig mehrere Jahrzehnte oder auch Jahrhunderte, weshalb auch hier zur Beschleunigung Simulationsmodelle eingesetzt werden, z.B. Klimamodelle (siehe [vSGH99]).

Die Techniken der Computersimulation bieten die Möglichkeit, das Verhalten eines komplexen Systems zu verstehen und evtl. vorauszusagen. Das geschieht, indem ein Modell entworfen wird, wobei vorher entweder von bestimmten festgelegten Annahmen, beobachtetem Verhalten oder gemessenen Daten bezüglich des Realsystems ausgegangen wird, die das Modell beschreiben. Man kann dann mit der Simulation mögliche Konsequenzen einer Systemveränderung voraussagen und dadurch das Systemverständnis vergrößern (siehe dazu die im Folgenden genannte Literatur). Dieser Text orientiert sich stark an [Pag92], [PLC00] und an meiner Studienarbeit [Str01]. Als weitere Literatur sind zusätzlich noch [Fis73], [Fis78], [Fis01], [NBBC66] und [Pri84] zu nennen, in denen jeweils eine Einführung in die Computersimulation mit unterschiedlichen Schwerpunkten gegeben wird.

Es werden in [NBBC66, S.5] vier wesentliche Schritte formuliert, um Aussagen über ein System machen zu können. Der erste Schritt ist die Beobachtung des realen Systems. Darauf folgt als zweiter Schritt die Formulierung von Hypothesen mit Hilfe von Modellen, die das System beschreiben. Der dritte Schritt ist die Vorhersage des Verhaltens auf Grund der Hypothesen. Der vierte verlangt die Durchführung des Experiments, um die Vorhersagen zu testen. Den vierten Schritt bezeichnet man auch als Validierung des Modells. Ist das Modell validiert, kann es für Simulationsexperimente eingesetzt werden. Diese vier Schritte

beschreiben im wesentlichen das Gebiet der Simulation.

Obwohl Systeme sehr unterschiedlich und sehr komplex sein können, können annähernd alle Systeme mit Hilfe der Simulation bzw. den Techniken der Modellbildung, des Software Engeneerings und der Statistik untersucht werden. Mit den Techniken der Simulation kann man demnach Systeme studieren.

Man kann die Simulation also als eine numerische Technik ansehen, die geeignet ist, um Experimente auf dem Computer durchzuführen, die bestimmte Typen von mathematischen und logischen Modellen beinhalten, um ein bestimmtes Verhalten eines Systems (in einem festgelegten Zeitintervall) bestimmen zu können (vgl. [Pag92]).

Eine sehr alte, aber noch immer zutreffende Definition der Simulation ist die von C. Churchman aus dem Jahre 1963, zitiert nach [NBBC66, S.2]:

Definition 1 *x simulates y, is true, if and only if*

- *x and y are formal systems,*
- *y is taken to be the real system,*
- *x is taken to be an approximation to the real system, and*
- *the rules of validity in x are non-error-free.*

Zu dieser Definition ist noch hinzuzufügen, dass es sich bei y nicht um ein formales System handeln muss. Jedes Realsystem kann simuliert werden. Lediglich x muss für die Computersimulation ein formales System sein.

„Ganz allgemein bedeutet Simulation das Experimentieren an Modellen, wenn bei der Systemanalyse ein Modell an die Stelle des Originalsystems tritt und Experimente am Modell durchgeführt werden“ ([Pag92, S.7]).

In diesem Kapitel wird zunächst ein Überblick über die Begriffe *System* und *Modell* gegeben. Danach werden die verschiedenen Sichtweisen der Modellierung bzw. der Simulation und ihre Einsatzgebiete besprochen. Zum Schluss wird noch ein Einblick in die Modellauswertung gegeben.

2.1 Was sind Systeme?

Ein wesentlicher Begriff im Bereich der Simulation ist der des *Systems*.

„Ein System besteht aus einer Menge von Entitäten (oder Objekten), die in einer bestimmten Beziehung zueinander stehen und zu einem bestimmten Zweck miteinander interagieren.“(J. Forrester zitiert nach [Pag92, S.2])

Ein System ist außerdem ein genau abgegrenzter Bereich innerhalb einer Umwelt. Der Bereich eines Systems umfasst demzufolge Grenzen und eine Umgebung, die sich außerhalb des Systems befindet. Ein System besteht aus einer Menge von Entitäten, die wiederum aus einer Menge von Attributen bestehen, die durch Werte charakterisiert sind (vgl. [Pag92, S.2]).

Ein wichtiger Begriff in Bezug auf Systeme ist der Begriff des *Zustands*. Der Zustand einer Entität wird durch die Werte ihrer Attribute gegeben, der Zustand des Systems durch die Zustände der Entitäten. Dabei ist zu beachten, dass ein

Attribut verschiedene Werte annehmen kann und eine Entität im Allgemeinen aus mehreren Attributen besteht. Daraus ergibt sich, dass der Zustand eines Systems aus einer Menge von Mengen entsteht.

Die Entitäten eines Systems können, wie oben schon erwähnt, in Beziehung zueinander stehen und untereinander interagieren. Man nennt diese Beziehungen Interaktionsbeziehungen. Eine Zustandsänderung einer Entität hätte damit u.U. auch eine Zustandsänderung einer anderen Entität des Systems zur Folge (vgl. [Pag92, S.2]). Die Entitäten eines Systems können, je nach Abstraktionsgrad, wiederum Systeme sein, bzw. Systeme können Entitäten übergeordneter Systeme sein. Man spricht in diesem Fall von einer *Systemhierarchie* (vgl. [Pag92, S.2]). Ein System, dass nur in einer ganz bestimmten bzw. genau definierten Umgebung existieren kann, nennt man im Bereich der Simulation *offenes System*. Ein offenes System interagiert mit seiner Umwelt. D.h., es unterhält mindestens eine Interaktionsbeziehung zu einem umgebenden System. Ein System, das auch in jeder anderen Umgebung existieren kann, nennt man *geschlossenes System*, da es nur mit Entitäten innerhalb seiner eigenen Grenzen interagiert; es ist also unabhängig von äußeren Einflüssen (siehe [Pag92, S.3]).[1]

In der Literatur, beispielsweise in [Pag92, S.1], werden verschiedene Typen von Systemen genannt, darunter u.a. natürliche Systeme wie beispielsweise bestimmte Ökosysteme, technische Systeme wie z.B. eine Industrieanlage, soziale Systeme wie z.B. das Rentensystem oder auch Systeme, die Ideengebilde sind.

Für die Analyse eines Systems ist die Veränderung seiner Zustände innerhalb eines bestimmten Zeitraums interessant. Man kann sich für die Analyse eines Systems einen speziell ausgewählten Bereich der Realität vorstellen, der unter einer speziellen Fragestellung betrachtet wird. Oft sind zu betrachtende Systeme sehr komplex und ihre Struktur und ihr Verhalten nicht unmittelbar durchschaubar. Damit ein komplexes System dennoch analysiert und ggf. beurteilt werden kann, müssen Vereinfachungen und Idealisierungen vorgenommen werden. Diese Vereinfachungen des Systems werden in Form von Modellen realisiert, wobei für jedes System, je nach Fragestellung, eine Menge von Modellen existieren kann (vgl. [Pag92, S.4]).

2.2 Modellbildung

Im Bereich der Simulation ist neben dem Begriff des *Systems* auch der Begriff des *Modells* von besonderer Wichtigkeit.

Modelle sind Systeme, die die Entitäten und Relationen des Ursprungssystems (Realsystems, Originalsystems) in veränderter (vereinfachter, idealisierter) Weise darstellen. Modelle dienen dazu, ein besseres Verständnis für die Vorgänge, die Wirkungsweisen, die Zusammenhänge und das Verhalten eines realen Systems zu entwickeln (vgl. [Pag92, S.4]). Dazu ist eine genaue Systemanalyse nötig, um die Systemstruktur und die Systemkomponenten zu erkennen. Ein Modell stellt ein Originalsystem vereinfacht dar. Es findet eine Abstraktion und

[1] Im Bereich von Wartesystemen bzw. -netzen ist die Beschreibung von offenen und geschlossenen Systemen anders. Hier haben offene Systeme eine veränderliche Füllung und geschlossene Systeme eine konstante (vgl. [JV87, S.359]

Idealisierung statt, durch die die Untersuchung komplexer Systeme erst handhabbar wird. Zu einem System können mehrere Modelle erstellt werden, die sich je nach Fragestellung, Art des Modells und subjektiver Sichtweise des Modellbildners unterscheiden (vgl. [Pag92, S.4]).

„Modelle sind materielle oder immaterielle Systeme, die andere Systeme so darstellen, dass eine experimentelle Manipulation der abgebildeten Strukturen und Zustände möglich ist“ ([Nie77, S.75], zitiert nach [PLC00, S.5]).

Bei der Entwicklung eines Modells ist es entscheidend, um welche Art Modell es sich handelt. In der Literatur sind deshalb verschiedene Gesichtspunkte angegeben, nach denen Modelle klassifiziert werden (siehe [Pag92] S.4-7). Man unterscheidet Modelle nach der Art ihrer Untersuchungsmethode, nach der Art ihres Abbildungsmediums, nach der Art ihrer Zustandsübergänge und nach der Art ihrer Verwendung.

Bei der Art der Untersuchungsmethode unterscheidet man zwischen *analytischen Modellen* und *Simulationsmodellen*. Modelle mit analytischem Lösungsansatz erlauben es, in ein Gleichungssystem, das die vorhandenen Systembeziehungen widerspiegelt, bestimmte angenommene Werte einzusetzen und in einem geschlossenen Lösungsdurchlauf den jeweiligen Systemzustand direkt zu berechnen. Analytische Untersuchungen sind wegen mathematischer Restriktionen jedoch auf Systeme begrenzt, die nicht zu komplex sind und für die ein analytischer Lösungsweg existiert. Bei komplexen Bedien-/Wartesystemen ist das beispielsweise nicht der Fall (vgl. Kapitel 8).

In Simulationsmodellen wird der Modellzustand dagegen Schritt für Schritt ermittelt und fortgeschrieben. Die Zwischenergebnisse der Berechnung können als Zwischenzustände des Originals interpretiert werden. Aus diesem Grund eignen sich Simulationsmodelle besonders gut zur Veranschaulichung des Systemverhaltens innerhalb eines Zeitintervalls.

Es existieren verschiedene Arten von Modellen, die nicht notwendigerweise in einem Formalismus dargestellt werden müssen. Deshalb ist es sinnvoll, Modelle nach Art ihres Abbildungsmediums zu unterscheiden. Es sind dies im wesentlichen:

- materielle Modelle, wie zum Beispiel Schiffsmodelle,
- verbale Modelle, das sind Beschreibungen in natürlicher Sprache,
- grafisch-deskriptive Modelle, wie Flussdiagramme und UML-Diagramme,
- mathematische Modelle, wie Gleichungssysteme und
- grafisch-mathematische Modelle, wie zum Beispiel Petrinetze, um die es in dieser Arbeit im Wesentlichen geht und die in den folgenden Kapiteln besprochen werden.

Bei der Klassifikation nach der Art der Zustandsübergänge unterteilt man in *statische* und in *dynamische Modelle*. Bei einem statischen Modell treten keine Zustandsänderungen auf. Sie sind im Bereich der Simulation und in dieser Arbeit deshalb nicht interessant. Bei dynamischen Modellen treten dagegen zeitabhängige Zustandsänderungen auf.

Die dynamischen Modelle werden wiederum unterteilt in *kontinuierliche* und *diskrete* Modelle. Die Zustandsvariablen eines kontinuierlichen Modells lassen sich durch stetige Funktionen beschreiben. Die diskreten Modelle ändern dagegen die Werte ihrer Zustandsvariablen sprunghaft zu ganz bestimmten diskret verteilten Zeitpunkten. In dieser Arbeit werden in den folgenden Kapiteln nur zeitdiskrete Simulationsmodelle betrachtet.

Sowohl bei den kontinuierlichen als auch bei den diskreten Modellen unterscheidet man zwischen *deterministischen* und *stochastischen* Modellen. Deterministisch heißt ein Modell dann, wenn seine Reaktion auf eine bestimmte Eingabe, ausgehend von einem bestimmten Zustand, eindeutig festgelegt ist. Wenn sich Reaktionen des Modells nur durch Wahrscheinlichkeitsverteilungen beschreiben lassen, nennt man es stochastisch. Auch hier muss angemerkt werden, dass diese Beschreibungen nur für den Bereich der Simulation gelten. Im Bereich der theoretischen Informatik und damit im Bereich der Petrinetze unterscheidet man zwischen *deterministischen* und *nicht deterministischen* Modellen (Automaten) (vgl. [HU79]). Man kann bei der Simulation die stochastischen Modelle einsetzen, da man durch die Beobachtung des Realsystems oder durch Messungen eine gewisse Kenntnis darüber besitzt, mit welcher Wahrscheinlichkeit ein Zustand eintritt.

Eine weitere Klassifizierung von Modellen ist die des Verwendungszwecks. Der Verwendungszweck eines Modells ist durch die Fragestellung und Zielsetzung einer Modellstudie festgelegt. Abhängig vom Verwendungszweck werden unterschiedliche Anforderungen an Modelle gestellt. Man unterscheidet Erklärungsmodelle, Prognosemodelle, Gestaltungsmodelle und Optimierungsmodelle.

2.3 Simulation

Was man unter Simulation versteht und wann man sie einsetzt, wurde bereits zu Beginn dieses Kapitels gezeigt. Zur Wiederholung: Simulation verwendet man beispielsweise, um Folgen eines Eingriffs in ein System abzuschätzen oder die Realisierbarkeit eines Projekts testen zu können. Darüber hinaus gibt es noch viele andere Gründe, die durchaus eine sehr subjektive Motivation haben können. Allgemein lässt sich aber sagen, dass die Simulation immer dann eingesetzt wird, wenn Manipulationen an einem System vorgenommen werden sollen. Wie oben schon diskutiert, ist es sinnvoll, ein Modell zu entwickeln, um ein System zu untersuchen. Dabei ist es wichtig, zu prüfen, ob das Modell die wesentlichen Eigenschaften des Systems abbildet. Dafür benötigt man Detailwissen über einzelne Objekte des Systems, d.h., man muss deren Eigenschaften kennen. Zusätzlich werden Kenntnisse über die Relationen, in denen die einzelnen Objekte zueinander stehen, benötigt. Man nennt dieses Wissen auch Strukturwissen. Ist die hinreichend korrekte Abbildung zwischen Original und Modell gesichert, so lassen sich Abläufe des realen, dynamischen Systems im Modell nachvollziehen und Kenntnisse über das Modellverhalten sammeln, die in gewissen Grenzen Rückschlüsse auf das Verhalten des Originals erlauben (vgl. [Pag92, S.7]).

Um komplexe Systeme verstehen zu können und handhabbar zu machen, findet, wie oben schon erklärt, eine Abstraktion des Realsystems auf das Modell

statt. Dadurch zeigt sich auch, welche Eigenschaften oder Aspekte eines Systems elementar sind und im Modell berücksichtigt werden müssen und welche unwichtig sind und weggelassen werden sollten. Ob ein Aspekt als wichtig oder unwichtig angesehen wird, hängt von der Fragestellung ab, auf die das System hin untersucht wird und von der subjektiven Sichtweise des Modellentwicklers. Im allgemeinen werden nur solche Modelle als Simulationsmodelle bezeichnet, die keine analytische Behandlung erlauben, d.h., Simulationsmodellen können keine mathematischen Methoden zugerechnet werden. Immer dann, wenn ein analytisches Verfahren keine adäquate Abbildung eines Systems mehr gestattet, bietet sich die Simulationsmethode an. Es werden hierbei mathematische Beziehungen beliebiger Art, algorithmische Beschreibungen in Form von logischen Verknüpfungen, Fallunterscheidungen oder Wiederholungen entwickelt, bis die angestrebte Abbildungsgenauigkeit erreicht ist (vgl. [Pag92, S.8]).

Simulationsmodelle haben nach [Pag92, S.8-9] gegenüber analytischen Modellen folgende Vorteile:

- Mit Simulationsmodellen lassen sich im Gegensatz zu analytischen Modellen auch komplexe Systemstrukturen untersuchen.
- Ohne vereinfachende Annahmen über Verteilungen, Zufälligkeit oder Unabhängigkeit kann das Simulationsmodell mit einem wesentlichen höheren Grad an Realitätsnähe versehen werden.
- Ein Simulationsmodell ermöglicht flexible Sensitivitätsuntersuchungen bezüglich der angenommenen statistischen Verteilungen.
- Simulation ist für den Anwender mathematisch weniger schwierig als die Verwendung analytischer Ansätze.
- Simulation ermöglicht eine anschauliche Darstellung des Systemverhaltens, weil die zeitliche Entwicklung des Systemzustandes Schritt für Schritt nachvollzogen wird.

Diesen Vorteilen stehen, ebenfalls nach [Pag92, S.9], auch einige Nachteile im Vergleich mit analytischen Modellen gegenüber.

- Im Gegensatz zu analytischen Verfahren ist bei der Simulation das Auffinden der optimalen Lösung nicht sichergestellt.
- Ein Simulationsmodell erfordert in der Regel höheren Entwicklungsaufwand als ein analytisches Modell.
- Computersimulation erfordert mehr Rechenzeit und Speicherplatz als die Anwendung analytischer Methoden.

Die Simulation ist also ein Instrument zur Analyse komplexer Systeme. Um ein Modell und damit die Simulation möglichst realistisch zu halten, ist es nötig, ausführliche statistische Tests zur Validierung oder *Plausibilitätstests* durchzuführen. Diese sind zur späteren Bewertung der Ergebnisse bezüglich ihrer Aussagekraft besonders wichtig. Es ist deshalb notwendig, ausreichend viele Daten

aus dem Realsystem zu sammeln und mit den Daten des Modells zu vergleichen. Bei zu großen Abweichungen wären die Ergebnisse der Simulation nicht aussagekräftig. Es muss davon ausgegangen werden, dass das Modell fehlerhaft ist. Bei geringeren Abweichungen oder bei etwa gleichem Verhalten zwischen Realsystem und Modell können die Parameter des Modells so nachjustiert werden, dass die gewünschte Änderung im Realsystem simuliert werden kann und die Ergebnisse aussagekräftig werden (vgl. [Pag92, S.16]).

2.4 Zeitdiskrete Simulation

Es gibt verschiedene Methoden der Simulation (z.B. kontinuierliche Simulation, diskrete Simulation oder quasi kontinuierliche Simulation). Für jedes System muss deshalb zunächst überprüft werden, welche Methode die geeignete ist. Thema dieser Arbeit ist die zeitdiskrete Simulation. Zeitdiskrete Simulationsmodelle treten in vielfältiger Form auf und werden für die unterschiedlichsten Problemstellungen eingesetzt. Sie sind häufig zufallsabhängig; es müssen daher statistische Verfahren zur Zufallszahlenerzeugung herangezogen werden. Bei zeitdiskreten Simulationsmodellen werden die Änderungen des Systemzustandes nur zu bestimmten, diskreten Zeitpunkten betrachtet. Es wird also durch eine endliche Folge von Zuständen die zeitliche Entwicklung des Systems beschrieben. Die Zustände, die zeitlich zwischen den diskret verteilten Zeitpunkten der Zustandsänderung liegen, werden nicht berücksichtigt und als konstant angenommen. Der Zeitpunkt der Neuberechnung der Zustandsvariablen wird in der zeitdiskreten Simulation auf das Eintreffen eines bestimmten Ereignisses gelegt. Die zeitdiskrete Simulation bietet so die Möglichkeit, bei relativ geringem Rechenaufwand eine recht genaue Abbildung eines Systems zu schaffen (vgl. [Pag92, S.25-26] und [PLC00, S.58]).

Die zeitdiskrete Simulation eignet sich besonders gut für Systeme, in denen keine stetigen Änderungen der Zustände auftreten, sondern eine eindeutige Änderung zu einem bestimmten Zeitpunkt. Ein zeitdiskretes Simulationsmodell besteht aus Systemobjekten bzw. Systemkomponenten. Diese Komponenten sind gleichfalls die Entitäten des Systems. Die Entitäten des Simulationsmodells stehen in Wechselwirkung zueinander, genau wie die Entitäten eines realexistierenden Systems auch. Bei der Entwicklung des Modells kann eine Entität auch als Objekt der objektorientierten Programmierung aufgefasst werden, dessen Verhalten im Verlauf einer bestimmten Simulationszeit definiert ist. Eine Entität wird zusätzlich durch ihren Zustand und durch Transformationsregeln charakterisiert. Diese Transformationsregeln verändern den Entitätszustand während der Simulationszeit. „Eine Entität ist ein Objekt, welches in der Lage ist, sich (aktiv) in der Simulationszeit fortzubewegen“ ([SH95], zitiert nach [PLC00, S.7]). Damit eine Zustandstransformation zu einem bestimmten Zeitpunkt stattfindet, ist es nötig, einige ausgewählte Methoden mit einem Zeitparameter zu versehen. Die in einem Simulationsmodell vergehende Zeit ist eine fiktive Modellzeit, die unabhängig von der realen Zeit und unabhängig von der Rechenzeit einer Simulation ist.

In der klassischen diskreten Simulation haben sich verschiedene Modellie-

rungsansätze entwickelt, die bei der Modellierung der Entitäten eine unterschiedliche Sichtweise einnehmen. Das sind der ereignisorientierte, der transaktionsorientierte, der prozessorientierte und der aktivitätsorientierte Ansatz. Die ersten beiden Ansätze bezeichnet man auch als *materialorientiert*, die beiden letztgenannten als *maschinenorientiert*. Ein Ereignis ist eine Zustandsänderung einer Entität, ein Prozess ist eine Folge von Ereignissen, die das Verhalten einer Entität widerspiegeln. Eine Transaktion ist der Übergang von einem Zustand in einen neuen, und eine Aktivität ist eine Ansammlung von Operationen, die eine Zustandsänderung herbeiführen. Ein Prozess setzt sich aus mehreren Aktivitäten zusammen. In der Praxis haben sich hauptsächlich der ereignisorientierte und der prozessorientierte Ansatz durchgesetzt. Der transaktionsorientierte Ansatz gilt als veraltet und hat deswegen eine geringere Bedeutung, und der aktivitätsorientierte Ansatz hat annähernd keine Bedeutung, da er dem prozessorientierten ähnlich ist. Man kann die Aktivitäten als Ereignisse innerhalb eines Prozesses betrachten.

Im Folgenden sollen deshalb nur die drei erstgenannten Ansätze und ihre Sichtweisen diskutiert werden.

2.4.1 Ereignisorientierung

Die ereignisorientierte Simulation ist der klassische Modellierungsstil der zeitdiskreten Simulation. Ereignisorientierte Modelle betrachten jeweils die Gesamtheit der Zustandsänderungen der Entitäten zu einem bestimmten Zeitpunkt. Das dynamische Verhalten des Systems wird durch eine Folge von Ereignissen dargestellt. Zu einem Ereigniszeitpunkt werden alle Zustandsänderungen einer oder mehrerer Entitäten zu einem Ereignis zusammengefasst. Die zwischen den Ereignissen liegenden Aktivitäten, die diese Zustandsänderungen auslösen, werden nicht direkt abgebildet und übersprungen. Auch wenn die Neuberechnung der Zustandsvariablen der Entitäten auf einem Computer Rechenzeit benötigt, erfolgt diese aus der Sicht des Modells konzeptionell zeitverzugslos, d.h. ohne Simulationszeitverbrauch.

Wie bei allen Simulationsansätzen müssen auch beim ereignisorientierten Ansatz als Erstes die relevanten Systemobjekte und ihre Attribute identifiziert werden. Der Modellierer nimmt bei der Beschreibung der ereignisorientierten Sichtweise eine „Vogelperspektive“ ein, da er alle Systemzustände, Ereignisse und Zustandsübergänge sämtlicher Entitäten des Systems zusammen überblicken und beschreiben muss. Dabei werden alle Systemzustände, die, durch Eintritt eines Ereignisses an Entitäten des Systems zum gleichen Zeitpunkt ausgelöst werden, zu Ereignissen eines bestimmten Typs zusammengefasst (vgl. [Pag92, S.29-30] und [PLC00, S.14-15]).

2.4.2 Prozessorientierung

Der prozessorientierte Modellierungsstil ist dadurch gekennzeichnet, dass auf eine Simulationsentität bezogene Aktivitäten mit den Entitätsattributen in ihrer Gesamtheit zu einem Prozess zusammengefasst werden. Der vollständige Lebenszyklus einer Entität kann auf einen Prozess abgebildet werden. Während

ein Prozess aktiv ist, kann Simulationszeit vergehen, d.h., eine Entität muss in der Lage sein, während ihr Lebenszyklus durchlaufen wird, die Kontrolle vorübergehend an die Zeitführungsroutine (Scheduler) der Simulation abzugeben, damit sie die Simulationsuhr weiterschalten kann. Aus der Sicht des Prozesses vergeht die Zeit, wenn er selber aktiv ist. Die zeitkonsumierenden Aktivitäten eines Prozesses und seine passiven Phasen werden durch inaktive Prozesszustände dargestellt. Es werden deshalb zwei Arten von inaktiven Phasen eines Prozesses unterschieden. Im ersten Fall bildet der Prozess eine zeitkonsumierende Tätigkeit ab; er ist also konzeptionell aktiv. Programmtechnisch ist er passiv, da er für diese Zeit die Kontrolle an einen anderen Prozess abgibt. Im zweiten Fall ist der Prozess in einem Wartezustand auf unbestimmte Zeit. Er hat nicht mehr die Programmkontrolle. Der Prozess ist jetzt sowohl konzeptionell als auch programmtechnisch inaktiv.

Während einer aktiven Prozessphase führt ein Prozess Zustandsänderungen durch, die konzeptionell zeitverzugslos ablaufen, d.h., während der aktiven Phase steht die Simulationsuhr still. Wenn ein Prozess aktiv ist, kann er beispielsweise Objektattribute aktivieren, neue Prozesse generieren, die Aktivierung anderer Prozesse zu bestimmten Zeitpunkten planen oder geplante Aktivierungen anderer Prozesse verschieben oder löschen, sich selbst deaktivieren, wobei die Kontrolle an einen anderen Prozess übergeht, und Prozesse beenden. Da in aktiven Prozessphasen keine Simulationszeit verbraucht wird, entsprechen diese den Ereignisroutinen. Der Unterschied ist der, dass ein Prozess nach der Durchführung einer Zustandsänderung zusätzlich die Möglichkeit hat, in einen inaktiven Zustand zu gelangen. Der Prozess kann dann zu einem späteren Zeitpunkt fortgesetzt werden, und der nächste Abschnitt eines Prozesses kann abgearbeitet werden. Bei der Ereignisroutine ist das nicht möglich.

Aus der Sicht des Schedulers kann ein Prozess nur zwei Zustände annehmen. Er ist entweder aktiv und rechnet solange, bis er von sich aus die Kontrolle abgibt, oder er wartet auf ein Ereignis, und ein anderer Prozess ist aktiv. Da in gängigen Programmiersprachen immer nur ein Prozess im aktiven Zustand sein kann[2], gibt es so etwas wie pseudo-parallele Abläufe beliebig vieler Prozesse zu einem bestimmten Simulationszeitpunkt. Pseudo-parallel sind sie deshalb, da sie nur sequenziell bearbeitet werden können, die Simulationsuhr aber nicht vor geschaltet wird. Daraus folgt, dass auch beliebig viele Entitäten parallel agieren können (vgl. [Pag92, S.30-32] und [PLC00, S.12-14]).

2.4.3 Transaktionsorientierung

Der transaktionsorientierte Ansatz hat in Deutschland keine so starke Bedeutung wie der ereignis- und der prozessorientierte Ansatz, dennoch soll er kurz beschrieben werden.

Zur Beschreibung des Systemverhaltens bei dieser Sichtweise verwendet man Blockdiagramme. Blöcke sind Elemente mit fest vorgegebenen Funktionen. Sie sind stets im System vorhanden und bilden somit die statische Systemstruktur. Die dynamische Systemstruktur wird durch Transaktionen dargestellt. Sie sind

[2] Ausnahme sind Multi-Threating Programme, wie z.B. Java.

temporäre Elemente, die durch die Blöcke wandern und in den Blöcken verändert werden. Der Systemzustand ändert sich also, indem die Parameter der temporären Elemente in einem Block geändert werden.

Der transaktionsorientierte Ansatz lässt sich auf den ereignisorientierten Ansatz zurückführen. Man kann das Zusammentreffen von Blöcken und Transaktionen als Ereignisse betrachten, die nach der Reihenfolge ihres Auftretens in einer Ereignisliste eingetragen werden (vgl. [Pag92, S.32]).

2.5 Modellauswertung

Nachdem im vorangegangenen Abschnitt geklärt wurde, was Simulationsmodelle sind, stellt sich nun die Frage, wie sie eingesetzt werden. Man erstellt ein Modell im Hinblick auf eine bestimmte Fragestellung, um ein Problem zu lösen bzw. eine Frage beantworten zu können. Man versucht deshalb, ein Modell zu erstellen, das möglichst realistisch ist und das so modifiziert werden kann, dass es die gestellte Frage an das Realsystem beantworten kann. Es ist dabei besonders wichtig zu bedenken, dass ein Modell nicht das Realsystem ist und sich eine Veränderung im Modell nicht unbedingt im Realsystem auswirkt. Da ein Modell eine Abstraktion und Idealisierung eines Realsystems ist, werden nicht alle Komponenten des Realsystems auf das Modell abgebildet. Das kann bei falscher Einschätzung des Realsystems zu Konstruktionsfehlern im Modell führen und damit auch zu Auswertungsfehlern.

Deshalb ist es wichtig, nicht nur die Endergebnisse einer Simulation zu betrachten und hinzunehmen, sondern die Ergebnisse müssen Schritt für Schritt analysiert werden. Es sollten Aussagen über die einzelnen Komponenten im Modell gemacht werden können und jeder Systemzustand einer Komponente sollte dokumentiert sein (vgl. [Pag92, S.16-18]).

Eine sehr häufig eingesetzte Komponente der Simulation ist beispielsweise die Warteschlange. In einer Warteschlange gibt es in der zeitdiskreten Simulation zu jedem Zeitpunkt eine bestimmte Füllung. Für die Auswertung ist es deshalb interessant, diese Füllung im Verlauf der Zeit zu beobachten: Ist die Kapazität einer Warteschlange immer ausgeschöpft, ist sie immer leer oder ist sie unterschiedlich stark gefüllt? Man kann daraufhin Parameter im System ändern, um eine gewünschte Füllung zu erhalten. Des Weiteren lässt sich unter Umständen beurteilen, ob eine gewählte Strategie wie zum Beispiel „lifo“ oder „fifo“[3] wirklich die richtige ist, oder ob eine andere Strategie nicht geeigneter wäre. Eine Auswertungskomponente einer Simulationssoftware sollte deshalb die Möglichkeit bieten, die einzelnen Systemzustände einer Systemkomponente aufzuzeigen, damit der Verlauf der Simulation nachvollzogen und mögliche Fehler gefunden werden können.

Man erreicht durch eine detaillierte Ergebnisausgabe eine größere Sicherheit bzgl. der Richtigkeit einer Simulation. Das heißt dennoch nicht, dass ein absolut richtiges Ergebnis gefunden wurde. Man sollte sich bewusst sein, dass es in der Simulation nur gut angenäherte Ergebnisse gibt und nicht optimale oder absolut richtige.

[3] last in first out, first in first out

Weitere wichtige Komponenten der zeitdiskreten Simulation sind Zufallszahlenströme. Sie werden häufig für Zwischenankunftszeiten temporärer Objekte im System eingesetzt oder um Bearbeitungszeiten bzw. allgemein den Zeitverbrauch darzustellen. Es ist in der Auswertung zu prüfen, ob passende Zufallszahlenströme eingesetzt wurden (Gleich-, Normal- oder Exponentialverteilung) und ob die simulierten Zeiten der Realität entsprechen (vgl. [Pag92, S.101-112]).

Neben den einzelnen Komponenten, die betrachtet werden müssen, ist es zusätzlich wichtig, den Simulationslauf zu beurteilen. Es muss festgestellt werden, ab welchem Zeitpunkt eine Simulation stabil läuft. Jeder Simulationslauf hat eine *Einschwingphase*, in der es zu Extremwerten kommen kann, die, werden sie in der Auswertung mit berücksichtigt, ein Ergebnis verfälschen können. Es muss deshalb geprüft werden, wann die Einschwingphase eines Simulationslaufes beendet ist. Nur die darauf folgenden Werte sollten für die Ergebnisbestimmung eingesetzt werden. Es werden in der Literatur (s.o.) unterschiedliche Verfahren beschrieben, mit der eine Einschwingphase berechnet werden kann. Es gibt Verfahren, die Teile eines Simulationslaufes häufig wiederholen, um die Einschwingphase nur einmal abwarten zu müssen. Diese Verfahren sind im allgemeinen schneller als Verfahren, bei denen für jeden Simulationslauf wieder ganz neu, inklusive Einschwingphase, angefangen werden muss. Sie haben aber den Nachteil, dass die einzelnen Simulationsläufe nicht unabhängig voneinander sind. Die Unabhängigkeit einzelner Simulationsläufe ist für die Auswertung oft wichtig, da sich sonst falsche Annahmen und Ergebnisse verschleppen und ein Endergebnis verfälschen können. In dieser Arbeit wird in den Kapiteln 8 und 9 näher auf diese Eigenschaft eingegangen (vgl. [Pag92, S.121]).

Es müssen meist viele Simulationsläufe durchgeführt werden, um ein aussagekräftiges Ergebnis zu bekommen. Die Frage, die sich stellt, ist: Wie viele sind notwendig, um ein gutes Ergebnis zu bekommen? Es müssen mindestens so viele sein, dass sich ein Konfidenzintervall berechnen lässt (vgl. [Pag92, S.129]). Ein Konfidenzintervall ist ein Intervall, mit dem sich die Genauigkeit eines Ergebnisses angeben lässt. Es sagt aus, mit welcher Wahrscheinlichkeit ein Wert in einem bestimmten Intervall liegt. Ziel ist es, eine möglichst hohe Wahrscheinlichkeit in einem möglichst kleinen Intervall zu bekommen. Für die Berechnung eines Konfidenzintervalls wird auf Kapiteln 8 und die Literatur verwiesen, z.B. [Hüb96].

Abschließend lässt sich feststellen, dass nicht nur bei der Modellentwicklung und Implementation sehr sorgfältig gearbeitet werden muss, sondern dass auch der Modellauswertung eine große Sorgfalt entgegengebracht werden muss. Das beste Modell und die beste Implementation sind nutzlos, wenn bei den Simulationsexperimenten und bei der Auswertung nicht sorgfältig gearbeitet wird.

Fazit

In diesem Kapitel wurden die für die Simulation wichtigen Begriffe System und Modell eingeführt und erklärt. Danach wurde auf die Simulation und speziell auf die zeitdiskrete Simulation eingegangen. Es wurden darüber hinaus die verschiedenen Sichtweisen der zeitdiskreten Simulation diskutiert. Dazu gehören die ereignisorientierte, die transaktionsorientierte, die prozessorientierte und die

aktivitätsorientierte Sichtweise. Im letzten Abschnitt dieses Kapitels wurde noch auf die Modellauswertung eingegangen. Es wurde gezeigt, was bei der Durchführung von Simulationsexperimenten zu berücksichtigen ist, und was bei der Auswertung der Ergebnisse der Simulationsexperimente beachtet werden muss.

3 Simulationsmodelle mit Petrinetzen

Das Gebiet der Petrinetze geht zurück auf Carl Adam Petri, der eine Beschreibungstechnik für Computeranwendungen entwickeln wollte, die universell ist und mit elementaren physischen Phänomenen realisiert werden kann (siehe dazu [Pet62]).

Eine grundlegende Einführung in die Theorie und Praxis der Petrinetze findet man in vielen Werken der Standardliteratur, s. z.B. [Bau90], [Rei85], [JV87] und [GV02].

Petrinetze dienen der Beschreibung verteilter, diskreter und nebenläufiger Systeme. Ein Petrinetz ist ein bipartiter Graph, der aus Objekten, Ereignissen und Kanten besteht. Es wird in Petrinetzen zwischen Zuständen und Zustandsänderungen unterschieden, in denen im Allgemeinen jeweils mehrere Teilzustände geändert werden. Zustände, Stellen genannt, werden in Diagrammen als Kreise oder Ellipsen dargestellt. Sie sind die statischen Elemente eines Petrinetzes. Ereignisse, Transitionen genannt, sind die aktiven bzw. dynamischen Elemente eines Netzes. Sie werden als Quadrat oder Rechteck dargestellt. Die Kanten des Graphen sind Pfeile, die entweder von einer Transition zu einer Stelle oder von einer Stelle zu einer Transition gehen.

Durch diese Beschreibung der Petrinetze kann man leicht die Nähe zu zeitdiskreten Simulationsmodellen erkennen. In beiden Modelltypen gibt es Zustände und Zustandsänderungen. Wie bereits erwähnt, beschreiben im Petrinetz die Stellen Zustände und die Transitionen Zustandsänderungen bzw. -übergänge. Im imperativen Ansatz werden Zustände durch die Werte der im Programm vorhandenen Zustandsvariablen ausgedrückt. Zustandsänderungen werden durch Änderung der Variablenwerte zu bestimmten Zeitpunkten ausgedrückt. Eine Zustandsänderung wird in der diskreten Simulation als Ereignis angesehen. Diese Ereignisse werden in einer Ereignisliste verwaltet. Die fortlaufende Änderung des Systemzustandes wird im imperativen Ansatz also durch die Abarbeitung einer Ereignisliste beschrieben. Die zeitliche Änderung des Systemzustandes beim Petrinetzansatz wird durch Marken beschrieben, die sich durch das Petrinetz bewegen.

In den folgenden Abschnitten soll nun der Petrinetzformalismus beschrieben werden. Es wird das Schaltverhalten von Petrinetzen erklärt und es werden Netzbeispiele gegeben. Danach werden die Vor- und Nachteile der Petrinetze für die Simulation gegenüber imperativen Ansätzen diskutiert.

3.1 Grundlagen der Petrinetze

Die einfachste Form eines Petrinetzes wird durch Stellen, Transitionen und die Flussrelationen beschrieben. Dabei sind auch unendliche Mengen zugelassen.

Formal kann man sagen:

Definition 2 *Ein Petrinetz ist das Tupel:*

$$N = (S, T, F)$$

- *S ist die Stellenmenge.*
- *T ist die Transitionsmenge, wobei $S \cap T = \emptyset$.*
- *F ist die Flussrelation: $F \subseteq (S \times T) \cup (T \times S)$.*

Die Flussrelation F eines Netzes kann als Kantenmenge eines Graphen aufgefasst und dargestellt werden. Kanten können nie von Stelle zu Stelle oder von Transition zu Transition führen. Ein einfaches Petrinetz besteht nur aus Stellen, Transitionen und Kanten, man nennt es auch Netzgraph. Die Stellen in einem Netz, die zu einer Transition führen, nennt man Eingangselemente. Die Stellen, die von einer Transition wegführen, nennt man Ausgangselemente. Nach [JV87, S.10] hat ein Petrinetz u.a. folgende Eigenschaften.

Definition 3 *Es sei N = (S,T,F) ein Netz.*

Für ein Element $x \in S \cup T$ bezeichnet ${}^\bullet x := \{y|(y,x) \in F\}$ die Menge der Eingangselemente, sowie $x^\bullet := \{y|(x,y) \in F\}$ die Menge der Ausgangselemente von x. Insbesondere heißt ${}^\bullet t$ bzw. $t^\bullet$ Menge der Vor- bzw. Nachbedingungen oder auch Eingangs- bzw. Ausgangsstellen von t. ${}^\bullet s$ bzw. $s^\bullet$ heißen Menge der Eingangs- bzw. Ausgangstransitionen von s.

Für eine Menge $A \subseteq S \cup T$ definiert man entsprechend ${}^\bullet A := \{y|\exists x \in A : (y,x) \in F\}$ sowie $A^\bullet := \{y|\exists x \in A : (x,y) \in F\}$.

Ist eine Stelle s gleichzeitig Eingangs- und Ausgangsstelle einer Transition t, also $s \in {}^\bullet t \cap t^\bullet$, dann heißt s Nebenstelle oder Nebenbedingung von t. N heißt nebenbedingungsfrei, rein oder auch schlingenfrei, falls N keine Nebenbedingungen enthält, d.h. $t^\bullet \cap {}^\bullet t = \emptyset$ gilt, $\forall t \in T$.

Ein Netz heißt schlicht, wenn keine zwei Knoten den selben Vor- und den selben Nachbereich haben, d.h. wenn $\forall x, y : {}^\bullet x = {}^\bullet y$ und $x^\bullet = y^\bullet \Rightarrow x = y$.

Zwei Transitionen t und t' sind unabhängig, gdw. $({}^\bullet t \cup t^\bullet) \cap ({}^\bullet t' \cup t'^\bullet) = \emptyset$, d.h. wenn sie unterschiedliche Eingangs- und Ausgangsstellen haben. Dies drückt die statische Unabhängigkeit zweier Transitionen aus.

Bedingungs-/Ereignis-Netze sind nach [Bau90] und EN-Systeme (elementare Netzsysteme) nach [RR98], Netze, an deren Markierungen man sehen kann, ob eine Bedingung gilt. D.h., es sind Netze, in denen lediglich die Gültigkeit (oder

die Ungültigkeit) von Bedingungen interessiert, die durch Ereignisse eintreten oder beendet werden.

Ob eine Stelle bzw. Bedingung gilt, sieht man an einer Markierung, die meist als ein schwarzer Punkt in der Stelle dargestellt wird. Diese Netze sind an ihren Stellen entweder mit einer Marke markiert oder unmarkiert. Eine Markierung M eines Netzes N ist eine Teilmenge von S: $M \subseteq S$. Eine Marke $s \in M$ heißt: „Bedingung gilt“, $s \notin M$ heißt „Bedingung gilt nicht“.

Man kann auch sagen, die Kapazität einer Stelle ist eins, da sie in der Lage ist eine Marke aufzunehmen. Die Verteilung von Marken in einem Netz zeigt den momentanen Zustand an, in dem sich ein System gerade befindet.

Transitionen sind die aktiven Elemente. Sie können, wenn sie aktiviert sind, schalten (feuern), d.h., sie können Marken in den Eingangstellen entfernen und in den Ausgangsstellen wieder erzeugen. Aktiviert ist eine Transition dann, wenn in all ihren Eingangsstellen eine Marke liegt und die Ausgangsstellen frei sind. Voraussetzung für das Schalten einer Transition ist, dass alle Eingangsstellen und Nebenbedingungen markiert sowie alle Ausgangsstellen unmarkiert sind.

Ein EN-System ist nach [RR98, S.23] ein Netz mit initialer Konfiguration.

Definition 4 *Ein* elementares Netzsystem *ist ein Quadrupel $M = (S, T, F, C_0)$, wobei (S, T, F) ein Netz bildet und $C_0 \subseteq S$ eine initiale Konfiguration ist.*

Eine Transition t ist aktiviert in M, notiert als $M \xrightarrow{t}$, falls gilt:

1. *Der Zustand M besitzt genug Marken: ${}^\bullet t \subseteq M$*
2. *Die Transition ist kontaktfrei: $t^\bullet \cap M = \emptyset$*

Eine aktivierte Transition t kann in die Nachfolgemarkierung M' schalten, die durch $M' = (M \setminus {}^\bullet t) \cup t^\bullet$ definiert ist. Dies wird als $M \xrightarrow{t} M'$ notiert.

Es gibt vier typische Muster an Schaltfolgen, die in Petrinetzen auftreten können (aus [Kum01]).

1. Bei einem Muster wird das Netz sequentiell ausgeführt. Es feuert eine Transition nach der anderen. Abbildung 3.1 zeigt so ein Verhalten.

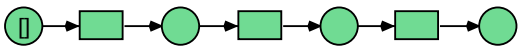

Abbildung 3.1: Eine sequentielle Schaltfolge

2. Im nächsten Muster findet eine Synchronisation statt, d.h., eine Transition vereinigt zwei oder mehr sonst unabhängige Äste eines Netzes. In Abbildung 3.2 ist so eine Schaltfolge zu sehen. Wichtig ist, dass eine Transition erst feuern kann, wenn alle Eingangsstellen markiert sind. Eine Synchronisation wird sozusagen erzwungen, denn sollte eine Stelle noch nicht bereit sein, da noch Marken fehlen, wartet die Transition mit dem Feuern so lange, bis jede Stelle die benötigten Marken bereitstellt.

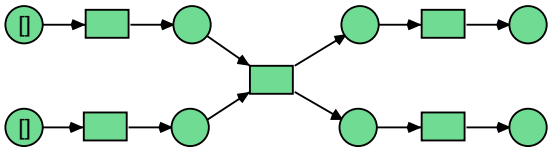

Abbildung 3.2: Eine Synchronisation

3. Ein drittes Muster beschreibt einen Konflikt (Abbildung 3.3). In diesem Fall gehen von einer Stelle mehrere Kanten weg, d.h., auf diese Stelle folgen verschiedene Transitionen. Es ist in so einem Fall nicht klar, welche Transition feuern, bzw. zuerst feuern wird. Unter Konflikt versteht man die nicht nebenläufige gleichzeitige Aktiviertheit von Transitionen.

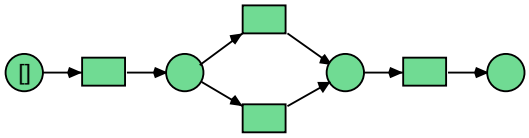

Abbildung 3.3: Eine Konfliktsituation

4. Das letzte Muster beschreibt ein nebenläufiges Verhalten. Von einer Transition gehen Kanten auf mehrere Stellen. Diese Stellen bilden im Netz jeweils den Anfang eines Astes im Netz. Abbildung 3.4 zeigt so ein Verhalten. Wenn die Transition feuert, bekommt jede dieser Stellen eine oder mehrere Marken, je nach Beschriftung der Kanten. Man kann für den weiteren Verlauf des Netzes nicht sagen, ob die folgenden Transitionen der einzelnen Äste gleichzeitig oder in einer beliebigen Reihenfolge schalten. Sollte es später wieder zu einer Synchronisation kommen, und muss die Transition mit dem Feuern warten, kann man erkennen, dass das Schaltverhalten der Transitionen der einzelnen Äste unabhängig voneinander stattfand.

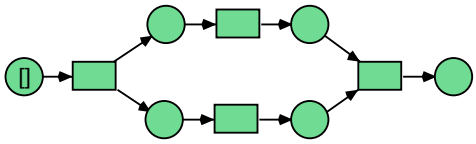

Abbildung 3.4: Ein nebenläufiges Verhalten

Stellen/Transitions-Netze sind Netze, in denen die Stellen eine endliche oder unendliche Kapazität für Marken aufweisen können. Es ist deshalb möglich, dass in einer Stelle eines S/T-Netzes, im Gegensatz zu Stellen eines B/E-Netzes, mehrere Marken liegen können, wenn mehrere Ressourcen des gleichen Typs zur Verfügung stehen. Die Kapazität einer Stelle kann in S/T-Netzen zwischen eins

und unendlich liegen. Das Kantengewicht kann ebenfalls größer als eins sein. Es können beim Schalten einer Transition also mehrere Marken aus einer Stelle entfernt und auch mehrere Marken in einer anderen Stelle wieder erzeugt werden, je nach Kantengewicht (auch Kantenbewertung genannt). Wichtig ist, dass es im S/T-Netz, im Gegensatz zu einem Netz, das nur aus Stellen, Transitionen und Kanten besteht, immer eine Anfangsmarkierung gibt. Man kann sagen, ein S/T-Netz ist eine Erweiterung eines B/E-Netzes, oder ein B/E-Netz ein Spezialfall der S/T-Netze.

Formal ist nach [JV87, S.37] ein S/T-Netz folgendermaßen definiert:

Definition 5 *Ein Stellen/Transitions-Netz – im folgenden kurz: S/T-Netz – ist das Tupel:*

$$N = (S, T, F, W, K, m_0)$$

- *S ist eine endliche Menge an Stellen.*
- *T ist eine endliche Menge an Transitionen mit $S \cap T = \emptyset$.*
- *F ist die Flussrelation: $F \subseteq (S \times T) \cup (T \times S)$.*
- *W ist die Kantengewichtungsfunktion $W : (S \times T) \cup (T \times S) \to \mathbb{N}$, wobei $W(x, y) = 0$ gdw. $(x, y) \notin F$.*
- *einer Kapazitätsfunktion $K : S \to \mathbb{N} \cup \{\omega\}$. $K(s) = \omega$ bedeutet, dass s keine endliche Kapazität besitzt,*
- *$m_0 : S \to \mathbb{N}$ ist die Anfangsmarkierung des Systems und $m_0(s) \leq K(s)$ gilt für alle $s \in S$.*

In der graphischen Darstellung werden $K(s)$, $W(f)$ bzw. $m_0(s)$ an bzw. in das entsprechende Element geschrieben. Soweit nicht explizit notiert, wird $K(s) = \omega$, $W(f) = 1$ bzw. $m_o(s) = 0$ angenommen. Ebenso kann K bzw. W in der Angabe von N entfallen, wenn $K(s) = \omega$ bzw. $W(f) = 1$ für alle $f \in F$, $s \in S$ gilt. Umgekehrt ist F durch die Angabe von W eindeutig festgelegt. Kapazitäten können äquivalent durch Komplementärstellen ausgedrückt werden.

Jede Kante kann mit einer Zahl, Kantengewicht genannt, versehen werden, die die Anzahl der Marken anzeigt, die beim Feuern in einer Stelle entfernt und in einer anderen wieder erzeugt werden. In Bedingungs/Ereignis-Netzen ist das Kantengewicht immer eins, da immer nur eine Marke „verschoben“ wird. Ein B/E-Netz ist immer schlingenfrei, schlicht, hat die Kapazität $K(s) = 1$ und das Kantengewicht $W(x, y) \leq 1$.

Die Beschreibung einer Markierung für ein S/T-Netz lautet formal nach [JV87, S.36]:

Definition 6 *Es sei $N = (S, T, W, K, m_0)$ ein S/T-Netz. Eine Markierung m von N ist eine Verteilung von Marken auf den Stellen oder formal eine Abbildung von S in die nicht negativen, ganzen Zahlen $\mathbb{N} := \{0, 1, 2, ...\}$:*

$$m : S \to \mathbb{N}$$

Dabei gibt $m(s)$ die Anzahl der Marken (englisch: tokens) in der Stelle $s \in S$ an.

Mit M_S bezeichnet man die Menge aller Markierungen über S.

Damit eine Transition eines S/T-Netzes schalten kann, muss sie zuvor aktiviert sein. Eine Transition $t \in T$ heißt aktiviert in m, $m \xrightarrow[N]{t}$, wenn für alle $s \in {}^\bullet t : m(s) \geq W(s,t)$ und für alle $s \in t^\bullet : m(s) \leq K(s) - W(t,s) + W(s,t)$.

Die Bedingungen für die Aktiviertheit sorgen dafür, dass beim Schalten einerseits genügend Marken entsprechend den Kantengewichten vorhanden sind, so dass die Markierung einer Stelle nicht unter Null sinkt, und andererseits die Kapazität einer Stelle nicht überschritten wird. D.h., auch nach dem Schalten erhält man wieder eine legale Markierung.

Man sagt t schaltet von m nach m' und schreibt $m \xrightarrow{t} m'$, wenn t unter m aktiviert ist und m' aus m durch Entnahme von Marken aus den Eingangsstellen und Ablage von Marken auf die Ausgangsstellen gemäß den Kantengewichten entsteht. m' heißt dann Folgemarkierung von m unter t. Die Aktivierungsbedingungen und die Definition der Folgemarkierungen bezeichnet man als Schaltregel.

Ein S/T-Netz ist nach [JV87, S.39] aktiviert und kann schalten, wenn folgendes gilt:

Definition 7 *Es sei $N = (S, T, F, K, W, m_o)$ ein S/T-Netz, $t \in T$ eine Transition und $m_1, m_2 \in M_S$ Markierungen.*
t heißt aktiviert in m_1, symbolisch $m_1 \xrightarrow[N]{t}$, falls $m_1(s) \geq W(s,t)$ und $m_1(s) - W(s,t) + W(t,s) \leq K(s)$ für alle $s \in S$ gilt.

Die Transition $t \in T$ schaltet m_1 zu m_2, symbolisch $m_1 \xrightarrow[N]{t} m_2$, falls t in m_1 aktiviert ist und $m_2(s) = m_1(s) - W(s,t) + W(t,s)$ für alle $s \in S$ gilt.

Ist N unzweifelhaft, so schreibt man auch $m_1 \xrightarrow{t}$ bzw. $m_1 \xrightarrow{t} m_2$.

Definition 8 *Ein S/T-Netz $N = (S, T, F, K, W, m_0)$ heißt kontaktfrei, wenn für alle erreichbaren Markierungen $m \in R(N)$ eine Transition $t \in T$ genau dann aktiviert ist, wenn $m(s) \geq W(s,t)$ für alle $s \in {}^\bullet t$ gilt.*

Für die Schaltregel gilt: Es sei (N, m_0) ein S/T-Netz und $w = t_1, ...t_n \in T^\star$. Man nennt eine Markierung $m'' \in M_S$ eine Folgemarkierung von m_0 unter w und schreibt $m_0 \xrightarrow{w} m''$, wenn gilt:

1. $w = \lambda$ und $m'' = m$ oder
2. Es gibt ein $m' \in M_S : m_0 \xrightarrow{t_1 ... t_{n-1}} m'$ und $m' \xrightarrow{t_n} m''$.

Man kann also sagen, eine Transition t eines S/T-Netzes ist dann aktivierbar, wenn eine Schaltfolge w mit $m_0 \xrightarrow{w}$ existiert. Durch wiederholtes Schalten erhält man eine Schaltfolge bzw. einen seriellen Prozess.

Weitere wichtige Eigenschaften von Petrinetzen sind Fairness, Fortsetzbarkeit und Lebendigkeit (siehe dazu Kapitel 8). Die Verklemmungsfreiheit eines Systems von Funktionseinheiten bedeutet nach [JV87], dass kein totaler Stillstand der ablaufenden Prozesse eintritt. Dadurch ist natürlich nicht ausgeschlossen, dass gewisse Funktionseinheiten oder Teile des gesamten Systems blockiert sind. Die Abwesenheit von solchen partiellen Verklemmungen nennt man Lebendigkeit des Systems. Ein S/T-Netz ist also lebendig, wenn nach einer beliebigen Schaltfolge jede Transition wieder zum Schalten gebracht werden kann. Sie ist fortsetzbar, wenn von m aus so geschaltet werden kann, dass jede Transition immer wieder schaltet. Daraus folgt, lebendige und fortsetzbare Schaltfolgen sind sicher. In einer sicheren oder fortsetzbaren Markierung gibt es immer mindestens eine Feuerfolge, die das System terminieren bzw. unbegrenzt weiterlaufen lässt. Ein lebendiges Netz verhält sich aber nicht immer fair. Ein S/T-Netz hat faires Verhalten, wenn in jeder unendlichen Schaltfolge jede Transition unendlich oft schaltet. Der Unterschied zur Lebendigkeit besteht darin, dass das S/T-Netz nicht nur frei von unvermeidbaren partiellen Verklemmungen ist, sondern dass es auch faktisch frei von partiellen Verklemmungen ist.

In diesem Abschnitt sollte eine allgemeine Einführung in Petrinetze gegeben werden. Das ist wichtig, da im Folgenden mit Petrinetzen modelliert werden soll und im nächsten Kapitel weitere, über den Formalismus der S/T-Netze hinausgehende Petrinetzformalismen vorgestellt werden.

3.2 Petrinetzmodelle

Wie oben schon erwähnt, eignen sich Petrinetze zur Modellierung und Implementierung von Simulationsszenarien. Wie man dabei vorgehen kann und welche Möglichkeiten einem für die Modellierung offen stehen, soll jetzt anhand eines Beispiels gezeigt werden. Das Beispiel stammt aus meiner Studienarbeit [Str01] und wird hier in leicht veränderter Weise wiedergegeben. Es werden in diesem Beispiel nur die bekannten S/T-Netze eingesetzt. Später wird gezeigt, wie mit höheren Petrinetzen kompaktere Simulationsmodelle implementiert werden können. Es handelt sich bei dem verwendeten Beispiel um einen LKW- Ent- und-Beladevorgang:

Ein LKW fährt in eine Ladezone, um entladen und wieder beladen zu werden. Ein Gabelstapler kommt ebenfalls in die Ladezone, um den LKW zu entladen und wieder zu beladen. Danach fährt der Gabelstapler an seinen Ausgangsstandort zurück und der LKW verlässt das System. Da es sich um ein stark vereinfachtes Beispiel handelt, werden nur ein LKW und ein Gabelstapler betrachtet. Das Beispiel soll auf drei mögliche Arten modelliert werden, als UML-Diagramm, genauer als UML-Aktivitätsdiagramm nach [GBB00], als einfaches S/T-Netz und als zwei S/T-Netze, die über gleichnamige Transitionen miteinander synchronisiert werden. Beim UML-Aktivitätsdiagramm, Abbildung 3.5, ist leicht zu erkennen, dass es in diesem System zwei Prozesse gibt: den LKW-Prozess und den Gabelstapler-Prozess. Für jeden einzelnen Prozess wird deutlich, wie er abläuft. Es wird aber nicht deutlich, wann und wie sich die beiden Prozesse synchronisieren, dazu müsste das Diagramm um Synchronisationskonstrukte

erweitert werden (Die grauen Linien in der Abbildung gehören nicht zum Diagramm, sie dienen nur als Beschreibungshilfe).

Im Gegensatz dazu kann man bei dem S/T-Netz (Abbildung 3.6) sofort erkennen, dass an den Transitionen *entladen_lkw* und *beladen_lkw* eine Synchronisation stattfindet. Der LKW- und der Gabelstapler-Prozess können in diesem Beispiel ebenfalls leicht erkannt werden. Es besteht aber die Gefahr, dass ein S/T-Netz, das mehrere Prozesse beinhaltet bzw. ein komplexeres Szenario darstellt, unübersichtlich wird.

Um diesem Problem aus dem Weg zu gehen, kann man auch jeden Prozess als eigenes S/T-Netz darstellen und die Transitionen, die synchron ablaufen sollen, über gleiche Namen synchronisieren, wie in den Abbildungen 3.7 und 3.8 zu sehen ist. Das hat den Vorteil, dass jeder einzelne Prozess leicht nachzuvollziehen ist und man trotzdem eine zu erkennende Synchronisationsmöglichkeit hat. In diesem Beispiel kommen die Transitionen *entladen_lkw* und *beladen_lkw* in beiden Netzen vor, d.h., an diesen Transitionen müssen sich die Netze synchronisieren (vgl. Kapitel 4.4). Dieses Beispiel sollte zeigen, dass sich Petrinetze zur Modellierung von Systemen durchaus einsetzen lassen. Sie verfügen über alle wesentlichen Modellierungseigenschaften der UML-Aktivitätsdiagramme.

3.3 Vorteile und Nachteile von Petrinetzen gegenüber imperativen Ansätzen

Warum Petrinetze zur Modellierung so gut geeignet sind, sieht man auf Grund ihrer graphischen Darstellung schnell ein. Ein Modell kann intuitiv graphisch erstellt werden. Zur Simulation eignen sie sich, da es eine Reihe von Petrinetzeditoren gibt, die auch einen Simulator beinhalten, mit dem ein modelliertes und implementiertes Szenario durchgespielt werden kann. Dies ist durch die mathematisch exakte Formulierung möglich. Es ist, wie schon im vorangegangenen Abschnitt erwähnt, mit Petrinetzen möglich, Synchronisationen verschiedener Netze vorzunehmen. Genau diese Synchronisationen sind es, die im imperativen Ansatz oft Probleme bereiten, da sie ein Modell verkomplizieren.

Im Petrinetzansatz ist das Modell gleichzeitig das Programm, d.h. die Modellierung und die Implementierung sind identisch. Jede Veränderung im Modell hat deshalb sofort auch eine Veränderung in der Implementierung zur Folge. Das hat den Vorteil, dass ein guter Gedanke, um den ein Modell erweitert wird, sofort in der Implementierung mit aufgenommen wird. Man muss sich keine Gedanken über die Umsetzung machen. Die Tatsache, dass Modellierung und Implementierung zusammenfallen, hat darüber hinaus auch den Vorteil, dass bei der Erstellung eines Simulationsmodells für ein bestimmtes Szenario viel Zeit gespart wird.

Bei der Entwicklung von Simulationsmodellen mit imperativen Programmiersprachen, wird im Allgemeinen erst ein UML- oder Flussdiagramm erstellt, das dann nachprogrammiert wird. Ein UML-Diagramm kann sehr einfach aussehen, sich aber bei der Programmierung als sehr kompliziert herausstellen. Das liegt daran, dass eine Programmiersprache ein Diagramm nicht eins zu eins übernimmt. Eine Programmiersprache stellt bestimmte Konstrukte für die Umset-

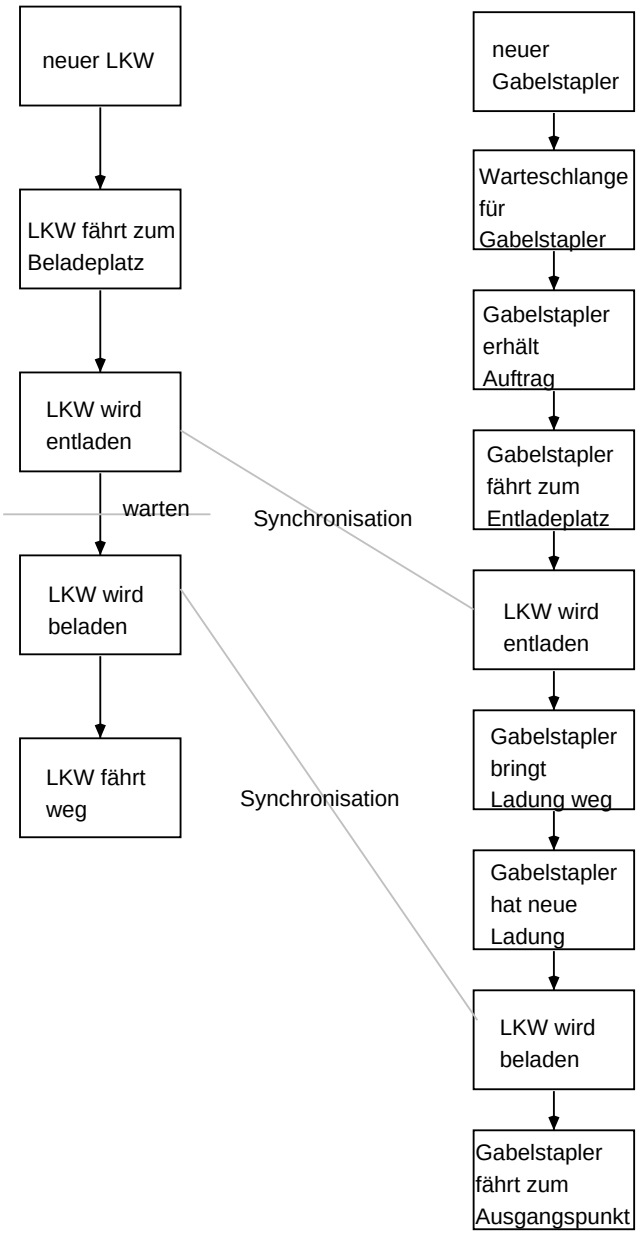

Abbildung 3.5: Das Szenario als UML-Aktivitätsdiagramm

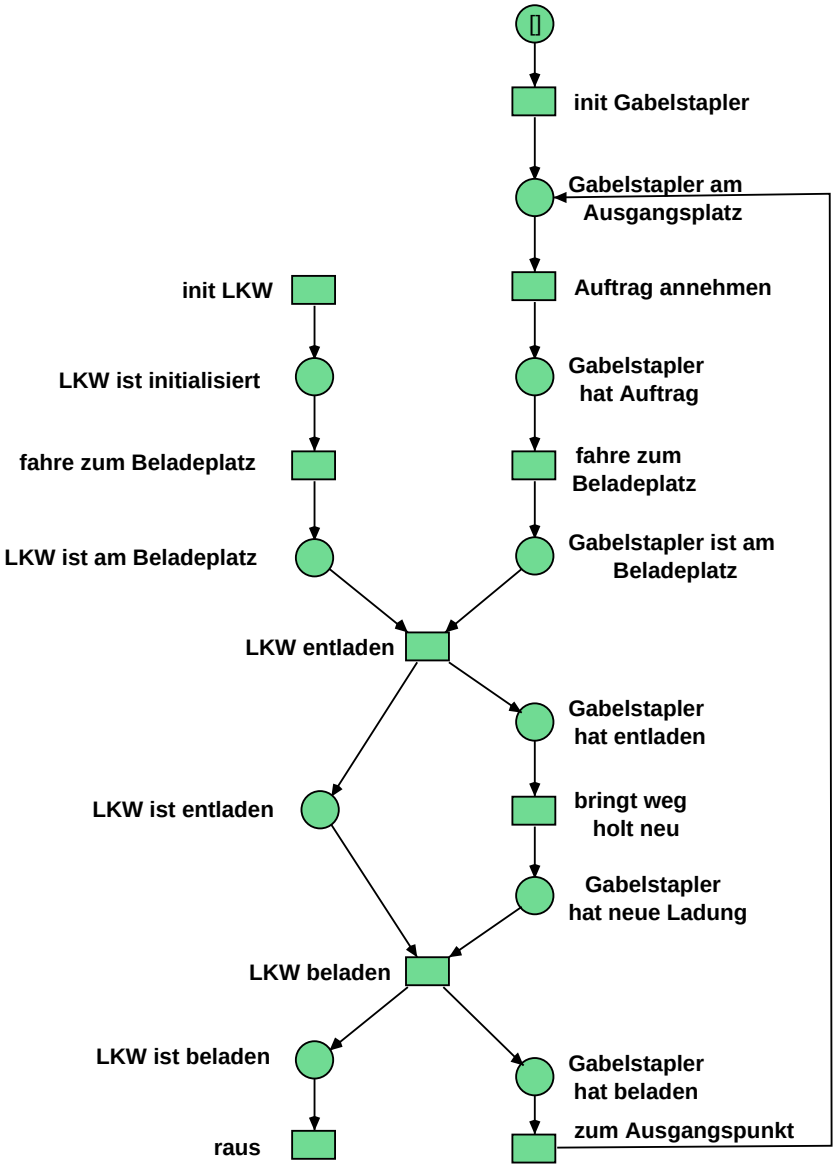

Abbildung 3.6: Das Szenario als S/T-Netz

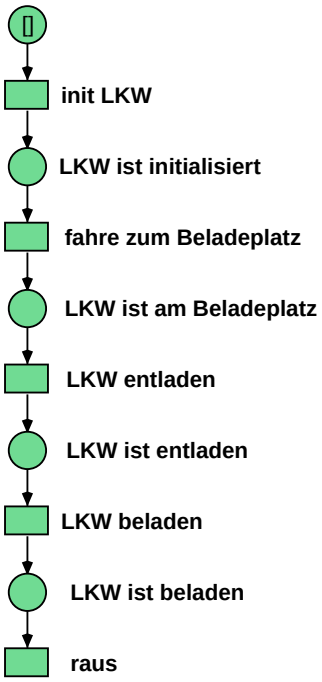

Abbildung 3.7: Das LKW-Netz

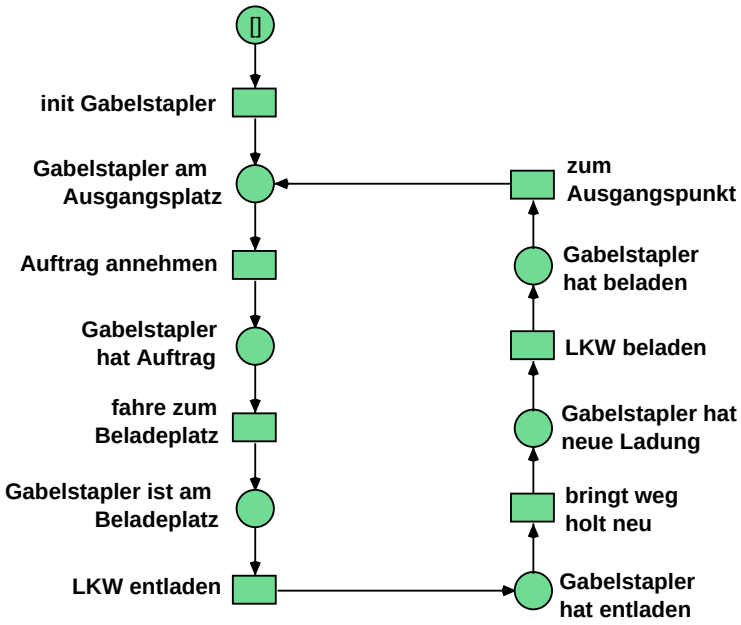

Abbildung 3.8: Das Gabelstapler-Netz

zung bestimmter Verhaltensweisen zur Verfügung. Diese Konstrukte muss man sehr gut kennen, um ein Simulationsmodell zu implementieren. Dadurch kann ein mit einer imperativen Programmiersprache erstelltes Simulationsmodell zwar die Funktionalität des vorher erstellten Diagramms erfüllen, es muss aber wie bereits erwähnt meist neu erstellt werden. Dazu ist es nötig, die verwendete Programmiersprache sehr gut zu beherrschen, da diese Art der Implementierung sehr fehleranfällig ist. Ein ebenfalls schon genanntes großes Problem der imperativen Programmiersprachen ist die Synchronisation. Eine im Diagramm modellierte Synchronisation kann mit Programmiersprachen oft nur sehr mühsam hergestellt werden, da diese Programmiersprachen nur einer sequentiellen Programmsteuerung folgen. Eine Parallelität, Gleichzeitigkeit und Nebenläufigkeit kann mit ihnen nicht durchgeführt werden (außer mit Multi-Threating-Programmen). Petrinetze dagegen unterstützen Nebenläufigkeit und Parallelität sehr gut, sie sind quasi dafür entwickelt worden (siehe Kapitel 3.1). Auch Synchronisationen können mit Petrinetzen sehr leicht durchgeführt werden, wie in Kapitel 3.2 gezeigt wurde.

Hat man einen Fehler gemacht, ist es in der imperativen Programmierung vergleichsweise schwierig, den Fehler zu finden, da es sich sowohl um einen Syntaxfehler als auch um einen logischen Fehler handeln kann. In modernen Programmiersprachen bereiten Syntaxfehler keine größeren Probleme, in älteren Sprachen, die durchaus noch verwendet werden, kann die suche nach einem Syntaxfehler aufwendig sein.

Es ist selbstverständlich auch möglich, bei der Implementierung mit Petrinetzen Syntaxfehler zu machen, da es sich im Bereich der Simulation meistens um höhere Petrinetze handelt, die einen erweiterten Formalismus darstellen und deshalb meistens auch eine erweiterte Syntax besitzen.Im Gegensatz zu den imperativen Programmiersprachen sind die Fehler hier aber meistens nicht ganz so schwierig zu finden, da die Syntax in den meisten Petrinetzformalismen und -editoren etwas überschaubarer ist als in imperativen Programmiersprachen. Diese Behauptung ist allerdings sehr subjektiv und mag für geübte gute Programmierer nicht gelten. Insbesondere dann nicht, wenn Petrinetze bis dahin unbekannt waren.

Das größere Problem bei der Implementierung eines Simulationsmodells bilden die logischen Fehler. Im imperativen Ansatz ist es schwierig, einen logischen Fehler zu finden, da man ausschließlich Code vor sich hat, der wenn er von einem Menschen gelesen wird schwer auf seinen sequenziellen Ablauf hin überprüft werden kann. Da ein imperatives Programm Zeile für Zeile aufgeschrieben wird, es also weder mit graphischer Unterstützung noch intuitiv erstellt werden kann und deshalb auch nicht anhand einer Grafik intuitiv überprüft werden kann, ist es im Vergleich zum Petrinetzansatz erheblich schwieriger, logische Fehler zu finden. Allerdings gibt es auch hier neuere Ansätze, die dieses Problem lösen. Im petrinetzorientierten Ansatz kann ein logischer Fehler im Programm schnell gefunden werden, da das Modell gleichzeitig auch das Programm ist und der Ablauf des Modells während der Simulation beobachtet werden kann. Man sieht die Marken, die temporäre Objekte darstellen, durch das Netz, das die statischen Komponenten darstellt, wandern. Ein Netz das schalten soll, es aber nicht tut ist fehlerhaft. Man kann im Petrinetz aber ablesen an welcher Stelle es zum

Fehler gekommen ist, und den Fehler deshalb meist schnell beheben.

Man kann eine Petrinetzsimulation immer visuell verfolgen und Unklarheiten deshalb schnell erkennen. Durch das Korrigieren des Modells hat man gleichzeitig die Implementierung korrigiert. Man kann in Petrinetzen in recht kurzer Zeit viele Veränderungen vornehmen, da im Allgemeinen nur Stellen und Transitionen weggenommen oder hinzugefügt werden müssen.

Abschließend kann man feststellen, dass es vergleichsweise einfach ist, mit einem Petrinetzeditor ein Simulationsmodell bzw. -programm zu entwickeln, da es nicht nötig ist, eine Programmiersprache perfekt zu beherrschen. Voraussetzung ist nur, dass man den vom Petrinetzeditor unterstützten Petrinetzformalismus kennt und einen entsprechenden Petrinetzsimulator zur Verfügung hat.

Ein Vorteil des imperativen Ansatzes besteht allerdings in der weiten Verbreitung von Simulationssoftware, die auf imperativen Sprachen beruht. Es gibt speziell Simulationssprachen, wie beispielsweise Simula, Simulations-Klassenbibliotheken, wie beispielsweise Desmo, und Simulatoren, die für eine bestimmte Anwendung entwickelt wurden und bei denen lediglich Parameter verändert werden müssen, wie beispielsweise Flugsimulatoren. Außerdem gibt es für einen großen Teil dieser Software geeignete Auswertungskomponenten, die bei Petrinetzsimulatoren oft nicht so umfangreich sind oder sogar ganz fehlen. Da der in dieser Arbeit für die Simulation eingesetzte Petrinetzeditor Renew auch keine Auswertungskomponente besitzt, wird in den Kapiteln 8 und 9 diskutiert, wie eine geeignete Auswertungskomponente für die Simulation mit Petrinetzen speziell in Renew aussehen sollte.

Fazit

In diesem Kapitel wurde gezeigt, wie Petrinetze für die Simulation eingesetzt werden können. Dazu wurde als Erstes eine allgemeine Einführung in die Grundlagen der Petrinetze gegeben. Es wurden einfache Netze, S/T-Netze und Bedingungs/Ereignis-Netze definiert. Außerdem wurde das allgemeine Schaltverhalten von Netzen erklärt. Netze können sequentiell, und nebenläufig schalten; des weiteren gibt es Konflikt- und Synchronisationssituationen. Danach wurde anhand von Beispielen gezeigt, wie sich Szenarien als Petrinetzmodell darstellen lassen. Damit konnten dann die Vor- und Nachteile der Petrinetzmodelle gegenüber imperativen Simulationsmodellen diskutiert werden. Es hat sich herausgestellt, dass sich Petrinetze zur Modellierung von Szenarien gut eignen.

4 Simulationsmodelle mit höheren Petrinetzen

Nachdem im vorangegangenen Kapitel der Formalismus einfacher S/T-Netze erläutert wurde, sollen jetzt erweiterte Petrinetzformalismen vorgestellt werden. Diese Petrinetzformalismen bezeichnet man zusammenfassend auch als höhere Petrinetze.

Höhere Netzkonstrukte erlauben es, Marken individuell zu unterscheiden, sodass komplexere Systeme oder Sachverhalte einfacher beschrieben bzw. ausgedrückt werden können als in einem S/T-Netz. Höhere Netztypen können als Faltung von S/T-Netzen aufgefasst werden. S/T-Netze und Substrukturen können wiederum als Verfeinerung von höheren Netzen angesehen werden. In einem höheren Netz kann eine Substruktur durch eine einzige Transition dargestellt werden. Man nennt diese Transition dann Funktionseinheit. Da ein höheres Netz Sachverhalte durch die Funktionseinheiten wesentlich kompakter ausdrücken kann als ein einfaches S/T-Netz, eignet es sich besonders gut zur Beschreibung von Systemen, was wiederum nützlich für den Einsatz in der Simulation ist.

Es gibt viele verschiedene Netzformalismen, die alle eine unterschiedliche Sichtweise auf Systeme haben und unterschiedliche Schwerpunkte beinhalten. In dieser Arbeit sollen der Formalismus der *gefärbten Petrinetze*, der *Objekt-Petrinetze*, der *Netze in Netzen*, der *Referenznetze* und der der *zeitbehaftete Petrinetze* beschrieben werden. Später wird gezeigt, dass sich alle Formalismen beim Einsatz in der Simulation wiederfinden.

Gefärbte Petrinetze stellen eine Faltung des S/T-Netz Formalismus dar. Verschiedene, aber gleichartige Transitionen oder auch ganze Teilnetze werden in einer Transition zusammengefasst. Die Marken werden durch zugeordnete Werte voneinander unterschieden. Beim Entwickeln des Netzes muss dringend darauf achtet werden, dass die unterschiedlichen Zustände, die das Netz vor der Faltung annehmen konnte, beibehalten werden. Deshalb muss für jede darzustellende Transition eine eigene Beschriftung einführt werden. Unterschiedliche Beschriftungen symbolisieren dann unterschiedliche Transitionen.

Eine Erweiterung der gefärbten Netze sind die Objekt-Petrinetze. Bei diesem Formalismus geht man von gefärbten Netzen aus und versucht Eigenschaften der Objektorientierung zu integrieren.

Die Idee der „Netze in Netzen“ besteht darin, als Marken eines Netzes wiederum Netze zu erlauben, wodurch eine Hierarchie von Netzen entsteht.

Referenznetze sind ein spezieller Fall des „Netze in Netzen“-Formalismus. Referenznetze sind sehr gut für den Einsatz in der Simulation geeignet, da sie über Referenzen zu anderen Netzen, Synchronisationskonstrukten und andere hilfreiche Eigenschaften für den konkreten Einsatz verfügen. Die Formalismen der gefärbten und der Objekt-Petrinetze sind in diesem Formalismus ganz oder

teilweise enthalten. In dieser Arbeit werden für die Simulation Referenznetze eingesetzt.

Abschließend soll noch der Formalismus der zeitbehafteten Petrinetze diskutiert werden. Dieser Formalismus kann auf jeden der bislang diskutierten Formalismen aufsetzen. Zusätzlich wird das Netz mit Zeitstempeln bzw. Funktionen für den Zeitverbrauch versehen, um das Schalten der Transitionen noch stärker beeinflussen zu können. In der zeitdiskreten Simulation ist der Zeitverbrauch von großer Wichtigkeit, weshalb man für die Simulation mit Petrinetzen einen Formalismus benötigt, der diesen Sachverhalt unterstützt.

4.1 Gefärbte Petrinetze

Um komplexe Systeme einfacher darzustellen bzw. große S/T-Netze zu vereinfachen, geht man von Netzen aus, bei denen anstelle der ununterscheidbaren Marken individuelle Objekte verwendet werden. In Systemen mit individuellen Marken bzw. Objekten haben die Marken einen bestimmten Typ, dem Werte zugeordnet werden. Durch die Werte können die Marken dann voneinander unterschieden werden. Den Wert der Marken bezeichnet man auch als Farbe. Dieses zusätzliche Ausdrucksmittel erlaubt es, Sachverhalte kompakter auszudrücken und Algorithmen zu modellieren. Die Kanten solcher Netze werden mit Variablen, z.B. x und y, bewertet, die diese Markentypen als Werte annehmen können. Bei diesen Variablen handelt es sich nicht um Variablen im Sinne imperativer Programmiersprachen, sondern um Platzhalter ähnlich wie beispielsweise in Prolog. Es kann sich dabei um Zahlen, Strings, Tupel oder andere komplexe Datentypen handeln. Es ist zusätzlich möglich, dass zwei verschiedene Marken den gleichen Wert annehmen. Aufgrund dieser kompakten Ausdrucksweise ist es mit gefärbten Netzen möglich, annähernd jeden Sachverhalt in einem im Vergleich zu S/T-Netzen kleineren Netz darzustellen.

Im einfachsten Fall ändert sich die Datenstruktur der Marke beim Feuern einer Transition im gefärbten Netz nicht, prinzipiell ist das aber möglich. Es muss in den Schaltregeln Bezug auf den Wert der Marken genommen werden, d.h., dem Schalten einer Transition t muss eine Zuweisung oder Bindung von Werten an diese Variablen vorausgehen.

Die Kanten von einer Stelle zu einer Transition und von einer Transition zu einer Stelle müssen mit Variablen bzw. Termen mit Variablen beschriftet sein. Wenn die Transition feuert, muss jede Variable an einen festen Wert gebunden sein. Die Variablenbelegungen entscheiden so, welche Werte (welche Marken) zur nächsten Stelle bewegt werden. Wie bei S/T-Netzen ist eine Transition t aktiviert, falls die zu entnehmenden Individuen vorhanden sind und eventuell vorgeschriebene Kapazitäten nicht überschritten werden. Um das Schalten einer Transition noch weiter kontrollieren und beeinflussen zu können, kann man *Guardfunktionen* oder kurz *Guards* einsetzen. Diese Guards sind boolesche Funktionen, die an eine Transition geschrieben werden. Eine Transition kann nur dann schalten, wenn die Guardfunktion *true* ergibt. Abbildung 4.1 zeigt ein einfaches gefärbtes Petrinetz.

Seien x und y Variablen. Eine mögliche Variablenbelegung ist $x =$ "A" und $y =$ "C". Für diese Variablenbelegung ist t aktiviert, da $G(t) = (x \neq y) = (true)$

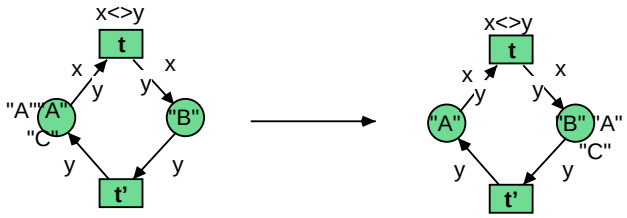

Abbildung 4.1: Das Schalten eines gefärbten Petrinetzes

gilt. Eine Variablenbelegung b der Variablen der Transition t, die das Prädikat $G(t)$ erfüllt, wird *Bindung* genannt, das Paar (t, b) wird *Bindungselement* genannt. Die Variablenbelegung $x =$ "C" und $y =$ "B" ist auch eine Bindung. Die Belegung $x = y =$ "A" ist keine Bindung, da $G(t)$ nicht gilt. Man beachte, dass eine Bindung der Variablen für t völlig unabhängig von Variablen anderer Transitionen (hier: t') ist.

Ein weit verbreiteter Formalismus der gefärbten Petrinetze sind die *Coloured Petri Nets (CPN)*. Sie werden von Jensen in [Jen92] vorgestellt. Sie sind wie folgt definiert:

Definition 9 *Ein gefärbtes Netz ist das Tupel:*

$$CPN = (\Sigma, P, T, A, N, C, G, E, I)$$

- Σ*: endliche Menge jeweils nichtleerer Farben*
- P*: endliche Menge von Stellen*
- T*: endliche Menge von Transitionen*
- A*: endliche Menge von Kanten, wobei* P*,* T *und* A *disjunkt sind:* $P \cap T = P \cap A = T \cap A = \emptyset$
- N*: Knotenfunktion* $N : A \to (P \times T) \cup (T \times P)$
- C*: Farbfunktion* $C : P \to \Sigma$
- G*: Guardfunktion* $G : T \to expr$*, wobei*

$$\forall t \in T : Type(G(t)) = \ bool, Type(Var(G(t))) \subseteq \Sigma$$

- E*: Kantenbewertung* $E : A \to expr$*, wobei*

$$\forall a \in A : Type(E(a)) = C(p(a))_{MS}, Type(Var(E(a))) \subseteq \Sigma$$

  Dabei bezeichne $p(a) = p_1$*, falls* $N(a) = (t_1, p_1)$ *bzw.* $N(a) = (p_1, t_1)$

- I*: Initialisierung* $I : P \to expr$*, wobei*

$$\forall p \in P : Type(I(p)) = C(p)_{MS}, Var(I(p)) = \emptyset$$

Hierbei sei $Var(G(t))$ und $Var(E(a))$ die Menge aller freien Variablen, die in Guard- und Kantenausdrücken vorkommen. $Type(expr)$ ist die Menge aller Typen, die Variablen in expr zugeordnet sind.

Die Ausdrücke sowie Typisierungen der Guardfunktion G und der Kantenbewertungsfunktion E sind meist einer Programmiersprache (z.B. ML für CPN) entnommen.

Das Schaltverhalten ist mit Hilfe von Variablenbindungen definiert. Sei b eine Bindung der Variablen einer Transition t. Die Menge aller Bindungen für eine Transition t wird mit $B(t)$ bezeichnet. Für einen Ausdruck E bezeichne $E\langle b\rangle$ die Auswertung von E durch die Variablenbelegung b.

Ein Schritt Y, d.h. eine Multimenge an Transitionen, ist in der Markierung m_1 unter der Bindung b aktiviert, wenn die Auswertung aller Kantenausdrücke $E(p,t)$ unter der Bindung b kleiner als die aktuelle Markierung ist:

$$\forall p \in P : m_1(p) \geq \sum_{(t,b)\in Y} E(p,t)\langle b\rangle$$

Ist in der Markierung m_1 der Schritt Y aktiviert, so kann dieser Schritt zur Markierung m_2 schalten, die wie folgt definiert ist:

$$\forall p \in P : m_2(p) = m_1(p) - \sum_{(t,b)\in Y} E(p,t)\langle b\rangle + \sum_{(t,b)\in Y} E(t,p)\langle b\rangle,$$

Die Markierung m_2 ist dann mit dem Schritt Y direkt aus der Markierung m_1 erreichbar, was als $m_1 \xrightarrow{Y} m_2$ notiert wird.

Zusammenfassend kann man sagen, dass gefärbte Netze einen komplexen Sachverhalt wesentlich kompakter und u.U. auch übersichtlicher ausdrücken können als S/T-Netze. Das macht sie zu einem geeigneten Werkzeug für den Einsatz in der Simulation und anderen systembeschreibenden Gebieten.

4.2 Objekt-Petrinetze

Um ein Simulationsmodell zu erstellen, bietet sich die objektorientierte Sichtweise an, da jede Entität im Modell als ein Objekt einer Klasse repräsentiert werden kann. Wie mit imperativen Sprachen objektorientiert programmiert werden kann, ist hinlänglich bekannt. Im Folgenden soll deshalb gezeigt werden, wie man Petrinetze für die Objektorientierung einsetzen kann. Wie ebenfalls allgemein bekannt ist, eignet sich die Objektorientierung aufgrund solcher Eigenschaften wie Datenkapselung, Vererbung, Polymorphie etc. besonders gut zur Modellierung komplexer Systeme.

Ziel ist es, diese Eigenschaften auf Petrinetze zu übertragen, da Petrinetze gegenüber objektorientierten Programmiersprachen nach [BM93] einige Vorteile haben, die dieses Vorgehen begründen.

- Sie haben einen mathematischen Hintergrund, der zwischen Grafik und algebraischer Präsentation eine eins zu eins Korrespondenz unterstützt.

- Jedes System kann mit Basiskonzepten beschrieben werden, d.h., aktiven und passiven Komponenten und deren Beziehungen zueinander.
- Der Grad der Abstraktion kann so gewählt werden, wie er benötigt wird.
- Die Netze können nach den Wünschen des Entwicklers oder der Anwendung verändert werden.
- Die Markierungen der Netze erlauben einen Blick auf die dynamischen Aspekte eines Systems.
- Die Modellierung von Nebenläufigkeit ist einfach, da Petrinetze für diesen Zweck entwickelt wurden (siehe auch Kapitel 3).

In [BM93], [BB95], [Lak95b] und in [Lak95a] findet man verschiedene Ansätze zur Beschreibung von Objekt-Petrinetzen. In dieser Arbeit wird der Formalismus von [BM93] vorgestellt.

Ein Objekt ist im allgemeinen ein eindeutig zu identifizierendes Gebilde, das einen Zustand besitzt und ein bestimmtes Verhalten zeigt. Dabei wird der Zustand eines Objektes durch Instanzvariablen und das Verhalten durch Methoden beschrieben.

Um ein Objekt als Petrinetz darstellen zu können, werden die Instanzvariablen auf Stellen abgebildet. D.h., die Stellen repräsentieren hier den Zustand eines Objektes. Der Typ der Variablen korrespondiert mit der Farbe der Stelle (siehe Kapitel 4.1). Das Verhalten eines Objektes wird durch das Netz in seiner Gesamtheit repräsentiert, wobei die statischen Komponenten des Netzes die statische Struktur des Objekts abbilden.

Wie in der Objektorientierung üblich, werden auch bei den Petrinetzen die Netze, die gleiches Verhalten und gleiche Struktur aufweisen, zu einer Klasse zusammengefasst. Die einzelnen Objekte werden deshalb durch identische Netze realisiert. Um eine Klasse zu erhalten, faltet man die identischen Objekt-Petrinetze zusammen. Die Klasse ist also eine Faltung der Objekt-Petrinetze. Um die Objekte weiterhin unterscheiden zu können, benutzt die Klasse verschiedene Farben für die einzelnen Objektidentitäten. Zusätzlich wird jeder Transition eine Stelle als Nebenbedingung zugeordnet, die alle Instanzen (Objekte) enthält (siehe Abbildung 4.2).

Objekte, die dynamisch erzeugt werden, werden als Untermengen der Markierung der Klasse dargestellt. Dadurch werden die dynamischen Komponenten eines objektorientierten Modells auf das Petrinetz übertragen.

Um eine Methode aufrufen zu können, muss eine Nachricht zu einem anderen Objekt geschickt werden. Damit ein Objekt in der Lage ist, Nachrichten zu empfangen und zu versenden, muss der Name des Empfängers bekannt sein. Die Kommunikation der Objekt-Petrinetze erfolgt via gefärbter Marken. Diese gefärbten Marken enthalten u.a. den Namen des Empfängers und den des Senders.

Mit dem Formalismus von [BM93] ist es nicht möglich, mit Petrinetzen Referenzen, so wie in objektorientierten Programmiersprachen, zu anderen Klassen zu repräsentieren. Man kann Klassen lediglich kopieren. Hat man eine Klasse

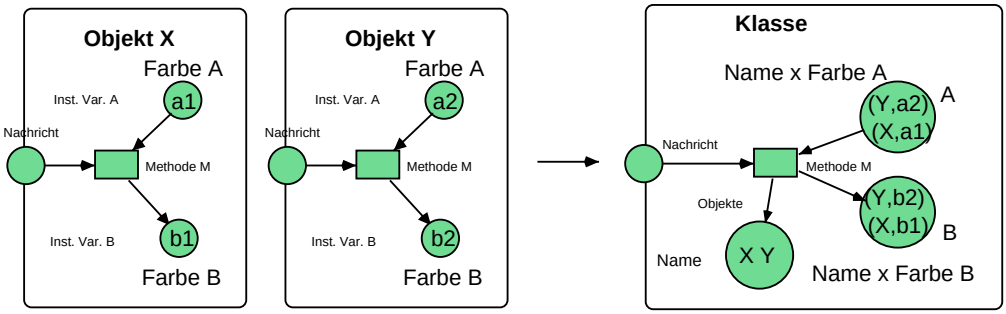

Abbildung 4.2: Objekt-Petrinetze, die zu einer Klasse zusammengefaltet werden.

X, und hat diese Klasse eine *is a - Beziehung* zu einer Klasse Y, dann zieht die Instantiierung eines Objektes einer Klasse X die Instantiierung eines Objektes einer Klasse Y nach sich. Das ist nötig, damit Nachrichten, die in X nicht definiert sind, an Y weitergeleitet, d.h. delegiert werden können. Diese Eigenschaft, Referenzen zu anderen Klassen bzw. Netzen zu unterstützen, ist ein wesentlicher Bestandteil der *Referenznetze*, die später in diesem Kapitel noch vorgestellt werden.

In diesem Abschnitt wurde gezeigt, dass die Objektorientierung auch in Petrinetzen wiederzufinden ist. Da die Objektorientierung für die Simulation sehr nützlich ist, wie im nächsten Kapitel noch ausführlich diskutiert wird, ist damit ein weiterer Grund gefunden, Petrinetze für die Simulation einzusetzen.

4.3 Netze in Netzen

Objektorientierte Modellierung heißt, dass Software als eine Menge von diskreten Objekten designt wird, die miteinander interagieren. Diese Interaktion macht das dynamische Verhalten von Objekten aus. Objekte der realen Welt werden modelliert, indem man die syntaktische Struktur, den momentanen Zustand, gewisse Attribute, benutzte Methoden, gespeicherte Daten, ausführende Prozeduren, Kommunikationsmechanismen und viele andere Funktionalitäten integriert (aus [Val00c]), wie auch in der Simulation üblich (siehe Kapitel 2).

Aus der Sicht eines Petrinetzes im *Netze in Netzen*-Formalismus werden dynamische Objekte jeweils als Netz modelliert, die Marken eines übergeordneten Petrinetz-Systems sind. In einem Petrinetz werden im Allgemeinen die Marken als der dynamische Teil angesehen. Um dem Petrinetz eine noch größere Dynamik und Ausdruckskraft zu verleihen, werden die Marken ebenfalls wieder als Netze dargestellt, wie in Abbildung 4.3 gezeigt. Man erhält dann die *Markennetze*, die auch als Objektnetze bezeichnet werden (nicht zu verwechseln mit den Objekt-Petrinetzen des vorangegangenen Abschnitts), und das übergeordnete Netz, das auch Systemnetz genannt wird. Ein System aus Systemnetz und Objektnetzen wird als elementares Objektsystem bezeichnet (siehe auch [Val00a, Val00b, Val00c, Val98]).

In [Val98] findet man eine Einführung in elementare Objektnetze. In [Val00c]

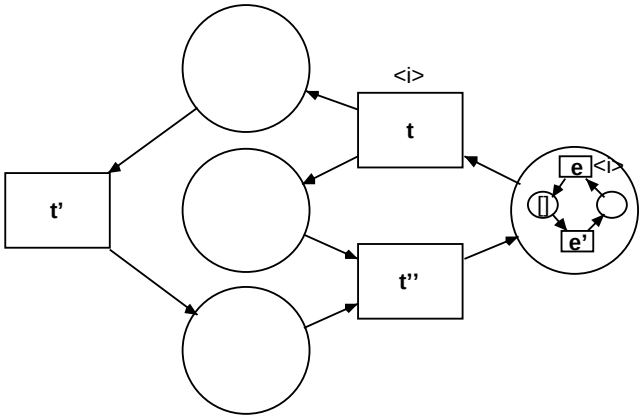

Abbildung 4.3: Ein Netz im Netz

sind elementare Objektsysteme wie folgt definiert:

Definition 10 *Ein elementares Objektsystem ist ein Tupel* $EOS = (SN, ON, \rho)$*, wobei*

- $SN = (P, T, W, M_0)$ *ein* EN-System *ist, genannt Systemnetz des* EOS*,*
- $ON = (B, E, F, m_0)$ *ein* EN-System *ist, genannt Objektnetz des* EOS *und*
- $\rho \subseteq T \times E$ *eine Interaktionsbeziehung ist.*

So wie alle Petrinetze können auch die Objektnetze ihre Markierung ändern. Die Veränderung der Objektnetzmarkierung kann unabhängig vom Systemnetz stattfinden, wie in Abbildung 4.4 zu sehen ist. So einen Schritt nennt man *autonomes Ereignis* der Transition. An der Markierung des Systemnetzes ändert sich dabei, von außen betrachtet, nichts. Wenn man in diese Stelle bzw. in die Marke, hineinschauen würde, könnte man sehen, dass sich etwas an der Markierung des Objektnetzes geändert hat. Es hat ein Schaltvorgang der Transition e' stattgefunden.

Es besteht im Objektsystem die Möglichkeit, dass eine Transition im Objektnetz, ausgelöst durch das Systemnetz, schaltet (siehe Abbildung 4.5). Bei so einem Ereignis heißt der Schritt *Interaktion*. Interaktion wird formalisiert als synchrone Veränderung einer Transition t des Systemnetz und einer Transition e des Objektnetzes [Val00c] mit $(t, e) \in \rho$ (in der Abbildung durch $< i >$ dargestellt). Diese Form der Synchronisation nennt man auch *rendezvous*-Synchronisation.

Eine weitere Möglichkeit der Änderung des Systemzustandes besteht darin, dass das Systemnetz schaltet, aber nicht das Objektnetz. So eine Änderung nennt man *Transport* (siehe Abbildung 4.6). Das Objektnetz wird im Systemnetz durch t' weiter transportiert, ohne dass das Objektnetz veranlasst wird zu schalten.

Im *Netze in Netzen*-Formalismus können sich Objektnetze an jeder Stelle des Systemnetzes befinden. Man unterscheidet aber zwischen Wertsemantik und Re-

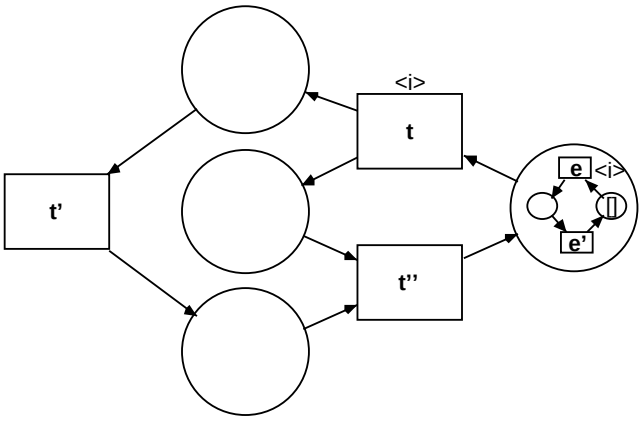

Abbildung 4.4: Ein autonomes Ereignis; das Objektnetz hat geschaltet (in Bezug auf Abb. 4.3)

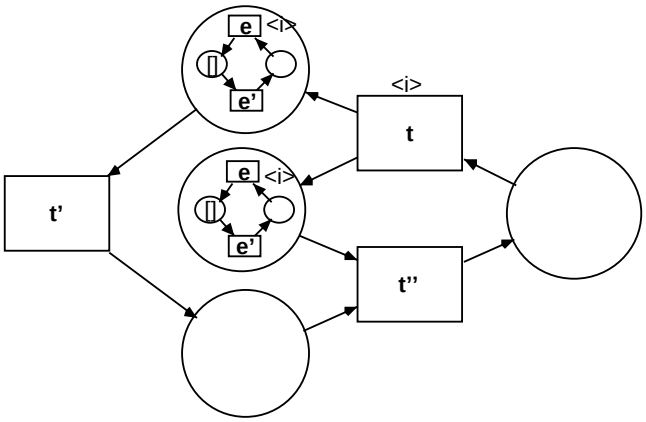

Abbildung 4.5: Eine Interaktion; Systemnetz und Objektnetz haben geschaltet (in Bezug auf Abb. 4.4)

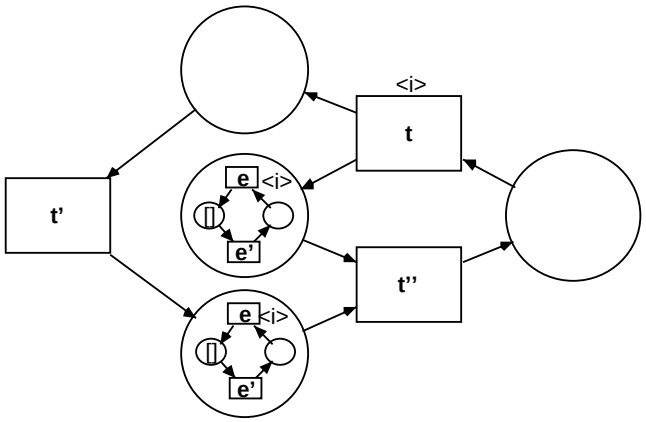

Abbildung 4.6: Ein Transport; das Systemnetz hat geschaltet (in Bezug auf Abb. 4.5)

ferenzsemantik. Der Unterschied besteht im Prinzip darin, dass bei der Referenzsemantik die Marken aus einer Referenz zu einem einzigen Objektnetz bestehen, während es sich bei der Wertsemantik um unabhängige Kopien handelt. Diese Kopien sind identisch in ihrer Netzstruktur, aber durchaus unterschiedlich in ihrer momentanen Markierung.

Die Ausdruckskraft der *Netze in Netzen* besteht zusammenfassend gesagt zum einen darin, dass auch die Marken Netze sind und sie damit ein bestimmtes Verhalten zeigen können. Zum anderen besteht in diesem Formalismus die Möglichkeit, tiefe Hierarchien zu bilden. Die Objektnetze können wiederum als Systemnetz ihrer eigenen Marken angesehen werden, die ebenfalls wieder Netze sein können [KR03]. Diese Hierarchien findet man auch bei der Modellierung von Systemen wieder, bei der ein Element auf einer niedrigeren Betrachtungsebene wieder ein System sein kann, und ein System auf einer höheren Betrachtungsebene ein Element (siehe Kapitel 2.1).

4.4 Referenznetze

Referenznetze beinhalten sowohl die Sicht der *gefärbten Netze*, der *Objekt-Petrinetze* als auch der *Netze in Netzen*. Sie sind von Olaf Kummer an der Universität Hamburg entwickelt worden (siehe auch [Kum02, Kum01, Kum99, Kum98]). In dieser Arbeit werden für die Simulation mit Petrinetzen immer Referenznetze eingesetzt, da sie, wie in diesem Abschnitt gezeigt wird, über alle bis hierhin vorgestellten Eigenschaften verfügen und zusätzlich noch weitere nützliche Eigenschaften aufweisen.

Für klassische Petrinetze existieren die Stellen und Transitionen, die gezeichnet wurden, exakt einmal während der Ausführung. Referenznetze verallgemeinern dies und erlauben mehrere Instanzen eines Netzes, damit verschiedene Instanzen eines Netzes ausgeführt werden können. Zusätzlich führt man Referenzen zu den Instanzen ein, bzw. Referenzsemantik (siehe auch Kapitel 4.3). Die Referenzen zu den Instanzen eines Netzes behandelt man so wie zuvor die Marken in gefärbten Netzen. Netze mit dieser Eigenschaft nennt man dann Referenznetze.

Die Ähnlichkeit zu *gefärbten Netzen* besteht darin, dass wie in den gefärbten Netzen Marken unterschieden werden und die Werte der Marken an Variablen gebunden werden. Damit eine Transition feuern kann, muss eine korrekte Kantenbeschriftung vorliegen (siehe Kapitel 4.1). Nur wenn in einer Stelle ein Wert liegt, und dieser Wert an die Variable gebunden ist, mit der die Kante beschriftet ist, kann eine Transition feuern. Bei den Referenznetzen ist das Verhalten ähnlich. Die Marken sind hier aber nicht ausschließlich Variablen mit zugeordneten Werten, sondern es sind auch Referenzen zu anderen Netzen möglich. Damit eine Transition feuern kann, muss auch im Referenznetz eine passende Beschriftungen an den Kanten gewährleistet sein. Liegt in einer Stelle eine Referenz zu einem bestimmten Netz, dann muss die Kante zur Transition auch mit Variablen dieses Netzes beschriftet sein, damit die Transition schalten kann.

Ein Referenznetz kann eine oder mehrere Instanzen eines Netzes erzeugen, indem eine Transition mit *new* und dem Namen des anderen Netzes beschriftet

wird. Damit die entstandene Referenz auch in einer Stelle gespeichert werden kann, muss die Beschriftung der Kante zu dieser Stelle an der Transition vor das *new* geschrieben werden (siehe Abbildung 4.7). Die Abbildungen in diesem Text stammen ursprünglich aus [Kum01] und sind für diese Arbeit leicht verändert worden.

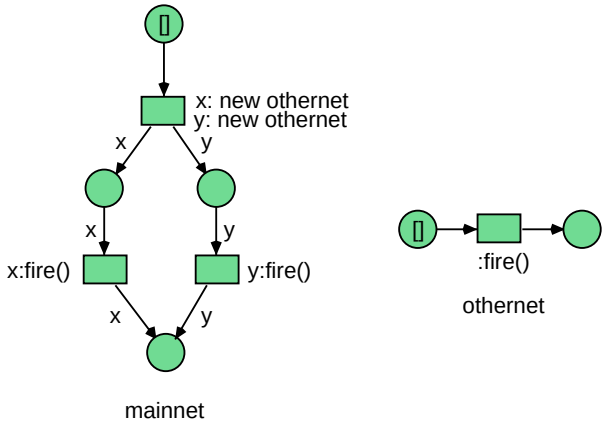

Abbildung 4.7: Ein Hauptnetz und ein anderes Netz

Die Ähnlichkeit zu den *Netzen in Netzen* besteht darin, das es sich bei den Referenznetzen im Prinzip um Netze in Netzen handelt, mit dem Unterschied, dass nicht die Objektnetze bzw. Objektnetzinstanzen selbst in einem Systemnetz als Marken liegen, sondern Referenzen zu den entsprechenden Objektnetzinstanzen. Mit Objektnetzinstanzen ist gemeint, dass es mehrere Instanzen eines Objektnetzes im Systemnetz erzeugt werden können. Darüber hinaus kann es verschiedene Objektnetze geben. Von jedem dieser Objektnetze können mehrere Instanzen existieren. Im Systemnetz können sich demzufolge Referenzen zu Instanzen unterschiedlicher Objektnetze befinden (siehe Abbildung 4.8).

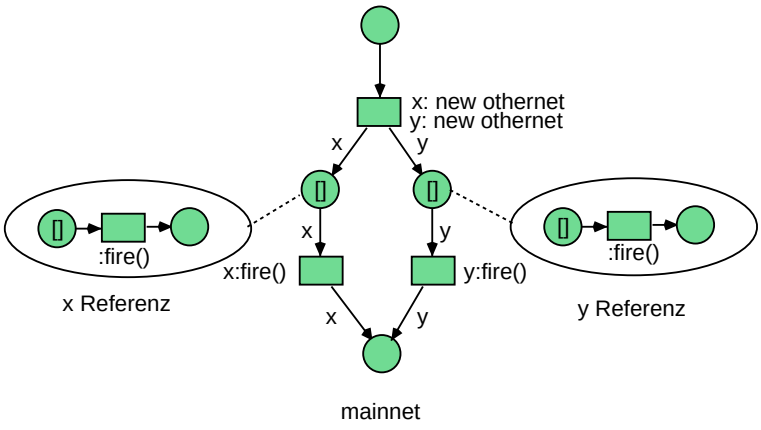

Abbildung 4.8: Im *mainnet* sind zwei Instanzen des *othernet* erzeugt worden.

Interaktionen drücken sich im Referenznetz nicht dadurch aus, dass eine Transition des Systemnetzes und eine Transition des Objektnetzes in einer Relation stehen, sondern das Schalten erfolgt über synchrone Kanäle. Synchrone Kanäle verbinden Transitionen dynamisch. Es können über synchrone Kanäle eine beliebig große Anzahl von Netzen miteinander synchronisiert werden, es muss lediglich gewährleistet sein, das jedes dieser Netze den entsprechenden Kanal besitzt. Geht man beispielsweise von zwei Netzen aus, muss ein synchroner Kanal in beiden Netzen vorhanden sein, d.h., die Transitionen müssen in beiden Netzen die gleiche Beschriftung aufweisen. In einem Netz kann dann an einer Transition das andere Netz durch seinen Namen und die Beschriftung der Transition aufgerufen werden. Solche Notationen nennt man deshalb synchrone Kanäle, da sie die Transition zwingen, synchron zu feuern und den Informationsfluss leiten. Vergibt man bei der Entwicklung sprechende Kanalnamen, kann man viele Synchronisationen gleichzeitig durchführen, ohne dass das Netz unübersichtlich wird (siehe Abbildung 4.9).

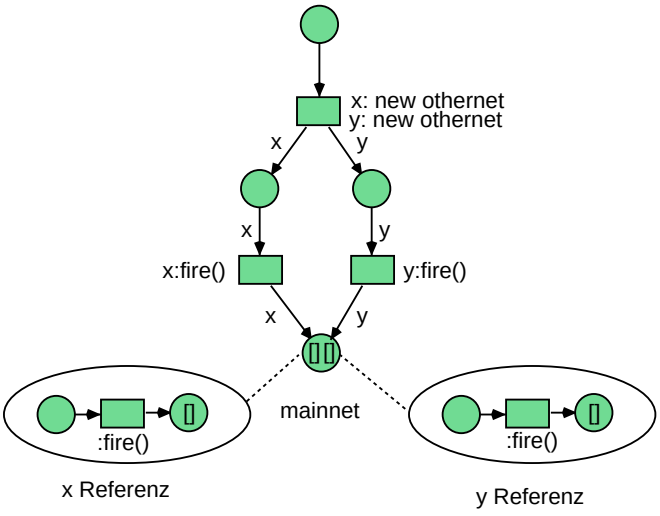

Abbildung 4.9: Über den synchronen Kanal *fire* haben die Transitionen in allen drei Instanzen geschaltet.

Möchte man die Vorteile eines höheren Netzes nutzen, wegen der Unübersichtlichkeit aber nicht alle Eigenschaften in einem einzigen Netz darstellen (wie in gefärbten Netzen üblich), kann man mehrere Netze zeichnen und sie über Kanäle miteinander synchronisieren.

Auch die Nähe zur Objektorientierung bleibt nach wie vor bestehen. Wie bekannt, haben Objekte eine eindeutige Objektidentität, eine bestimmte Menge an Zuständen, ein gewisses Verhalten und sie berücksichtigen das Geheimnisprinzip. Die Referenznetze unterstützen genau diese Eigenschaften auch. Es gibt im Referenznetzformalismus eindeutige Exemplaridentitäten. Jede Instanz eines Netzes hat eine eindeutige Identität, über die sie identifiziert werden kann. Die verschiedenen Zustände, die ein Objekt in der Objektorientierung annehmen kann, werden in Referenznetzen durch die Stellen beschrieben bzw. durch

die Markierung der Stellen. Jede Markierung im Netz zeigt einen bestimmten Zustand. Das Verhalten der Referenznetze wird durch die Transitionen beschrieben. Das Schalten einer Transition wird als aktive Phase gesehen, die das Referenznetz von einem Zustand in einen neuen überführt. Diese Eigenschaft korrespondiert zu den Methoden der Objektorientierung. Die bis zu dieser Stelle aufgezählten Eigenschaften werden zum Teil auch von anderen Formalismen unterstützt, beispielsweise vom *Objekt-Petrinetz-* oder vom *Netze in Netzen*-Formalismus. Das in der Objektorientierung bekannte Geheimnisprinzip (auch „information hiding“ genannt), das im nächsten Kapitel noch ausführlicher besprochen wird, unterstützen diese Formalismen aber nicht. Der Referenznetzformalismus unterstützt dies mit den synchronen Kanälen (siehe auch Kapitel 6 oder speziell Abschnitt 6.5).

Es wurde in diesem Abschnitt gezeigt, dass die Referenznetze eine Vielzahl von Eigenschaften besitzen, die sie zu einem annähernd universell einsetzbaren Werkzeug für die Beschreibung von Systemverhalten machen. Aus diesem Grund werden die Referenznetze in dieser Arbeit weiterhin eingesetzt, speziell für den Einsatz in der Simulation mit Petrinetzen.

4.5 Zeitbehaftete Petrinetze

Die zeitbehafteten Petrinetze spielen für die Simulation eine sehr wichtige Rolle, da ein mit Petrinetzen erstelltes Simulationsmodell auch zur Auswertung geeignet sein sollte. Wie schon oben erwähnt ist der Zeitverbrauch in der Simulation sehr wichtig, weshalb man ihn in der Modellierung unbedingt mit berücksichtigen sollte. Eine Zustandsänderung im Realsystem kann länger dauern als eine andere, was bei der Simulation mit Petrinetzen widergespiegelt werden muss, indem eine neue Markierung erst nach einer bestimmten Zeit erreicht werden kann. Für die Auswertung ist der Zeitverbrauch wichtig, da evtl. Durchsätze und Verweilzeiten benötigt werden, um Aussagen über das Systemverhalten machen zu können. In dieser Arbeit sollen Ansätze zur Modellierung zeitbehafteter Petrinetze vorgestellt werden.

Die bis jetzt betrachteten Petrinetze berücksichtigen keine Zeitangaben, d.h., aktivierte Transitionen feuern zeitlos und warten nicht, bis ein bestimmtes Zeitintervall vergangen ist. Es ist nicht möglich, mit Petrinetzen die Funktionalität eines Systems inklusive des zeitlichen Verhaltens zu analysieren. Die bis jetzt vorgestellten Petrinetze können daher nicht für die quantitative Analyse von Systemen eingesetzt werden.

Seit den siebziger Jahren gibt es einige veröffentlichte Anregungen zur Integration von zeitverbrauchenden Funktionen in Petrinetzen. In [BK96] werden zwei unterschiedliche Möglichkeiten zur Klassifizierung zeitbehafteter Petrinetze vorgestellt:

1. das Spezifizieren einer Verweilzeit der Marken in den entsprechenden Stellen und

2. das Spezifizieren einer Verzögerung beim Feuern der aktivierten Transitionen.

Der erste Fall sieht vor, dass Marken, die auf einer Stelle s liegen, für alle Ausgangstransitionen dieser Stelle s nicht zur Verfügung stehen, bis ein bestimmtes Zeitintervall vergangen ist. Nach diesem Zeitintervall können die Marken von den aktivierten Ausgangstransitionen verbraucht werden. Solche Petrinetze nennt man *Timed Places Petri Nets (TPPNs)*.

Im zweiten Fall stehen die Marken den aktivierten Transitionen sofort zur Verfügung, sie schalten aber erst, wenn ein bestimmtes Zeitintervall vergangen ist. Diesen Typ von Petrinetzen nennt man *Timed Transitions Petri Nets (TTPNs)*. Diese Petrinetze können in zwei Klassen eingeteilt werden:

1. *Preselection models* und
2. *Race models*.

Bei den *Preselection models* nimmt bzw. reserviert eine aktivierte Transition alle Marken, die sie zum Schalten benötigt, von ihren Eingangsstellen, so dass diese Marken für andere Transitionen des Netzes nicht mehr zur Verfügung stehen. Die Transition wartet mit dem Feuern solange, bis ein bestimmtes, vorher festgelegtes Zeitintervall vergangen ist. Danach feuert sie sofort und entfernt dabei die gesammelten bzw. reservierten Marken in den Eingangsstellen und erzeugt entsprechend den Schaltregeln neue Marken in ihren Ausgangsstellen.

Bei den *Race models* werden die Marken nicht von einer Transition reserviert. Wenn eine Transition aktiviert ist, wartet sie, bis das vorgegebene Zeitintervall vergangen ist und feuert dann. Es ist bei diesem Petrinetztyp durchaus möglich, dass eine Transition nicht mehr aktiviert ist, wenn ein Zeitintervall vergangen ist. Das liegt daran, dass die Marken nicht reserviert wurden und andere aktivierte Transitionen, die ein kürzeres Zeitintervall gewartet haben, die Marken verbrauchen können. In so einem Petrinetz können Transitionen andere Transitionen deaktivieren. Man kann sagen, dass die Transitionen um die Marken der Eingangsstellen wetteifern; man nennt sie deswegen *Race models*.

Die um Zeitangaben erweiterten Petrinetze TPPN und TTPN können weiter in deterministische oder stochastische Petrinetze klassifiziert werden. Die deterministischen heißen *Timed Petri Nets (TPN)*, die stochastischen *Stochastic Petri Nets (SPN)*. In *Timed Petri Nets* wird der Zeitverbrauch in das Netz an die Stellen oder Transitionen geschrieben. Da es sich um ein deterministisches Modell handelt, bleibt der Zeitverbrauch immer gleich, d.h., ist der Zeitverbrauch einmal festgelegt, dann ändert er sich nicht mehr. Es gibt in *Timed Petri Nets* keine Zufallsverteilungen. Bei den stochastischen Petrinetzen *SPN* ist das anders.

In [BK96] werden stochastische Petrinetze als ein Tupel $SN = (N, \Lambda)$ beschrieben, bestehend aus einem S/T-Ntz $N = (S, T, F, K, W, m_0)$ und einer Menge $\Lambda = \{\lambda_1, \ldots, \lambda_m\}$, wobei λ_i die markierungsabhängige Übergangsrate der Transition t_i ist.[1] Die Feuerungszeit ist exponentiell verteilt. Die Verteilung F_{χ_i} der Feuerungszeit einer Transition ist:

$$F_{\chi_i}(x) = 1 - e^{-\lambda_i x}$$

[1] In [Mar95] werden *Generalized Stochastic Petri Nets (GSPN)* vorgestellt, die neben anderen Eigenschaften auch eine Zeitkomponente beinhalten.

In Verbindung mit der Tatsache, dass die Möglichkeit der Zustandsänderung unabhängig von der Verweilzeit ist, heißt das, dass ein stochastisches Petrinetz einen Markovprozess beschreibt. Die quantitative Analyse eines stochastischen Petrinetzes kann durchgeführt werden, indem man den korrespondierenden Markovprozess analysiert. Markovketten eines stochastischen Petrinetzes können von dem Erreichbarkeitsgraphen des zugrundeliegenden S/T-Netzes erzeugt werden. Diese Eigenschaft der *Stochastic Petri Nets* soll an einem kleinen Beispiel aus [BK96] gezeigt werden.

In der Abbildung 4.10 sieht man ein gewöhnliches S/T-Netz. Es besteht aus fünf Stellen und fünf Transitionen. Die Transitionen t_1 und t_5 zeigen sequenzielles Schaltverhalten, die Transitionen t_2 und t_3 nebenläufiges. Zwischen den Transitionen t_4 und t_5 gibt es eine Konfliktsituation. Bei t_1 kommt es zu einer Gabelung, bei t_5 zu einer Vereinigung. Das Betrachten der Transitionen ist wichtig für die Konstruktion des Erreichbarkeitsgraphen.

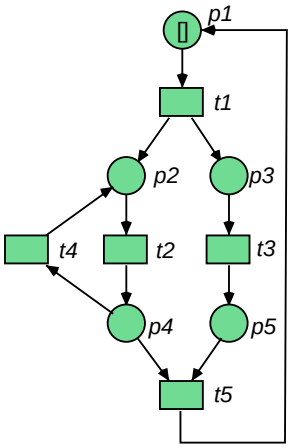

Abbildung 4.10: Ein S/T-Netz

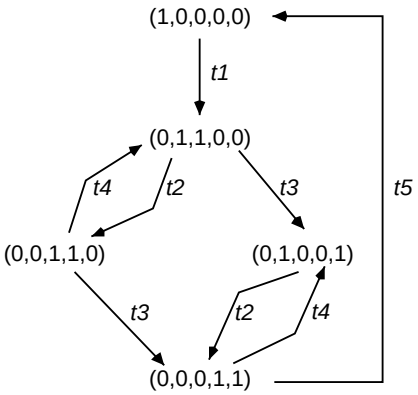

Abbildung 4.11: Der Erreichbarkeitsgraph zum Netz aus Abbildung 4.10

Zu diesem S/T-Netz gibt es den zugehörigen Erreichbarkeitsgraphen, wie

in Abbildung 4.11 zu sehen ist. Das Netz hat zu Beginn die Markierung $M_0 = (1, 0, 0, 0, 0)$ und t_1 ist aktiviert. Die Zeit, die vergeht, bis t_1 feuert, ist exponentiell verteilt, mit der Rate λ_1. Nachdem t_1 gefeuert hat, erhält man die Markierung $M_1 = (0, 1, 1, 0, 0)$. Hier sind t_2 und t_3 aktiviert, und nach einer gewissen Zeit wird eine der beiden Transitionen schalten. Man erreicht darauf die Markierung $M_2 = (0, 0, 1, 1, 0)$ oder $M_3 = (0, 1, 0, 0, 1)$. Je nachdem, welche Transition schneller ist. Die Wahrscheinlichkeit, das t_2 feuert ist:

$$P[t_2 \text{ schaltet zuerst in } M_1] = P[\chi_2 < \chi_3] = \frac{\lambda_2}{\lambda_2 + \lambda_3}$$

oder umgekehrt

$$P[t_3 \text{ schaltet zuerst in } M_1] = P[\chi_3 < \chi_2] = \frac{\lambda_3}{\lambda_3 + \lambda_2}$$

Diese Eigenschaft zeigt, dass die Änderung der Markierung M_1 in eine andere Markierung abhängig von der Zeit ist, die in M_1 vergangen ist. Die vergangene Zeit in M_1 ist gegeben durch das Minimum der unabhängigen exponentiell verteilten Schaltzeiten der beiden Transitionen, nämlich:

$$P[\min(\chi_2, \chi_3) \leq x] = 1 - e^{-(\lambda_2+\lambda_3)x}$$

Daraus folgt, dass die Zeit, die in M_1 vergeht, auch exponentiell verteilt ist, mit dem Parameter $(\lambda_2 + \lambda_3)$. In Kombination mit der Tatsache, dass die Möglichkeit, den Zustand zu wechseln, unabhängig von der vergangenen Zeit ist, heißt das, dass ein *SPN* einen Markovprozess beschreibt.

Die zugehörige Markovkette (isomorph zu Abbildung 4.11) ist in Abbildung 4.12 zu sehen.

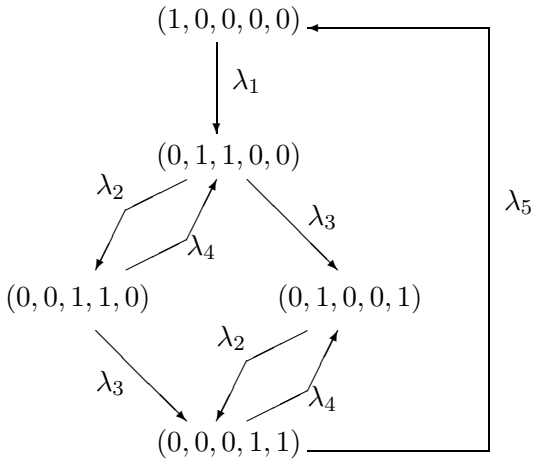

Abbildung 4.12: Die Markovkette zum Netz aus Abbildung 4.10

Um eine quantitative Analyse eines *SPNs* durchzuführen, reicht es, den korrespondierenden Markovprozess zu analysieren. Man muss dafür jede Markierung des Erreichbarkeitsgraphen als einen Zustand im Markovprozess darstellen und die Feuerungsrate λ_i der Transition t_i als Kanten hinzufügen. Für die Analyse einer Markovkette wird auf die Literatur verwiesen. Für die Analyse stochastischer Prozesse siehe z.B. [HLuK78], [Bor76], oder [Hüb96], speziell für *SPNs* siehe z.B. [BK96] oder [MBC+95], siehe auch Kapitel 8.

Dieser Abschnitt hat gezeigt, dass Zeitstempel für die Analyse von Systemen eine entscheidende Rolle spielen. Dies wurde an einem S/T-Netz gezeigt, es ist aber grundsätzlich möglich, jeden der in diesem Kapitel vorgestellten Formalismen zusätzlich mit einer Zeitkomponente auszustatten. Abschließend zu diesem Kapitel kann festgestellt werden, dass ein Referenznetz mit zusätzlichen Zeitfunktionen ein idealer Formalismus ist, um Systeme zu beschreiben und zu analysieren. Diese Eigenschaft wird in den Kapiteln 8 und 9 noch ausführlich diskutiert.

Fazit

In diesem Kapitel wurde auf erweiterte Petrinetzformalismen eingegangen. Diese erweiterten Petrinetzformalismen werden allgemein als höhere Petrinetze bezeichnet. Es gibt viele verschiedene höhere Petrinetze, hier wurden aber speziell die gefärbten Netze, die Objekt-Petrinetze, die Netze in Netzen, die Referenznetze und die zeitbehafteten Petrinetze diskutiert. Es hat sich herausgestellt, dass die höheren Petrinetze, speziell die Referenznetze einen geeigneten Formalismus bereitstellen, um Simulationsmodelle zu entwickeln.

5 Grundlagen des Petrinetzsimulators Renew

Renew ist ein in Java implementierter und an der Universität Hamburg von Olaf Kummer, Frank Wienberg und Michael Duvigneau entwickelter Petrinetzeditor und -simulator, der die vorgestellten Petrinetzformalismen unterstützt (siehe [KWD] und [KW03]).

Renew besitzt eine Toolbar, in der alle zur Verfügung stehenden grafischen Elemente mit einem Button dargestellt werden (siehe Abbildung 5.1). Um ein Netz zu zeichnen, wählt man mit der Maus ein Symbol aus und klickt es an, danach klickt man in den Editor und es erscheint exakt das ausgewählte Symbol. Auf diese Weise kann Schritt für Schritt durch einfache Mausklicks das gewünschte Netz entwickelt werden.

Abbildung 5.1: Die Renew Toolbar

Es stehen in Renew neben den gewöhnlichen Stellen, Transitionen und Pfeilen zusätzlich im Menü noch eine Menge anderer Funktionen zur Verfügung. Es ist möglich, unterschiedliche Arten von Kanten einzusetzen, um ein gewünschtes Verhalten zu realisieren. Darüber hinaus kann man Stellen und Transitionen mit einem Namen versehen, was für die Realisierung komplexer Netze nützlich sein kann, da Netze bei entsprechender Größe unübersichtlich werden können. Renew unterstützt alle im vorangegangenen Kapitel vorgestellten Netzformalismen. Um Referenznetze zu realisieren, besteht in Renew die Möglichkeit, Stellen- und Transitionsanschriften hinzuzufügen, um beispielsweise neue Netzinstanzen zu erzeugen. Zusätzlich besitzt Renew einen Simulator, mit dem das dynamische Verhalten der erstellten Netze simuliert werden kann. Der Netzsimulator ist nützlich, um u.a. die Richtigkeit des erstellten Netzes zu überprüfen.

Die Vorteile von Renew liegen darin, dass der Quelltext zugänglich ist und dadurch Erweiterungen jederzeit möglich sind. Es können beispielsweise spezielle Netzanschriften hinzugefügt oder sogar ganz neue Netzformalismen ergänzt werden. Darüber hinaus ist Renew in der Lage, Java-Klassen zu benutzen, d.h., aus den Netzen heraus können Java Objekte bzw. Methoden aufgerufen werden. Die Netze können demzufolge Java-Klassen benutzen. Diese Eigenschaft ist sehr

nützlich, da mit dem Zugriff auf Java-Klassen an Ausdrucksstärke hinzugewonnen wird. Daraus folgt, dass alle Möglichkeiten der Programmierung in Java auch in Renew zur Verfügung stehen. Da die in Renew unterstützten Referenznetze ebenfalls Java-Klassen sind, ist ein Aufrufen von Netzen in Java-Code ebenfalls möglich. Ein weiterer Vorteil ist, dass Renew in Java geschrieben, und daher auf annähernd allen Betriebssystemen lauffähig ist.

Neben den sehr speziellen Möglichkeiten, die Renew bietet, und die im Folgenden besprochen werden, ist mit Renew auch einfaches Zeichnen möglich. Man kann in Renew Graphiken oder Diagramme erstellen, die keinem Petrinetzformalismus gehorchen. Dafür stehen in der Toolbar verschiedene geometrische Formen und Textfelder zur Verfügung. Diese Zeichentools eignen sich u.a. für das Erstellen von UML-Diagrammen.

Man kann bereits sehen, das Renew sehr universell einsetzbar ist. Es ist erweiterbar, spezielle Netztypen werden unterstützt, es eignet sich zum Zeichnen und damit auch zum Modellieren und es besitzt einen Simulator, der eine Zeitkomponente beinhaltet. Die Zeitkomponente ist eine Kantenanschrift, die Transitionen verzögert schalten lassen kann. In den folgenden Abschnitten soll auf diese Eigenschaften eingegangen werden.

5.1 Modellieren mit Renew

In Renew ist das objektorientierte Modellieren möglich, da Netzinstanzen realisiert werden können. Man kann ein gezeichnetes Netz als Klasse ansehen, da Instanzen von dem gezeichneten Netz aufgerufen werden können, die man als Objekte ansehen kann. Die Syntax von Renew ist den objektorientierten Programmiersprachen dabei sehr ähnlich. Man ruft an einer Transition mit *new* und einer Variablen den Namen des Netzes auf, von dem eine neue Instanz erzeugt werden soll, z.B. *x:new othernet*, diese Instanz erscheint als Marke in dem aufrufenden Netz (siehe auch Kapitel 4.4).

Um die Möglichkeiten der Modellierung zu erhöhen, gibt es in Renew nicht nur einfache *Eingangs- (input)* und *Ausgangskanten (output arcs)* (siehe Abbildung 5.2), sondern es gibt zusätzliche Kantentypen, die ein spezielles Verhalten unterstützen. Dazu gehören *Reservierungskanten (reserve arcs)*, die an jedem Ende eine Pfeilspitze haben und eine entnommene Marke sofort wieder während des Schaltvorganges zurücklegen. *Testkanten (test arcs)* sind Kanten, die gar keine Pfeilspitze haben und nur testen, ob eine Marke prinzipiell zur Verfügung steht. In den Abbildungen 5.3 und 5.4 sieht man das gleiche Netz einmal mit Reservierungskanten und einmal mit Testkanten gezeichnet. Der Unterschied der beiden Kantentypen besteht darin, dass die Transitionen t und t' in dem Netz mit den Testkanten beide nebenläufig aktiviert sind und beide Transitionen schalten können. Im Netz mit den Reservierungskanten ist im Gegensatz dazu nur eine Transition t oder t' aktiviert.

Neben diesen gängigen und häufig benutzten Kantentypen gibt es zusätzlich noch die *flexible Kante (flexible arc)*, die eine unterschiedliche Anzahl von Marken transportieren können, *Löschkante (clear arcs)*, die alle Marken von einer Stelle nehmen und *Inhibitorkante (inhibitor arcs)*, die eingesetzt werden, um

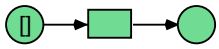

Abbildung 5.2: Ein Netz mit einer Eingangs- und einer Ausgangskante

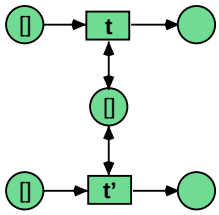

Abbildung 5.3: Ein Netz mit Reservierungskanten

boolesche Ausdrücke mit einfachen *anonymen Marken (black Tokens)* zu realisieren. Auf diese Kantentypen wird nicht näher eingegangen, da sie ein sehr spezielles Verhalten zeigen (siehe aber [KWD]).

In Renew können Kanten optional mit Anschriften versehen werden. Die Anschriften werden ausgewertet, wenn eine Transition feuert. Je nach Auswertung werden die Marken dann durch die Transitionen bewegt. Es können also nur Marken bewegt werden, die an die Variablen der Beschriftung gebunden sind.

Um beim Zeichnen ein Netz einfacher und besser strukturiert erscheinen zu lassen, gibt es in Renew die Möglichkeit *virtuelle Stellen* zu benutzen. Virtuelle Stellen repräsentieren eine bereits gezeichnete Stelle an einem anderen Platz. Sie werden eingesetzt, wenn ein Netz durch zu viele Kanten droht, unübersichtlich zu werden. Stellen können optional Stellentypen und eine unbestimmte Anzahl von Initialisierungsausdrücken bekommen. Die Initialisierungsausdrücke werden nach dem Eintragen sofort ausgewertet und auf der entsprechenden Stelle als Marke abgelegt. Eine Marke, ohne Substruktur, bezeichnet man als anonyme Marke (black Token). Anonyme Marken werden in Renew mit [] symbolisiert. Marken können beispielsweise Java-Werte, Referenzen, Werte selbstdefinierter Typen oder anonyme Marken sein. In Abbildung 5.5 sieht man, dass der ganz links stehenden Stelle ein Typ zugeordnet wurde. Es handelt sich um den Typ *int* (integer). Die Marken, die in dieser Stelle liegen, dürfen deshalb nur von diesem Typ sein, was in diesem Fall durch die Marke gewährleistet ist, die den Wert 42 besitzt.

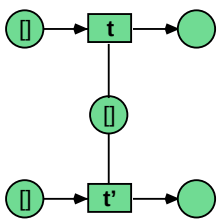

Abbildung 5.4: Ein Netz mit Testkanten

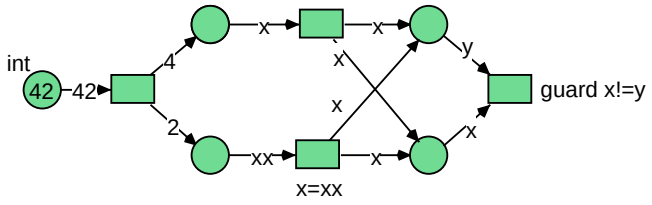

Abbildung 5.5: Ein Netz mit Stellentyp, Kantenanschriften, expression inscription und guard Funktion

Kantenanschriften sind einfache, Java-ähnliche Ausdrücke wie in Abbildung 5.5. Es können aber zusätzlich auch alle Java-Methoden benutzt werden. Auf diese Eigenschaft wird weiter unten noch genauer eingegangen.

Für Transitionen stellt Renew ebenfalls einige mögliche Anschriften, die Java-Code ähnlich sind, zur Verfügung. Diese Anschriften, die das Schaltverhalten beeinflussen und die Ausdruckskraft des Netzes stark erhöhen, können ganz einfache Ausdrücke sein oder auch Java-Methoden. Prinzipiell ist jeder Ausdruck möglich, wenn er ausgewertet werden kann. Die *expression inscriptions* sind einfache Anschriften, die ausgewertet werden, während der Netzsimulator nach einer Bindung der Transition sucht. *Guard inscriptions* sind Ausdrücke, die mit dem reservierten Wort *guard* versehen werden. Eine Transition kann feuern, wenn alle guard inscriptions nach der Auswertung ein boolesches *true* ergeben. In Abbildung 5.5 ist die untere mittlere Transition mit einer expression inscription versehen. Diese Transitionsanschrift bedeutet, dass x den gleichen Wert wie xx besitzt, woraufhin in den folgenden Stellen jeweils eine *2* abgelegt wird. Die ganz rechte Transition ist mit einer guard Funktion beschriftet. Diese Funktion stellt in diesem Fall sicher, dass x und y unterschiedlich sind. D.h., es kann nur zu $x = 4$ und $y = 2$ oder $x = 2$ und $y = 4$ ausgewertet werden. Es existieren zusätzlich noch weitere Transitionsanschriften, die etwas spezieller sind und im folgenden Abschnitt vorgestellt werden.

5.2 Referenznetze in Renew

In diesem Abschnitt soll darauf eingegangen werden, wie mit Renew Referenznetze erstellt werden und wie sie in Renew ablaufen. Was Referenznetze sind und wodurch sie sich von anderen höheren Netztypen unterscheiden, wurde bereits in Kapitel 4.4 und im vorangegangenen Abschnitt ausführlich diskutiert. Jetzt geht es darum, wie diese Netze in Renew realisiert werden und wie Renew mit diesen Netzen umgeht.

Renew unterstützt Referenznetze. D.h., es werden alle Eigenschaften unterstützt, die die Referenznetze auszeichnen. Dazu gehören vor allem die zur Kommunikation von Netzen verwendeten synchronen Kanäle.

In Anlehnung an die klassische objektorientierte Programmierung wird bei den Referenznetzen vorausgesetzt, dass der Initiator einer Synchronisation die anderen Netzinstanzen kennt. Die Transition, von der die Synchronisation ausgeht, benötigt eine bestimmte Beschriftung, die sogenannten *downlinks*. Ein

downlink macht eine Anfrage an das benannte untergeordnete Netz. Ein downlink besteht aus einem Namen, der zu einer Netzreferenz ausgewertet wird, einem Doppelpunkt, dem Namen des Kanals, einer öffnenden Klammer, einer Liste von Parametern und einer schließenden Klammer. Der Ausdruck *net:ch(1,2,3)* bedeutet, das sich die Transition mit dieser Beschriftung mit einer Transition *ch* in der Instanz *net* synchronisieren möchte und die Parameter *1, 2* und *3* übergeben möchte. Die Liste der Parameter ist optional, es kann auch ein leeres Klammerpaar angegeben werden. Als Gegenstück zum downlink muss es in der Netzinstanz den passenden *uplink* geben. Ein uplink bedient die Anfragen jedes Netzes und jeder Instanz. D.h., die Transition, die durch ein uplink gekennzeichnet ist und aufgerufen wird, braucht ihren Initiator nicht zu kennen. Der uplink muss deshalb keinen Instanznamen enthalten. Vom Doppelpunkt an unterscheidet sich der uplink nicht mehr vom downlink. Lediglich die Parameter vom uplink und vom downlink müssen zusammenpassen, beispielsweise könnte *:ch(x,y,z)* ein passender uplink für das Beispiel weiter oben sein. Kanäle sind bidirektional, d.h., die Parameterübergabe kann vom Instanznetz zum Hauptnetz und umgekehrt ablaufen, wenn die Parameter zusammenpassen. Die Parameter von *net:ch(1,2,x)* und *ch(1,y,3)* könnten übergeben werden und insgesamt zu den Parametern *1, 2* und *3* ausgewertet werden.

In Renew müssen dazu die zu synchronisierenden Transitionen mit den entsprechenden up- und downlinks beschriftet werden, um einen synchronen Kanal zu erzeugen. In Abbildung 5.7 synchronisiert sich das Netz *mainnet* mit dem Netz *othernet* über den synchronen Kanal *fire*. Damit ist gemeint, dass das aufrufende Netz an der beschrifteten Transition mit seinem Kanal *fire* eine Instanz von Typ *othernet* aufruft, die ebenfalls den Kanal *fire* besitzt und seine Transition schalten soll.

Wenn in einem Netz mehrere Transitionen mit dem gleichen Namen bzw. Kanal aktiviert sind, dann ist nicht festgelegt, welche Transition zuerst feuert. Es ist in Renew auch möglich, mit dem reservierten Wort *this* zu kennzeichnen, dass in der aktuellen Instanz eine Transition gefeuert werden soll (siehe Abbildung 5.6).

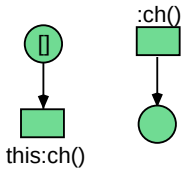

Abbildung 5.6: Ein Netz mit dem reservierten Wort *this*.

Ein ganz wesentlicher Mechanismus der Referenznetze in Renew sind die schon mehrfach erwähnten Netzinstanzen. Ein Netz, das im Editor gezeichnet wurde, hat zunächst eine statische Struktur. Wenn aber eine Netzsimulation gestartet wird, erzeugt der Simulator, ausgehend von der obersten Instanz, Netzinstanzen, die Markierungen haben, die sich im Verlauf der Zeit ändern können, und dadurch das dynamische Verhalten zeigen. Es ist in Renew möglich, mehrere In-

stanzen eines Netzes zu erzeugen. Die Instanzen haben ihre eigene Markierung und können unabhängig von anderen Instanzen feuern. Jedes Netz in Renew bekommt einen Namen. Dieser Name ergibt sich aus dem Namen, unter dem das Netz abgespeichert worden ist. Neue Netzinstanzen werden erzeugt, indem eine Transition eine *creation inscription* erhält. Die creation inscriptions enthalten eine Variable, einen Doppelpunkt, das reservierte Wort *new* und den Namen des Netzes, z.B. *x:new othernet*. Diese Bezeichnung bedeutet, dass x an eine neue Instanz des Netzes *othernet* gebunden wird (siehe Abbildung 5.7). Instanzen eines Netzes sind im Wesentlichen das Gleiche wie Objekte einer objektorientierten Programmiersprache, die eine Instanz einer Klasse sind. Daher behandelt man Netzinstanzen im Allgemeinen auch wie Objekte. Sie haben eine Identität wie Objekte und sie haben einen Zustand, der sich im Verlauf der Zeit ändern kann. Ihre Stellen können als Attribute angesehen werden, sie kapseln ihre Daten und sie können von anderen Netzinstanzen referenziert werden. Kanäle entsprechen Methoden, um eine Netzinstanz als ein Objekt ansprechen zu können, wie es aus objektorientierten Programmiersprachen bekannt ist. Es gibt in Renew dafür die Möglichkeit, ganze Java-Packages einzubinden, um diese Einschränkung auszugleichen.

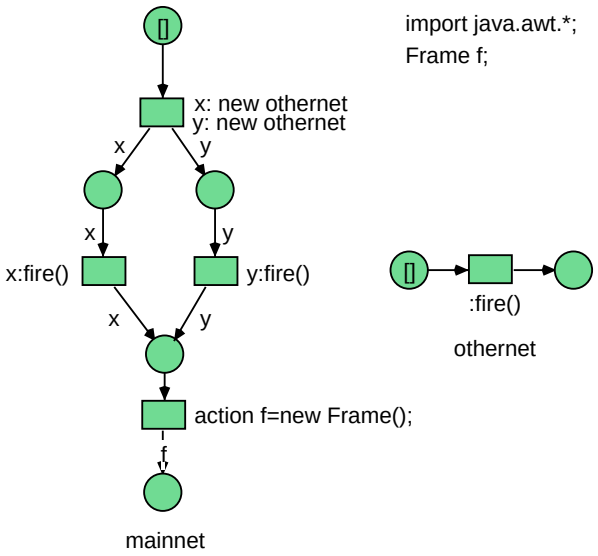

Abbildung 5.7: Ein Netz mit synchronem Kanal zu einem zweiten Netz, creation inscription, action inscription und eingebundenem Java-Package

Action inscriptions sind Anschriften, für die das reservierte Wort *action* vorgesehen ist und die im Gegensatz zu einfachen expression inscriptions garantiert nur einmal während des Feuerns der Transition ausgewertet werden. Einfache Ausdrücke können während der Bindungssuche beliebig häufig evaluiert werden. Immer dann, wenn auszuwertende Ausdrücke, z.B. Java Methoden, nur einmal ausgewertet werden sollen und es zu keinen Seiteneffekten kommen soll, eignet sich der Einsatz von action inscriptions (siehe Abbildung 5.7 *action f=new Frame();*).

Eine ausführliche Beschreibung der Referenznetze in Renew findet man in [KW03] und [KWD]. Es sollte hier nur ein kleiner Überblick über die Mächtigkeit der Referenznetze in Renew gegeben werden.

5.3 Simulation in Renew

Wie schon erwähnt, ist Renew ein Netzsimulator. Hat man ein Netz gezeichnet, ist es möglich, sich das dynamische Verhalten des Netzes anzuschauen, indem man es simuliert. Man kann dabei beobachten, wie die Marken durch das Netz wandern.

Es ist möglich, die Simulation zu starten und laufen zu lassen. Das Problem dabei ist, dass die Simulation so schnell voranschreitet, dass man nicht jeden Schritt verfolgen und das Netz nicht auf seine Richtigkeit überprüfen kann. Dennoch ist diese Methode sinnvoll, um beispielsweise *deadlocks* oder andere Fehler im Netz zu finden.

Um zu überprüfen, ob es im Netz einen semantischen Fehler gibt, ist es sinnvoll, die Simulation Schritt für Schritt durchzuführen. Dafür bietet Renew die Möglichkeit, mit Hilfe von *short cuts* oder der Maus die Simulation jeweils einen Schritt fortlaufen zu lassen, indem man durch ein Klick das nächste Feuern einleitet. Es ist in diesem Modus auch möglich, auf eine Stelle, auf der eine Marke liegt (eine Netzinstanz) doppelt zu klicken, um sich die in dieser Stelle liegende Netzinstanz in einem weiteren Fenster anzeigen zu lassen. Es ist prinzipiell möglich, sich jede Netzinstanz während der Simulation anzuschauen. Für jedes Netz wird dafür ein neues Fenster geöffnet, in dem die Simulation der entsprechenden Instanz abläuft. Diese speziellen Fenster, in der die Simulation läuft, können nicht manipuliert werden. Es sind keine Editorfenster.

Während eine Simulation läuft, protokolliert Renew mit, welche Transitionen gefeuert haben und welche Marken verbraucht und erzeugt wurden.

Zusätzlich gibt es in Renew die Möglichkeit, eine Transition verzögert schalten zu lassen. Eine Verzögerung erhält man, indem man an eine Kantenanschrift das Symbol @ und einen Ausdruck, der die Anzahl der verzögerten Zeitschritte angibt, hinzufügt. Der Ausdruck $x@t$ bedeutet beispielsweise, dass der Markenwert x erst nach t Zeiteinheiten bewegt wird. Diese Art eine Verzögerung zu realisieren gibt es in Renew nur für Ausgangskanten (vgl. Abbildung 5.8). Für Eingangskanten muss eine Verzögerung in einer action inscription angegeben werden.

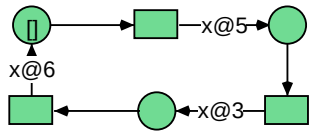

Abbildung 5.8: Ein Netz mit drei Schaltverzögerungen

Renew verwendet die *earliest time fireing rule*, d.h. Transitionen werden zum frühestmöglichen Aktivierungszeitpunkt geschaltet. Die Simulationszeit wird

nur vorgestellt, wenn zum aktuellen Zeitpunkt keine Transition aktiviert ist.

Im zeitbehafteten Modus zeigt Renew während der Simulation die vergangene Simulationszeit an. Terminiert eine Simulation, kann man die Simulationszeit ablesen. In Renew werden alle Transitionen, die zum gleichen Zeitpunkt aktiviert sind, auch zum gleichen Zeitpunkt gefeuert. Dieses Verhalten von Renew passt gut zur zeitdiskreten Simulation, in der ebenfalls alle Ereignisse, die zum gleichen Zeitpunkt stattfinden, gleichzeitig ausgeführt werden.

Alles in allem stellt der Renew Simulator sehr viele Funktionalitäten für die Simulation von Referenznetzen zur Verfügung. Lediglich eine Auswertungskomponente für die Simulation fehlt derzeit noch. Deshalb wird in Kapitel 9 eine Auswertungskomponente für Renew entwickelt, die für die Simulation eingesetzt werden kann.

Fazit

In diesem Kapitel wurde der Petrinetzeditor und -simulator Renew vorgestellt. Renew verfügt sowohl über Symbole zum Zeichnen von Netzen als auch zum Zeichnen anderer geometrischer Formen. Zusätzlich hat Renew noch verschiedene Kantentypen integriert, mit denen vielfältige Verhaltensweisen dargestellt werden können. Es wurde gezeigt wie Renew zur Modellierung von Simulationsszenarien eingesetzt werden kann, bzw. wie man Netze mit Renew zeichnet, die einem Szenario entsprechen. Renew unterstützt Referenznetze; deshalb wurde explizit erklärt, wie sich Referenznetze mit Renew realisieren lassen. Im letzten Abschnitt wurde dann gezeigt, wie sich diese Netze mit Renew simulieren lassen. Renew besitzt einen integrierten Simulator, weshalb es sinnvoll ist Renew zur Simulation von Szenarien mit Referenznetzen einzusetzen. Außerdem besitzt Renew eine Komponente, die Transitionen verzögert schalten lassen kann und damit einen Zeitverbrauch realisiert. Die genannten Eigenschaften machen Renew zu einem nützlichen Werkzeug für die zeitdiskrete Simulation.

6 Objektorientierte Simulation

Da die Simulation Entitäten der realen Welt betrachtet, passt der objektorientierte Ansatz sehr gut zur Modellierung von Simulationsszenarien und damit auch zur zeitdiskreten Simulation. Als Beispiel dafür, dass die Simulation einen engen Bezug zur Objektorientierung hat, soll die Sprache Simula angeführt werden. Die Sprache Simula, die hauptsächlich aus Algol hervorgegangen ist, und die schon in den sechziger Jahren für den Einsatz in der Simulation entwickelt wurde, gilt als die erste objektorientierte Sprache, da sie als erste die Begriffe Klasse und Objekt beschrieb. In Simula konnten Komponenten eines Objektes jedoch nicht, wie in modernen objektorientierten Sprachen üblich, verborgen werden. Simula vermengte außerdem das Konzept des Objektes mit den davon unabhängigen Konzepten des Verweises und der Koroutine. Aber sie führte das wesentliche Konzept der Vererbung ein (zu den objektorientierten Konzepten siehe auch [Wat96, S. 231-249]). Eine Objektklasse konnte Operationen einer anderen Objektklasse erben. Man sieht ein, dass Simula keine objektorientierte Sprache in dem Sinne ist, wie man objektorientierte Sprachen heute versteht. Da sie aber die wesentlichen Eigenschaften solcher Sprachen geprägt hat, kann man durchaus behaupten, die Objektorientierung habe ihren Ursprung in der Simulation.

„Unter objektorientierter Simulation versteht man die Simulationsmodellierung eines Systems mit objektorientierten Entwurfs- und Implementierungsmethoden.“ ([RD98], zitiert nach [PLC00, S.19]).

In Kapitel 2.2 wurde schon die Modellbildung in der Simulation diskutiert. Jetzt geht es darum, einen Zusammenhang zwischen objektorientierter Simulation, objektorientierter Modellierung, objektorientierten Sprachen und Petrinetzen herzustellen. Es wird sich im Verlauf dieses Kapitels zeigen, dass alle vier Bereiche einen engen Bezug zueinander haben. Zuerst wird der Zusammenhang zwischen Simulation und Objektorientierung anhand der Systemmodellierung hergestellt. Danach wird allgemein das Konzept der objektorientierten Programmierung mit imperativen Programmiersprachen erklärt. Zum Schluss wird auf Petrinetze eingegangen und deren Nähe zur Objektorientierung. Diese Sichtweise soll mit Referenznetzen erläutert werden.

6.1 Objektorientierte Modellierung

Wenn man die Modellierung eines Simulationsmodells betrachtet, wird die Nähe zur Objektorientierung deutlich. In der Modellierung versucht man, relevante Entitäten eines Realsystems auf ein Modell abzubilden, d.h. im Modell wird jede abzubildende Entität als Objekt dargestellt (siehe auch Kapitel 2).

Bildet man ein System der realen Welt auf ein Modell ab, müssen dabei verschiedene Sachverhalte berücksichtigt werden, die in verschiedenen Diagramm-

typen festgehalten werden können. Jedes Diagramm ist eine Abstraktion des Realsystems und hat seinen eigenen Schwerpunkt. Es wird entweder die statische Struktur, die dynamische Struktur, die Kommunikation mit anderen Objekten oder die Nebenläufigkeit abgebildet. Alle Diagrammtypen zusammen ergeben dann eine Abbildung des Realsystems. Diese Vorgehensweise, einzelne Entitäten herauszuarbeiten und deren Verhalten zu beschreiben, ist eine wesentliche Aufgabe in der Simulation und wird als objektorientierte Modellierung bezeichnet.

Im Folgenden soll ein Einblick gegeben werden, wie man bei der objektorientierten Modellierung vorgeht.

Als erster Schritt müssen die Struktur und das Verhalten der einzelnen Entitäten erfasst werden. Dabei muss bedacht werden, dass ein Modell immer eine Abstraktion und eine Idealisierung des Realsystems darstellt. Daraus folgt, dass ein Diagramm, in dem die Informationen des Modells festgehalten werden, ebenfalls nur eine Abstraktion ist.

Die Struktur eines Systems lässt sich verhältnismäßig leicht erfassen, da die Entitäten des Realsystems auf das Modell im Allgemeinen eins zu eins übertragen werden können. Zum Festhalten der Entitäten eignen sich *UML-Klassendiagramme* (siehe auch [GBB00, S.99-102]). Jeder Entitätstyp wird als Klasse abgebildet. Der Name der Klasse ist der der Entität. Einige Klassen erben Eigenschaften übergeordneter Klassen, und einige benutzen Methoden anderer Klassen. Diese Beziehungen müssen im Klassendiagramm gezeigt werden. Hat man ein Klassendiagramm erstellt, hat man die statische Struktur des Systems erfasst.

Das Erfassen der dynamischen Struktur ist komplizierter. Klassendiagramme geben nur Auskunft über die Objekte an sich, welches Objekt sie erzeugt hat und welches Objekt welches andere benutzt. Sie verraten nichts über die Dynamik eines Systems, da sie das Zusammenwirken der einzelnen Objekte nicht zeigen. Um das Modell zu vervollständigen, müssen die Eigenschaften der Entitäten ergänzt werden. Man sollte deshalb jede Entität genau auf ihre Verhaltensweisen hin untersuchen und versuchen, daraus ein Aktivitätsdiagramm für die Entitäten zu erzeugen.

Aktivitätsdiagramme beschreiben Sequenzen, Alternativen und Parallelen von Abläufen. Im Aktivitätsdiagramm kann man ablesen, unter welchen Voraussetzungen sich eine Entität wie verhält. Jedes Objekt im Modell wird so durch seine Eigenschaften ergänzt (siehe auch [GBB00, S.69-83]). Man kann ein Aktivitätsdiagramm auch als Prozess auffassen, wodurch die Nähe der objektorientierten Modellierung zur Simulation deutlicher wird.

In dem Beispiel aus Kapitel 3.2 gibt es die Entitäten *LKW* und *Gabelstapler*. Um realitätsnah zu bleiben, bildet man im Simulationsmodell den LKW mit seinen Eigenschaften als ein Objekt ab und den Gabelstapler mit seinen Eigenschaften als ein anderes Objekt. Diese Vorgehensweise findet man in der objektorientierten Programmierung wieder, man nennt dieses Konzept Klassifikation.

Der Vollständigkeit halber muss noch die Interaktion der Entitäten modelliert werden. Es muss gezeigt werden, welche Entität von welcher Nachrichten empfängt und an welche sie Nachrichten verschickt. Dazu eignen sich *Sequenz-*

diagramme. Sequenzdiagramme zeigen den zeitlichen Ablauf der Interaktionen in einem Modell. Es werden nicht die gesamten Abläufe mit allen Verzweigungen und Parallelitäten dargestellt, sondern nur die Nachrichten, die zwischen den beteiligten Entitäten in zeitlicher Reihenfolge ausgetauscht werden (vgl. [GBB00, S.83-91]).

Abschließend kann man feststellen, dass jedes Diagramm einen ganz bestimmten Abstraktionsgrad liefert. Umso mehr versucht wird, Information in ein Diagrammtyp zu stecken, desto abstrakter wird das Diagramm. Für eine detailliertere Sicht ist es nötig, verschiedene Diagrammtypen einzusetzen, die zusammen einen Sachverhalt exakt darstellen können.

Durch die oben beschriebene Art der Modellierung wird es einfacher, ein System zu verstehen, da man durch die einzelnen Objekte eine Struktur erhält. Veränderungen an einem Objekt betreffen im Allgemeinen nur ein oder einige wenige andere Objekte und nicht das ganze Modell, wodurch es leichter zu verändern, bzw. beim Auftreten von Fehlern leichter zu korrigieren ist. Ein weiterer Vorteil dieser Art zu modellieren ist, dass die Modellentwicklung Schritt für Schritt vorgenommen werden kann. Zu Beginn reicht es aus, die Objekte zu benennen. Im Verlauf der Modellbildung können dann nach und nach die Eigenschaften, z.B. in Form von Funktionen, hinzugefügt werden.

Der objektorientierte Entwurf hat nach [Val98] folgende Vorteile:

- bei der Software-Entwicklung erhält man durch Objekte eine Verallgemeinerung,
- man hat einen Sprachen-unabhängigen Entwurf,
- man bekommt ein besseres Verständnis,
- man hat einen klaren Entwurf und
- man hat leicht wartbare Systeme.

„Das Objektkonzept hat sich als leicht verständlich und als extrem flexibel gezeigt. In vielen Fällen lassen sich durch Objekte Konzepte aus der realen Welt abbilden, so dass der Wunsch nach einem starken Bezug zur Wirklichkeit erfüllt wird“ [Kum02, S.20].

Dieser Satz zeigt deutlich die Nähe zur Simulation und begründet den Einsatz der objektorientierte Modellierung in diesem Gebiet.

6.2 Objektorientierte Programmiersprachen

Um ein objektorientiertes Simulationsmodell implementieren zu können, benötigt man eine geeignete Programmiersprache. Im Prinzip kann ein Simulationsmodell mit jeder Programmiersprache implementiert werden. Allerdings gibt es Sprachen, die besser geeignet sind als andere. Für ein objektorientiertes Simulationsmodell eignen sich die objektorientierten Programmiersprachen am besten. Das liegt daran, dass sie die gleichen Ausdrucksmittel benutzen, wie im vorangegangenen Abschnitt deutlich wurde. Wie ebenfalls schon beschrieben benutzte

die Sprache Simula bereits Objekte. Es gibt viele Sprachen und Klassenbibliotheken, die extra für die Simulation entwickelt wurden. Dazu gehören neben Simula u.a. auch Slam, GPSS I und II, Simscript, Gasp, Dynamo und Desmo. Diese Sprachen sollen in dieser Arbeit nicht weiter berücksichtigt werden, da sie oft speziell für bestimmte Arten von Simulationsszenarien geeignet sind (siehe aber [PLC00], [NBBC66], [Fis73] und [Pri84]; in [PBH+88] wird die Simulation mit modernen objektorientierten Sprachen eingeführt). In dieser Arbeit soll es um einen allgemeinen Ansatz der objektorientierten Implementierung gehen. Es werden in diesem Abschnitt die typischen Konzepte der objektorientierten Programmierung, wie Klassifikation, Kapselung, Nachrichtenaustausch, Vererbung, Polymorphie, dynamisches Binden und Nebenläufigkeit erklärt. Ziel ist es, ein Verständnis für diese Art der Implementierung zu schaffen.

In der objektorientierten Programmierung ist ein Objekt nach [Bla97, S.420] wie ein Verbund aufgebaut. Es enthält Werte und Methoden. Um die inneren Details eines Objektes vor unbefugtem Zugriff zu schützen, werden Objekte meistens als Datenkapseln angesehen, die nur bestimmte Operationen nach außen zur Verfügung stellen, ihre Werte aber vor der Umwelt verbergen.

Eine Klasse ist nach [Bla97, S.421] eine Zusammenfassung gleichartiger Objekte. Klassen haben in objektorientierten Sprachen mehrere Aufgaben:

- sie definieren Datentypen,
- sie bilden Mengen, deren Elemente die Objekte sind und
- sie dienen als Mechanismus zur Erzeugung von Objekten.

In der objektorientierten Programmierung würde man für das Beispiel aus Kapitel 3.2 eine Klasse *LKW* und eine Klasse *Gabelstapler* bilden, die jeweils mit bestimmten Attributen und Methoden versehen würden. Aus diesen Klassen können dann die Objekte erzeugt werden. Es können pro Klasse mehrere Objekte erzeugt werden. Für das Beispiel bedeutet das, wenn das Realsystem z.B. mehrere Gabelstapler vorsieht, können aus der Klasse Gabelstapler mehrere Gabelstapler-Objekte erzeugt werden, die alle die gleichen Eigenschaften besitzen, sich aber in ihren Werten unterscheiden. Man nennt diese Objekte auch Instanzen.

Die Prozesse in einem Realsystem werden zu bestimmten Zeitpunkten oder zu bestimmten Ereignissen miteinander kommunizieren, um Informationen auszutauschen oder um sich zu synchronisieren. Im Simulationsmodell müssen der Austausch von Informationen und die Synchronisation ebenfalls modelliert werden. In der Objektorientierung gibt es dafür das Konzept des *Nachrichtenaustausches*. Objekte können Nachrichten an andere Objekte verschicken und selber Nachrichten von anderen Objekten empfangen. Dieser Nachrichtenaustausch geschieht mit Hilfe von Methoden, die bestimmte Attribute bzw. Werte von Attributen übergeben. Eine Nachricht kann verschiedene Wirkungen haben. Sie kann Eigenschaften eines Objektes abfragen, Objekte zu Aktionen veranlassen oder sie kann den Zustand eines Objektes verändern. Eine Nachricht ist dem Aufruf einer Zugriffsroutine einer Datenkapsel ähnlich, sie ist jedoch immer an ein Empfängerobjekt gerichtet (vgl. [Bla97, S.412]).

Wie schon erwähnt, beschäftigt sich die Objektorientierung mit Objekten, Klassen und Nachrichtenaustausch. Ein weiteres wichtiges und nützliches Instrument ist die *Vererbung*. Hat man ein objektorientiertes Modell, stellt man oft fest, dass sich viele Klassen nur in Details unterscheiden. Damit man nicht jede Klasse mit allen Eigenschaften neu programmieren muss, ist es nützlich, die Eigenschaften einer anderen Klasse, die gleich sind, an die neue Klasse, also eine untergeordnete Klasse, zu vererben. Es wird bei der Vererbung in objektorientierten Sprachen jedoch nicht der Code kopiert, sondern er wird in der untergeordneten Klasse mitbenutzt (vgl. [Bla97, S.423]). Wie oben schon gesagt, verfügte bereits Simula über dieses Konzept. Die beiden Klassen *LKW* und *Gabelstapler* aus dem Beispiel könnten beide von einer übergeordneten Klasse, auch Superklasse genannt, *Fahrzeug* erben. Es könnte eine Methode *fahre* vererbt werden. Die Klassen *LKW* und *Gabelstapler* sind sogenannte Spezialisierungen der übergeordneten Klasse *Fahrzeug*. Es entsteht durch die Vererbung letztendlich eine Klassenhierarchie.

Die *Polymorphie*, oder auch Vielgestaltigkeit, ist nach [Bla97, S.425-427] die Eigenschaft einer Variablen, für Objekte verschiedener Klassen stehen zu können. In typisierten Sprachen ist diese Möglichkeit durch eine Sprachregel so eingeschränkt, dass eine Variable eines Typs K nur Objekte der Klasse K oder davon abgeleiteter Klassen bezeichnen kann. Um sicherzustellen, dass Variablen immer zulässige Objekte bezeichnen, gibt es in typisierten objektorientierten Sprachen Kompatibilitätsregeln, mit denen ungültige Zuweisungen verhindert werden. Diese Sprachen bieten dadurch zur Laufzeit eine höhere Sicherheit als nicht typisierte Sprachen. Da in nicht typisierten objektorientierten Sprachen Variablen nicht mit Typen verknüpft werden, findet bei der Kompilierung auch keine Typüberprüfung statt. Daraus folgt, dass jede Variable im Prinzip jede Methode aufrufen kann. Wenn ein durch eine Variable aufgerufenes Objekt keine passende Methode besitzt, entsteht ein Laufzeitfehler. Mit untypisierten Sprachen hat man also die Freiheit, Kombinationen von Variablen und Objekten vorzunehmen. Das hat den Vorteil, dass die Ausdruckskraft der Sprache erhöht wird, und den Nachteil, dass man eine größere Unsicherheit bekommt.

Unter *Bindung* versteht man nach [Bla97, S.428] die Verknüpfung eines Methodenaufrufs mit dem aufgerufenen Code. In nicht objektorientierten Sprachen kann diese Verknüpfung statisch vom Compiler oder Binder vorgenommen werden, sodass sich während der Laufzeit die Bindung nicht mehr ändert. In objektorientierten Sprachen kann wegen der Polymorphie eine Variable für Objekte verschiedener Klassen stehen. Deshalb ist es sinnvoll, dass die Bindung dynamisch zur Laufzeit erfolgt.

Um Nebenläufigkeit in Programmiersprachen ausdrücken zu können, unterstützen objektorientierte Sprachen das Prinzip des Multithreatings. Threats sind sogenannte leichte Prozesse, die jeweils einen eigenen Stack besitzen, die aber alle auf dem gleichen Speicherbereich arbeiten und deshalb parallel bzw. nebenläufig ablaufen können (siehe auch [Tan95]). In allgemeinen imperativen Programmiersprachen wird dieses Konzept nicht unterstützt, hier können Prozesse nur sequenziell abgearbeitet werden. Das Konzept des Multithreatings macht die objektorientierten Sprachen zu einem geeigneten Werkzeug, Simulationsmodelle realitätsnah abzubilden.

Um ein Simulationsmodell mit einer objektorientierten Programmiersprache zu implementieren, ist es sinnvoll, die oben beschriebenen Konzepte zu kennen. Um effizient programmieren zu können, muss bekannt sein, welche Konzepte in der gewählten Programmiersprache wie unterstützt werden. Daraus folgt, dass das Implementieren eines Simulationsmodells mit objektorientierten Programmiersprachen das gute Beherrschen der gewählten Sprache voraussetzt. Dennoch ist die Implementierung von Simulationsmodellen mit objektorientierten Programmiersprachen weit verbreitet.

6.3 Objektorientierung und Petrinetze

Ein weiteres Mittel, um Entitäten der realen Welt zu implementieren, sind Petrinetze, speziell Objekt-Petrinetze (siehe auch [ACR00]) oder Referenznetze, da sie dem Modellierer adäquate Beschreibungsmittel liefern, wie in Kapitel 4 bereits erläutert.

Jede Entität im Realsystem kann durch ein Petrinetz beschrieben werden. Die Zustände und das Verhalten einer Entität werden im Netz mit Hilfe von Stellen und Transitionen ausgedrückt. Man kann auch sagen, um den Bezug zur Objektorientierung herzustellen, dass jedes Netz als ein Objekt angesehen werden kann.

„Sowohl Objekte als auch Petrinetze wollen mit möglichst einfachen, leicht verständlichen Mitteln ein Realsystem abbilden. Lediglich die Methoden sind unterschiedlich. Objekte benutzen Attribute und Methoden, Petrinetze benutzen Stellen und Transitionen. Außerdem sind die Grundeigenschaften der Objektorientierung und der Petrinetze so ähnlich, das eine Verbindung nicht mehr von der Hand zu weisen ist. Beide Techniken streben nach Einfachheit und Natürlichkeit, und sie verfolgen die gleichen Ziele, Beschreibung und Ausgestaltung von Berechnungsvorgängen“ [Kum02, S.20].

So wie auch Objekte in der Objektorientierung kommunizieren müssen, müssen auch die verschiedenen Netze kommunizieren. Petrinetze benutzen für den Nachrichtenaustausch das Konzept der Synchronisation. Dieses Konzept existiert für Referenznetze in Form von synchronen Kanälen, die das gemeinsame Schalten zweier oder auch mehrerer Transitionen erzwingen (siehe auch Kapitel 4.4).

Lediglich das Vererbungs- und das Subtypisierungskonzept sind für Referenznetze noch nicht realisiert. Diese Konzepte sind zwar angedacht, sie haben aber, da für Petrinetze auch Vererbung von Verhalten von Interesse ist, noch keinen endgültigen und zufrieden stellenden Status erlangt (vgl. [Kum02, S.22]).

Ein für die Simulation sehr nützliches Konzept, das Konzept der Nebenläufigkeit, ist, im Gegensatz zu den imperativen Programmiersprachen, ein grundlegendes Konzept der Petrinetze. Jedes Netz kann für sich nebenläufig zu anderen Netzen ausgeführt werden. Petrinetze sind geradezu prädestiniert für nebenläufiges Verhalten. In objektorientierten Programmiersprachen wird das Konzept der Nebenläufigkeit zwar unterstützt, die Programmierung kann aber sehr aufwändig und fehleranfällig sein. Erstellt man ein Petrinetz, modelliert man die Nebenläufigkeit automatisch mit. Die einzige Voraussetzung bei der Modellie-

rung ist, dass man ein geeignetes Werkzeug hat, um die gewünschten Netze mit ihren Eigenschaften zu erstellen.

Für die Realisierung von Simulationsmodellen mit Petrinetzen eignen sich die im vorangegangenen Kapitel vorgestellten Referenznetze besonders gut, da sie die Konzepte der Objektorientierung fast vollständig widerspiegeln.

Es ist deutlich geworden, dass auch Petrinetze typische Konzepte der Objektorientierung unterstützen, wie z.B. Kapselung, Nachrichtenaustausch und Nebenläufigkeit. Man spricht deshalb auch von objektorientierten Petrinetzen (siehe auch Kapitel 4.2).

6.4 Integration der Simulationsmodellierungsstile in Petrinetzen

Es soll hier gezeigt werden, wie sich die Modellierungsstile oder Sichtweisen der Simulation, die in Kapitel 2 vorgestellt wurden, auf Petrinetze übertragen lassen. Es werden nur der ereignisorientierte Ansatz und der prozessorientierte Ansatz betrachtet. Der transaktionsorientierte Ansatz hat nicht die Bedeutung wie die anderen genannten und soll hier deshalb nicht berücksichtigt werden.

Wie in Kapitel 2.3 schon gezeigt wurde, sind der ereignisorientierte und der prozessorientierte Ansatz zwei verschiedene Sichtweisen auf ein System. Man nennt den ereignisorientierten Ansatz auch *materialorientierte* Sicht, und den prozessorientierten Ansatz *maschinenorientierte* Sicht. Ab dieser Stelle wird nur noch von der material- und der maschinenorientierten Sicht gesprochen, da diese Namen zur Erklärung der Unterschiede der Sichtweisen geeigneter sind. Die Sichtweisen werden anhand eines Bedienungssystems erklärt. Es gibt Werkstücke und es gibt Maschinen. Die Werkstücke wandern durch die Maschinen und werden dort bearbeitet.

Bei der materialorientierten Sicht geht man von dem Inneren eines Systems aus. Man betrachtet die Ereignisse, die als Nächstes stattfinden. Es können Ereignisse nacheinander oder zeitgleich auftreten. Es könnte z.B. in einer Maschine A und in einer Maschine B jeweils ein Werkstück verändert werden. Diese beiden Ereignisse treten zeitgleich auf, d.h., in der Simulation wird die Simulationsuhr nur einmal weitergeschaltet, nicht zweimal. Die Ereignisse, die zu einem bestimmten Zeitpunkt stattfinden, müssen im System nicht miteinander in Beziehung stehen. Für diese Sichtweise müssen sie lediglich zum gleichen Zeitpunkt stattfinden. Danach kann ein Werkstück in einer Maschine C weiter bearbeitet, und dann in ein Lager gebracht werden. Die nächsten Ereignisse sind also „bearbeiten in einer Maschine C“ und danach, „Werkstück ins Lager bringen“. Hier wird die Simulationsuhr zweimal weitergeschaltet, da die Ereignisse nacheinander stattfinden. Das wesentliche Merkmal dieses Ansatzes ist, dass diese Sicht die Reihenfolge der Abläufe innerhalb eines Systems betrachtet. Es ist nur interessant, welche Ereignisse als Nächstes stattfinden.

Bei der maschinenorientierten Sicht geht man von Maschinen aus, die bestimmte „Aktionen“, die zu einem Prozess zusammengefasst werden können, ausführen. Es wird dabei jede Maschine für sich allein betrachtet. Es gibt, um beim vorherigen Beispiel zu bleiben, drei Prozesse, da es drei Maschinen gibt.

Diese Prozesse können sequenziell (Maschine A bearbeitet vor Maschine C ein Werkstück) oder nebenläufig (Maschine B bearbeitet ein Werkstück und Maschine C möglicherweise ein anderes) ablaufen. Man betrachtet bei dieser Sicht nur die Prozesse der einzelnen Maschinen. Die Liste der als Nächstes stattfindenden Ereignisse ist für diese Sichtweise nicht entscheidend, da jeder Prozess seine eigene Ablaufsteuerung besitzt.

Diese Sichtweisen sollen nun auf Petrinetze übertragen werden. Es wurde weiter oben schon angedeutet, dass man sich den Ablauf eines Petrinetzes auch als einen Prozess vorstellen kann. Das Verhalten einer Maschine kann deshalb durch ein Petrinetz dargestellt werden. Nimmt man eine Vogelperspektive ein, dann betrachtet man die drei Maschinen, nur diesmal jeweils als Petrinetz dargestellt. Die Marken in den einzelnen Netzen stellen die Werkstücke dar. Man sieht schnell ein, dass sich das maschinenorientierte Weltbild sehr leicht auf Petrinetze übertragen lässt.

Um die materialorientierte Sichtweise als Petrinetz darzustellen, sollten die Maschinen durch Petrinetze spezifiziert werden. Es eignen sich dazu die Referenznetz gut, da ihr Formalismus eine große Ausdruckskraft besitzt, die hier benötigt wird. Die Werkstücknetze sind hier die Marken in den Maschinennetzen, die wiederum als Petrinetz (hier als Referenznetz) dargestellt werden können. Zusätzlich muss es noch ein Umgebungsnetz geben, in dem die Maschinennetze die Marken sind. Die Ereignisliste ergibt sich jetzt aus den Transitionen, die als Nächstes schalten. Ein sequenzielles Schaltverhalten wird dadurch realisiert, dass in einer Transition im Umgebungsnetz immer die entsprechende Transition im Maschinennetz aufgerufen wird. Es schalten also je eine Transition im Maschinennetz und im Umgebungsnetz durch Synchronisation. Die Ereignisliste wird durch das Umgebungsnetz selbst repräsentiert. Sollten mehrere Ereignisse zeitgleich stattfinden, schalten auch mehrere Transitionen zeitgleich. Wenn z.B. Maschine A und Maschine B zeitgleich schalten sollen, werden in einer Transition im Umgebungsnetz die gleichnamigen Transitionen in den Maschinennetzen für Maschine A und Maschine B aufgerufen.

Man kann in den Petrinetzen die materialorientierte Sicht und die maschinenorientierte Sicht auch zusammengefasst betrachten. Denn die Maschinennetze bilden immer die Maschinenprozesse, und die Synchronisationen an den Transitionen finden auch in beiden Sichten statt. Die Simulation mit Referenznetzen ist also eine Zusammenfassung der material- und der maschinenorientierten Sicht, was die Referenznetze zu einem universell einsetzbaren Simulationswerkzeug macht.

6.5 Simulationsbeispiel

In diesem Abschnitt soll gezeigt werden, wie sich ein objektorientiertes Simulationsmodell mit Referenznetzen realisieren lässt. Es soll in diesem Beispiel wieder das LKW-Gabelstapler-Beispiel aus den vorangegangenen Kapiteln eingesetzt werden.

LKW fahren an einen Ort, an dem sie von Gabelstaplern entladen und wieder beladen werden. Danach fahren die LKW aus dem System hinaus und die Gabel-

stapler werden zurück in einen Wartepool geschickt. Der Ort an dem der Entladevorgang stattfindet, wird durch ein Kontroll-Netz dargestellt. Dieses Kontroll-Netz verwaltet zusätzlich den Ablauf des Vorgangs. Die LKW und die Gabelstapler werden jeweils durch Netzinstanzen des LKW- bzw. des Gabelstapler-Netzes im Kontroll-Netz dargestellt.

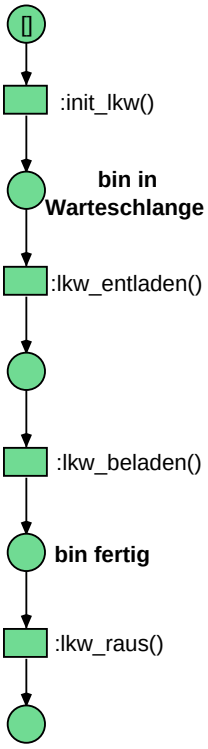

Abbildung 6.1: Das LKW-Netz

In Abbildung 6.3 ist das Kontroll-Netz dargestellt. Man sieht, dass sowohl die Aktivitäten der LKW als auch die der Gabelstapler abgebildet sind. Abbildung 6.1 und Abbildung 6.2 zeigen das LKW- und das Gabelstapler-Netz. Die LKW und die Gabelstapler sind im Kontroll-Netz als Marken dargestellt. Diese Marken sind, wie schon oben erwähnt, Netzinstanzen. Allerdings erst dann, wenn die Transitionen mit den Kanälen *lkw:new lkw2* bzw. *gabelstapler:new gabelstapler2* im Kontroll-Netz geschaltet haben (siehe Abbildung 6.3). An diesen Transitionen werden die Marken an einen bestimmten Typ gebunden; hier an einen LKW bzw. einen Gabelstapler. An den folgenden Transitionen *lkw:init_lkw()* und *gabelstapler:init_gabelstapler()* werden die LKW und Gabelstapler initialisiert. Diese Transitionen synchronisieren sich mit den gleichnamigen Transitionen aus den Netzinstanzen über synchrone Kanäle. Erst wenn die Transitionen im LKW-Netz und im Kontroll-Netz aktiviert sind, können sie schalten. Für das Gabelstapler-Netz gilt das analog.

Da der Ort, an dem ein Ladevorgang stattfindet, im Allgemeinen nur über eine begrenzte Kapazität verfügt, muss dies auch im Modell berücksichtigt werden. Dazu verwendet man Warteschlangen, die über eine bestimmte Kapazität

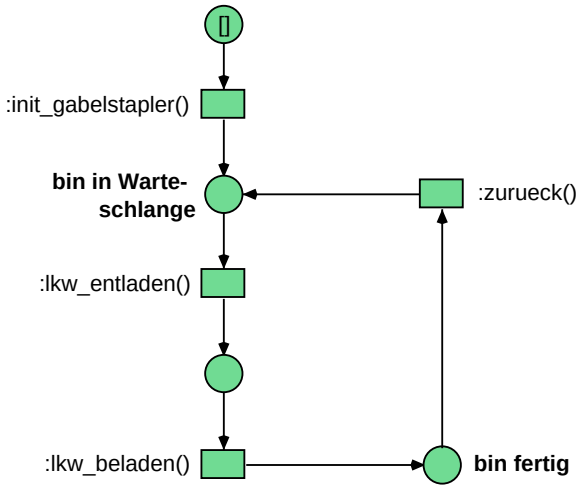

Abbildung 6.2: Das Gabelstapler-Netz

verfügen. In einem Petrinetz kann man eine Warteschlange durch eine einfache Stelle ausdrücken. Diese Stelle hat dann eine unendliche Kapazität, wie in der Warteschlange für die Gabelstapler zu sehen ist. Um eine Warteschlange mit einer endlichen Kapazität zu modellieren, benutzt man zusätzlich Komplementärstellen. Die LKW-Warteschlange im Beispiel hat die Kapazität drei: Zu jeder Stelle ist eine Komplementärstelle hinzugefügt worden, sodass die Stellen nur noch eine Kapazität von eins haben. Da es drei Stellen sind, ergibt sich insgesamt eine Kapazität von drei.

Jetzt folgen im Kontroll-Netz die Transitionen, an denen ein LKW von einem Gabelstapler entladen und danach wieder beladen werden soll. Diese Transitionen sind beschriftet mit *lkw:lkw_entladen()* bzw. *gabelstapler:lkw_entladen()*. Das heißt, dass es die Transition *lkw_entladen()* sowohl im Kontroll-Netz, im LKW-Netz als auch im Gabelstapler-Netz gibt und sich alle drei Netze an dieser Transition synchronisieren müssen. Es kann also erst entladen werden, wenn ein LKW angekommen ist und ein Gabelstapler bereit ist, diesen zu entladen. Für den Beladevorgang gilt das Gleiche wie für den Entladevorgang. Nach dem Beladevorgang trennen sich die Wege des LKWs und des Gabelstaplers. Der LKW verlässt das System und der Gabelstapler fährt zurück in seinen Wartepool, um für neue Aufträge bereitzustehen (siehe Abb. 6.3).

Die Beschriftungen an den Kanten zeigen an, welche Arten von Marken weitergegeben wird. Es sind auf der linken oberen Seite nur LKW, auf der rechten Seite nur Gabelstapler und zwischen den Entlade- und Beladevorgängen LKW und Gabelstapler. Das bedeutet, dass eine Marke, die an einen ganz bestimmten Typ gebunden ist, nur weitergeschaltet werden kann, wenn die Kante diesen Typ zulässt. Es können beispielsweise in den Wartepool der Gabelstapler niemals LKW gelangen, da die Kantenbeschriftung dies nicht zulässt.

Das @-Zeichen an den Kantenbeschriftungen steht für einen Zeitverbrauch. Zum Beispiel bedeutet *lkw@2*, dass zwei Zeiteinheiten vergehen müssen, bis die Transition die Marke wieder auf die nächste Stelle legt.

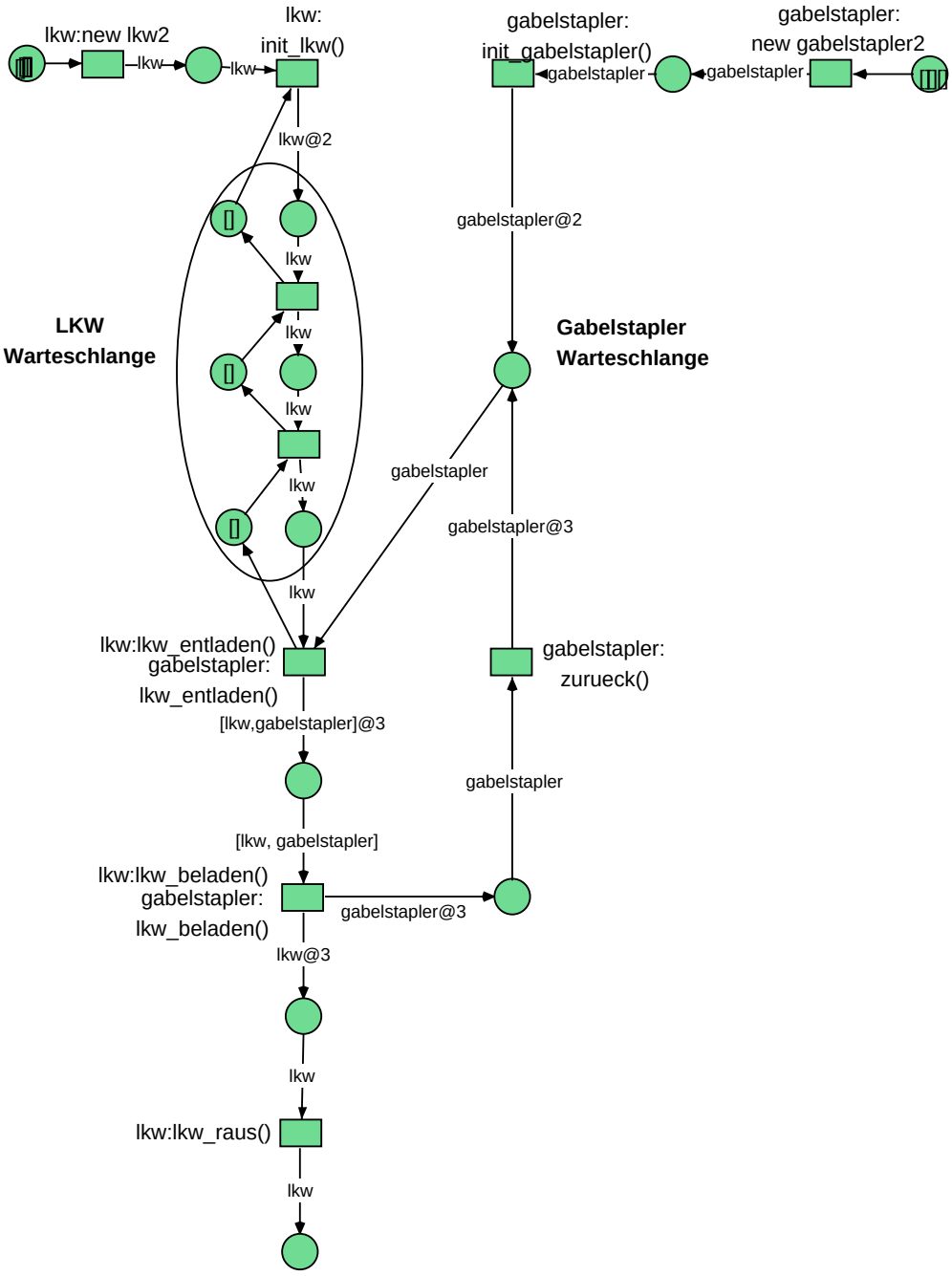

Abbildung 6.3: Das Kontroll-Netz

Schaut man sich die verwendeten Netze an, kann man leicht die Vorteile gegenüber imperativen Ansätzen erkennen. Die einzelnen Prozesse sind sofort zu erkennen und leicht zu verstehen. Synchronisationen sind einfach durch Beschriftungen der Transitionen zu erreichen. Es muss nicht kompliziert programmiert werden. Programmänderungen können durch Hinzufügen oder Weglassen von Stellen und Transitionen durchgeführt werden. Fehler lassen sich beim Ablauf des Netzes leicht finden, da man sehen kann, wenn ein Netz nicht mehr schaltet.

Zusammenfassend kann festgestellt werden, dass Simulationsmodelle, die mit Petrinetzen erstellt wurden, schnell verständlich und einfach zu handhaben sind.

Fazit

In diesem Kapitel wurde gezeigt, welchen Bezug die Simulation zur Objektorientierung hat und welchen Bezug Petrinetze dazu haben. Es wurde zunächst die Objektorientierung im Allgemeinen erklärt. Dabei wurde festgestellt, dass die Objektorientierung ihren Ursprung in der Simulation hat. Die erste Objektorientierte Sprache war die Sprache Simula. Später wurde gezeigt, dass auch Petrinetzmodelle einen engen Bezug zur Objektorientierung haben. Es wurde dann diskutiert, wie sich die Modellierungsstile der zeitdiskreten Simulation (ereignisorientierter und prozessorientierter Modellierungsstil) auf Petrinetze übertragen lassen. Am Ende des Kapitels wurde anhand eines Beispiels konkret gezeigt, wie ein objektorientiertes Simulationsmodell mit Petrinetzen aussieht.

7 Agentenorientierte Simulation

In diesem Kapitel sollen Agenten und ihr Bezug zur Simulation beschrieben werden. Es soll gezeigt werden, was Agenten sind und welches ihre wesentlichen Merkmale sind. Darüber hinaus soll gezeigt werden, wie man sie mit Petrinetzen realisiert und wie man sie dann zur Simulation einsetzen kann. Nach einer allgemeinen Einführung in das Thema Agenten wird in einer kurzen Einführung auf Multiagentensysteme eingegangen. Es soll gezeigt werden, wie sich ein Multiagentensystem mit Petrinetzen mit Hilfe des Petrinetzeditors Renew realisieren lässt (siehe dazu [Röl99], [MR00], [KMR01] und [KMR02]). An einem Beispiel wird gezeigt, wie sich ein bestimmtes Szenario als Petrinetzmultiagentensystem erstellen und simulieren lässt. Als Ergebnis dieses Abschnitts soll gezeigt werden, dass mit Agenten und Petrinetzen zwei sehr gute Formalismen existieren, die sich, miteinander verbunden, als sehr gutes Simulationswerkzeug einsetzen lassen. Dieses Kapitel wird sich stark an dem Überblickswerk [Fer01] orientieren.

7.1 Was ist ein Agent?

Der Begriff Agent kommt ursprünglich aus dem Bereich der künstlichen Intelligenz. Hier ist es in erster Linie wichtig, dass der Agent intelligent war, bzw. logische Schlussfolgerungen ziehen konnte. Andere Eigenschaften spielen in der künstlichen Intelligenz nur eine untergeordnete Rolle. Ein Agent ist hier ein Expertensystem.

Im Gegensatz zur klassischen Künstlichen Intelligenz sind Agenten der Verteilten Künstlichen Intelligenz (VKI) nicht nur in der Lage, logische Schlussfolgerungen zu ziehen, sondern sie können auch in ihrer Umwelt agieren und kommunizieren. Das tun sie, indem sie Aktionen ausführen und Nachrichten senden, empfangen oder auf Nachrichten mit einer Aktion reagieren, sie also verarbeiten. Durch Aktionen verändern die Agenten ihre Umwelt, was gleichzeitig bedeutet, dass sie die Voraussetzungen für zukünftige Entscheidungen selbst beeinflussen. Agenten besitzen also eine gewisse Autonomie. Sie werden durch eine bestimmte Menge von Absichten gesteuert und sie haben einen gewissen Handlungsspielraum. Sie können Aktionen aber nicht an jeder Stelle ausführen, sondern ihr Handlungsspielraum beschränkt sich auf Aktionen in einer bestimmten ihnen bekannten Umwelt. Ein Agent muss seine Umwelt also kennen, um agieren zu können. In einer unbekannten Umgebung könnte er keinerlei Aktionen ausführen und nicht kommunizieren (vgl. [Fer01, Kap.1]).

Damit ein Agent tatsächlich agieren kann, muss gewährleistet sein, dass er auf Ressourcen zugreifen kann, wie zum Beispiel Energie, Speicher, die CPU und bestimmte Informationsquellen. Er muss demzufolge mit einer Menge von Verfügungsrechten und Wissen ausgestattet sein. Das ist wichtig, da ein Agent bei einer Aktion beispielsweise Informationen an andere Agenten weitergeben soll,

an die er vorher selber herangekommen sein muss. Damit er autonom handeln kann, sollte er zu jeder Zeit über die Ressourcen verfügen können, die er zur Bearbeitung benötigt. Er muss deshalb mit Zugriffsrechten für externe Elemente ausgestattet sein. Das ist der Grund warum man Agenten sowohl als offene als auch als geschlossene Systeme ansehen kann. Als offenes deshalb, da er, wie eben beschrieben, externe Elemente zum Überleben braucht (Speicher, CPU oder auch andere Agenten). Als geschlossenes System kann man einen Agenten ansehen, da, wie ebenfalls schon gesagt, die Austauschaktionen mit der Umwelt nur begrenzt möglich sind (begrenzte Kenntnis der Umwelt, nur bestimmte Aktionen können ausgeführt werden). Es fehlt ihnen der Gesamtüberblick über ihre Umwelt. Sie kennen nicht die Handlungsmöglichkeiten und das Wissen anderer Agenten (vgl. [Fer01, S.30]).

Um zu zeigen, was ein Agent in der VKI ist, wird eine Definition aus [Fer01, S.29] gegeben.

Definition 11 *Man kann einen Agenten als eine physische oder auch als eine virtuelle Entität ansehen,*

- *die selbstständig in einer Umwelt agieren kann,*
- *die direkt mit anderen Agenten kommunizieren kann,*
- *die durch eine Menge von Absichten angetrieben wird,*
- *die eigene Ressourcen besitzt,*
- *die fähig ist, ihre Umwelt wahrzunehmen,*
- *die nur eine partielle Repräsentation ihrer Umwelt besitzt,*
- *die bestimmte Fähigkeiten besitzt und Dienste offerieren kann,*
- *die sich ggf. selbst reproduzieren kann,*
- *deren Verhalten darauf ausgerichtet ist, ihre Ziele unter Berücksichtigung der ihr zur Verfügung stehenden Ressourcen und Fähigkeiten zu befriedigen und die dabei auf ihre Wahrnehmung, ihre internen Modelle und ihre Kommunikation mit anderen Agenten angewiesen ist.*

Wie schon oben erwähnt, ist die Kommunikation eine wesentliche Eigenschaft eines Agenten der VKI. Die Kommunikation besteht im Wesentlichen aus dem Versenden von Nachrichten an andere Agenten und aus dem Empfangen von Nachrichten. Dem Versenden kann ein eigenständiges Generieren einer Nachricht vorausgegangen sein. Eine Nachricht kann einfach weitergeleitet werden oder der Agent kann mit einer Aktion auf eine Nachricht reagieren, sie also verarbeiten und dann eine Antwortnachricht verschicken. Ähnlich verhält es sich mit dem Empfangen von Nachrichten. Agenten können auf eine empfangene Nachricht mit einer Aktion reagieren oder sie können eine Nachricht lediglich zu Informationszwecken empfangen. Letztendlich geht es bei der Kommunikation zwischen Agenten immer um Informationsaustausch. Den Informationsaustausch bzw. das

wechselseitige Ausführen von Aktionen zweier oder mehrerer Agenten bezeichnet man als *Interaktion*. Eine Interaktionsbeziehung ist eine zusammengehörige Menge von Verhaltensweisen, die aus der Gruppierung von Agenten entstehen, die agieren müssen, um ihre Ziele zu erreichen, und dabei zugleich ihre mehr oder weniger begrenzten Ressourcen und Fähigkeiten zu beachten haben (vgl. [Fer01, Kap.2]). Das Konzept der Interaktion setzt nach [Fer01, S.84] voraus,

- dass es eine Menge von Agenten gibt, die miteinander agieren und/oder miteinander kommunizieren können,
- dass es Situationen gibt, in denen sich Agenten begegnen und zusammenarbeiten, gemeinsam Ressourcen benutzen und den Gruppenzusammenhalt regulieren,
- dass es dynamische Elemente gibt, die lokale und zeitlich begrenzte Beziehungen zwischen Agenten erlauben und
- dass es einen gewissen Spielraum gibt in den Beziehungen zwischen Agenten, d.h., sie sollen nicht nur in der Lage sein Beziehungen einzugehen, sondern sie sollen Beziehungen auch wieder auflösen können.

Diese aufgezählten Punkte sind entscheidend für die individuelle Autonomie eines Agenten.

Damit wird klar, dass ein Agent, um eingesetzt werden zu können, eine Mindestmenge an Eigenschaften aufweisen muss. [JW95] haben diese wesentlichen Eigenschaften zusammengefasst:

- Autonomie, damit der Agent eigenständig Aktionen durchführen bzw. Entscheidungen treffen kann.
- Sozialverhalten, damit er mit anderen Agenten oder u.U. Benutzern kommunizieren kann.
- Reaktivität, damit er seine Umwelt wahrnehmen kann und auf Veränderungen reagieren kann.
- Proaktivität, damit er Ziele verfolgen kann und damit auf seine Umwelt selbst einwirken kann.

Darüber hinaus sollen Agenten streng genommen noch weitere Eigenschaften wie Wissen, Glauben, Ziele, Verpflichtungen und Emotionen haben. Das wird deutlich, wenn man sich klar macht, dass ein Agent nur Aktionen aus einem bestimmten Grund heraus, also zu einem bestimmten Zweck, ausführt. Er verfolgt also ein Ziel, das er nur verfolgen kann, wenn er über ein bestimmtes Wissen verfügt. Je nach Sichtweise können diese Eigenschaften wiederum nach [JW95] zusätzlich um die folgenden Eigenschaften erweitert werden:

- Mobilität, damit sich der Agent in einem Netzwerk frei bewegen kann,
- Wahrhaftigkeit, damit der Agent niemals bewusst Unwahrheiten weitergibt,

- Wohlwollen, damit der Agent keine widersprüchlichen Ziele verfolgt und ihm übertragene Aufgaben tatsächlich ausgeführt werden und
- Rationalität, damit der Agent so handelt, dass seine Ziele auch wirklich ausgeführt werden können, solange sein Wissen und Glauben dies gewährleistet.

Bevor man einen Agenten modelliert, muss man sich über seine Eigenschaften im Klaren sein. Agenten sind häufig verteilte und mobile Objekte mit einem gewissen Grad an Autonomie. D.h., die minimale Anforderung an einen autonomen Agenten ist die Fähigkeit, asynchron zu arbeiten. Außerdem soll er in der Lage sein, mehrere Aufträge gleichzeitig zu verwalten oder zu bearbeiten. Er sollte z.B. während er Nachrichten verschickt, gleichzeitig noch in der Lage sein, weitere Nachrichten anzunehmen. Nebenläufiges Verhalten ist für einen Agenten demzufolge von großer Wichtigkeit, was bei der Modellierung berücksichtigt werden muss, da ein autonomer Agent selbstständig mit Daten, externen Anwendungen, speziellen Diensten und anderen Agenten interagieren können muss.

Die Intelligenz eines Agenten lässt sich modellieren, indem man die Benutzerwünsche als Vorgabe nimmt und das Verhalten des Agenten darauf hin optimiert. Des Weiteren soll der Agent sich an seiner Umwelt orientieren, d.h., das Verhalten anderer Agenten, die sich ebenfalls in seiner Umwelt befinden und mit denen er interagieren kann, muss in der Modellierung mit aufgenommen werden. Man kann den Grad der Intelligenz beeinflussen, indem man festlegt, ob ein Agent nur knapp vorgegebene Präferenzen als Mindestmaß erfüllen soll, oder ob es sich um einen lernfähigen Agenten handeln soll. Ein lernfähiger Agent ist ein Agent, der Schlüsse aus vorangegangenen Aktionen ziehen kann und sein Verhalten daraufhin ändert bzw. optimiert.

Eine Eigenschaft, die ebenfalls schon genannt wurde, ist die Mobilität. Statische Agenten sind nicht mobil, sondern an ihren Standort gebunden. Ein gewisser Grad an Mobilität kann durch dynamische Agentenerzeugung auf entfernten Standorten simuliert werden. Mobile Skripte können auf verschiedenen Rechnern gleichzeitig und unabhängig voneinander und von ihrem ursprünglichen Standort ausgeführt werden. Mobile Objekte sind auch zur Ausführungszeit mobil, können also mit samt ihrem augenblicklichen Datenzustand von einem Rechensystem zu einem anderen transportiert werden, wo sie ihre Ausführung fortsetzen (vgl. [Röl99]).

7.2 Multiagentensysteme

Im vorangegangenen Kapitel wurde beschrieben, was ein Agent ist und welche Eigenschaften er haben sollte. Es ist bisher noch nichts darüber gesagt worden, wie und warum Agenten eingesetzt werden.

Genau wie in der Objektorientierung, in der man nicht nur von einem Objekt ausgeht, sondern von einer Menge von Objekten, geht man auch bei Agentensystemen in der Regel nicht von nur einem Agenten aus, sondern von mehreren Agenten. Wie bei der Objektorientierung sollen in Agentensystemen bestimmte Aufgaben erfüllt werden, nur dass in diesem Fall an Stelle von Objekten Agenten

mit den schon beschriebenen Eigenschaften eingesetzt werden. Ein Agentensystem, in dem eine Menge von Agenten agiert, bezeichnet man allgemein als Multiagentensystem.

Der Begriff des Multiagentensystems kommt ursprünglich aus der Verteilten Künstlichen Intelligenz. Das Ziel eines Multiagentensystems ist es, möglichst effizient zu arbeiten, d.h., es sollen Aufgaben möglichst schnell und sinnvoll erledigt werden. Es ist deshalb wünschenswert, dass die eingesetzten Agenten bei der Erledigung einer Aufgabe zusammenarbeiten und sich gegenseitig unterstützen. Darüber hinaus sollen Aufgaben auch verteilt bearbeitet werden können, d.h., die Aufgaben sollen auf unterschiedlichen Rechnern erledigt werden können.

Um genau zu verstehen, was ein Multiagentensystem ist, wird im Folgenden wieder eine Definition aus [Fer01, S.31] gegeben.

Definition 12 *Ein Multiagentensystem ist ein System, das aus folgenden Elementen besteht:*

- *einer Umwelt E, wobei E ein Raum ist, der im Allgemeinen ein Volumen V hat,*
- *einer Menge von Objekten O, die situiert sind, was bedeutet, dass zu einem beliebigen Zeitpunkt jedem Objekt eine Position in E zugewiesen werden kann. Agenten können Objekte wahrnehmen, erzeugen und löschen,*
- *einer Menge von Agenten A, die die aktiven Objekte im System repräsentieren ($A \subseteq O$),*
- *einer Menge von Beziehungen R, die die Objekte miteinander verbinden,*
- *einer Menge von Operationen Op, mit denen Agenten Objekte empfangen, erzeugen, konsumieren, verändern und löschen können.*
- *Außerdem gibt es eine Menge von Operationen mit der Aufgabe, die Anwendungen dieser Operationen und die Reaktion der Umwelt auf die entsprechenden Veränderungsversuche darzustellen. Man bezeichnet diese Operationen als Gesetz des Universums.*

Zusätzlich zu dem eben beschriebenen Multiagentensystem gibt es noch Spezialfälle. Wenn zum Beispiel die Menge der Agenten $A = O$ und der Raum $E = \emptyset$ ist, dann handelt es sich um ein Netzwerk. Jeder Agent ist dann direkt mit einer Menge anderer Agenten verbunden, die als seine Bekannten bezeichnet werden. Sie werden als rein kommunizierende Multiagentensysteme bezeichnet, und kommen oft in der Künstlichen Intelligenz vor (aus [Fer01, S.31]).

Wie man durch die vorangegangene Beschreibung feststellen konnte, sind Multiagentensysteme sehr komplexe Systeme. Es stellt sich die Frage, als was man einen einzelnen Agenten im Multiagentensystem betrachten soll bzw. was er können soll. Ein Agent soll in Bezug auf ein gegebenes Problem definiert werden. D.h., es stellen sich die Fragen, welcher Bezug besteht zwischen Agent und Aufgabe, welche Aufgaben müssen insgesamt, um ein Problem lösen zu können,

ausgeführt werden, wie sollen die Aufgaben unter den Agenten aufgeteilt werden und wie groß ist nach der Aufteilung die Redundanz?

Die wichtigste Entscheidung, die beim Entwurf eines Multiagentensystems zu treffen ist, ist die Entscheidung, wie die nötigen Aufgaben unter den Agenten so aufgeteilt werden, dass das System möglichst effizient arbeitet. Es muss beim Entwurf deshalb geklärt werden, ob es besser ist, Agenten einzusetzen, die spezialisiert sind, d.h. Agenten, die nur eine Aufgabe beherrschen, die aber sehr gut bzw. effizient. Oder ob es nützlicher ist, eine Menge von „Alleskönnern“ einzusetzen. Setzt man Alleskönner ein, kommt es im System zu großer Redundanz. Setzt man Spezialisten ein, kann es zu Engpässen kommen (vgl. [Fer01, S.89 und S.130]).

Wenn man eine Fähigkeit eines Agenten a mit C_a bezeichnet, als C_p die benötigten Fähigkeiten zur Durchführung einer Aufgabe P und als C_A die Menge der Fähigkeiten aller Agenten A, dann kann die Aufgabe P ausgeführt werden, wenn $C_p \subseteq C_A$ ist. Alle zur Erfüllung der Aufgabe P nötigen Aufgaben müssen dann in A vorhanden sein. Ein Agent ist totipotent (Alleskönner), wenn er alle Fähigkeiten zur Ausführung von P besitzt. Mit C^P bezeichnet man alle Fähigkeiten, die der Agent a zur Ausführung von P benötigt $C_a^p = C_a \cap C^P$. Existiert kein totipotenter Agent, so müssen sich mehrere Agenten abstimmen (vgl. [Fer01, S.138]).

Die Grundlage für Interaktion zwischen Agenten bildet, wie im vorangegangenen Kapitel schon vorgestellt, die *Kommunikation*. Ohne Kommunikation ist ein Agent taub, stumm und blind gegenüber anderen Agenten und nur ein einzelnes isoliertes Element in einer Menge von Elementen. Erst durch die Kommunikation ist es den Agenten möglich zu kooperieren, Aktionen zu koordinieren und Aufgaben gemeinsam zu lösen. Wenn mehrere Agenten zusammenarbeiten, hat das jedoch zur Folge, dass einige zusätzliche Aufgaben bearbeitet werden müssen, die nicht direkt produktiv sind, sondern dazu dienen, die Zusammenarbeit zu verbessern. Diese zusätzlichen Aufgaben bilden einen Teil des organisatorischen Systems. Man nennt sie auch *Koordinationsaufgaben*. Koordination ist notwendig, sobald sich mehrere autonome Agenten, die jeweils ihre eigenen Ziele verfolgen, in einer gemeinsamen Umgebung aufhalten. Damit Aufgaben produktiv erfüllt werden können, müssen also zuerst eine Reihe von Koordinationsaufgaben bearbeitet werden. Man kann die Koordination von Aktionen auch als eine Menge zusätzlicher Aktivitäten beschreiben, die in einer Multiagentenumgebung ausgeführt werden müssen und die ein einzelner Agent, der dieselben Ziele verfolgt, nicht selbst ausführen könnte.

Kooperation bietet zwar nach [Fer01, Kap.2] Vorteile im Hinblick auf die quantitative Effizienz und das Entstehen von Qualität. Es ergeben sich dabei aber auch Fragen bezüglich der Verteilung von Aufgaben zwischen den Agenten. Diese Probleme sind schwerwiegend, da sie mehrere Parameter involvieren. Das sind u.a.:

- die kognitiven Fähigkeiten eines Agenten und seine Möglichkeit, Verpflichtungen einzugehen,
- die operativen Fähigkeiten des Einzelnen,

- die Art der Aufgaben,
- die Effizienz,
- die Kosten der Kommunikation und
- die sozialen Strukturen, innerhalb derer sich die Agenten bewegen.

Aus diesem Grund stellt die Verteilung von Aufgaben und Ressourcen eines der Hauptarbeitsgebiete im Bereich der Multiagentensysteme dar. Da in Multiagentensystemen die verteilten Zuweisungen von Aufgaben sowie die Konzepte von Vertrag und Verpflichtung im Vordergrund stehen, formulieren sie das Problem der Modellierung von Aktivitäten sowohl in einer sozialen als auch in einer computerspezifischen Sicht. Die Untersuchung der Verteilung von Aufgaben, Daten und Ressourcen dient dazu, folgende Fragen zu beantworten: Wer hat was zu tun und verbraucht welche Ressourcen, welche Ziele und Fähigkeiten hat der Agent und welche kontextspezifischen Bedingungen liegen dem zugrunde?

Damit aus einer Menge von Agenten ein funktionierendes bzw. kooperierendes Multiagentensystem wird, müssen mehrere Vorkehrungen getroffen werden. Diese Vorkehrungen betreffen Agenteneigenschaften, die weitgehend unabhängig vom inneren Aufbau der Agenten sind, wie Verwaltung und Interaktion. Zur Verwaltung gehören unter anderem Erzeugung und Vernichtung von Agenten, bei mobilen Agenten die Migration und beispielsweise Freigabe oder Verweigerung von Systemressourcen. Interaktion fasst die direkte Kommunikation und indirekte Einflussnahme zusammen. Indirekte Effekte treten beispielsweise bei gleichzeitigem Zugriff auf exklusive Ressourcen auf (vgl. [Fer01, S.87]).

Es muss noch geklärt werden, wie Agenten kommunizieren. Es gibt zwei Arten Kommunikation zu steuern: direkte Kommunikation zwischen den Agenten oder Kommunikation, die über Vermittler gesteuert wird (siehe auch [GK94]). Der erste Fall hat den Vorteil, dass kein extra Programm nötig ist. Allerdings besteht ein höherer Bandbreiten- und Speicherplatzbedarf. Dieser erhöhte Bedarf an Ressourcen macht in kleinen Systemen keine Probleme. Wird ein System aber größer, gibt es erhebliche Probleme bezüglich der Belastung des Übertragungsmediums. Das liegt daran, dass ein Agent möglicherweise Anfragen an jeden anderen Agenten im System stellen muss, wenn er den Agenten mit der richtigen Antwort nicht kennt, was zu einer großen Belastung führt. Um effizient zusammenarbeiten zu können, müssen die Aufgaben der Agenten sinnvoll untereinander aufgeteilt sein. Daraus folgt zwangsläufig, dass jeder Agent ein Bild eines jeden anderen Agenten gespeichert haben muss, was zu einem erhöhten Speicherplatzbedarf führt. Ein weiterer Nachteil besteht darin, dass jeder Agent sämtliche Gesprächs- und Verhaltensprotokolle beherrschen muss, um erfolgreich kommunizieren zu können.

Um diese in größeren Systemen auftretenden Nachteile zu vermeiden, kann man sogenannte Vermittlungsinstanzen einsetzen. Solche Systeme haben eine hierarchische Struktur, d.h., es findet keinerlei direkte Kommunikation zwischen den Agenten statt. Die Kommunikation wird immer von der Vermittlungsinstanz, die auch *Facilitator* genannt wird, ermöglicht (siehe [Gen97]). Diese Facilitator sind in der Lage, untereinander zu kommunizieren und ermöglichen

dadurch die Funktion des Multiagentensystems, ohne den Bedarf der Bandbreite zu stark zu erhöhen. Da sich die Agenten untereinander nicht kennen, sondern jeder nur seine Facilitator, wird auch zu großer Speicherplatzbedarf vermieden.

Die beiden Abschnitte 7.1 und 7.2 haben einen kurzen Überblick über die wesentlichen Charaktereigenschaften und Merkmale von Agenten und Multiagentensystemen gegeben. Zusammenfassend kann man sagen, dass der Einsatz von Multiagentensystemen zur Beschreibung eines Systems der Realität näher kommt als die Beschreibung eines Systems mit Objekten. Das liegt an den zusätzlichen Kommunikationsmöglichkeiten, der Autonomie und den anderen möglichen oben beschriebenen Eigenschaften, die Agenten im Gegensatz zu Objekten besitzen. Es ist leicht einzusehen, dass sich Multiagentensysteme aus diesem Grund auch hervorragend für den Einsatz in der Simulation eignen, da es in diesem Gebiet besonders wichtig ist, ein System möglichst realitätsnah abzubilden.

7.3 Agenten und Petrinetze

Neben den verwendeten Hochsprachen, wie zum Beispiel Java, mit denen Agenten häufig implementiert werden, gibt es auch die Möglichkeit, Petrinetze einzusetzen.

In diesem Kapitel soll gezeigt werden, wie sich Petrinetze für die Implementierung von Agenten und zur Erstellung einer Agentenarchitektur einsetzen lassen. Um die Petrinetzagenten zu beschreiben, orientiert sich dieser Text an [KMR02], [KMR01] und [MR00]. Die verwendeten Netzbeispiele entstammen ebenfalls diesen beiden Literaturangaben. Für die Implementierung dieser Beispiele, wurden die in Kapitel 4.4 eingeführten Referenznetze eingesetzt. Als Editor eignet sich der in Kapitel 5 vorgestellte Petrinetzeditor Renew, insbesondere mit dem Plug-In des Mulan-Viewers (siehe [Car03])

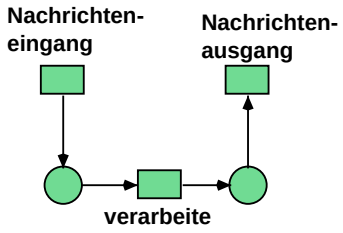

Abbildung 7.1: Ein vereinfachter Agent

In Abbildung 7.1 kann man einen als ein stark vergröbertes Referenznetz dargestellten Agenten sehen. Man sieht, dass dieses Referenznetz bzw. dieser Agent nur aus drei Transitionen und zwei Stellen besteht. Trotzdem kann dieser Agent noch als sinnvoll erachtet werden, da er die wesentlichen Merkmale eines Agenten erfüllt. Die beiden Transitionen *Nachrichteneingang* und *Nachrichtenausgang* bilden die Schnittstellen, über die der Agent Nachrichten empfängt und verschickt. Sie sind gleichzeitig die Schnittstellen, an denen der Agent nach

außen hin zu erkennen ist. Die Kommunikation an diesen beiden Transitionen kann bei Referenznetzen über synchrone Kanäle realisiert werden.

Die dritte Transition, die Transition *verarbeite*, ist eine starke Vergröberung, kann aber individuell, je nach Agent, verfeinert werden. In dieser Transition laufen alle Vorgänge ab, die in einem Agenten vorgehen können. Es können hier z.B. Nachrichten neu generiert oder verarbeitet werden. Der Agent entscheidet hier, ob er eine Konversation aufnimmt. Dazu benötigt er Protokolle, und zwar für jede Art von Konversation ein passendes. Protokolle sind Programme, die die Kommunikation steuern. Man kann auch sagen, eine Konversation ist ein aktiviertes Protokoll.

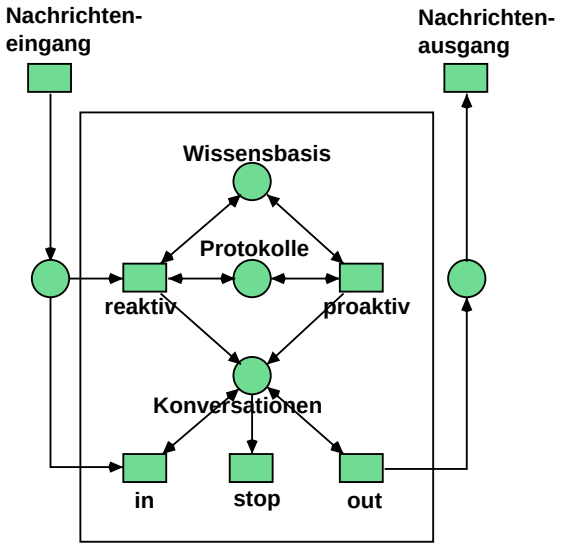

Abbildung 7.2: Der Agent, verfeinert

In Abbildung 7.2 sieht man eine Verfeinerung des ersten Netzagenten. Die Transition *verarbeite* ist um weitere Transitionen und Stellen verfeinert worden, die, wiederum vergröbert, die Aktivitäten des Agenten darstellen. Als wesentliche Aktivität ist hier die Auswahl des richtigen Protokolls zu nennen. Ein Agent verfügt über eine Menge von Protokollen, die er bei Bedarf aufrufen kann. Diese Protokolle sind Subnetze, d.h., jedes Protokoll ist ebenfalls als ein Referenznetz dargestellt. Die Kommunikation mit den Protokollen geschieht über synchrone Kanäle. Die Protokolle sind Bestandteil des Agenten und für andere Agenten von außen nicht sichtbar. Die Protokollauswahl lässt sich unterteilen in proaktive und reaktive Auswahl. Proaktiv heißt, dass der Agent von sich aus eine Konversation aufnehmen möchte, reaktiv heißt, dass er auf eine eingegangene Nachricht reagiert. Es gibt deshalb in der Abbildung die Transitionen *reaktiv* und *proaktiv*. Die Transition *reaktiv* kann, wie man sieht, nur schalten, wenn vorher auch eine Nachricht eingegangen ist. Die Transition *proaktiv* kann immer dann schalten, wenn der Agent über ein bestimmtes Wissen verfügt, welches das Generieren einer neuen Nachricht begründet. Dazu dient die *Wissensbasis*, die hier im Netz als Stelle dargestellt ist, aber ebenfalls wie die Protokolle als Subnetz ausgeführt

werden kann. Als Nebenbedingung ist noch das Vorhandensein eines geeigneten Protokolls gefordert. Aus Vereinfachungsgründen sind die Beschriftungen an der Transition weggelassen worden. Prinzipiell ist es aber so, dass Transitionen mit unterschiedlichen Bindungen (bzw. verschiedenen Beschriftungen an der Transition) beliebig häufig nebenläufig zu sich selbst schalten können. Dadurch wird im Netzagenten eine gewisse Autonomie ausgedrückt.

Jede Konversation hat eine eindeutige Identifikation. Bei einer Konversation können die Agenten auf eine Identifikation Bezug nehmen. Dadurch kann festgestellt werden, ob es sich um eine neue Nachricht handelt oder um eine Antwort auf eine vorangegangene. Erhält ein Agent eine Nachricht, die sich auf eine vorangegangene bezieht, dann schaltet die Transition *in*. Die Transition *in* reicht diese Nachrichten an das entsprechende Protokoll weiter. Es ist zu bedenken, dass sich dieses Protokoll gerade in Ausführung befindet. Möchte ein Agent eine Nachricht verschicken, geschieht dieses über die Transition *out* zur Transition *Nachrichtenausgang*. Es ist zu beachten, dass der Agent nicht direkt über synchrone Kanäle aus der Transition *out* Nachrichten verschicken kann. Der Agent hat nach außen nur die Transitionen *Nachrichteneingang und -ausgang* zur Verfügung.

Da schon viel über die Wichtigkeit der Protokolle gesagt wurde, soll jetzt auf die Protokollnetze etwas näher eingegangen werden. Dabei werden ein *Erzeuger*- und ein *Verbraucher*-Protokoll als Beispiel gegeben.

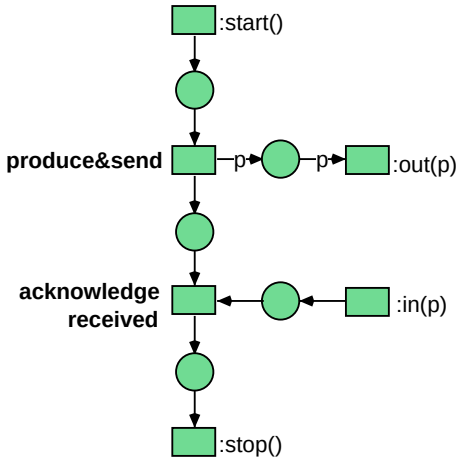

Abbildung 7.3: Das Erzeuger-Protokoll

Die Abbildung 7.3 zeigt ein Protokollnetz, das einen Erzeuger darstellt. D.h., es können Nachrichten erzeugt und verschickt werden, wie an den Transitionen *produce&send* und *:out(p)* zu erkennen ist. Das *(p)* ist die Nachricht bzw. die Identifikation der Nachricht. Es kann aber auch auf eingehende Nachrichten reagiert werden, wie an den Transitionen *:in(p)* und *acknowledge received* zu erkennen ist. Die Transitionen *:in(p)* und *:out(p)* sind über synchrone Kanäle mit dem Hauptnetz verbunden (siehe Transitionen *in* und *out* in Abbildung 7.2). Es müssen noch die Transitionen *:start()* und *:stop()* erwähnt werden. Die Transition *:start()* wird von der Transitionen *reaktiv* oder *proaktiv* des Netzagenten

über synchrone Kanäle aufgerufen. Zusätzlich können an dieser Transition für die Konversation wichtige Parameter übergeben werden. Ein Protokoll wird hier gestartet. Über die Transition *:stop()* kann ein Protokoll gelöscht werden.

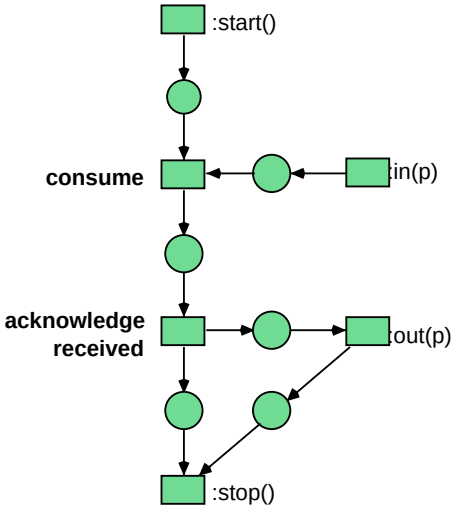

Abbildung 7.4: Das Verbraucher-Protokoll

Die Abbildung 7.4 zeigt ein Verbrauchernetz. Wie auch das Erzeugerprotokoll hat auch das Verbraucherprotokoll die schon besprochenen Transitionen *:start()* und *:stop()*. Des Weiteren besitzt der Verbraucher auch die Transitionen *:in(p)* und *:out(p)*. Das Verbraucherprotokoll kann nur im Hauptnetz von der Transition *reaktiv* aufgerufen werden, da eine Nachricht, die schon eingegangen ist, mit diesem Protokoll lediglich verarbeitet (verbraucht) wird. Dafür steht die Transition *consume*. Als Reaktion darauf kann er eine Nachricht mit der Transition *send acknowledge* über den Kanal *:out(p)* an das Hauptnetz übergeben, von wo aus die Nachricht dann an einen anderen Agenten verschickt werden kann.

Damit Agenten agieren können, benötigen sie eine Plattform, auf der sie erzeugt werden und die die Organisation der Agenten übernimmt. In Abbildung 7.5 ist ein Multiagentensystem abgebildet (links oben), das mehrere Agentenplattformen als Marken beinhaltet. Auf jeder Stelle liegt eine Agentenplattform. Zusätzlich gibt es die Transitionen *communication structure*, die für die Kommunikation von Agenten verschiedener Plattformen zuständig ist. Der „Zoom-Pfeil“ bedeutet, dass in eine Stelle hineingeguckt bzw. eine Marke angeschaut wird, um eine vereinfachte Agentenplattform zu betrachten (oben rechts). Die Plattform enthält eine zentrale Stelle und vier Transitionen. Die Transitionen *new* und *destroy* dienen dazu, einen Agenten zu erzeugen oder zu löschen. Die Transition *internal communication* wird für die Kommunikation von Agenten innerhalb einer Plattform benötigt. Die Kommunikation verläuft hier über synchrone Kanäle. Die Transition *external communication* ist für die Kommunikation von Agenten unterschiedlicher Plattformen vorgesehen. Hier wird zusätzlich die Transition *communication structure* des erst genannten Netzes benötigt. In der Stelle liegen alle jene Agenten als Marken, die zu

dieser Plattform gehören. Der nächste „Zoom-Pfeil“ zeigt ein Agentennetz (unten rechts), wie es schon besprochen wurde. In jedem Agenten liegen Protokolle (unten links), die für die verschiedenen Handlungen eines Agenten zuständig sind.

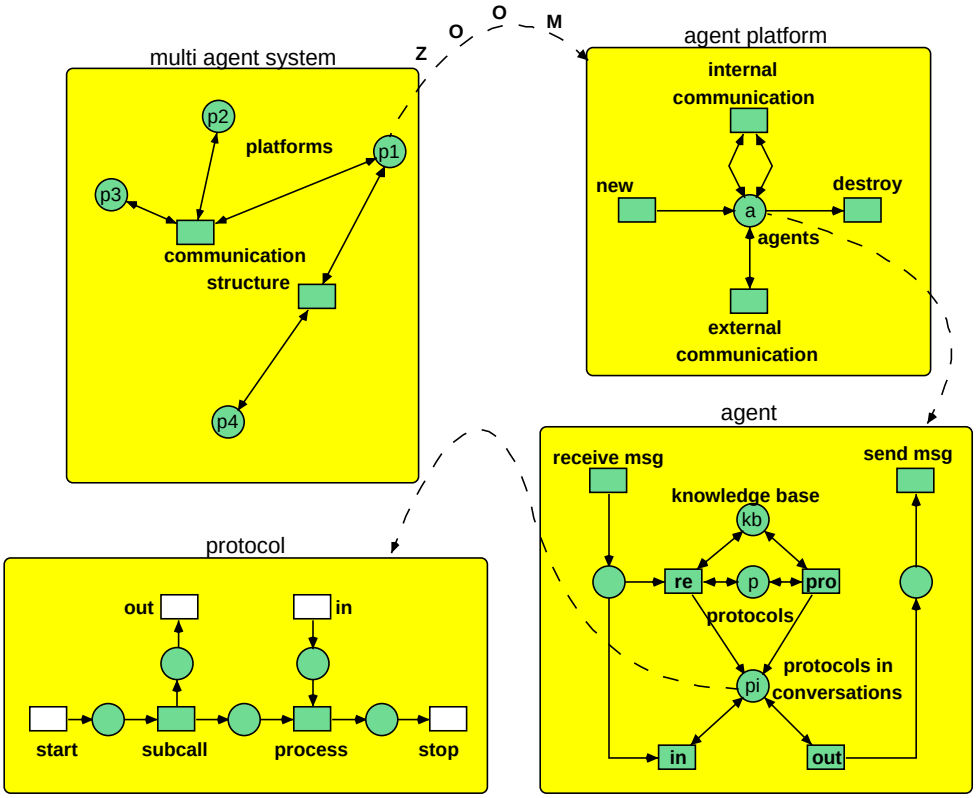

Abbildung 7.5: Ein Multiagentensystem, dass als Netze in Netzen dargestellt wird

Darüber hinaus benötigt man unter Umständen neben den vier besprochenen Netzen noch weitere Netze, um das Multiagentensystem zu vervollständigen. Damit ein Agent agieren kann, muss er über bestimmte Informationen verfügen. Diese Informationen, sein Wissen, wird in der *Wissensbasis* verwaltet. Die Wissensbasis kann ebenfalls ein Referenznetz sein, das, um an Informationen zu kommen, auf eine Datei zugreift, die speziell zu dem Agenten gehört, der eine Anfrage an die Wissensbasis gestellt hat. In dieser Datei sind alle Informationen gespeichert, über die ein Agent verfügen muss. Beim Entwickeln eines Agenten muss diese Datei ebenfalls angelegt und mit Informationen gefüllt werden. Damit ein Agent jederzeit die richtige Aktion ausführt, muss er auf das jeweils richtige Protokoll zugreifen. Die Protokolle, die Referenznetze sind, liegen in der Protokollfabrik, die ebenfalls ein Referenznetz ist. Die Protokollfabrik verwaltet die Protokolle.

Eine Umgebung für Petrinetzagenten wurde an der Universität Hamburg von Heiko Rölke im Rahmen seiner Dissertation entwickelt und Mulan (Multiagen-

tennetze) getauft. Diese Entwicklungsumgebung hat den Vorteil, dass der Entwickler lediglich die Daten für die Wissensbasis anlegen und die zu verwendenden Protokolle anpassen muss. Alle anderen Netze sind bereits entwickelt und können jederzeit wiederverwendet werden. Bei der vorgestellten Entwicklungsumgebung handelt es sich um eine vereinfachte Version.

7.4 Beispielszenario

Nachdem in den vorangegangenen Abschnitten gezeigt wurde, dass sich Petrinetze als Agenten einsetzen lassen, soll dies an einem Beispiel konkretisiert werden. Als Beispiel für die Agentensimulation soll das Szenario einer Spedition dienen. Eine Spedition besitzt mehrere LKW, die, wenn sie einen Auftrag übernehmen, quer durch Deutschland fahren. Das Ziel der Spedition ist es, den Gewinn zu maximieren. Es ist sinnvoll, die LKW immer beladen fahren zu lassen, da eine Leerfahrt Geld kostet und keinen finanziellen Gewinn einbringt. Dennoch müssen die LKW Leerfahrten machen, um an den Startort eines Auftrags zu gelangen. Es wird deshalb der LKW ausgewählt, der dem Startort am nächsten ist. Es besteht die Möglichkeit, dass der am dichtesten stehende LKW den Auftrag nicht übernehmen möchte, d.h., es muss ein anderer LKW ausgewählt werden, der die Fahrt übernimmt. Möglicherweise auch einer, der sehr weit weg ist, die Fahrt aber unbedingt übernehmen möchte. Möglicherweise ist auch kein LKW bereit, den Auftrag zu übernehmen, dann wird der Auftrag abgelehnt.

Hier sollen die LKW nun als Agenten dargestellt werden. Die LKW brauchen also Intelligenz, eine gewisse Autonomie und die Fähigkeit, mit anderen Agenten (hier LKW) kommunizieren zu können. Die Intelligenz benötigen sie, da sie selber wissen müssen, an welchem Ort sie sich gerade befinden und wie groß die Entfernung zwischen ihnen und einem neuen Auftragsstartort ist. Sie müssen deshalb als Information so etwas wie eine Landkarte bekommen. Autonomie haben sie dadurch, dass sie selber entscheiden müssen, ob sich die Übernahme eines Auftrags für sie lohnt, aus welchem Grund auch immer. Allgemein gilt, die Kosten sollten nicht größer sein als die Einnahmen durch einen Auftrag. Die LKW müssen kommunizieren können, da sie regeln müssen, wer den Auftrag letztendlich übernimmt. Dazu muss zuerst jeder LKW von dem Auftrag erfahren. Ein Auftrag kommt ins System und wird an jeden LKW bekannt gegeben. Jeder LKW rechnet aus, wie weit er vom Auftragsstartpunkt entfernt ist und entscheidet dann für sich, ob er den Auftrag übernehmen will. Diese Information wird an einen Entscheidungsagenten bzw. Entscheidungs-LKW weitergegeben, der den geeignetsten LKW heraussucht und ihm den Auftrag erteilt. Alternativ könnten die LKW auch alle gleichberechtigt sein und jeder LKW könnte jedem anderen seine Daten zuschicken. Diese Lösung hat allerdings den Nachteil, dass zu viele Nachrichten erzeugt werden und das System zu stark belastet wird. Deshalb wird ein LKW-Agent ausgewählt, an den alle Nachrichten geschickt werden und der dann den geeignetsten LKW heraussucht.

Dieses Beispiel zeigt kooperierende LKW. Es wird das gemeinsame Ziel verfolgt, der Spedition möglichst viel Gewinn zu verschaffen. Es ist alternativ auch möglich, ein Beispiel mit konkurrierenden LKW zu zeigen. Hier muss jeder ein-

zelne LKW seinen Gewinn maximieren. Jeder LKW möchte deshalb jede Fahrt, die ihm noch einen finanziellen Gewinn bringt, übernehmen. Da pro Auftrag durchaus mehrere LKW einen Gewinn erzielen könnten, stehen diese LKW in einem Konkurrenzverhältnis zueinander. Jeder der LKW kann ein Gebot abgeben, für welchen Preis er den Auftrag annimmt. Der billigste bekommt den Zuschlag. Für einen LKW besteht die Aufgabe darin, billiger zu sein als alle anderen, aber dennoch in der Gewinnzone zu bleiben.

Im Folgenden soll die erste Alternative des Beispiels wiedergegeben werden. Als erstes muss dafür ein Interaktionsdiagramm erstellt werden, damit klar wird, welche Nachrichten empfangen und welche verschickt werden müssen. In Abbildung 7.6 kann man die Interaktionen sehen, die ein LKW im System zu leisten hat. Gibt es n LKW im System, müssen alle n LKW informiert werden, wenn ein neuer Auftrag ins System kommt, es müssen also n Nachrichten verschickt werden. Danach muss jeder der n LKW seine Daten an den Entscheidungs-LKW verschicken. Da der Entscheidungs-LKW keine Daten zu verschicken braucht, werden weitere $n-1$ Nachrichten verschickt. Zum Schluss wird eine Nachricht an den ausgewählten LKW geschickt, um ihm mitzuteilen, dass er den Auftrag übernehmen soll. Es werden bei n LKW also nur $2n$ Nachrichten verschickt. Man kann überlegen, ob die anderen LKW benachrichtigt werden sollten, wenn sie einen Auftrag nicht erhalten. Dann wären es $3n-2$ Nachrichten, die insgesamt verschickt werden würden.

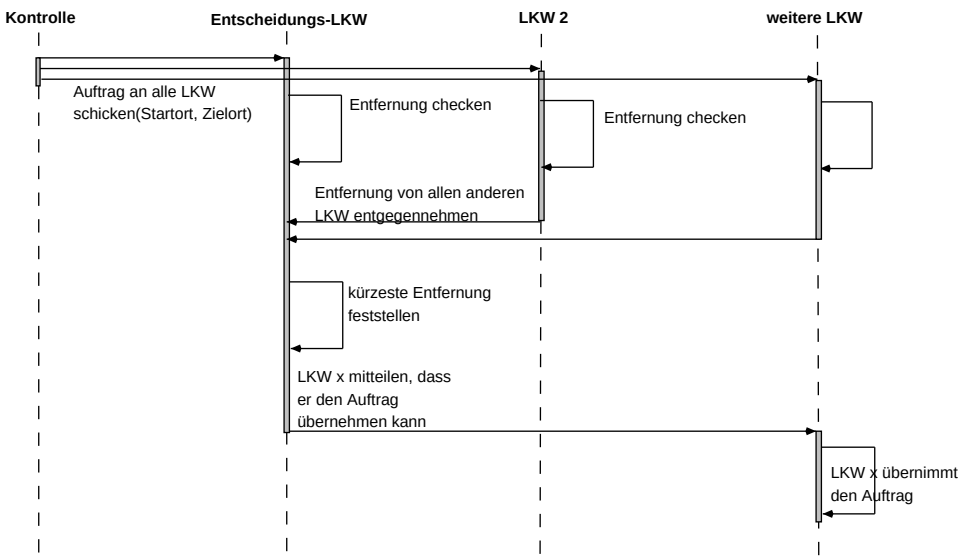

Abbildung 7.6: Das Interaktionsdiagramm des Beispiels

Hat man das Interaktionsdiagramm erstellt, lassen sich daraus die Protokolle ableiten. Die Abbildung 7.7 zeigt das Anfrageprotokoll und Abb. 7.8 zeigt das Protokoll der LKW für das Beispiel.

Beide Protokolle ähneln den oben in diesem Kapitel gezeigten Erzeuger- und Verbraucherprotokollen. Die Ankunft eines Auftrags passiert in dem Netz aus Abbildung 7.7 in der Transition *erzeuge Auftragsbeschreibung*. Hier handelt es

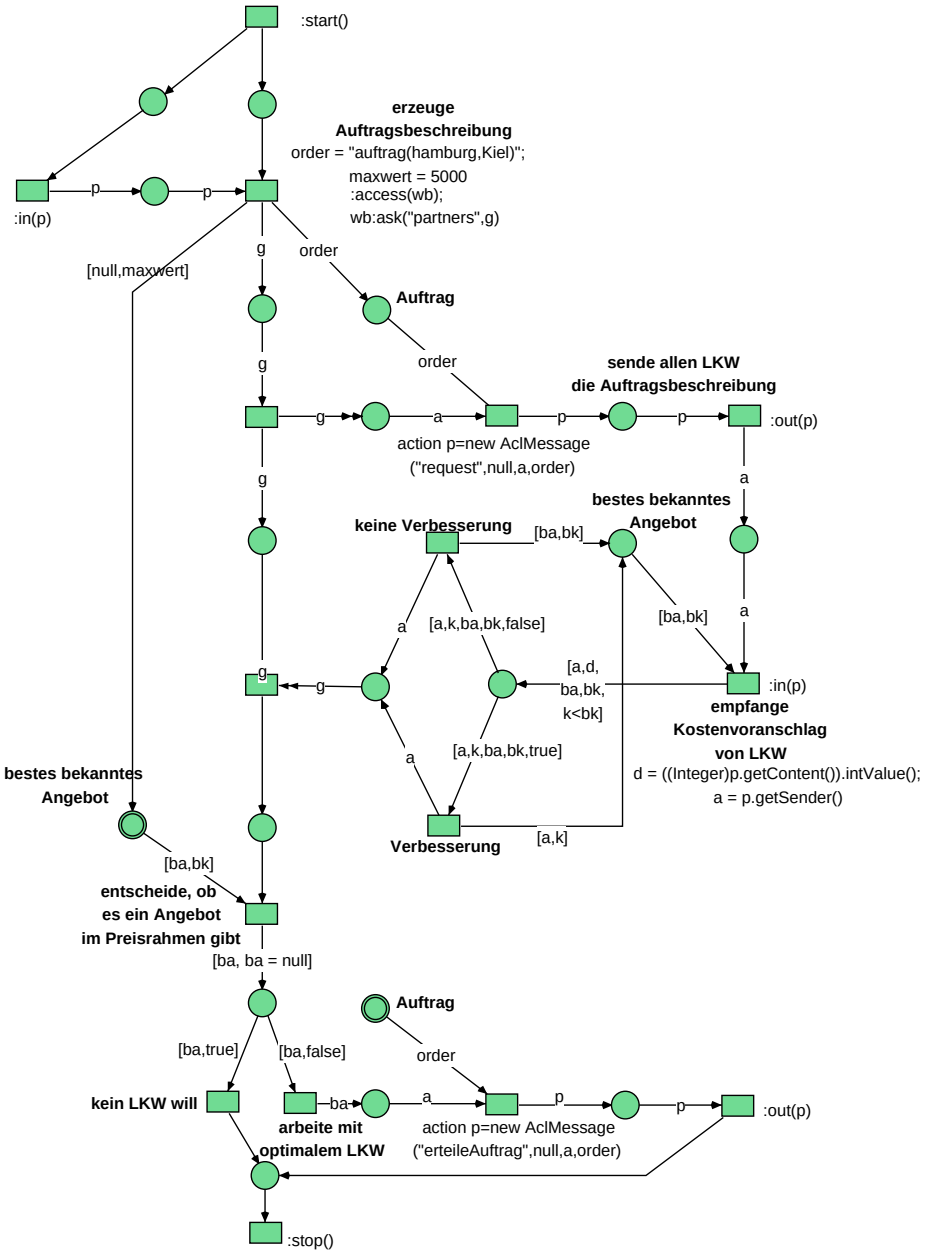

Abbildung 7.7: Interaktionsbeispiel: Generierung eines Auftrags und Bestimmung des geeignetsten LKW durch Auswertung der Gebote

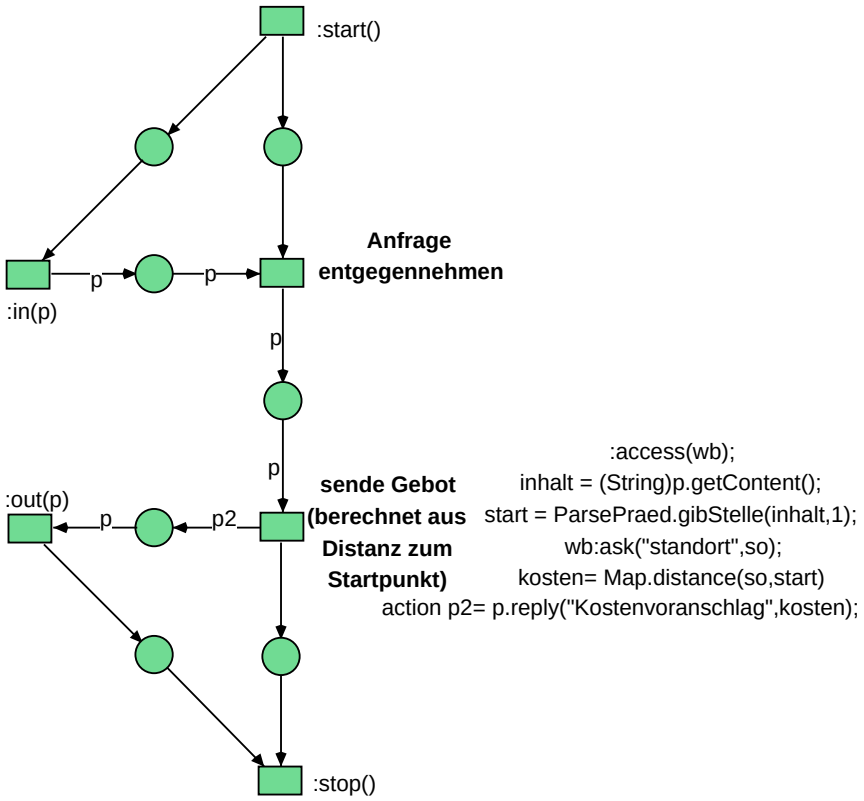

Abbildung 7.8: Interaktionsbeispiel: Versenden des Gebots anhand der relativen Entfernung

sich um einen „von Hamburg nach Kiel“ Auftrag. Dieser Auftrag wird dann über die Transition *:out(p)* an alle LKW verschickt. Vorher wird die Nachricht an einer Transition mit *action*-Anschrift in einen bestimmten Typ umgewandelt: in den Typ *AclMessage*. Diese *AclMessage* beinhaltet u.a. die Nachricht als String und den Agenten, an den die Nachricht verschickt wird. In diesem Fall wird die Nachricht n-mal verschickt, an jeden LKW einmal. Um eine *AclMessage* erzeugen zu können, müssen die Eingangsstellen dieser Transition markiert sein. Bei einer Eingangsstelle wurden dafür *flexible arcs* eingesetzt. Mit *flexible arcs* ist es möglich, eine unterschiedliche Anzahl von Marken zu transportieren. Damit die Auftragsnachricht auch an alle Agenten verschickt werden kann, müssen die einzelnen LKW bekannt sein. Diese Information erhält der Nachrichten verschickende Agent, und damit das Protokoll von der Agentenplattform, auf der alle Agenten verwaltet werden (vgl. Anhang A). Einer der LKW ist der Entscheidungs-LKW, der noch zusätzliche Aufgaben zu erfüllen hat, weshalb auch das Protokoll anders aussieht. Das Protokoll bzw. Netz, in dem die Aufträge erzeugt werden, ist auch gleichzeitig das Protokoll des Entscheidungs-LKWs. An der Transition *:in(p)* empfängt der Entscheidungs-LKW die Antwortnachrichten der anderen LKW. Danach prüft er, ob einer der LKW geeignet ist, den Auftrag zu übernehmen. Die LKW schicken eine Zahl an den Entscheidungs-LKW, die aus der Entfernung zwischen dem LKW-Standort und dem Auftragsstartort, multipliziert mit einem Interessenfaktor, berechnet wird. Ist diese Zahl kleiner als ein Initialwert, dann ist eine Verbesserung eingetreten (Transition *Verbesserung*) und der Wert des LKWs wird als bestes Angebot aufgenommen (Stelle *bestes bekanntes Angebot*). Die Stelle *bestes bekanntes Angebot* kommt als virtuelle Stelle noch an einer anderen Stelle des Netzes vor. Man benutzt virtuelle Stellen, um das Netz übersichtlich zu halten. Danach wird die Antwort des nächsten LKW mit dem derzeit besten Angebot verglichen, und je nach dem, ob dieser einen kleineren Wert hat als der vorherige, wird der LKW ausgetauscht oder nicht (Transitionen *Verbesserung* oder *keine Verbesserung*). So wird jeder LKW überprüft und der Entscheidungs-LKW findet den LKW mit dem kleinsten Wert, der als am geeignetsten angesehen wird. Danach wird überprüft, ob gegenüber dem Initialwert eine Verbesserung eingetreten ist (Transition *entscheide, ob es ein Angebot im Preisrahmen gibt*). Gibt es keinen LKW, der einen Wert hat, der kleiner als der Initialwert ist, wird der Auftrag abgelehnt (Transition *kein LKW will*). Gibt es eine Verbesserung, wird der entsprechende LKW wieder mit einer *AclMessage* über die Transition *:out(p)* benachrichtigt.

In der Abbildung 7.8 ist das Protokoll eines LKW abgebildet. An der Transition *:in(p)* kommt eine Auftragsnachricht ins Netz. An der Transition *Auftrag entgegennehmen* wird die Nachricht überprüft und das Angebot berechnet. Danach wird der Wert des berechneten Angebots als Nachricht über die Transition *:out(p)* an den Entscheidungs-LKW zurückgeschickt.

Um dieses Agenten-Szenario zu simulieren, sind noch weitere Netze zur Unterstützung nötig. Im Anhang A werden diese Netze gezeigt. An diesem Beispiel kann man sehen, wie komplex ein kleines Agentenszenario in der Modellierung sein kann. Es ist mit Hilfe von Renew aber möglich, durch virtuelle Stellen und verschiedene Kantentypen ein Netz übersichtlich zu halten (siehe auch Kapitel 5).

Fazit

In diesem Kapitel wurde die agentenorientierte Simulation eingeführt. Dazu wurde zunächst erklärt, was ein Agent im Allgemeinen ist. Danach wurden Multiagentensysteme eingeführt. Später wurde gezeigt, dass sich auch Petrinetze als Agenten einsetzen lassen. Es wurde diskutiert, wie ein Petrinetzagent aussehen muss, und wie sein Wissen und sein Verhalten realisiert werden. Darüber hinaus wurde Mulan vorgestellt. Mulan ist eine Architektur für Petrinetzagenten, die in der Universität Hamburg von Heiko Rölke entwickelt wurde. Zum Schluss wurde an einem Beispiel gezeigt, wie ein agentenorientiertes Simulationsmodell mit Petrinetzen aussieht. In diesem Beispiel wurde allerdings noch keine Zeitkomponente berücksichtigt; prinzipiell ist das aber mit dem Einsatz des @-Zeichens auch in diesem Beispiel möglich, da es mit Renew entwickelt wurde.

8 Analyse und Auswertung von Simulationsmodellen

In den vorangegangenen Kapiteln wurde die Simulation in erster Linie dazu eingesetzt, um Modelle von Systemen zu entwickeln. Es ging hauptsächlich darum, ein System zu verstehen, es handhaben zu können und durch animierte Petrinetze den Ablauf beurteilen zu können. Es ging in den vorherigen Kapiteln demnach um die Qualität von Simulationsmodellen.

Um die Qualität von Petrinetzsimulationsmodellen genauer zu untersuchen, werden die Begriffe Verklemmungsfreiheit, Lebendigkeit und Fairness eingeführt. Diese Begriffe sind für das Schaltverhalten der Netze und damit auch für die Simulation mit Petrinetzen wichtig, da Verklemmungen ausgeschlossen werden sollen und sich ein System in der Simulation fair verhalten soll.

In den folgenden Abschnitten wird es zusätzlich um die quantitative Analyse von Systemen gehen. Das Ziel dieses Kapitels ist es, Simulationsergebnisse zu erzeugen und zu bewerten.

Für ein gegebenes System und eine gegebene Belastung ist es eine wichtige Aufgabe, den Auftragsverkehr zu ermitteln. Der Auftragsverkehr ist bei der Analyse von Systemen von großer Wichtigkeit, um sie z.B. im Hinblick auf ihre Wirtschaftlichkeit zu optimieren.

Diese Aufgabe stellt sich beim Entwurf eines neuen Systems, etwa durch Einbringung neuer Funktionseinheiten oder durch den Austausch vorhandener, durch Änderung der Strategie des Systems und schließlich bei Veränderung der Belastung. Unter Belastung versteht man – unter Verallgemeinerung des Begriffs des Auftragsankunftsprozesses – Folgen von Aufträgen, geordnet nach Auftragsankunftszeitpunkten.

Die Aufgabe, den Auftragsverkehr zu ermitteln und gegebenenfalls ein System zu optimieren, gehört zum Gebiet des Operation Research. Die spezifische Interpretation der Simulation ist ein wichtiges Instrument des Operation Researches. Die Simulation kann deshalb im Hinblick auf die Optimierung eines Systems als ein Teilgebiet des Operation Research angesehen werden. Sie wird immer dann eingesetzt, wenn keine leistungsfähigen analytischen Lösungsverfahren zur Verfügung stehen. Das gilt in der Praxis insbesondere für einen großen Teil der Operation Research Aufgaben, die stochastischer Natur sind. Der Auftragsverkehr kann demzufolge auch durch Simulation bestimmt werden.

Bei Problemen, die mit Hilfe der Simulation gelöst werden sollen, handelt es sich in den meisten Fällen um Bedien-/Warte-Systeme, um ausfallanfällige Systeme oder auch um Lagerhaltungssysteme. Auf den ersten Blick unterscheiden sich die genannten Systeme deutlich voneinander. Auf den zweiten Blick kann man feststellen, dass es sich in allen drei Fällen um Wartesysteme handelt. Aufgrund der stochastischen Natur der Warteschlangensysteme ist die Behand-

lung von komplizierteren Aufgaben meist nicht mehr mit analytischen Methoden möglich. Die Simulation spielt demnach bei der Lösung praktischer Warteschlangenprobleme eine sehr große Rolle (vgl. [Neu77]).

„Der weitaus größte Teil der Warteschlangentheorie befasst sich mit der Beschreibung des Verhaltens von Wartesystemen in Abhängigkeit von der Zeit oder im Gleichgewichtszustand. Dabei sind der Strom der an den Service-Stationen ankommenden Einheiten und die Verteilung der Bedienzeiten im Allgemeinen vorgegeben. Gesucht sind in der Regel die Verteilung der Wartezeiten und die Anzahl der Kunden im Wartesystem“[Neu77, S.366].

Die Warteschlangentheorie wurde von 1908-1922 von dem Dänen K. Erlang begründet und hat sich seitdem stark weiterentwickelt, da sie sich als sehr vielseitig herausgestellt hat. Kassen, Schalter, Fernsprechvermittlungen, Lagerhaltung und Programme, die auf Prozessorzeit warten, sind Beispiele für die Verbreitung von Wartesystemen.

In Abschnitt 8.2 werden Wartesysteme beschrieben. Es wird eine Einführung in die elementaren Wartesysteme gegeben, dann wird auf Wartenetze eingegangen und auf hierarchische Bediensysteme.

Im darauf folgenden Abschnitt 8.3 werden Wartesysteme bezüglich der Zugänge, Durchsätze, Verweilzeiten und Füllungen analysiert. Es wird auf die Analyse elementarer Wartesysteme durch Markovprozesse eingegangen und auf die Analyse von Wartenetzen. Darüber hinaus soll geklärt werden, wann ein Simulationsdurchlauf abgebrochen werden kann, wann das Ergebnis befriedigend ist und wie man die Streuung der Ergebnisse minimieren kann.

Abschließend soll in Abschnitt 8.4 auf die Darstellung von Simulationsergebnissen eingegangen werden.

8.1 Verklemmungsfreiheit, Lebendigkeit, Fairness

Ein Simulationsmodell, das mit Petrinetzen erstellt worden ist, soll bei der Simulation alle Marken, die in das System (Netz) hineinkommen, verarbeiten und jederzeit schalten können. Kann das Netz, obwohl noch Marken vorhanden sind, die verarbeitet werden sollen, nicht mehr schalten, dann liegt eine Verklemmung (deadlock) vor. Wenn es zu Verklemmungen im System kommt, wurde bei der Konstruktion ein Fehler gemacht, der behoben werden muss. Verklemmungen kann es auch in Simulationsmodellen geben, die mit imperativen Sprachen erstellt worden sind. Sie sind in diesem Fall besonders schwierig zu finden, da der Fehler sich irgendwo im Quelltext verbirgt und nicht sofort zu lokalisieren ist. In Petrinetzsimulatoren ist es durch die grafische Darstellung leichter, eine Verklemmung zu finden, da sich die Marken an einer bestimmten Stelle sammeln und von dort nicht weiter transportiert werden. Die Simulation steht also still.

Nach [JV87, S.196] ist eine Verklemmung:

Definition 13 *Es sei $N = (S, T, F, K, W, m_0)$ ein S/T-Netz. Eine Markierung m heißt Verklemmung (deadlock), falls keine Transition aktiviert ist:*

$$\neg\exists t \in T : m \xrightarrow{t}$$

N heißt verklemmungsfrei, falls keine erreichbare Markierung $m \in R(N)$ eine Verklemmung ist.

In einer sicheren oder fortsetzbaren Markierung hingegen gibt es immer mindestens eine Feuerfolge, die das System terminieren bzw. unbegrenzt weiterlaufen lässt.

„Die Verklemmungsfreiheit eines Systems von Funktionseinheiten bedeutet, dass kein totaler Stillstand des ablaufenden Prozesses eintritt. Dadurch ist natürlich nicht ausgeschlossen, dass gewisse Funktionseinheiten oder Teile des gesamten Systems blockiert sind. Die Abwesenheit von solchen partiellen Verklemmungen nennt man Lebendigkeit des Systems. Ein Netz ist dann lebendig, wenn nach einer beliebigen Schaltfolge jede Transition wieder zum Schalten gebracht werden kann.“ [JV87, S.226]

Formal ist ein Netz nach [JV87] dann lebendig, wenn folgendes gilt:

Definition 14 *Es sei $N = (S, T, F, K, W, m_0)$ ein S/T-Netz, $m \in M_S$ eine Markierung und $E \subseteq T$ eine Teilmenge von Transitionen.*

- *E heißt lebendig in m, falls $\forall t \in E \forall u \in E^* \exists v \in T^* : m \xrightarrow{u} \Rightarrow m \xrightarrow{uvt}$*
- *m heißt lebendig, falls T in m lebendig ist.*
- *N heißt lebendig, falls m_0 lebendig ist.*
- *m heißt E-fortsetzbar, falls $\exists w \in T^\omega : m \xrightarrow{w}$ und alle $t \in E$ in diesem w unendlich oft vorkommen.*

„Lebendigkeit bedeutet demnach Freiheit von unvermeidbaren partiellen Verklemmungen. N ist lebendig $\Leftrightarrow \forall t \in T \forall m \in R(N) \exists w \in T^\omega : m \xrightarrow{w} \wedge |w|_t = \infty$.“ [JV87, S.230]

Die Verklemmungsfreiheit und die Lebendigkeit von Petrinetzen als Simulationsmodell ist eine notwendige Voraussetzung, damit das System fehlerfrei ablaufen kann. Diese beiden Bedingungen sind aber noch nicht ausreichend. Zusätzlich muss das Netz auch faires Verhalten zeigen. „Fairness bedeutet Freiheit von faktischen partiellen Verklemmungen. N verhält sich fair $\Leftrightarrow \forall t \in T \forall m \in R(N) \forall w \in T^\omega : m \xrightarrow{w} \Rightarrow |w|_t = \infty$.“ [JV87, S.230]

D.h., jede Transition, die theoretisch schalten kann, muss auch schalten. Die Wichtigkeit dieser Eigenschaft wird klar, wenn man an Bedien-/Wartesysteme denkt. Ein Kunde, der bedient werden kann, muss auch irgendwann bedient werden, sonst wird das Simulationsmodell unrealistisch und kann für die Analyse eines Systems nicht verwendet werden.

Nach [JV87, S.230] zeigt ein Netz faires Verhalten, wenn folgendes gilt:

Definition 15 *Ein S/T-Netz $N = (S, T, F, K, W, m_0)$ hat ein faires Verhalten, oder verhält sich fair, wenn in jeder unendlichen Schaltfolge $w \in F_\omega(N)$ jede Transition $t \in T$ unendlich oft vorkommt.*

In diesem Abschnitt sollte gezeigt werden, dass die sorgfältige Konstruktion eines Petrinetzsimulators bezüglich seines Schaltverhaltens für die Verwendbarkeit zur Analyse ein wesentlicher Bestandteil der Simulation mit Petrinetzen ist.

8.2 Hierarchische Bediensysteme

In dem folgenden Abschnitt wird das Modell der *hierarchischen Bediensysteme* (HBS) betrachtet, die als konzeptionelle Grundlage den auszuwertenden Simulationsmodellen zugrunde liegen. Hierarchische Bediensysteme verallgemeinern die bekannten Modelle der Wartesysteme bzw. der Wartenetze durch die Möglichkeit, Substrukturen zu beschreiben. Bevor das allgemeine Modell der HBS eingeführt wird, werden zuvor Wartesysteme und -netze betrachtet, insbesondere in Hinblick auf deren Anwendbarkeit als theoretisches Modell und auf die Randbedingungen, unter denen diese analytisch zu behandeln sind.

8.2.1 Elementare Wartesysteme

Elementare Wartesysteme (EWS) beschreiben ein System, das sich durch den Ankunft- und Bedienprozess von Aufträgen kennzeichnen lässt. Beispiele für Wartesysteme sind: Postschalter, Autowaschanlage, Kassen etc. (siehe oben). Sie charakterisieren die Bedienzeit lediglich durch die Bedienzeit an einem Typ von Instanz (bzw. an einer ganz bestimmten Funktionseinheit FE, z.B. einem Postschalter). Das Warten von Aufträgen steht im Vordergrund des Interesses.

Ein System, in dem der Abbruch von Aufträgen zulässig ist, heißt Verlustsystem. Wenn ein Auftrag lediglich deshalb aktuell nicht erledigt werden kann, weil die Kapazität der beauftragten Bedieneinheit (oder FE) begrenzt ist, dann kann die zugehörige Anweisung in einem dafür bestimmten Wartekanal lagern, bis der Auftrag zur Ausführung übernommen werden kann, d.h. der Auftrag wartet. Solche Lagerzeit heißt Wartezeit. Ankommende Aufträge warten in einer Warteschlange, bevor sie von den Bedieneinheiten bearbeitet werden. Die Größe der Warteschlange k und die Anzahl der Bediener m sind weitere Parameter zur Beschreibung des Wartesystems.

Ein System, in dem aktuell nicht ausführbare Aufträge warten, ohne abgebrochen zu werden, heißt Wartesystem. Ein Wartesystem, das nur aus einer Instanz besteht, heißt elementares Wartesystem (vgl. [JV87, S.348]).

Abbildung 8.1 zeigt die Darstellung eines EWS in Form eines Petrinetzes. Die Transition *enter* charakterisiert den Ankunftsprozess, der einen Auftrag (eine Marke) auf der Stelle *queue* erzeugt. Die Transition *leave* bearbeitet einen Auftrag, indem sie eine Marke von *queue* entfernt. Die Anzahl m der Bedienstationen ergibt sich anhand der Initialmarkierung der Stelle *servers*, in diesem Beispiel als $m = 2$. Die Kapazität des Warteraums ergibt sich anhand der Stelle *capacity* zu $k = 3$.

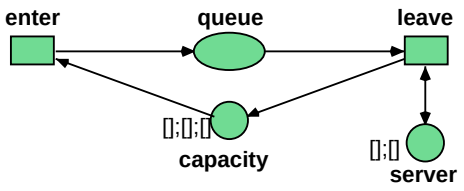

Abbildung 8.1: Ein Wartesystem

Der Ankunftsprozess beschreibt die Art und Weise, in der die Forderungen in dem Wartesystem eintreffen. Die Reihenfolge in der die Forderungen vom Bedienungsmechanismus abgearbeitet werden, wird durch die Bedienstrategie bestimmt. Die Bedienstrategie legt fest, welcher im elementaren Wartesystem verweilende Auftrag jeweils bedient wird. Die am häufigsten gebrauchte Bedienstrategie ist *fifo*, first in first out. Sie besagt, dass Forderungen in der Reihenfolge ihres Eintreffens bedient werden. Eine Alternative ist *lifo*, last in first out. Sie bedeutet, dass die zuletzt angekommene Forderung zuerst bedient wird. Bei *siro*, service in random order, werden die Forderungen in einer Reihenfolge abgearbeitet, die einem Zufallsmechanismus unterliegt. Für *pri*, priority, eingehende Forderungen werden Prioritäten zugeordnet und entsprechend ihrer Rangfolge abgearbeitet, wobei die Bearbeitung innerhalb einer Priorität meist wieder nach fifo durchgeführt wird (vgl. [Neu77]).

Darüber hinaus gibt es zusätzlich noch Bedienstrategien, die von der Bearbeitungszeit der einzelnen Forderungen abhängen (z.B. *shortest job first*). Bedienstrategien heißen nicht verdrängend, wenn keine Forderung aus der Bedienstation verdrängt wird, d.h., jeder Auftrag, dessen Bedienung bereits begonnen hat, wird erledigt, bevor in der selben Bedieneinheit ein anderer Auftrag bzw. eine andere Forderung bedient wird; sonst heißt sie verdrängend (vgl. [JV87, S.452-499]). Sie heißt produktiv, wenn sie keinen Auftrag warten lässt, solange die Füllung der Bedienstation kleiner als die Kapazität ist, fair gegen Aufträge einer Klasse K, wenn sie keinen Auftrag aus K, der eine endliche Bedienzeit hat, unendlich lange warten lässt, aber einen später ankommenden Auftrag einer anderen Klasse nur endlich warten lässt. Sie heißt Overhead-frei, wenn sie selbst nicht zur Füllung der Bedienstation beiträgt (vgl. [JV87, S.351]).

Ankommende Forderungen, die nicht sofort bedient werden, reihen sich in eine Warteschlange vor der Bedienstation ein. Hier müssen zwei Fälle unterschieden werden. Einmal besteht die Möglichkeit, dass der Warteraum, in dem die Forderungen auf ihre Bearbeitung warten physikalisch begrenzt oder überhaupt nicht vorhanden ist, und daher nur eine endliche Anzahl von Forderungen vor der Bedienstation warten können. Ist der Warteraum besetzt, so müssen weiter hinzukommende Forderungen abgewiesen werden. Diese gehen somit dem Wartesystem verloren. Ist diese physikalische Begrenzung nicht gegeben, so spricht man von einem unendlich großen Warteraum.

Geht man von kontinuierlichen Zeitpunkten aus, so ist ein Ankunftsprozess charakterisiert durch eine monoton wachsende Abbildung $z : \mathbb{N} \to \mathbb{R}$. Dabei beschreibt $z(n)$ den Ankunftszeitpunkt des n-ten Auftrags. Monotonie von z bedeutet, dass für alle $m \leq n$ auch $z(m) \leq z(n)$ gelten muss, d.h. nachfolgende Aufträge kommen zu keinem früheren Zeitpunkt. Analog ist ein Bedienprozess eine monotone Abbildung $b : \mathbb{N} \to \mathbb{R}$, wobei $b(n)$ den Bedienzeitpunkt des n-ten Auftrags beschreibt. Da nicht mehr Aufträge als vorhanden bedient werden können, muss $\sum_{i=0}^{n} b(i) \leq \sum_{i=0}^{n} z(i)$ für alle Anzahlen n gelten (vgl. [JV87, S.342-343]).

Für stochastische elementare Wartesysteme betrachtet man nicht die Zeitpunkte der Ankünfte, sondern gibt die Verteilung der Abstände zwischen Ankunftsereignisse an: die Zwischenankunftszeit. Analog wird nur die Verteilung der Bedienzeiten betrachtet.

Die Verteilung der Zwischenankunftszeiten kann beispielsweise einer Exponentialverteilung M, einer deterministischen Verteilung D, einer Erlangverteilung E_k oder einer allgemeinen Verteilung GI gehorchen. Für die Verteilung der Bedienzeiten gilt das Gleiche.

Ein besonderer Fall liegt vor, wenn sowohl Zwischenankunfts- als auch Bedienzeiten exponentiell verteilt sind, also der Verteilung $F(t) = 1 - e^{\lambda t}$ gehorchen, wobei λ die Ankunftsrate bzw. die Bedienrate ist. Es gilt, dass ein System mit exponentiell verteilten Zwischenankunft- und Bedienzeiten die Markoveigenschaft besitzt: Die Aufträge sind also stochastisch unabhängig.[1] Elementare Wartesysteme mit exponentieller Verteilung, m Bedienern und der Warteraumkapazität k werden als $M/M/m/k$ notiert. Ist der Warteraum potentiell unendlich groß ($k = \infty$), so wird statt $M/M/m/\infty$ kurz $M/M/m$ notiert.

Ein M/M System besitzt den Vorzug, dass seine charakteristischen Größen – wie Wartezeiten, Füllung etc. – analytisch berechnet werden können, sodass für diese Systeme eine Simulation nicht notwendig ist (siehe dazu z.B. [Neu77], [HLuK78] oder [Bor76]).

8.2.2 Wartenetze

Ein Wartenetz ist nach [JV87, S.392] gekennzeichnet durch

- eine Menge $\mathbb{M}$ von elementaren Wartesystemen $i (i \in [1..|\mathbb{M}|])$, genannt die Knoten des Wartenetzes, mit je m_i gleichen Bedieneinheiten, Kapazität k_i und Bedienstrategie BS_i

- seriell ausgeführte Auftragssysteme, deren Teilaufträge bei Bedienung am Knoten i durch Zugehörigkeit zu einer Klasse $r \in K$ (Klassenmenge) mit der klassenspezifischen Bedienzeit b_{ir} und der klassenspezifischen Folgegesetzmäßigkeit $(i, r) \rightarrow (j, s)$ gekennzeichnet sind $(i, j \in \mathbb{M}, r, s \in K)$

Offensichtlich ist jedes elementare Wartesystem ein Spezialfall eines Wartenetzes mit nur einem Knoten.

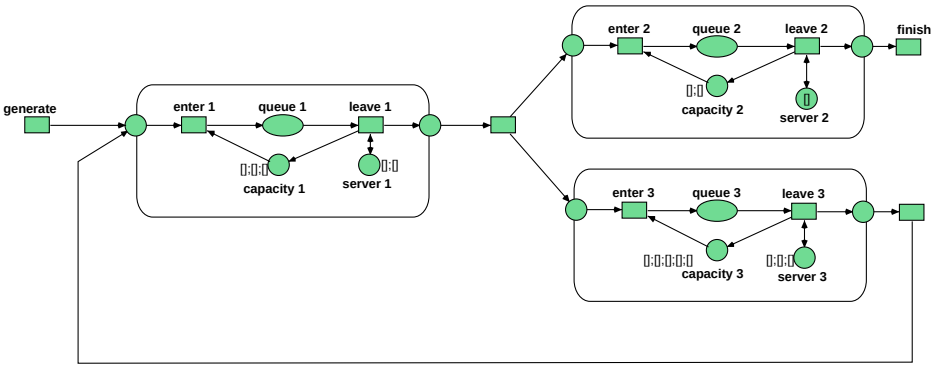

Abbildung 8.2: Ein Wartenetz

[1] Für diskrete Zeit ergeben sich für die geometrische Verteilung analoge Aussagen (siehe [Hüb96]).

Abbildung 8.2 zeigt die Darstellung eines Wartenetzes in Form eines Petrinetzes. Die Knoten des Wartenetzes sind elementare Wartesysteme wie in Abb. 8.1. Sie sind mit einem Kasten umrandet. Es ist zu erkennen, dass das Verlassen eines Teilsystems Folgeaufträge für andere Teilsysteme generiert. Beispielsweise erzeugt das Verlassen des ersten Knotens durch die Transition *leave 1* zwei neu Aufträge, die in dem zweiten bzw. dritten Knoten bearbeitet werden.

Die Darstellung eines Wartenetzes in Abbildung 8.2 weist durch die Replikation der Wartesystem-Struktur eine hohe Redundanz auf. Diese Strukturähnlichkeit kann zu einem gefärbten Petrinetz (siehe Kapitel 4) zusammengefaltet werden. Es ergibt sich eine Struktur wie in Abbildung 8.3, bei der die Wegewahl durch Paare (x, y) auf der Stelle *Wegewahlpaare* bestimmt wird.

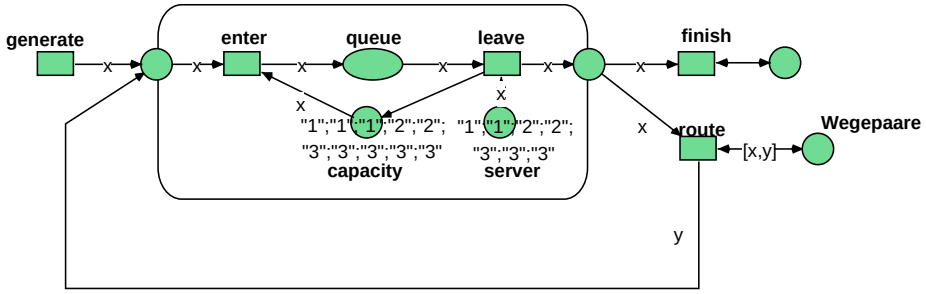

Abbildung 8.3: Ein gefärbtes Wartenetz

8.2.3 Hierarchische Bediensysteme

Die bislang betrachteten Wartenetze besitzen keinerlei Substrukturen. Im Folgenden betrachten wir hierarchische Bediensysteme (HBS). Ein HBS ist ein Netz, dessen Knoten – im Gegensatz zu Wartnetzen – wiederum HBS sein dürfen. Das einfachste HBS ist ein elementares Wartesystem. Wartenetze sind ein Spezialfall der HBS, bei dem jeder Knoten ein elementares Wartesystem ist, also keinerlei rekursive Substruktur besteht. Ein Beispiel für ein HBS ist in Abbildung 8.4 dargestellt.

Die Substrukturen beschreiben Verfeinerungen der Knoten. HBS erlauben es, unterschiedlich detaillierte Sichten auf ein System einzunehmen. Dies ist sinnvoll, da man im Allgemeinen auch simulative Analysen verschiedener Granularitätsstufen benötigt.

Die hierarchische Beschreibung der HBS korrespondiert zu dem Konzept der Verfeinerung/Vergröberung der Petrinetztheorie. Wie man in Abbildung 8.4 erkennen kann, sind die drei Teil-HBS stellenberandete Gebiete, die durch die Wegewahltransitionen ihrerseits zu einem strukturgleichen HBS aggregiert werden.

Das Modell der HBS ist von großer Bedeutung, da es sehr abstrakt und daher allgemein einsetzbar ist. Seine rekursive Natur ermöglicht die Definition von Verfeinerungsoperatoren zur Erzeugung von HBS. Dies geschieht analog zu der Erzeugung wohlgeformter Workflownetze nach [vdA95] durch Operatoren,

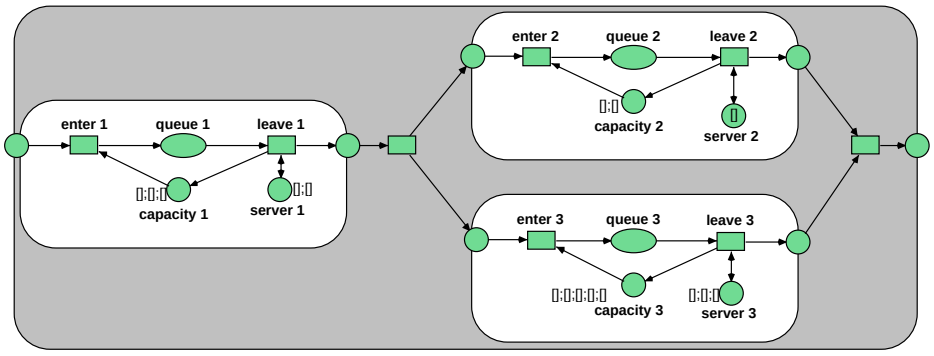

Abbildung 8.4: Ein hierarchisches Wartenetz

wie der Sequenz, der Alternative, nebenläufiger fork-join Strukturen etc. Diese Ähnlichkeit ist naheliegend, da Workflows auch Bearbeitungsstrukturen von Aufträgen darstellen, wenn auch im Allgemeinen ohne die Berücksichtigung von Zeit.

8.3 Standardauswertungen

Eine Handlung in einem System kommt durch die Verpflichtung einer Funktionseinheit FE durch einen entsprechenden Auftrag zustande. Verschiedene Aufträge bzw. Funktionseinheiten können durch Verknüpfung komplexe Strukturen bilden (vgl. Abb. 8.1 und 8.2), wie im vorherigen Abschnitt gezeigt wurde.

In diesem Abschnitt geht es um die konkrete Analyse von Funktionseinheiten. Es werden die Eigenschaften Zugang, Verweilzeit, Durchsatz und Füllung einer FE und ihr Zusammenhang diskutiert. Danach wird die Analyse von elementaren Wartesystemen durch Markovprozesse gezeigt und die Analyse von Wartenetzen eingeführt.

Im Zusammenhang mit der Simulation bzw. mit Simulationsergebnissen sind darüber hinaus noch Abbruchkriterien der Simulationsläufe, Konfidenzintervalle und Methoden zur Varianzverringerung interessant, die ebenfalls in einem Abschnitt eingeführt werden.

8.3.1 Kenngrößen

Die Analyse der Bearbeitung zusammengesetzter Aufträge in einem System, das aus Funktionseinheiten besteht, ist die Aufgabe dieses Abschnitts.

Aufträge kommen in ein System und durchwandern es. Jede Funktionseinheit bearbeitet einen Teilauftrag. Daraus ergibt sich, dass es zu nebenläufigen Teilaufträgen kommen kann (vgl. Abb. 8.2). In einem Zeitintervall kann eine FE Aufträge verarbeiten, die Verweilzeiten y_i (engl.: dwell time) erfordern[2]. Während eine FE einen Auftrag bearbeitet, bleibt der Auftrag in der FE. Die Zeit, die die FE benötigt, um den Auftrag zu bearbeiten, heißt Verweilzeit $y = (t_2 - t_1)$.

[2]Das i bei der Bezeichnung y_i dient zur Unterscheidung der Verweilzeiten y der verschiedenen Aufträge.

Die Belegtzeit $x = \sum_{j=1}^{n} y_j$ einer FE ist die Summe der Verweilzeiten[3]. Sie ist die Zeit innerhalb eines Betrachtungsintervalls, in der die Funktionseinheit belegt ist. In einem Zeitintervall (t_1, t_2), mit $t_1 < t_2$, kann eine bestimmte Anzahl von Aufträgen bearbeitet werden. Daraus ergibt sich der Durchsatz $d(t_1, t_2)$ (engl.: throughput) s.u. der FE. Von Aufträgen dieser Charakteristik kann die Funktionseinheit einen Grenzdurchsatz $c = \frac{1}{\bar{y}}$, mit $\bar{y} = \frac{x}{n}$, leisten. Daraus lässt sich die Auslastung $\rho = \frac{d}{c}$, auch relative Belegtzeit genannt, berechnen. Die Füllung f (engl.: fill) einer Funktionseinheit ist die Anzahl der Aufträge, die der Funktionseinheit zu einem Zeitpunkt übergeben wurden, die aber noch nicht erledigt worden sind. Bei Füllung $f = 0$ heißt eine FE frei, bei $f > 0$ beschäftigt. $f > 1$ bedeutet Simultanbetrieb. Existiert eine maximale Füllung (engl.: maximum fill), dann heißt diese Kapazität k der FE. Ist $k = 1$, dann heißt die FE einfach, bei $f = k$ belegt. Der Quotient $\rho = f/k$ heißt relative Füllung der FE. Die Bedienzeit eines Auftrags in einer Funktionseinheit ist die Verweilzeit bei Füllung 1, d.h. wenn kein anderer Auftrag in der Funktionseinheit verweilt (vgl. [JV87, S.62 und S.67]). Die Kapazität kann im Netz durch Komplementärstellen ausgedrückt werden (vgl. Beispiel aus Abb. 6.3).

Sei $\#z(t)$ die Anzahl der Zugänge (engl.: input) von Aufträgen im Intervall $[t_0, t)$ und sei $\#a(t)$ die Anzahl der Abgänge (engl.: output) von Aufträgen im Intervall $[t_0, t)$. Die Füllung ist die Anzahl der aktuell im System vorhandenen Aufträge:

$$f(t) := \#z(t) - \#a(t)$$

Abbildung 8.5 illustriert die Füllung anhand von Zu- und Abgängen.

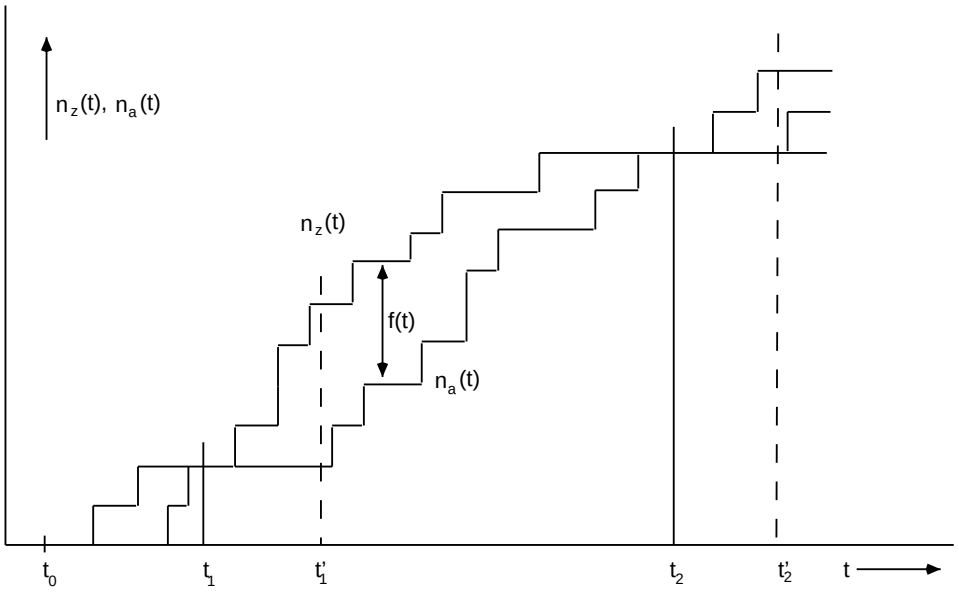

Abbildung 8.5: Beispielgraph für Zu- und Abgänge

[3]Das n ist die Anzahl der verschiedenen Aufträge.

Der Zugang von Aufträgen im Intervall $[t_1, t_2]$ ist definiert als der Quotient:

$$z(t_1, t_2) := \frac{\#z(t_2) - \#z(t_1)}{t_2 - t_1}$$

Der Durchsatz von Aufträgen ist der Quotient:

$$d(t_1, t_2) := \frac{\#a(t_2) - \#a(t_1)}{t_2 - t_1}$$

Für Systeme im *Flussgleichgewicht* kommen genau so viele Aufträge in das System wie nach der Bearbeitung aus dem System. Es gilt also: $z(t_1, t_2) = d(t_1, t_2)$ und somit auch $\#z(t_2) - \#z(t_1) = \#a(t_2) - \#a(t_1)$. Es werden dabei nur FE betrachtet, die keine verdrängenden Bedienstrategien verfolgen.

Die von $\#z$ und $\#a$ eingeschlossene Fläche in Abbildung 8.5 lässt sich auf zweierlei Weise beschreiben: Zum einen als der Verlauf der Füllung f über die Zeit (horizontal), zum anderen als der Verlauf der Verweilzeit y_i der einzelnen Aufträge. Dieser Zusammenhang ist als das Gesetz von Little bekannt. Sei die mittlere Füllung definiert als $\bar{f}(t_1, t_2) = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} f(t)\mathrm{d}t$, die mittlere Verweilzeit als $\bar{y}(t_1, t_2) = \frac{1}{\#z(t_2) - \#z(t_1)} \sum_{i=\#z(t_1)+1}^{\#z(t_2)} y_i$. Für die Mittelwerte von Füllung $\bar{f}(t_1, t_2)$ und Verweilzeit $\bar{y}(t_1, t_2)$ gilt wegen der Flächengleichheit:

$$\bar{y}(t_1, t_2) = \frac{1}{\#z(t_2) - \#z(t_1)} \sum_{i=\#z(t_1)+1}^{\#z(t_2)} y_i = \frac{1}{\#z(t_2) - \#z(t_1)} \int_{t_1}^{t_2} f(t)\mathrm{d}t$$

Dies ergibt das Gesetz von Little:

$$\bar{f}(t_1, t_2) = d(t_1, t_2) \cdot \bar{y}(t_1, t_2)$$

Die Little'sche Formel gilt auch im stochastischen Modell. Dort gilt – analog zu den Mittelwerten – für die Erwartungswerte:

$$E[F] = E[D] \cdot E[Y]$$

Mit den Formeln von Little lässt sich das Grenzverhalten von Systemen bei sehr kleiner bzw. sehr großer Füllung charakterisieren.

Als klein bezeichnet man eine Füllung des Systems, bei der die Verweilzeit nicht wesentlich von der Bedienzeit verschieden ist, d.h. $y(f) \approx b$ für kleine f. Bei großer Füllung wird der Durchsatz d des Systems sich dem Grenzdurchsatz $d \to c$ nähern, so dass mit dem Gesetz von Little $y(f) \approx f/c$ für $f \to \infty$.

Der Durchsatz wächst bei kleiner Füllung proportional dem Mittelwert der Füllung: $d \approx f/b$. Bei großer Füllung des Systems wird der Durchsatz durch den Verkehrsengpass begrenzt: die mittlere Verweilzeit wächst bei großer Füllung proportional dem Mittelwert der Füllung.

Offenbar zeigt das System den asymptotischen Verlauf von Durchsatz und mittlerer Verweilzeit als Funktion der mittleren Füllung (vgl. Abb. 8.6). Der Schnittpunkt der beiden Asymptoten liegt bei der mittleren Füllung, dieser Wert heißt Sättigungsfüllung.

In einem System aus m Funktionseinheiten heißen die Funktionseinheiten mit der Auslastung $\rho_{max} = max/\rho_i$ Verkehrengpässe (bottlenecks) des Systems. Sollen alle in (t_1, t_2) eingegangenen Aufträge bis t_2 erledigt sein, dann muss nach Definition des Grenzdurchsatzes $\rho_{max} \leq 1$ sein (vgl. [JV87, S.342-389]).

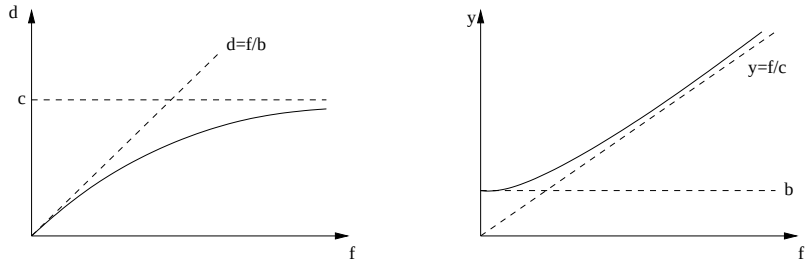

Abbildung 8.6: Darstellung der Verweilzeit bei kleiner und großer Füllung

8.3.2 Analyse elementarer Wartesysteme durch Markovprozesse

Die einfachste Form der abhängigen Koppelung und gleichzeitig die wichtigste für die Modellierung mehrstufiger praktischer Probleme ist die sogenannte Markovkopplung, benannt nach A.A. Markov, der solche Modelle zu Beginn des letzten Jahrhunderts untersucht hat.

Hängen bei einem mehrstufigen Versuch die Übergangswahrscheinlichkeiten nicht von der vollen Vorgeschichte ab, sondern nur vom letzten beobachteten Wert, so spricht man von Markovkopplung. Die Folge der Beobachtungen bildet dann einen Markovprozess, im diskreten Fall auch Markovkette genannt.

Sei $S : \mathbb{R} \to X$ eine Zustandsvariable. Die Zustandsvariable S hat die *Markoveigenschaft*, wenn die Wahrscheinlichkeit, im Zeitpunkt t vom Zustand i in den Zustand j zu wechseln nur vom derzeitigen Zustand $S(t)$, nicht aber von den vorherigen Zuständen $S(t-a)$ abhängt:

$$P(S(t+\Delta t) = j | S(t) = i \wedge S(t-a) = k) = P(S(t+\Delta t) = j | S(t) = i)$$

Ein *homogener* Markovprozess liegt vor, falls die Übergangswahrscheinlichkeit nicht vom Zeitpunkt t abhängt:

$$P(S(t+\Delta t) = j | S(t) = i) =: p_{ij}(\Delta t)$$

Insbesondere lassen sich $M/M/k$ Systeme durch Markovketten formalisieren. Die Zustände sind die möglichen Füllungen $\{0, \ldots, k\}$ des Wartesystems. Jeder Markovprozess, für den gilt, dass von jedem Zustand jeder Zustand erreichbar ist, heißt irrreduzierbar. Jeder irrreduzierbarer Markovprozess besitzt eine konvergierende Zustandsverteilung. Das Markovsystem für ein $M/M/k$ System ist stark zusammenhängend, sodass ein stationärer Grenzprozess, d.h. ein Gleichgewichtszustand existiert. Berechnet man die Wahrscheinlichkeiten $p(f), f \in \{0, \ldots, k\}$ der Zustände für den stationären Grenzprozess, so ergeben sich die Wahrscheinlichkeiten für die jeweiligen Füllungen, woraus der Erwartungswert der Füllung $E[F]$ folgt. Aus $E[F]$ ergibt sich dann mit dem Gesetz von Little die Wartezeit der Aufträge.

Ein einfach zu analysierendes System ergibt sich für einen unendlichen Warteraum und einen Bediener, kurz $M/M/1$. Sei λ die Ankunftsrate und μ die Bedienrate. Der Erwartungswert des Durchsatzes $E[D]$ ist gleich dem Erwartungswert der Anzahl der ankommenden Aufträge, da sich das System im Gleichgewichtszustand befindet: $E[D] = \lambda$. Der Erwartungswert der Bedienzeit ist

$E[B] = 1/\mu$. Der Erwartungswert der Füllung F, der Verweilzeit Y und der Wartezeit W ergibt sich als:

$$\begin{aligned} E[F] &= \frac{\rho}{1-\rho} \\ E[D] &= \lambda \\ E[B] &= 1/\mu \\ E[Y] &= \frac{E[F]}{E[D]} = \frac{\rho}{1-\rho}\,\frac{1}{E[D]} = \frac{E[B]}{1-\rho}, \text{ da } \frac{\rho}{E[D]} = \frac{\lambda/\mu}{\lambda} = E[B] \\ E[W] &= E[Y] - E[B] = \frac{\rho}{1-\rho}E[B] \end{aligned}$$

Sind die Schaltzeiten aller Transitionen eines zeitbehafteten Petrinetzes exponentiell verteilt, so ist der Erreichbarkeitsgraph isomorph zur Markovkette des beschreibenden Wartesystems (vgl. [BK96] oder Kapitel 4.5).

Das stochastische Petrinetz (SPN) aus Abbildung 8.7 beschreibt ein $M/M/1$-System und hat eine Markovkette als Erreichbarkeitsgraphen. Die Abbildung 8.8 beschreibt den dazugehörigen Markovprozess als Erreichbarkeitsgraphen.

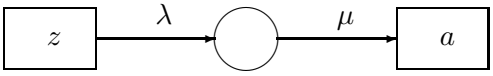

Abbildung 8.7: Das stochastische Petrinetz (SPN) als M/M/1-System

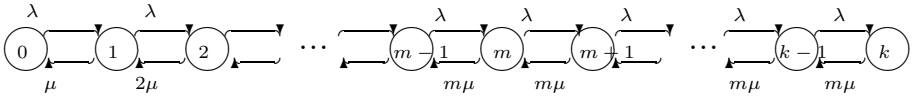

Abbildung 8.8: Die zugehörige Markovkette als Erreichbarkeitsgraph

8.3.3 Analyse von Wartenetzen

Eine Analyse von Wartenetzen durch Markovprozesse ist im Allgemeinfall nicht möglich, da durch die Kopplung von Wartesystemen die stochastische Unabhängigkeit der Ankunftsprozesse nicht gegeben ist, also gilt die Markoveigenschaft (die „Gedächtnislosigkeit“) gar nicht.

Unter gewissen Zusatzannahmen lassen sich die Erwartungswerte der Kenngrößen dennoch analytisch berechnen. [JV87, S.397 und S.401] folgend kann man hier die Jackson- und Gordon-Newell-Netze nennen, die beide auf genau einem Auftragtyp operieren. Die Wegewahl durch beide Typen von Wartenetzen wird durch eine Wegewahlwahrscheinlichkeit beschrieben.

Jackson-Netze sind offene Systeme mit unendlichen Warteräumen und exponentiell verteilten Bedienzeiten. Für Jackson-Netze ergibt sich das bemerkenswerte Ergebnis, dass die Füllungen der einzelnen Systeme alle wie ein $M/M/k$ Systeme verteilt sind – und dies sogar trotz der stochastischen Abhängigkeit.

Gordon-Newell-Netze sind dagegen geschlossene Systeme, d.h. die Anzahl der Aufträge im System ist konstant, da bearbeitete Aufträge als neue Folgeaufträge erneut in das System gelangen. Die Bedienzeiten sind exponentiell verteilt. Die Füllungswahrscheinlichkeiten lassen sich berechnen, die Wahrscheinlichkeiten weichen jedoch von denen der $M/M/k$ Systeme ab. Bemerkenswert ist weiterhin, dass der Zustandsraum eines Gordon-Newell-Netzes aufgrund der konstanten Anzahl an Aufträgen in einem geschlossenen System endlich ist – im Gegensatz zum unendlichen Zustandsraum eines Jackson-Netzes.

Für Wartenetze, die nicht den obigen Einschränkungen (genau ein Auftragstyp, exponentielle Verteilung der Ankunft- und Bedienprozesse, Routing der Aufträge durch Wegewahlwahrscheinlichkeiten) genügen, ist dagegen auf simulative Verfahren zurückzugreifen.

8.3.4 Einschwingphase, Abbruchkriterium, Konfidenzintervall, Varianz

Für die Analyse von Simulationsmodellen werden Simulationsexperimente durchgeführt. Diese Simulationsexperimente liefern Werte, die Simulationsergebnisse. Um über ein Systemverhalten Aussagen machen zu können, müssen die Simulationsergebnisse ausgewertet werden.

Für die Auswertung interessant sind die verschiedenen Kenngrößen wie Durchsatz (engl.: throughput), mittlere Verweilzeit (engl.: mean dwell time), mittlere Bedienzeit (engl.: mean service time) und mittlere Füllung (engl.: mean fill). Darüber hinaus können Abweichungen der Mittelwerte, Maximal- und Minimalwerte von Warteräumen, Wartezeiten (engl.: waiting time) und Bedienzeiten interessieren. In manchen Fällen ist auch das Fortschreiben von Werten wichtig, d.h., bei jeder Werteänderung wird der Zeitpunkt und der neue Wert abgefragt. In manchen Experimenten sind Akkumulationen von Werten gefragt, die dann angezeigt werden müssen. Wie all diese Werte abgefragt werden, hängt davon ab, welchen Formalismus man für die Implementierung des Simulators eingesetzt hat. Für Referenznetze wird in Kapitel 9 darauf eingegangen (für imperativ erstellte Simulatoren wie Desmo siehe z.B. [PLC00]).

Jeder Simulationslauf geht von einem Anfangszustand aus, der bei der Initialisierung des Modells hergestellt wird. In der Regel gilt das Interesse einer Simulationsstudie der stationären Zustandsverteilung eines Systems. Ein stochastischer Prozess wird als stationär bezeichnet, wenn die Wahrscheinlichkeit von X_t nicht von t abhängt. Viele stochastische Prozesse konvergieren gegen einen stationären Prozess. Wenn man ein Simulationsexperiment über einen längeren Zeitraum durchführt, stellt sich nach der Anlaufphase, auch Einschwingphase genannt, im allgemeinen ein stationärer Prozess ein, der unabhängig vom Anfangszustand ist und dessen Ergebnisse für die Auswertung herangezogen werden können. Ergebnisse der Einschwingphase (transiente Phase) dürfen nicht für die Auswertung herangezogen werden, da sie zu Extremwerten neigen können und die Analyse dadurch verfälschen. Es gibt keine exakten Verfahren zur Bestimmung des Beginns des stationären Prozesses, es existieren lediglich heuristische Verfahren. Glaubt man den Beginn der stationäre Phase gefunden zu haben, gibt es statistische Testverfahren, die diesen Zeitpunkt bestätigen oder

widerlegen können (vgl. [Pag92, 111-118]).

Damit die Simulationsergebnisse für die Analyse herangezogen werden können, muss eine ausreichende Anzahl von Simulationsdurchläufen durchgeführt werden. Es wird also untersucht, wie sich die Anzahl der Simulationsdurchläufe bei einer stochastischen Simulation auf die Genauigkeit der gewünschten Resultate auswirkt. Zur Vereinfachung wird hierzu eine zufällige Ergebnisvariable herangezogen, beispielsweise X, von der der Erwartungswert $E[X] = \sum_{k\in\Omega} k \cdot P(X = K)$, mit Ω = Zufallszahlenraum, ermittelt werden soll. Zunächst wird die Anzahl der erforderlichen Simulationsdurchläufe bestimmt, wenn $E[X]$ bis auf einen vorgegebenen Fehler genau berechnet werden soll (Abbruchkriterium). Danach wird das Konfidenzintervall für den Erwartungswert $E[X]$ angegeben. Zum Schluss werden Möglichkeiten angegeben, die Anzahl der benötigten Simulationsdurchläufe mittels geeigneter statistischer Techniken zu verringern (varianzreduzierende Methoden).

Der folgende Abschnitt orientiert sich im Wesentlichen an [Pag92, S.121-143] und [Neu77].

Führt man N Simulationsdurchläufe durch, erhält man für die Ergebnisvariable X die Stichprobenwerte $x_1, ..., x_N$. Als Schätzwert für den Erwartungswert $E[X]$ kann man dann das arithmetische Mittel der x_i,

$$\bar{x} := \frac{1}{N} \sum_{i=1}^{N} x_i,$$

verwenden. Der Stichprobenumfang sollte so bestimmt werden, dass mit der Sicherheit von $Q\%$ der absolute Fehler $|\bar{x} - E[X]|$ höchstens gleich einer vorgegebenen Fehlerschranke F ist. Wird der Stichprobenwert x_i entsprechend den Stichprobenvariablen mit $X_i (1 \leq i \leq N)$ bezeichnet und sei analog

$$\bar{X} := \frac{1}{N} \sum_{i=1}^{N} X_i,$$

dann soll

$$P(|\bar{X} - E[X]| \leq F) = \frac{Q}{100}$$

gelten. Im Allgemeinen wird N genügend groß gewählt ($N \geq 50$), sodass $\bar{X}$ auch näherungsweise als $\left(E[X], \frac{\sigma(X)}{\sqrt{N}}\right)$-normalverteilt angesehen werden kann. Es muss dabei allerdings gewährleistet sein, dass die Stichprobenvariablen X_i voneinander unabhängig sind. $\sigma(X)$, oder kurz σ, ist hier die Standardabweichung von X. Dabei bezeichnet var$X$ die Varianz der Zufallsvariablen X, d.h. die erwartete quadratische Abweichung vom Erwartungswert:

$$\sigma^2 = \text{var}X := E[(X - E[X])^2]$$

Dann gilt

$$P(|\bar{X} - E[X]| \leq \lambda_{Q\%} \frac{\sigma}{\sqrt{N}}) = \frac{Q}{100}$$

mit $\lambda_{Q\%}$ als $Q\%$-Grenze der $(0,1)$-Normalverteilung. Diese Gleichung ist auf jeden Fall dann erfüllt, wenn

$$\lambda_{Q\%}\frac{\sigma}{\sqrt{N}} \leq F$$

und damit

$$N \geq N^* := \frac{\sigma^2\lambda_{Q\%}^2}{F^2}$$

ist.

Eine Halbierung des maximalen absoluten Fehlers F erfordert demnach eine Vervierfachung der benötigten Zahl der Simulationsdurchläufe. Zu beachten ist, dass die Gleichung nur dann gilt, wenn die Standardabweichung σ von X bekannt ist. Bei unbekanntem σ wird statt σ^2 der Schätzwert s^2 betrachtet:

$$s^2 := \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

Es soll abschließend noch bemerkt werden, dass anstelle des maximalen absoluten Fehlers F häufig auch der relative Fehler $f := \frac{F}{E[X]}$, oder bei unbekanntem $E[X]$ auch $f := \frac{F}{\bar{x}}$, verwendet wird.

Zur genauen Interpretation des Simulationsergebnisses $\bar{x}$ und zur Beantwortung der Frage, wie genau die gesuchte Größe $E[X]$ durch das arithmetische Mittel $\bar{x}$ der Stichprobenwerte $x_1, ..., x_n$ angenähert werden kann, ist es sinnvoll, ein Konfidenzintervall (Vertrauensintervall) für $E[X]$ anzugeben. Hierzu sind gewisse Kenntnisse über die Verteilung von X bzw. $\bar{X}$ erforderlich. Ist X normalverteilt, was etwa mit Hilfe des χ^2-Test (siehe [Pag92]) oder durch Einzeichnen der Stichprobenwerte $x_1, ..., x_n$ in Wahrscheinlichkeitspapier o.Ä. nachgeprüft werden kann, so erhält man bei bekanntem σ für das $Q\%$-Konfidenzintervall

$$\bar{x} - \frac{\sigma\lambda_{Q\%}}{\sqrt{N}} \leq E[X] \leq \bar{x} + \frac{\sigma\lambda_{Q\%}}{\sqrt{N}}.$$

Ist die Anzahl der Simulationsdurchläufe genügend groß ($N \geq 50$), so darf $\bar{X}$ in der Regel auch als näherungsweise normalverteilt angesehen werden, wenn X keiner Normalverteilung genügt. Die Ungleichung lässt sich dann auch im Fall einer beliebigen Verteilung von X (die in der Regel unbekannt ist) verwenden.

Es ist aus den bisher gegebenen Gleichungen zu erkennen, dass die bei der vorgegebenen Fehlerschranke erforderliche Mindestanzahl N^* von Simulationsdurchläufen bzw. bei gegebener Zahl der Simulationsdurchläufe die Länge l des Konfidenzintervalls für den Erwartungswert $E[X]$ entscheidend von der Größe der Streuung (Varianz) $\sigma^2(X) = \mathrm{var}X$ der Ergebnisvariablen X abhängt. N^* ist dabei proportional zu $\sigma^2(X)$ und l proportional zu $\sigma(X)$. Eine Verkleinerung der Streuung von X bewirkt also eine Verbesserung der Genauigkeit der Simulationsergebnisse bzw. bei unveränderter Genauigkeit eine Verringerung des Rechenaufwandes.

Maßnahmen, die Streuung der Ergebnisvariablen zu reduzieren, sind unter dem Namen varianzreduzierende Methoden bekannt. Sie setzen in der Regel

relativ weitgehende Kenntnisse über die stochastischen Eigenschaften des betreffenden Problems voraus, die in der Praxis meist nicht gegeben sind, und beinhalten im Allgemeinen ganz spezielle, auf das Problem zugeschnittene statistische Techniken. Außerdem verursachen diese statistischen Techniken einen zusätzlichen Rechenaufwand, der die Einsparung durch die Verminderung der Anzahl der Simulationsdurchläufe z.T. wieder kompensieren kann. Aus diesen Gründen werden varianzreduzierende Methoden in der Praxis nur selten angewendet. Es soll hier deshalb nur eine einfache Möglichkeit der Varianzreduktion skizziert werden. Für weitere Methoden wird auf die Literatur verwiesen, z.B. [Pag92] oder [Neu77].

Bei vielen Operations Research Problemen sind zwei oder mehrere Alternativen miteinander zu vergleichen und die bezüglich eines gewissen Gütekriteriums Beste der Alternativen auszuwählen. In diesem Beispiel wird angenommen, dass die durch zwei verschiedene Alternativen bedingten Kosten den Ergebnisvariablen X bzw. Y einer stochastischen Simulation entsprechen. Es soll festgestellt werden, welche der beiden Alternativen zu geringeren mittleren Kosten führen, d.h., $E[X]$ und $E[Y]$ sind miteinander zu vergleichen. Mit je N Simulationsdurchläufen kann man dann Stichprobenwerte $x_1, ..., x_N$ bzw. $y_1, ..., y_N$ für X bzw. Y und damit Schätzwerte

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \text{ und } \bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i$$

für $E[X]$ bzw. $E[Y]$ erhalten. Unter der Annahme, dass die zu $\bar{x}$ und $\bar{y}$ gehörenden Stichprobenvariablen $\bar{X}$ und $\bar{Y}$ näherungsweise normalverteilt sind, ergibt sich das $Q\%$-Konfidenzintervall für die Größe $E[X] - E[Y] = E[X - Y]$ zu

$$\bar{x} - \bar{y} - \sqrt{\frac{\mathrm{var}(X-Y)}{N}}\lambda_{Q\%} \leq E[X] - E[Y] \leq \bar{x} - \bar{y} + \sqrt{\frac{\mathrm{var}(X-Y)}{N}}\lambda_{Q\%}.$$

Zwischen den gegebenen Alternativen gibt es einen entscheidenden Unterschied (bzgl. des Kostenkriteriums), wenn das in der Gleichung festgelegte Intervall nicht den Nullpunkt beinhaltet. Je kleiner die Varianz $\mathrm{var}(X - Y)$ ist, umso kleiner ist das Konfidenzintervall, d.h., umso eher ist bei einer geringeren Anzahl von Simulationsläufen ein Unterschied zwischen den beiden Alternativen feststellbar.

Werden die den beiden Alternativen entsprechenden Simulationen völlig unabhängig voneinander durchgeführt, dann können X und Y als stochastisch unabhängige Zufallsgrößen betrachtet werden, es gilt dann

$$\mathrm{var}(X - Y) = \mathrm{var}X + \mathrm{var}Y.$$

Werden jedoch alternativ die beiden Simulationen mit den gleichen Zufallszahlen durchgeführt und lässt man auch im Übrigen die beiden Simulationen – abgesehen von den verschiedenen Alternativen und deren Auswirkungen – in völlig gleicher Weise ablaufen, so sind die Zufallsgrößen X und Y voneinander abhängig, es gilt dann

$$\mathrm{var}(X - Y) = \mathrm{var}X + \mathrm{var}Y - 2\mathrm{cov}(X, Y).$$

Hierbei ist cov$X$ die Kovarianz der Zufallsvariablen X und Y:

$$\mathrm{cov}(X,Y) := E[XY] - E[X]E[Y]$$

Die Kovarianz beschreibt die Abhängigkeit der beiden Zufallsvariablen X und Y. Insbesondere impliziert die stochastische Unabhängigkeit von X und Y auch $\mathrm{cov}(X,Y) = 0$ – wohingegen die Umkehrung im Allgemeinen nicht gilt.

Vor allem wenn die beiden Alternativen nicht stark voneinander abweichen, sind die Zufallsvariablen X und Y positiv korreliert, also $\mathrm{cov}(X,Y) > 0$. Die zweite Möglichkeit liefert dann eine kleinere Varianz $\mathrm{var}(X-Y)$ und ist folglich der ersten Möglichkeit vorzuziehen. Falls die Größen var$X$, var$Y$, $\mathrm{cov}(X,Y)$ nicht bekannt sind, können sie auch durch die Schätzwerte

$$\frac{1}{N-1}\sum_{i=1}^{N}(x_i-\bar{x})^2, \frac{1}{N-1}\sum_{i=1}^{N}(y_i-\bar{y})^2 \text{ und } \frac{1}{N-1}\sum_{i=1}^{N}(x_i-\bar{x})(y_i-\bar{y})$$

ersetzt werden (vgl. [Neu77]).

8.4 Darstellung von Auswertungen

Neben der Richtigkeit der Simulationsergebnisse ist auch die Darstellung der Ergebnisse für die Analyse nicht unwichtig. Sie ist nicht so entscheidend wie die Aspekte der vorangegangenen Abschnitte, dennoch kann eine gute Darstellung der Simulationsergebnisse die Analyse eines Systems erleichtern.

Simulationsergebnisse sollten in erster Linie gut lesbar sein, d.h. sie sollten nicht in einem Text versteckt werden, sondern besser in Tabellenform dargestellt werden. Damit eine Tabelle gut lesbar ist, sollten im Quelltext sprechende Namen verwendet werden. Die verschiedenen Ankunftsprozesse, Bedienzeiten verschiedener Entitäten oder Warteschlangen sollten anhand ihres Namens eindeutig identifizierbar sein. Eine Möglichkeit wäre beispielsweise, in einer Tabelle eine Spalte für die verschiedenen Namen zu reservieren und weitere Spalten für die verschiedenen Werte, wie z.B. maximale oder durchschnittliche Länge einer Warteschlange.

Ist ein Simulationsmodell sehr groß, ist es meist sinnvoll, die Ergebnisse in verschiedenen Tabellen darzustellen, z.B. eine Tabelle für Ankunftszeiten, eine für Bedienzeiten und eine für Warteschlangen.

Tabellen sind ein gutes Medium, um Werte bequem abzulesen. Sie lassen aber keinerlei Interpretation der Ergebnisse zu. Um das zu erreichen, kann man Diagramme einsetzen. Ein Diagramm, in dem die Zugänge und Abgänge gegen die Zeit aufgetragen werden (Plot von $z(t)$, $a(t)$, $f(t)$), gibt sofort Auskunft über die Füllung des Systems zu jedem Zeitpunkt bzw. im zeitlichen Verlauf (vgl. Abb 8.5). In so einem Diagramm sind Engpässe gut zu erkennen, da die Füllung immer größer wird. Weitere Diagramme, die nützlich sind, sind Diagramme, bei denen der Durchsatz gegen die Verweilzeit einmal im Verlauf der Zeit und einmal in Abhängigkeit von der Füllung aufgetragen wird (Plot von $d(t)$, $y(t)$ und Plot $d(f)$, $y(f)$).

In [Pri84] wird vorgeschlagen, die Daten in Klassen oder Zellen zu gruppieren und in Tabellenform anzugeben. Beispielsweise kann die Wartezeit vor einer

Entität aufgeteilt werden, in kleine, mittlere, lange und sehr lange Wartezeiten. Den Unterschied zwischen dem oberen und dem unteren Klassenlimit nennt man Klassenweite. Zu den angegebenen vier Klassen kann dann die jeweilige Anzahl an z.B. Kunden in einer weiteren Spalte dazugeschrieben werden. Die Einträge in dieser Spalte nennt man die *Frequenz*. Die entstandene Tabelle nennt man deshalb *frequency distribution table*. Sie gibt im Allgemeinen einen guten Überblick über die Verteilung der Daten.

Waiting Time	Numbers of Customers
$0 \rightarrow 20$	21
$20 \rightarrow 40$	35
$40 \rightarrow 60$	42
$60 \rightarrow 80$	35
$80 \rightarrow 100$	19
$100 \rightarrow 120$	10
> 120	10

Tabelle 8.1: Die Anzahl der Ereignisse pro Klasse

Es gibt verschiedene Variationen der *frequency distribution tables*. Eine Variation, die hier noch kurz vorgestellt werden soll, ist die *cumulative frequency table*. Man erhält diese Form der Tabelle durch schrittweise Addition der Frequenzen.

Waiting Time	Numbers of Customers
20	21
40	56
60	98
80	133
100	152
120	167
∞	172

Tabelle 8.2: Die kumulierten Ereignisse

Die oben vorgestellten Tabellen lassen sich auch in Form eines Histogramms darstellen, in dem die Frequenzen als Rechtecke dargestellt werden, deren Länge proportional der Klassenfrequenz ist. In Abbildung 8.9 wird das zu den vorangegangenen Tabellen passende Histogramm gezeigt.

Die Schwierigkeit beim Erstellen solcher Frequenztabellen und Histogrammen ist, herauszufinden, welche Anzahl an Klassen benötigt wird und wo das obere und untere Klassenlimit jeder Klasse liegt. In [Pri84] werden dafür Richtlinien angegeben:

- Die Klassenweite sollte bei allen Klassen gleich lang sein, mit Ausnahme der ersten und der letzten Klasse, die offen sein können.
- Klassenintervalle sollten sich nicht überlappen und alle Daten sollten in eine, und nur in eine der angegebenen Klassen fallen.

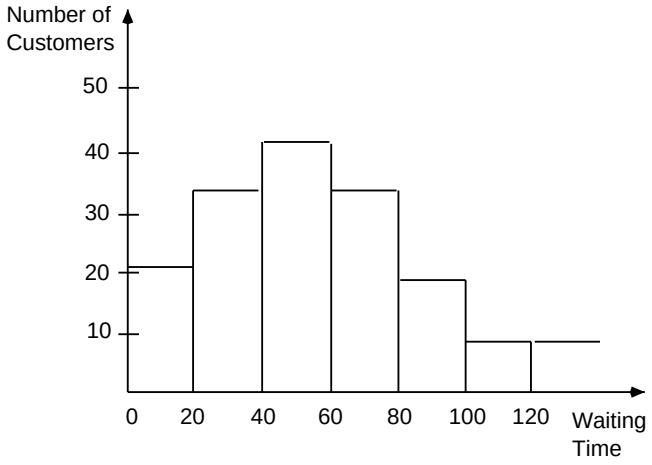

Abbildung 8.9: Das Histogramm eines Warteprozesses

- Im Allgemeinen werden mindestens fünf und höchstens zwanzig Klassen eingesetzt.

Abschließend zu diesem Abschnitt lässt sich feststellen, dass Tabellen ein geeignetes Mittel sind um viele verschiedene Werte übersichtlich festzuhalten und dass Diagramme ein geeignetes Mittel sind, um erste Interpretationen vornehmen zu können.

Fazit

In diesem Kapitel wurde beschrieben, was bei der Durchführung von Simulationsexperimenten und der Auswertung von Simulationsergebnissen zu beachten ist. Dazu wurden zunächst die Begriffe Verklemmungsfreiheit, Lebendigkeit und Fairness eingeführt und ihre Wichtigkeit in Bezug auf Petrinetzsimulationsmodelle erklärt. Danach wurde auf elementare Wartesysteme, Wartenetze und hierarchische Bediensysteme eingegangen. Es wurde dabei u.A. gezeigt, wie wichtig Wartesysteme für die Simulation sind, und wie man sie mit Petrinetzen umsetzen kann. Zusätzlich wurden die für die Analyse von Wartesystemen wichtigen Kenngrößen, wie Füllung, Durchsatz und Verweilzeit, erklärt. Danach wurde die Analyse elementarer Wartesysteme durch Markovprozesse und die Analyse von Wartenetzen erläutert. Später wurden die Begriffe Einschwingphase, Abbruchkriterium, Konfidenzintervall und Varianz eingeführt und erklärt. Diese vier Begriffe sind für die Durchführung von Simulationsexperimenten wichtig. Zum Schluss dieses Kapitels wurde zusätzlich gezeigt, wie sich Simulationsergebnisse geeignet darstellen lassen, damit eine Analyse der Ergebnisse möglich wird.

9 Auswertung von Simulationsmodellen in Renew

Nachdem gezeigt wurde, wie man Referenznetze und Renew zur Simulation einsetzen kann und im vorangegangenen Abschnitt die Analyse von Wartesystemen vorgestellt wurde, soll in diesem Kapitel auf die konkrete Simulationsauswertung mit Renew eingegangen werden. Dazu sollen zunächst die verschiedenen Möglichkeiten diskutiert werden, die Renew bietet, um Simulationsergebnisse zu gewinnen. Es existieren mehrere Möglichkeiten, um die Eintritts- und Austrittsereignisse aus dem Petrinetzmodell zu extrahieren. Entweder gewinnt man die Daten direkt aus dem Petrinetzprozess, so wie er durch Renew mitprotokolliert wird, oder die Ereignisse werden als Teil des Simulationsmodells mitverarbeitet. Danach werden die verwendeten Klassen, die für die Auswertung programmiert worden sind, vorgestellt. Es handelt sich dabei im Wesentlichen um Klassen, die Methoden enthalten, die als Transitionsanschriften die Zugänge und Abgänge im Netz mitprotokollieren. Um die Auswertung der Petrinetzsimulation zu verdeutlichen, wird das Objektnetz-Beispiel aus Kapitel 6 eingesetzt, um einen Simulationslauf mit entsprechender Auswertung durchzuführen. Da Renew grundsätzlich erweiterbar ist, kann überlegt werden, wie man die Auswertung in Renew vereinfachen kann. Es wäre beispielsweise sehr benutzerfreundlich, wenn es in der Renew Toolbar in einem Menü entsprechende Auswertungsbaussteine gäbe, die durch einfache Mausklicks eine Transition so verändern, dass sie bezüglich der Auswertung gewisse Eigenschaften erhält.

9.1 Extraktion aus dem Netzsimulationslauf

Wird in Renew ein Referenznetz simuliert, dann wird das Schalten jeder Transition im Renew-Output auf der Konsole mitprotokolliert. Es ist deshalb möglich, aus dem Renew-Output Ergebnisse abzugreifen. Das Netz muss zuvor lediglich mit Anschriften versehen werden, die die verschiedenen Zeitverbräuche, wie Zwischenankunftszeiten oder Bearbeitungszeiten, realisieren. Der Renew-Output ist allerdings sehr unkomfortabel, um konkrete Ergebnisse zu erhalten, da alles, was im Netz passiert, also jeder Schaltvorgang, fortlaufend mitprotokolliert wird, d.h. es werden auch Informationen gezeigt, die für die Auswertung u.U. uninteressant sind. Die gewünschten Informationen müssen in diesem Fall aus dem Renew-Output selbstständig herausgesucht werden.

Eine wesentlich komfortablere Lösung ist es, ausgewählte Transitionen mit Anschriften zu versehen, die entweder für einen Eintritt in eine Warteschlange oder für den Austritt aus einer Warteschlange verantwortlich sind. Diese Anschriften sollten dafür sorgen, dass wenn ein genanntes Ereignis eintritt, ein Statistikobjekt aufgerufen wird, das die Eintritts- und Austrittsereignisse ver-

waltet. Während eine Simulation in Renew läuft, wird die Rechenzeit mitprotokolliert. Aus den Eingangs und Ausgangsereignissen lassen sich dann mit Hilfe der Rechenzeit die Verweilzeit, der Durchsatz und die Füllung einer bestimmten Entität berechnen. Diese Ergebnisse können dann in einer Datei oder gesondert im Renew-Output gespeichert und jederzeit für eine Analyse herangezogen werden.

Abbildung 9.1 zeigt, wie die Transitionen, die die Zu- und Abgänge beschreiben, mit einem Statistikobjekt gekoppelt werden. Es handelt sich bei diesem Netz um das gleiche Wartesystem wie in Abbildung 8.1. Jedes Schalten der Transition *enter* in dem Netz führt die Aktion (*action*-Anschrift) *t = getSimulationTime())* am Statistikobjekt *so* aus. Mit *so.registerStart(x,t)* wird der Ereigniseintritt festgehalten. Für das Schalten der Transition *leave* gilt das analog, mit dem Unterschied, dass beim Festhalten des Ausgangsereignisses die Prozedur *so.registerStop(x,t)* eingesetzt wird.

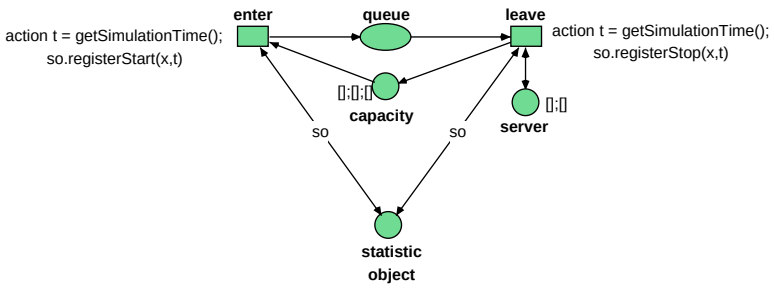

Abbildung 9.1: Kopplung von Zu- und Abgängen mit einem Statistikobjekt

Für ein Wartenetz wie in Abbildung 9.1 existiert pro Wartesystemknoten ein eigenes Statistikobjekt. Nachteilig ist hierbei, dass das Simulationsmodell mit den Auswertungsaktionen vermischt wird. Für ein Wartenetz sind zudem noch zusätzliche Stellen und Kanten zu zeichnen, die das Modell unübersichtlich machen.

Eine Möglichkeit, Statistikobjekte zu verwenden und die Übersichtlichkeit beizubehalten, ist der Einsatz von virtuellen Stellen (vgl. Abbildung 9.2).

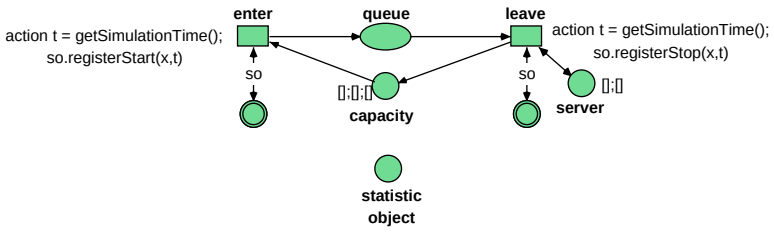

Abbildung 9.2: Kopplung von Zu- und Abgängen mit einem Statistikobjekt durch virtuelle Stellen

Durch den Einsatz von synchronen Kanälen kann die Kopplung aufgetrennt werden. Abbildung 9.3 zeigt die Verbindung von Zu- und Abgängen mit einem Statistikobjekt durch den Einsatz synchroner Kanäle. Auf der Stelle *statistic*

objekt werden u.U. verschiedene Statistikobjekte verwaltet. Die Transition *enter* gibt über den synchronen Kanal *enter* bekannt, welches Statistikobjekt benötigt wird (hier: *sys1*). Zusätzlich wird eine Kennung übergeben (hier: x), da alle Transitionen die gleichen synchronen Kanäle benutzen und der Aufruf eines Statistikobjektes richtig zugeordnet werden muss. Alle anderen Funktionen sind identisch mit dem oben genannten Beispiel. Man kann sich leicht vorstellen, dass mit dieser Lösung bei größeren Netzen an Übersichtlichkeit hinzugewonnen wird.

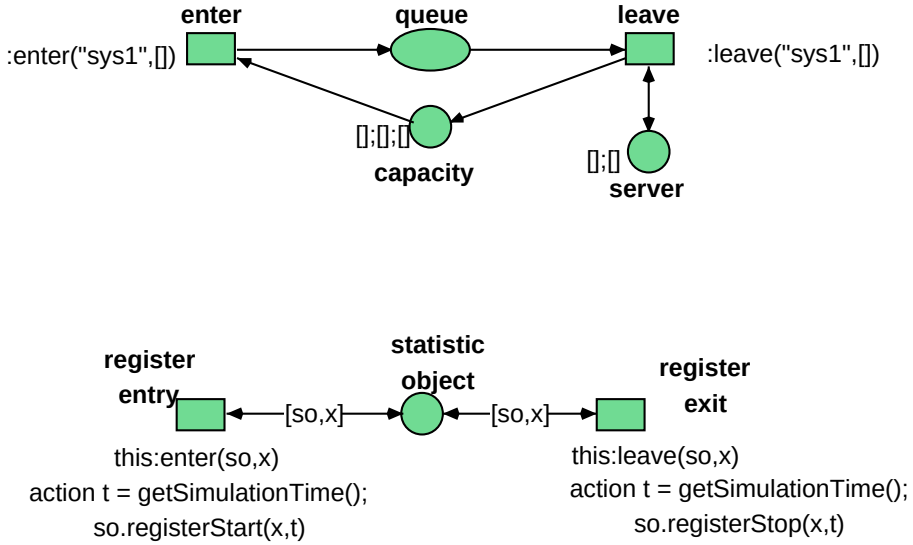

Abbildung 9.3: Kopplung von Zu- und Abgängen mit einem Statistikobjekt durch synchrone Kanäle mit einem Warteknoten

Da der Auswertungsmechanismus für alle Teilsysteme gleich ist, konnte in Abbildung 9.3 ein generischer Mechanismus vorgestellt werden, der durch das System (hier: *sys1*) parametrisiert ist. Es existiert daher auch für Wartenetze mit mehreren Knoten nur ein Bereich zur Statistikverwaltung (vgl. Abb. 9.4).

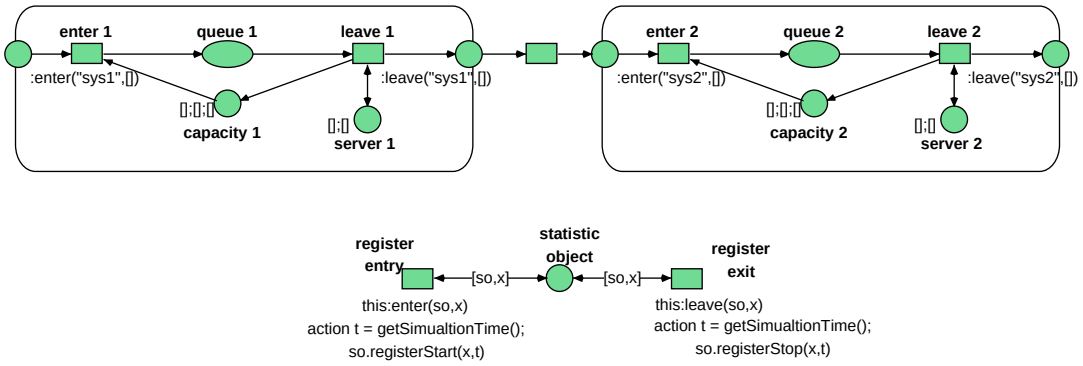

Abbildung 9.4: Kopplung von Zu- und Abgängen mit einem Statistikobjekt durch synchrone Kanäle mit zwei Warteknoten

Es wurde hier gezeigt, wie sich Auswertungsmechanismen für Referenznetze

in Renew integrieren lassen und welche Entwurfsalternativen existieren. Dazu wurden das Konzept des synchronen Kanals, das Konzept der *action*-Anschriften und das Konzept der virtuellen Stellen verwendet.

9.2 Auswertungsklassen

In diesem Abschnitt sollen die im Rahmen dieser Diplomarbeit programmierten und verwendeten Javaklassen zur Simulationsauswertung mit Referenznetzen in Renew vorgestellt werden. Es wird gezeigt, wie Warteschlangen realisiert wurden, wie Standardmaße damit gewonnen werden und wie die Ergebnisse dargestellt werden.

Der Entwurf und die Programmierung der Auswertungskomponente wurden im Rahmen eines Projektes mit großer Unterstützung von Heiko Rölke realisiert. Es wurden zwei Javaklassen programmiert, die für die Auswertung nötig sind: die Klasse *Auswertung* und die Klasse *ColoredTransitionEvent*. Die Klasse *Auswertung* beinhaltet Prozeduren *transInputEvent(...)*, *transOutputEvent(...)* und *simulationStop(...)*. Die beiden erstgenannten Prozeduren sind für die Verwaltung der Zu- und Abgänge im System zuständig, die dritte für die Ausgaben der Simulationsergebnisse. Die Klasse *ColoredTransitionEvent* beinhaltet im Wesentlichen Prozeduren und Funktionen für das Berechnen der Kenngrößen und die Darstellung der Ergebnisse. In Anhang B ist der vollständige Quellcode angegeben.

9.2.1 Warteschlangen

Wie im vorangegangenen Abschnitt schon gezeigt wurde, werden die Zugangs- und Abgangsereignisse durch *action*-Anschriften an den Transitionen realisiert. Es wird deshalb die Klasse *Auswertung* benötigt, die Prozeduren für Eintritts- und Austrittsereignisse (*transInputEvent(String key, Object value, double time)* und *transOutputEvent(String key, Object value, double time)*) bereithält.

Hier wird, um festzuhalten, dass ein neues Objekt ins System gekommen ist, die Transition mit *action awt.transInputEvent("Name des Objektes", value, SearchQueue.getTime())* beschriftet. Die Anschrift *action* ist eine schon in Kapitel 5 vorgestellte Anschrift. Das *awt* bedeutet, dass am Objekt *awt* vom Typ *Auswertung* die Prozedur *transInputEvent(String key, Object value, double time)* aufgerufen wird. Als Parameter werden also ein String als Key, der Wert des eingetroffenen Objektes und die Ereigniseintrittszeit übergeben. Mit *SearchQueue.getTime()* wird die momentane Simulationszeit extrahiert. Diese Prozedur ist bereits Bestandteil des Renew-Packages und kann für das Erfassen der Zeit hier eingesetzt werden.

An der Ausgangstransition wird die Prozedur *awt.tranOutputEvent(String key, Objekt value, double time)* aufgerufen.

Demzufolge wird die Zeit beim Eintritt eines Objektes in ein Wartesystem und beim Austritt aus dem Wartesystem festgehalten. In Abbildung 9.5 ist dargestellt, wie in Renew die Beschriftung der Transitionen aussehen muss, um ein Wartesystem zu realisieren.

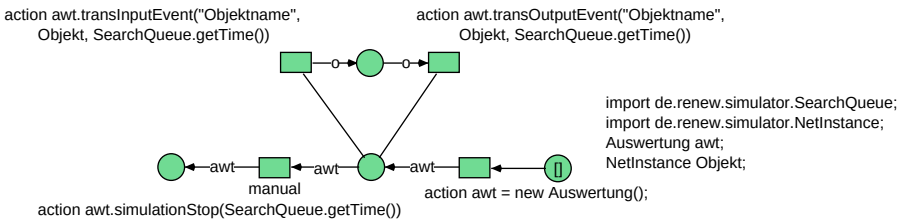

Abbildung 9.5: Ein einfaches Wartesystem mit einer Renew-Auswertungskomponente

Es ist darüber hinaus möglich, die Transitionsanschriften beliebig zu verschachteln, um eine Hierarchie von Wartesystemen zu realisieren. Ein großes Wartenetz kann beispielsweise mehrere kleinere enthalten, ein kleineres wiederum kleinere.

Ist ein Simulationslauf beendet, wird die Prozedur *awt.simulationStop(double time)* aufgerufen. Sie ist ebenfalls mit einer *action*-Anschrift an eine Transition gebunden. Sie darf nur einmal, am Ende einer Simulation, schalten. Sie veranlasst damit, dass eine Ergebnisausgabe produziert wird (vgl. Abschnitt 9.2.3. Die Transition wird manuell vom Benutzer ausgelöst (Anschrift „manual“).

9.2.2 Statistik

Für die Erzeugung der Simulationsergebnisse wird die Klasse *ColoredTransitionEvent* benötigt. Diese Klasse besitzt Prozeduren zur Berechnung von Kenngrößen. Die Prozedur *in(Object o, double t)* erhält den Zeitpunkt des Ereigniseintritts, zählt bei einem neuen Eintrittsereignis die Anzahl der Ereignisse um eins hoch und erhöht den Wert der momentanen Füllung. Dann wird die maximale Füllung mit der momentanen Füllung verglichen und die maximale Füllung gegebenenfalls korrigiert.

Die Prozedur *out(Object o, double oT)* erhält den Zeitpunkt des Ereignisaustritts, zählt die Anzahl der Austrittsereignisse (Durchsatz), ermittelt die momentane Füllung und berechnet die Verweilzeit des verlassenden Objektes (vgl. Kapitel 8.3.1). Zusätzlich werden die Zugänge und die Abgänge pro Zeiteinheit, die mittlere Füllung, die mittlere Verweilzeit, die Streuung und die Varianz der Verweilzeit berechnet (vgl. Kapitel 8.3.4).

Ein wesentlicher Bestandteil der Simulation ist das Erzeugen von Zufallszahlen. Für die Zwischenankunftszeiten von Objekten in ein System kann auf verschiedene Javafunktionen für Verteilungen von Zufallszahlen in einem bestimmten Intervall zurückgegriffen werden. Beispielsweise kann *Dist.negexp(int x)* benutzt werden. Diese Funktion kann als Kantenanschrift verwendet werden und so für die entsprechende Verteilung der Eintrittsereignisse sorgen. Sie ist ebenfalls bereits im Renew-Package enthalten. Für die Bedienzeiten gilt das analog. Es kann in Renew jede Javafunktion benutzt werden, es ist lediglich darauf zu achten, dass das entsprechende Java-Package in das Netz importiert wurde. In Abbildung 9.6 ist eine Exponentialverteilung mit $\lambda = 5$ für die Zwischenankunftszeiten eingesetzt worden. Die jeweilige Verteilung wird in Renew

wie auch deterministische Zeitverbräuche hinter einem @-Zeichen angegeben.

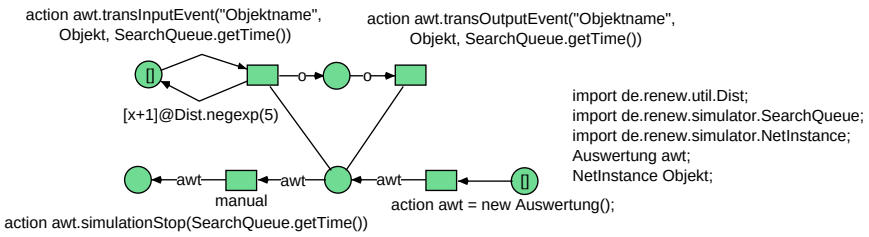

Abbildung 9.6: Eine Exponentialverteilung als Zwischenankunftszeit an einer Warteschlange

9.2.3 Darstellung

Nachdem Kenngrößen im Petrinetzsimulator berechnet wurden, müssen sie für die Analyse in geeigneter Form dargestellt werden. Dazu wird die Prozedur *simulationStop(double time)* der Klasse *Auswertung* benutzt.

Die Transition, an die die Ausgabekomponente gebunden werden soll (i.A. am Ausgang eines Systems), wird mit der Anschrift *action awt.simulationStop(SearchQueue.getTime())* versehen. Die Prozedur hält mit dem Zeitparameter fest, zu welchem Zeitpunkt die Simulation gestoppt wird. Zusätzlich ruft diese Prozedur die Prozedur *reportHTML(String title, double stopTime)* der Klasse *ColoredTransitionEvent* auf, die wiederum die Prozedur *analyseHTML(String title, double sT)* aufruft. Die Prozedur *analyseHTML(String title, double sT)* sorgt

dafür, dass eine HTML-Datei erstellt wird, die die Kenngrößen in Tabellenform ausgibt. Es werden die Größen Ereigniseintritt, Ereignisaustritt, Simulationszeit, mittlere Verweilzeit, mittlere Füllung, maximale Füllung, Durchsatz etc. (siehe oben) aufgelistet. Derzeit wird für jede Warteschlange eine HTML-Datei erzeugt[1]. Zusätzlich wird in der Prozedur *simulationStop(double time)* ein Plot der Füllung im Verlauf der Zeit und ein Plot der Verweilzeit in Abhängigkeit der Füllung erzeugt. Diese Diagramme sind, wie in Kapitel 8.4 diskutiert, ein geeignetes Mittel, um erste Aussagen über das Verhalten eines Systems zu machen.

Da die Transition *action awt.simulationStop(double time)* nur einmal zum Ende der Simulation schalten darf, wird sie zusätzlich mit der Anschrift *manual* versehen. Diese Anschrift gewährleistet, dass die entsprechende Transition nur manuell (per Mausklick o.ä.) schalten kann (vgl. Abb 9.5).

Um die Auswertungskomponente von Renew benutzen zu können, wurden die wesentlichen Aspekte in Teilen vorgestellt. Um die Benutzung und die Auswertung noch deutlicher zu zeigen, wird im folgenden Abschnitt ein Szenario simuliert, ausgewertet und ausführlich erklärt.

[1] Man kann überlegen, ob es sinnvoll wäre, alle Warteschlangen in einer HTML-Datei zu speichern; prinzipiell ist das möglich.

9.3 Auswertungsbeispiel

Da die einzelnen Komponenten der Renew-Auswertung bereits oben diskutiert wurden, werden sie jetzt eingesetzt, um konkret ein Szenario zu simulieren und auszuwerten. Es wird das bekannte OO-Szenario aus Kapitel 6 eingesetzt. In Abbildung 9.7 ist das Netz mit Integration der Auswertungskomponente gezeigt. Es wird darauf verzichtet, das LKW-Netz und das Gabelstapler-Netz darzustellen, da in diesen Netzen keine Auswertungskomponenten enthalten und deshalb keine Veränderungen aufgetreten sind. Es wird hierfür auf die Abbildungen 6.1 und 6.2 verwiesen.

An der Transition *lkw:new lkw* kommen neue LKW in das System. Es wird also an dieser Transition für jeden neu ankommenden LKW eine neue Netzinstanz vom Typ *lkw* erzeugt. In diesem Beispiel kommen die LKW nicht mehr deterministisch nach einem bestimmten Zeitintervall ins System, sondern die Ankunftszeiten sind hier exponential verteilt mit $\lambda = 5$ (*[]@Dist.negexp(5)*).

An der nächsten Transition fährt der neue LKW in eine Warteschlange. Damit erfasst wird, wann der LKW hineingefahren ist, wird die Transition mit *action awt.transInputEvent("LKW-Warteschlange", lkw, SearchQueue.getTime())* beschriftet. Diese Transition stellt damit den Eingang in die Warteschlange dar, da hier eine Ereigniseintrittsprozedur aufgerufen wird. Die folgende Transition ist mit *action awt.transOutputEvent("LKW-Warteschlange", lkw, SearchQueue.getTime())* beschriftet. Hier wird der Austrittszeitpunkt eines *lkw*-Objektes festgehalten. Es wird also an dieser Transition eine Ereignisaustrittsprozedur aufgerufen. Mit den Anschriften dieser beiden Transitionen kann die Warteschlange später analysiert werden. Der LKW kann nur aus der Warteschlange herausfahren, wenn ein Gabelstapler für dieser LKW bereit steht. Gibt es noch keinen Gabelstapler, muss der LKW warten. Gibt es sofort einen Gabelstapler, dann muss der LKW trotzdem eine Zeit lang warten, bis er weiterfahren darf (*lkw@Dist.negexp(6)*).

Wird die Füllung dieser Warteschlange immer größer, dann ist es evtl. nötig, zusätzliche Gabelstapler einzusetzen. An der gleichen Transition, an der sich der Warteschlangenausgang befindet, befindet sich der Eingang in eine Entladestation. Die Transition ist deshalb zusätzlich noch mit *action awt.transInputEvent("Entladevorgang", lkw, SearchQueue.getTime())* beschriftet. Die Zeit, die vergeht, bis der LKW entladen ist, ist in diesem Beispiel ebenfalls exponential verteilt, mit $\lambda = 10$ (*lkw@Dist.negexp(10)*). Danach verlässt der LKW die Entladestation wieder. Die Transition ist mit *action awt.transOutputEvent("Entladevorgang", lkw, SearchQueue.getTime())* beschriftet. Die Zeit, die vergeht, während ein LKW sich in der Entladestation befindet, kann als die Bedienzeit angesehen werden.

Beim Beladevorgang wiederholt sich die Vorgehensweise des Entladevorgangs. Die Ausgangstransition des Entladevorgangs ist gleichzeitig die Eingangstransition des Beladevorgangs (*action awt.transInputEvent("Beladevorgang", lkw, SearchQueue.getTime())*).

Die Bedienzeit ist hier exponential verteilt mit $\lambda = 12$ (*lkw@Dist.negexp(12)*). Die folgende Transition ist die Ereignisaustrittstransition (*action awt.transOutputEvent("Beladevorgang" lkw, SearchQueue.getTime())*)

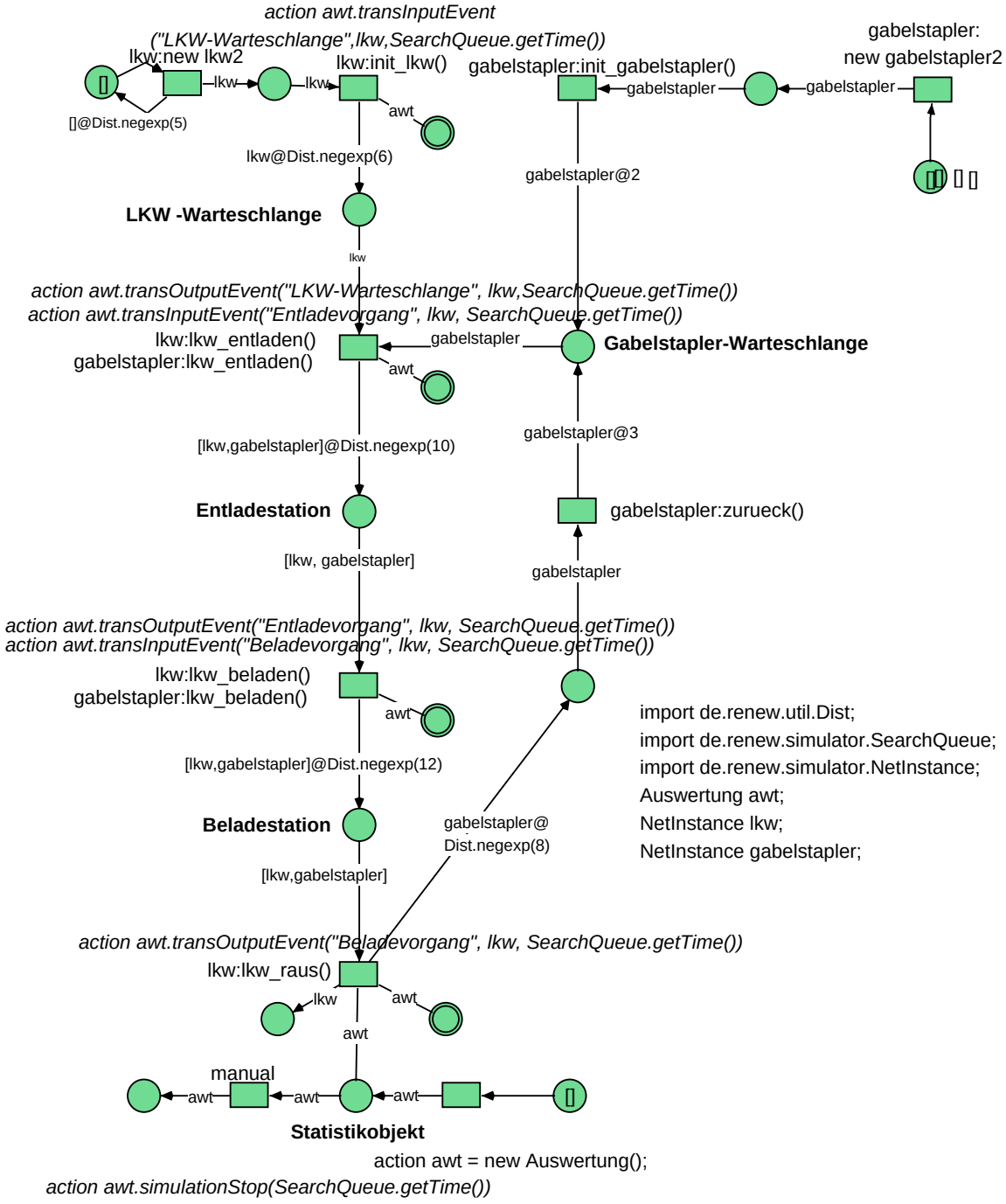

Abbildung 9.7: OO-Simulationsbeispiel

Unterhalb dieser Transition liegt eine Stelle, auf der ein *awt*-Objekt liegt. Das *awt*-Objekt wurde zu Beginn der Simulation einmal erzeugt (*action awt = new Auswertung()*) und danach in die Stelle *Statistikobjekt* gelegt. Die Stelle *Statistikobjekt* existiert als virtuelle Stelle an jedem Eintritts- und Austrittsereignis. Die Transition links neben der Stelle *Statistikobjekt* kann nur manuell per Mausklick geschaltet werden, weshalb sie mit der Anschrift *manual* versehen wurde.

Da an dieser Transition die Prozedur zum Stoppen und Auswerten der Simulation aufgerufen wird (*action awt.simulationStop(SearchQueue.getTime())*), darf sie nur ein einziges Mal am Ende der Simulation schalten. Hätte man auf die Anschrift *manual* verzichtet, dann würde die Transition schon zu Beginn des Simulationslaufs feuern[2]. Das hätte zur Folge, dass die Ergebnisdateien erstellt werden und das Netz nicht mehr schalten kann, es sich also in einem Deadlock befindet, da auf den virtuellen Stellen keine Marken bzw. kein *awt*-Objekte mehr liegen.

Alle anderen Eigenschaften des Netzes sind die gleichen wie die in dem Netz in Abbildung 6.3. Aus Übersichtlichkeitsgründen wurde lediglich auf die Kapazität der LKW-Warteschlange verzichtet.

Hat man einen Simulationslauf beendet, dann gibt Renew Gnu-Plots und HTML-Dateien aus. Die Gnu-Plots erscheinen selbstständig auf dem Bildschirm und zeigen für jedes Wartesystem die Füllung im Verlauf der Zeit und die Verweilzeit in Abhängigkeit von der Füllung an. In den Abbildungen 9.8, 9.9, 9.10, 9.11, 9.12 und 9.13 sind die Gnu-Plots für dieses Beispiel angezeigt.

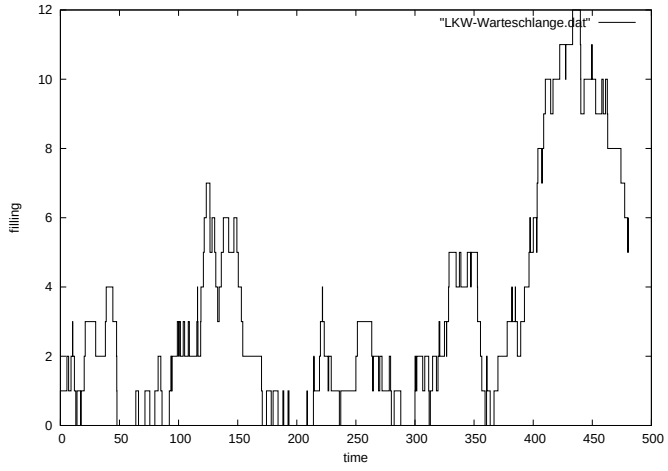

Abbildung 9.8: Die Füllung der Warteschlange im Verlauf der Zeit

Da die Füllung der LKW-Warteschlange immer größer wird und die mittlere Verweilzeit der LKW in dieser Warteschlange sehr groß ist, könnte bzgl. der Analyse des Systems überlegt werden, ob ein zusätzlicher Gabelstapler eingesetzt werden sollte.

Da es sich hier um ein Beispiel handelt, kann man diese Ergebnisse nicht zur Analyse einsetzen, da die Ergebnisse bzgl. der Einschwingphase, des Ab-

[2] nach dem Feuern der Transition mit dem Aufruf *action awt=new Auswertung();*

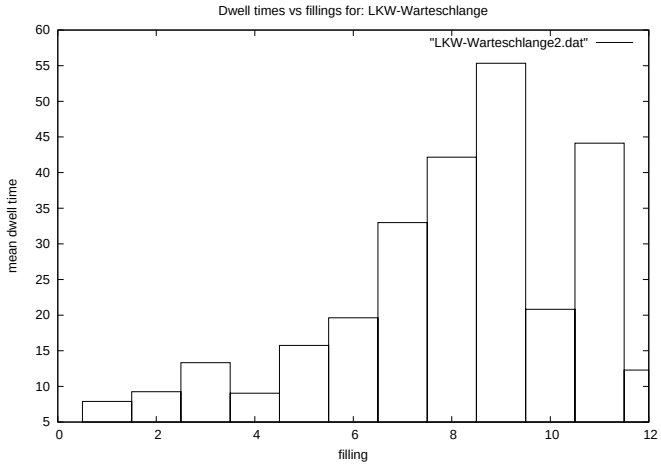

Abbildung 9.9: Die mittlere Verweilzeit in der Warteschlange in Abhängigkeit der Füllung

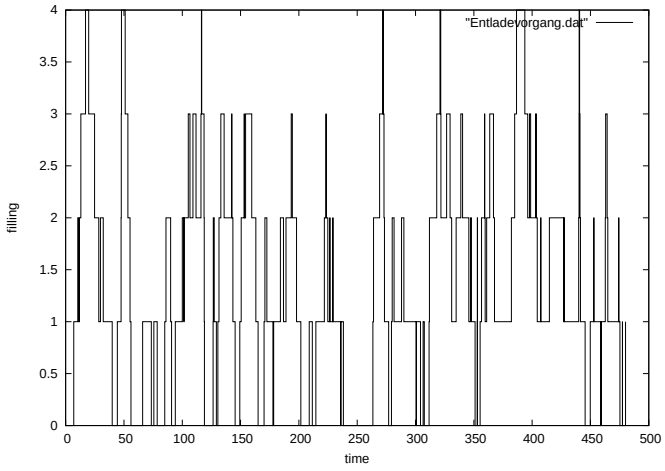

Abbildung 9.10: Die Füllung der Entladestation im Verlauf der Zeit

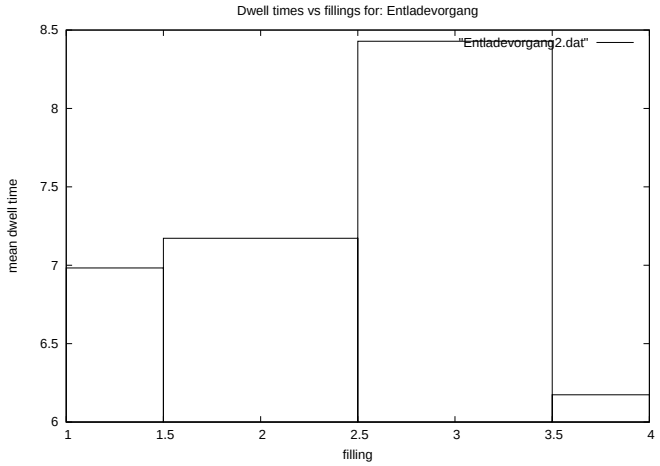

Abbildung 9.11: Die mittlere Verweilzeit in der Entladestation in Abhängigkeit der Füllung

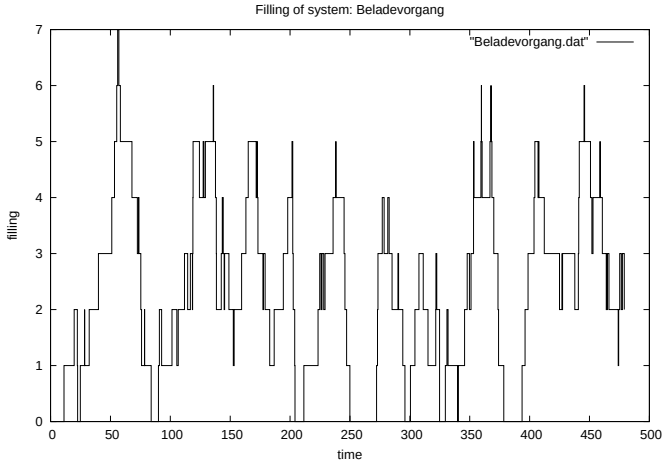

Abbildung 9.12: Die Füllung der Beladestation im Verlauf der Zeit

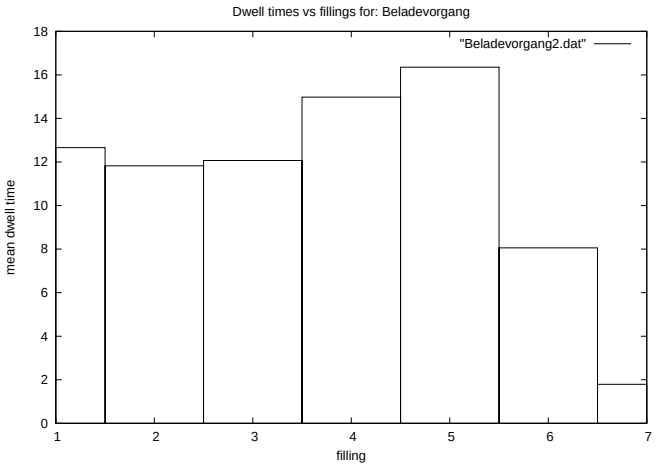

Abbildung 9.13: Die mittlere Verweilzeit in der Beladestation in Abhängigkeit der Füllung

bruchkriteriums und des Konfidenzintervalls hin überprüft werden müssten. Es müssten deshalb noch sehr viel mehr Simulationsläufe durchgeführt werden.

Die HTML-Dateien werden in dem Verzeichnis gespeichert, in dem auch das Netz gespeichert ist. Es gibt wiederum für jedes Wartesystem eine HTML-Datei. Die HTML-Dateien enthalten die genannten Kenngrößen tabellarisch aufgelistet. In Abbildung 9.14, 9.15 und 9.16 sind die HTML-Ausgaben für dieses Beispiel gezeigt.

Starting time	0.0
Stopping time	480.77884971766247
Simulation time	480.77884971766247
Total input	93
Total throughput	87
Maximal fill	12
Mean input	0.1934361298435118
Mean throughput	0.18095637953102717
Mean dwell time	14.670571807352676
Mean dwell time approximated variance	264.7193545697131
Mean dwell time approximated deviation	16.27019835680294
Mean fill	2.654733559908498

Abbildung 9.14: Kenngrößen der LKW-Warteschlange

9.4 Integration in die Benutzeroberfläche

Hat man sich das Beispiel-Szenario aus dem vorangegangenen Abschnitt klargemacht, kann man erkennen, dass bestimmte Komponenten immer wieder benutzt werden. Beispielsweise werden die Ereigniseintritts- und die Ereignisaustrittskomponente mehrfach benutzt. Ebenso wird die Exponentialverteilung häufiger eingesetzt. Das oben beschriebene Beispiel ist ein sehr kleines Szenario,

Starting time	0.0
Stopping time	480.77884971766247
Simulation time	480.77884971766247
Total input	87
Total throughput	86
Maximal fill	4
Mean input	0.18095637953102717
Mean throughput	0.17887642114561306
Mean dwell time	7.359362755725247
Mean dwell time approximated variance	55.76261860888688
Mean dwell time approximated deviation	7.46743721827555
Mean fill	1.3164164716564488

Abbildung 9.15: Kenngrößen des Entladevorgangs

Starting time	0.0
Stopping time	480.77884971766247
Simulation time	480.77884971766247
Total input	86
Total throughput	84
Maximal fill	7
Mean input	0.17887642114561306
Mean throughput	0.17471650437478484
Mean dwell time	12.933253184268612
Mean dwell time approximated variance	133.98153009416737
Mean dwell time approximated deviation	11.575039096874246
Mean fill	2.259652786549467

Abbildung 9.16: Kenngrößen des Beladevorgangs

weshalb es hier wenig Aufwand bereitet, die Transitionen bei der Entwicklung des Simulationsmodells selbständig mit den jeweiligen Anschriften zu versehen. In einem komplexeren Szenario, kann diese Vorgehensweise aber sehr mühsam und fehleranfällig werden. Es ist deshalb zu überlegen, ob nicht bestimmte immer wiederkehrende Komponenten in einem Menü oder als Button in Renew bereitgestellt werden sollten.

Eine Möglichkeit wäre beispielsweise, in Renew einen Transitionsbutton einzuführen, der ein elementares Wartesystem darstellt. In einem Menü sollten dann nur der Name und der Wert vom Entwickler angegeben werden müssen. Es würde sich bei so einer Transition um eine Faltung des schon vorgestellten Warteschlangennetzes handeln. In der Verfeinerung wäre ein elementares Wartesystem zu sehen, das aus zwei Transitionen und einer Stelle dazwischen besteht und die oben vorgestellten Transitionsanschriften trägt (vgl. Abb. 9.17). Eine solche Werkzeugunterstützung konnte bereits für den Kontext der Erstellung von Agentenprotokollnetzen (siehe Kapitel 7 und Anhang B) erstellt werden (siehe [Cab02, CMR03]).

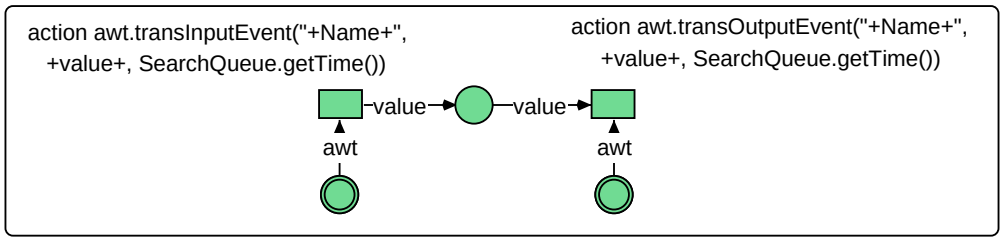

Abbildung 9.17: Eine Wartesystemkomponente mit virtuellen Stellen für das *awt*-Objekt

Für elementare Wartesysteme ist die Idee der Wartesystemtransition sicher nützlich. Problematisch wird diese Lösung aber dann, wenn es sich um hierarchische Bediensysteme HBS handelt. Da Simulationsszenarien im Allgemeinen sehr komplex sind, sind Hierarchien von Wartesystemen keine Seltenheit. Es muss deshalb eine Lösung gewählt werden, die auch HBS unterstützt. Eine Möglichkeit wäre, den Ereigniseintritt vom Ereignisaustritt zu trennen. Im Menü müsste bei dieser Lösung abgefragt werden, ob es sich um ein Ereigniseintritt oder -austritt handelt. Es wäre nützlich, wenn zusätzlich eine Fehlerüberprüfung stattfindet, damit ein Ereignis nicht zweimal in ein System eintritt, ohne es vorher verlassen zu haben, und umgekehrt (vgl. Abb. 9.18). Alternativ kann auch hier statt eines Menüs ein Button für eine Eintrittstransition und ein Button für eine Austrittstransition, gewählt werden.

Zwischenankunftszeiten und Bedienzeiten könnten ebenfalls in einem Menü zur Verfügung gestellt werden. Dafür könnte als erste Alternative ein zusätzlicher Kantentyp eingeführt werden, beispielsweise der Kantentyp *Exponentialverteilung*. Man müsste die Kante dann lediglich mit dem λ-parameter manuell beschriften. Bei dieser Lösung müssten konsequenterweise für jede in der Simulation gängige Verteilung Kantentypen eingeführt werden. Das kann dazu führen, das die Renew Toolbar unübersichtlich wird. Eine Alternative wäre, einen Kan-

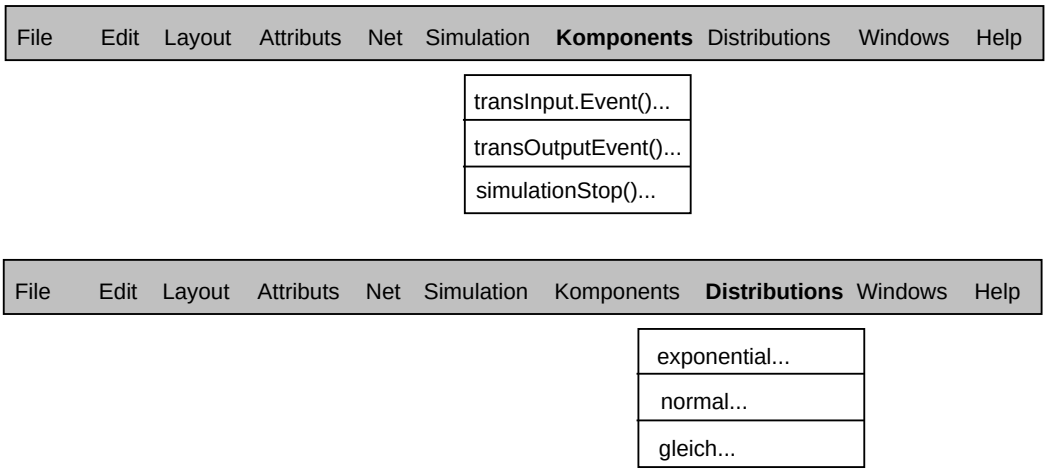

Abbildung 9.18: Ein Renew-Menü mit Prozeduren für die Simulationsprozeduren und die verschiedenen Verteilungen

tentyp *Verteilung* einzuführen. In einer Menüabfrage kann dann die gewünschte Verteilung ausgewählt werden (vgl. Abb. 9.18). Es müssen auch hier nur die notwendigen Parameter vom Entwickler angegeben werden.

Die Prozedur für das Beenden einer Simulation könnte ebenfalls in einem Menüpunkt zur Verfügung gestellt werden. Dort, wo das Eintritts- und das Austrittsereignis zur Disposition stehen, könnte auch das Simulationsendeereignis zur Auswahl bereitstehen. Dieser Menüpunkt darf in einem Netz nur ein einziges Mal aufgerufen werden. Renew könnte, falls es in einem Netz bereits ausgewählt wurde, diesen Punkt von dem Menü streichen, bzw. in heller Schrift darstellen und die Funktionalität sperren, wie allgemein üblich. Das würde gewährleisten, dass dieses Ereignis auch tatsächlich nur einmal verwendet wird (vgl. Abb 9.18). Auch hier kann alternativ über einen Simulationsendebutton nachgedacht werden.

Da für die Auswertung ein neues *awt*-Objekt erzeugt werden muss, wäre es sinnvoll, bei der Auswahl des Simulationsendeereignis aus dem Menü zusätzlich auch ein neues *awt*-Objekt zu erhalten. Die Testkanten zu den Eintritts- und Austrittsereignissen müssen nicht vom Entwickler gezeichnet werden. Man kann diesen Punkt mit Hilfe von virtuellen Stellen oder durch den Einsatz synchroner Kanäle lösen, die schon in den genannten Ereignissen integriert sind. Für diese Alternative wäre ein Button nötig, der den Netzteil abbildet, in dem das *awt*-Objekt erzeugt wird, das Simulationsendeereignis auftritt und die Stelle, aus der sich die virtuellen Stellen in den Warteschlangenkomponenten ableiten (vgl. Abb. 9.19).

Ist eine Simulation beendet, wäre es schön, wenn Renew auch die Darstellung bereithalten würde. Nützlich wäre beispielsweise ein Button oder ein Menüpunkt *Ergebnisse* o.ä., der den Namen des Netzes abfragt und dann die entsprechenden Ergebnistabellen und Diagramme zeigt. Dieser Menüpunkt ist in etwa mit dem Menüpunkt *Open File* zu vergleichen, nur mit dem Unterschied, dass der Speicherplatz, an dem die Ergebnisse abgelegt werden, immer gleich bleibt, es

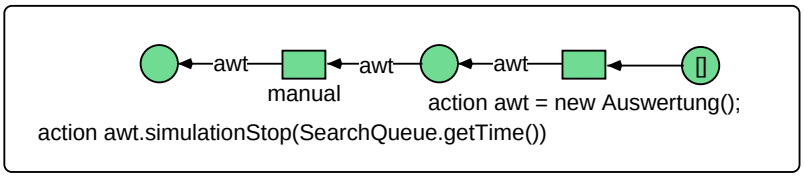

Abbildung 9.19: Eine zusammengefasste Simulationsendeereignis-Komponente

muss also kein Pfad angegeben werden. Es sind selbstverständlich auch hier Alternativen möglich.

Die Punkte zur Vereinfachung der Simulation mit Renew sind bislang nur Ideen, wie Renew in Zukunft noch erweitert werden könnte. In dieser Arbeit sollte nur gezeigt werden, wie eine Auswertungskomponente generell aussehen kann und wie sie einzusetzen ist.

Fazit

In diesem Kapitel wurde gezeigt, wie sich eine Auswertungskomponente für den Petrinetzeditor und -simulator Renew realisieren lässt. Es wurden deshalb zunächst die verschiedenen Möglichkeiten beschreiben, Ergebnisse aus einem Simulationslauf in Renew zu extrahieren. Danach wurde beschrieben, wie sich Warteschlangen mit Renew umsetzen lassen, und was eine Auswertungkomponente beinhalten müsste. Danach wurde die, speziell für Renew, implementierte Auswertungskomponente beschrieben und erklärt, wie man sie in einem Netz, das mit Renew erstellt wurde, benutzt. Es handelt sich bei der Auswertungskomponente um zwei Klassen, die in Java geschrieben wurden. Zum Schluss des Kapitels wurde an einem Beispiel erklärt, wie die implementierte Auswertungskomponente benutzt wird, und welche Ergebnisse sie in welcher Darstellung liefert.

10 Zusammenfassung und Ausblick

In dieser Arbeit wurde die Frage diskutiert, wie sich Petrinetze in der Modellierung und zur Simulation einsetzen lassen. Dazu sollte der Bezug zwischen Petrinetzen, Modellierung und Simulation herausgearbeitet werden. Zusätzlich wurde eine Auswertungskomponente für den Einsatz von Petrinetzen in der Simulation entwickelt und für den Petrinetzeditor Renew umgesetzt.

10.1 Zusammenfassung

Zunächst wurden in Kapitel 2, einer allgemeinen Einführung in das Gebiet der Simulation, die beiden für die Simulation wichtigen Begriffe System und Modell erklärt. Danach wurde die zeitdiskrete Simulation mit ihren verschiedenen Sichtweisen diskutiert. Es wurde dabei festgestellt, dass die zeitdiskrete Simulation eine weitverbreitete Technik ist, die zum Verstehen und Analysieren von Systemverhalten eine immer wichtigere Rolle einnimmt.

Da die Simulation ein ständig wachsendes Einsatzgebiet aufweisen kann, wird es immer wichtiger, eine Möglichkeit zu finden, Simulationsmodelle möglichst effizient entwickeln zu können. D.h. es soll möglich sein, Simulationsmodelle in kurzer Zeit entwickeln zu können, ohne vertiefte Programmierkenntnisse besitzen zu müssen. In dieser Arbeit wurde deshalb in Kapitel 3 der Einsatz von Petrinetzen vorgeschlagen.

Petrinetze besitzen einen grafisch-mathematischen Formalismus und sind deshalb in zweierlei Hinsicht zur Modellierung von Simulationsmodellen geeignet.

Aufgrund ihrer grafischen Darstellung ist es möglich, ein Systemverhalten aufzuzeichnen. Da eine Graphik eine große Menge an Information kompakt darstellen kann, erhält man ein erstes Modell in sehr kurzer Zeit. Anhand dieser graphischen Darstellung können dann alle Prozesse innerhalb eines Systems detailliert betrachtet und optimiert werden.

Der mathematische Hintergrund garantiert bei der Entwicklung von Simulationsmodellen die eindeutige Verwendung von grafischen Symbolen (Kreise, Rechtecke, Pfeile), die eine einfache Syntax darstellen und zudem ein eindeutiges Verhalten beschreiben. Darüber hinaus existieren erweiterte Petrinetzformalismen, die durch ihren jeweiligen mathematischen Hintergrund ein bestimmtes Systemverhalten gewährleisten oder Sachverhalte kompakter ausdrücken können. Es kann beispielsweise die Objektorientierung oder ein bestimmtes zeitliches Verhalten integriert werden. In Kapitel 4 werden erweiterte Petrinetzformalismen, höhere Petrinetze genannt, vorgestellt.

Zusammenfassend lässt sich feststellen, dass der Einsatz von höheren Petrinetzen zur Modellierung von Systemen eine sinnvolle Methode darstellt, da Petrinetze eine recht überschaubare Syntax besitzen und deshalb intuitiv in kurzer Zeit erstellt werden können.

Um ein Petrinetz zu entwickeln und es danach simulieren zu können, benötigt man einen Petrinetzeditor, der einen Simulator beinhaltet. In dieser Arbeit wurde in Kapitel 5 der Petrinetzeditor Renew vorgestellt und eingesetzt, der von Olaf Kummer, Frank Wienberg und Michael Duvigneau an der Universität Hamburg entwickelt wurde. Renew ist ein Petrinetzeditor, der speziell Referenznetze gut unterstützt. Referenznetze sind höhere Petrinetze, die ebenfalls von Olaf Kummer entwickelt wurden und die einen objektorientierten Modellierungsstil erlauben. Darüber hinaus besitzt Renew einen Simulator, der es erlaubt, das dynamische Verhalten von Petrinetzen darzustellen.

In der Simulation ist das Gebiet der Objektorientierung ein wichtiges Thema, da sich der Ursprung der Objektorientierung in der Simulation wiederfindet. In beiden Fällen betrachtet man Entitäten bzw. Objekte, ihr Verhalten und ihre Beziehungen zueinander. Simulation und Objektorientierung sind demnach untrennbar miteinander verbunden. Da auch Referenznetze eng mit der Objektorientierung verbunden sind, liegt es nahe, Simulationsmodelle mit Referenznetzen zu erstellen. Es wurden in dieser Arbeit deshalb ein Simulationsmodell als Referenznetz entwickelt (vgl. Kapitel 6).

Da es in der Simulation wichtig ist, Modelle möglicht realitätsnah zu entwickeln, wurde die agentenorientierte Sichtweise eingeführt. Agenten verfügen im Gegensatz zu Objekten u.A. über die Eigenschaft der Autonomie und des Wissens. Dadurch ist es Agenten möglich selber Entscheidungen zu treffen, was wiederum dazu führt, das Szenarien u.U. realitätsnäher modelliert werden können. Nach einer allgemeinen Einführung in Agenten und Multiagentensysteme wurde diskutiert, wie Petrinetzagenten eingesetzt werden können, um ein Simulationsszenario abzubilden. Dazu wurde Mulan, eine von Heiko Rölke ebenfalls an der Universität Hamburg entwickelte Multiagentensystem-Architektur verwendet. Es hat sich dabei gezeigt, dass auch sehr komplexe Szenarien, die mit imperativen Programmiersprachen nur unter sehr großem Zeitaufwand zu implementieren sind, recht schnell und sauber wiedergegeben werden können (vgl. Kapitel 7).

Es ist bereits deutlich geworden, dass die Simulation mit Petrinetzen, speziell mit Referenznetzen, ein geeignetes Mittel ist, um das Verhalten eines Systems zu modellieren und zu verstehen. D.h. es können Aussagen bezüglich seines Verhaltens gemacht werden. Es ist aber darüber hinaus für einen guten Simulator wichtig, auch quantitative Analysen durchführen zu können. Dazu ist es notwendig, dass eine Auswertungskomponente für die quantitative Analyse existiert. Für Renew wurde in dieser Arbeit eine Auswertungskomponente entwickelt, da diese noch nicht existierte.

Für die Analyse von Systemen ist es wichtig zu wissen, welche Daten interessant sind und welche nicht. Deshalb wurde in dem Kapitel 8 auf die grundsätzliche Analyse von Systemen eingegangen. Es wurde herausgearbeitet, welche Aspekte bzgl. der Analyse schon bei der Modellierung beachtet werden sollten, wie z.B. Verklemmungsfreiheit, Lebendigkeit und Fairness. Es wurde diskutiert, welche Kenngrößen (Füllung, Verweilzeit, Durchsatz etc.) berechnet werden müssen, um Aussagen über ein System machen zu können und es wurde gezeigt, was während eines Simulationsexperiments beachtet werden muss (Einschwingphase, Abbruchkriterium, Varianzreduktion).

Nachdem diskutiert wurde, welche Aspekte bei der Analyse zu beachten sind, wurde in Kapitel 9 eine Auswertungskomponente für Renew integriert. Es hat sich gezeigt, dass es sich dabei im Wesentlichen um die Behandlung von Wartesystemen handelt. Es wurden deshalb Klassen geschrieben, die Methoden enthalten, die die Zu- und Abgänge mitprotokollieren. Diese Methoden werden dann mit Hilfe von Transitionsanschriften aufgerufen. Zusätzlich wurde eine Klasse zum Berechnen der Kenngrößen und zur Darstellung der Ergebnisse geschrieben. Die Ergebnisse werden in Form von Diagrammen und Tabellen ausgegeben.

Das Ergebnis dieser Arbeit ist ein Petrinetzeditor, der ein mit Referenznetzen modelliertes Szenario simulieren kann und eine umfangreiche quantitative Ausgabe der Simulationsergebnisse liefert.

10.2 Ausblick

Zusätzlich kann man darüber nachdenken, wie Renew, erweitert um die Auswertungskomponente, zur Analyse von GSPNs herangezogen werden kann. Es ist bis jetzt nur möglich, Mittelwerte auszurechnen; man berechnet bei den GSPNs aber Erwartungswerte. Die Auswertungskomponente von Renew müsste also dahingehend verändert bzw. ergänzt werden, dass Markovketten ausgerechnet werden können. Dies könnte ein eigenes Modul in Renew sein oder durch einen Export zu einem Analysewerkzeug wie z.B. GreatSPN realisiert werden.

Ein anderes Einsatzgebiet für Renew mit Auswertungskomponente ist die Agentensimulation. In Kapitel 7 wurde bereits *Mulan* vorgestellt (siehe auch Anhang A). Es wäre möglich, die Auswertungskomponente in Mulan als Performance-Meter einzusetzen. Das hätte den Vorteil, dass man anhand von Ergebnissen erkennen kann, welche Auswirkungen unterschiedliche Vorgehensweisen bei der Modellentwicklung haben.

Es kann zusätzlich überlegt werden, ob der Einsatz von Renew mit der Auswertungskomponente auch für die Workflow-Engine Renews eingesetzt werden kann, da auch für Workflows die Performance von Modellen bzw. Szenarien im Vordergrund steht.

Darüber hinaus kann Renew noch mit Grafiken oder Makros erweitert werden, sodass ein umfangreiches objektorientiertes Simulationswerkzeug entsteht, das Prozesse nicht nur simuliert, sondern auch visualisiert und animiert. Durch den Einsatz von Grafiken bekäme man ein Simulationswerkzeug, das Stellen nicht nur als Kreise, Transitionen als Rechtecke und Marken als Klammerpaar abbildet, sondern jede Entität als Grafik so darstellt, wie sie in der Realität wirklich aussieht. Beispielsweise könnte eine Transition als Maschine, eine Stelle als Parkplatz und eine Marke als LKW dargestellt werden. Das Ergebnis wäre ein realitätsnahes, animiertes objektorientiertes Simulationswerkzeug.

A Mulan

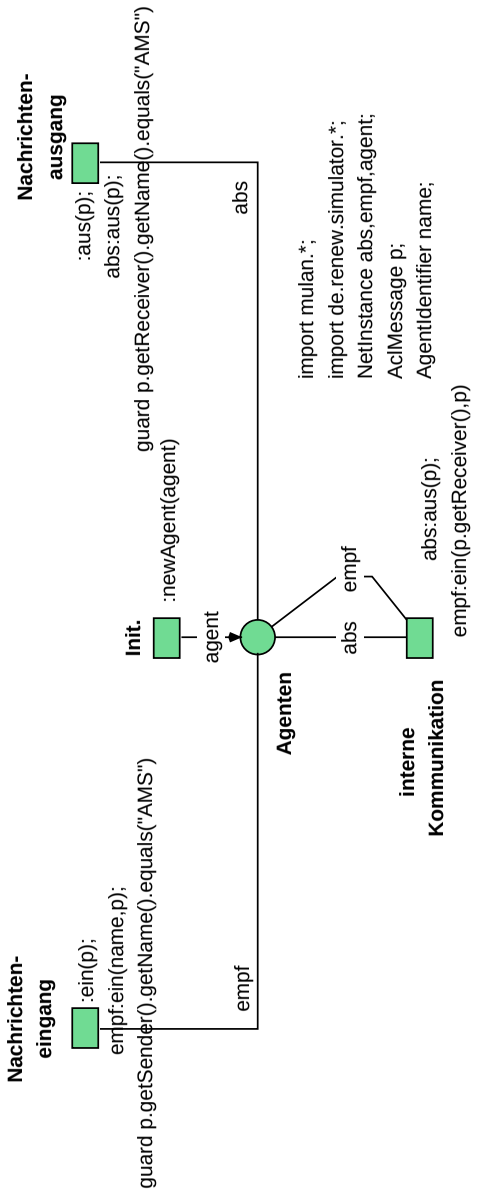

Abbildung A.1: Die Agentenplattform

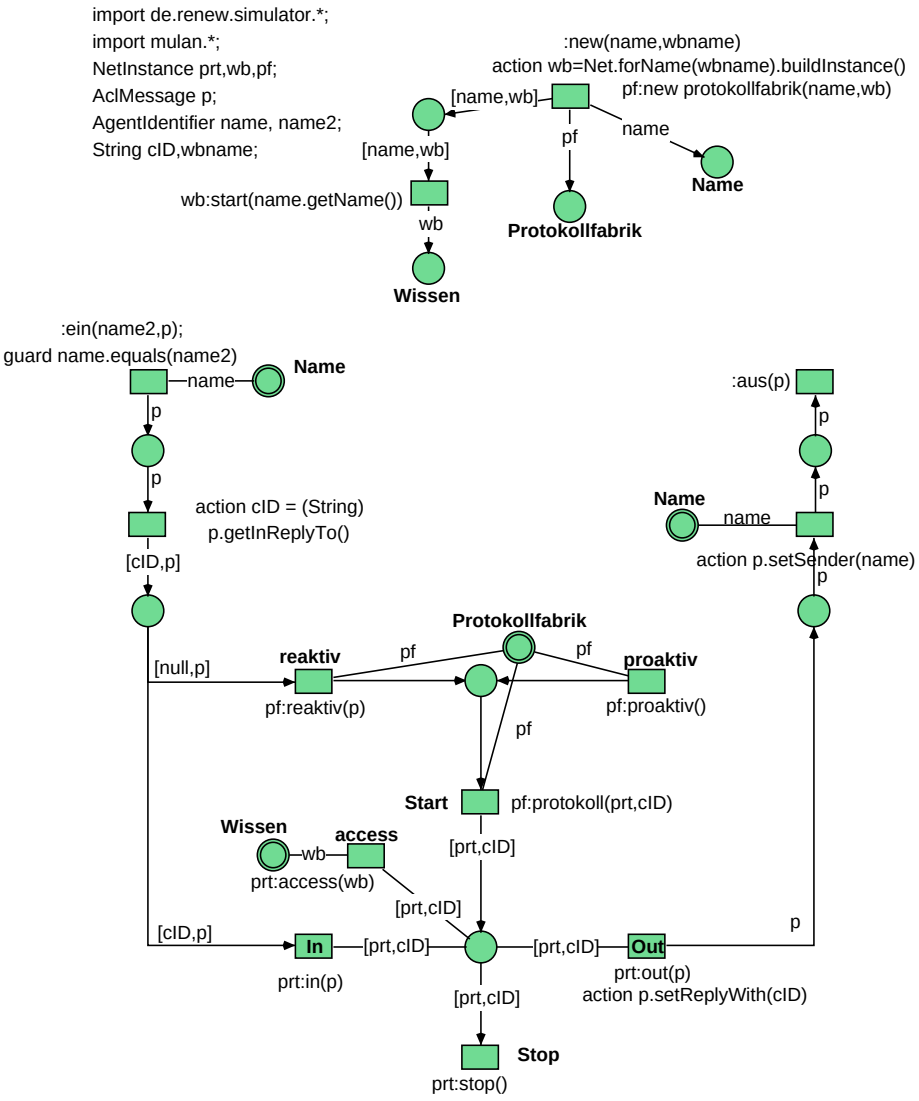

Abbildung A.2: Der Agent

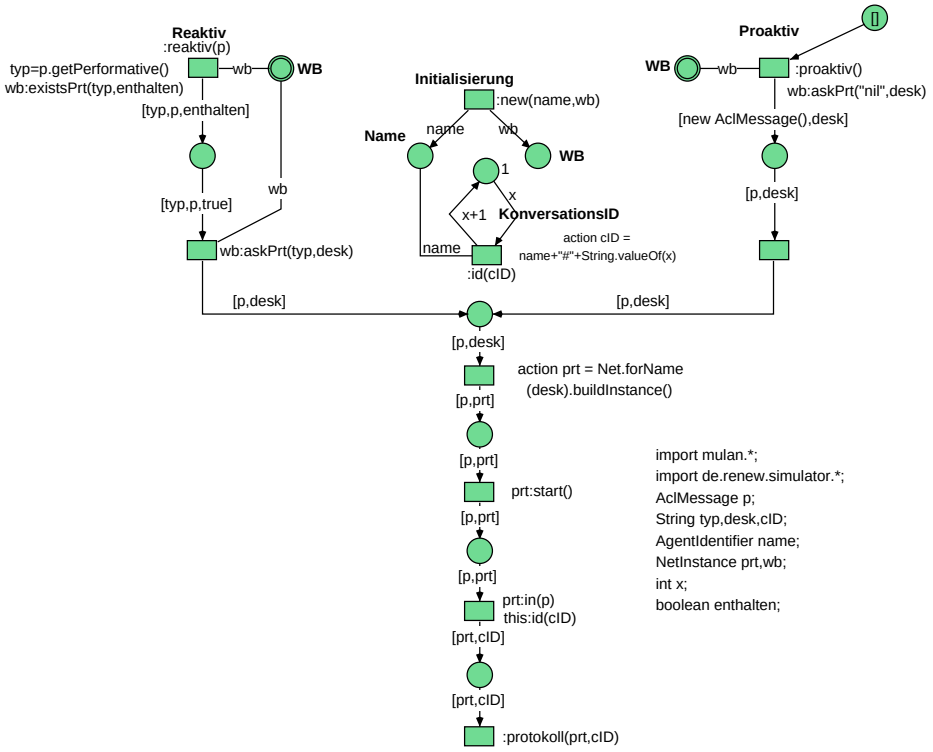

Abbildung A.3: Die Protokollfabrik

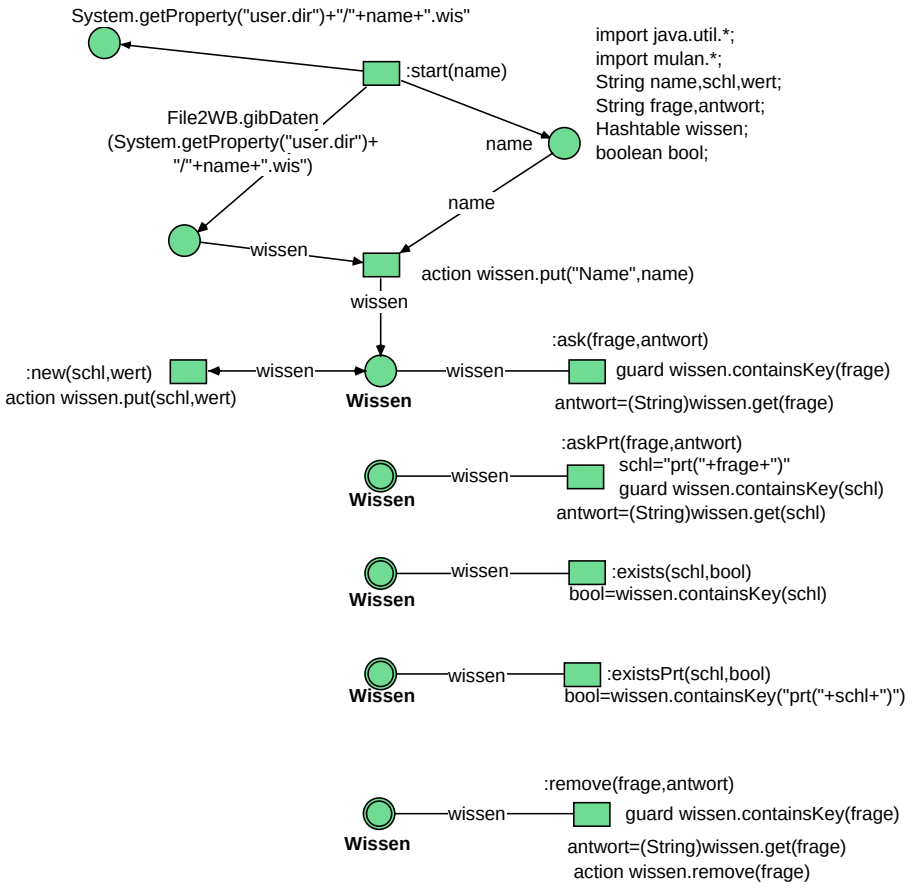

Abbildung A.4: Die Wissensbasis

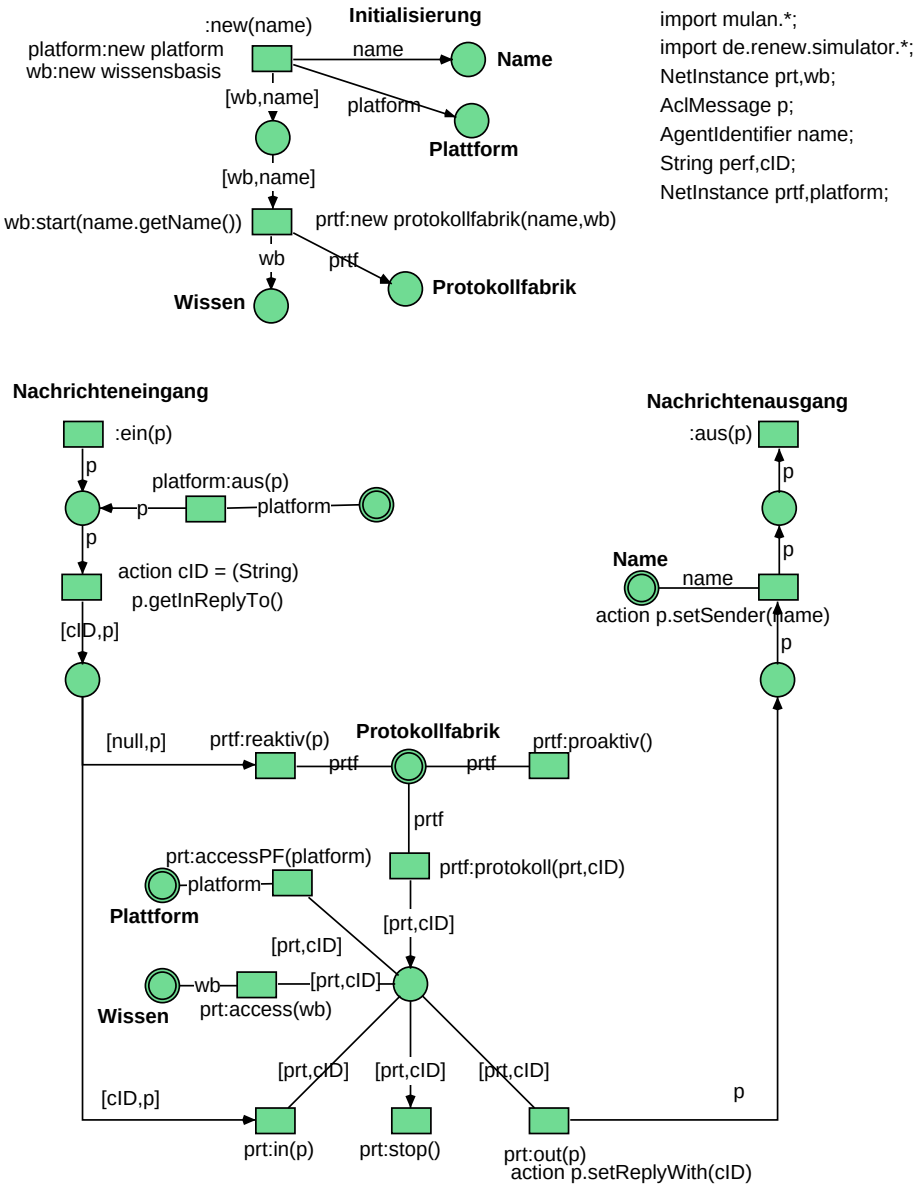

Abbildung A.5: Der AMS-Agent

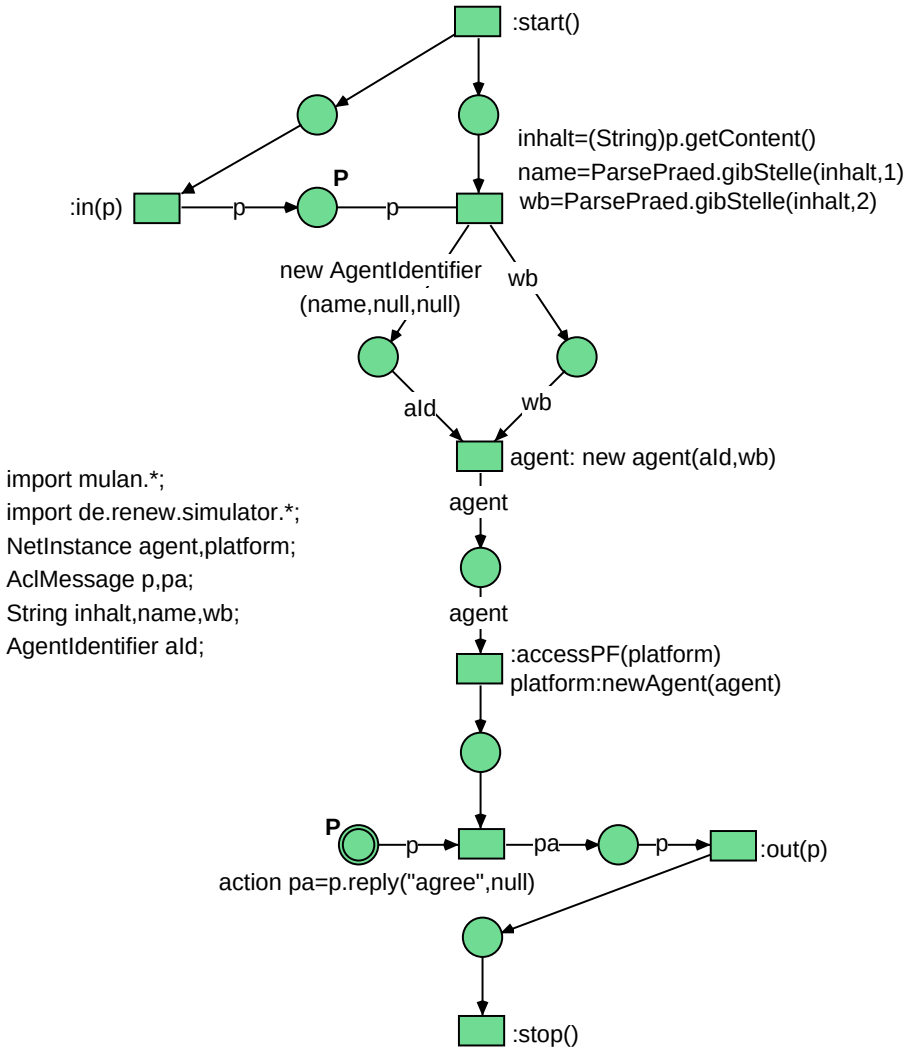

Abbildung A.6: Das Protokoll zur Agentenerzeugung – ein Plattformservice

B Quelltext

Der nachfolgend abgedruckte Quelltext, ist der Quelltext, der in Kapitel 9 für die Analyse von Simulationsmodellen mit Petrinetzten für Renew entwickelt, eingesetzt und erklärt wurde.

```
/**
 * Auswertung.java
 *
 * if real time analysis is wished, it is necessary to start with constructor
 *      Auswertung(double time)
 *
 * @author <a href="mailto: "Heiko Roelke</a>
 */

import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Vector;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class Auswertung {

    private Hashtable transEvents;
    private Hashtable placeEvents;
    private double startingTime = 0.0;
    public Auswertung() {
        transEvents = new Hashtable();
    }

    //if the starting time differs from zero
    public Auswertung(double time) {
        transEvents = new Hashtable();
        startingTime = time;
    }

    //colored (default)
    public void transInputEvent(String key, Object value, double time){
        if (transEvents.containsKey(key))
            ((ColoredTransitionEvent)transEvents.get(key)).in(value, time);
        else {
            ColoredTransitionEvent te = new ColoredTransitionEvent(startingTime);
            te.in(value, time);
            transEvents.put(key,te);
        }
    }

    //black token: to be replaced by new object
    public void transInputEvent(String key, double time){
        transInputEvent(key, new Object(), time);
```

```
}

//realtime...
public void transInputEvent(String key){
    transInputEvent(key,new Long(System.currentTimeMillis()).doubleValue());
}

public void transInputEvent(String key, Object value){
    transInputEvent(key, value, new Long(System.currentTimeMillis()).doubleValue());
}

//colored (default)
public void transOutputEvent(String key, Object value, double time) {
    if (transEvents.containsKey(key))
        ((ColoredTransitionEvent)transEvents.get(key)).out(value, time);
    else {
        //this should not happen...
        ColoredTransitionEvent te = new ColoredTransitionEvent(startingTime);
        te.out(value, time);
        //now the Queue is emptier than empty :)
        transEvents.put(key,te);
    }
}

//black token
public void transOutputEvent(String key, double time) {
    if (transEvents.containsKey(key))
        ((ColoredTransitionEvent)transEvents.get(key)).out(time);
    else {
        //this should not happen...
        ColoredTransitionEvent te = new ColoredTransitionEvent(startingTime);
        te.out(time);
        //now the Queue is emptier than empty :)
        transEvents.put(key,te);
    }
}

//realtime
public void transOutputEvent(String key) {
    transOutputEvent(key, new Long(System.currentTimeMillis()).doubleValue());
}

public void transOutputEvent(String key, Object value) {
    transOutputEvent(key, value, new Long(System.currentTimeMillis()).doubleValue());
}

public void simulationStop(double time) {
    Enumeration allTransEvents = transEvents.keys();
    while(allTransEvents.hasMoreElements()){
        String key = (String)allTransEvents.nextElement();
        System.out.println("\nReporting for system: "+key+"\n");
        ColoredTransitionEvent te = (ColoredTransitionEvent)transEvents.get(key);
        //te.report(time);
        String htmlString = te.reportHTML(key,time);
        schreiben(key+".html",htmlString);
        if (true) { //fillings
            String header = "#!/usr/local/bin/gnuplot -persist\n"+
```

```
                   "set title \"Filling of system: "+key+"\"\n"+
                    "set xlabel \"time\"\n"+
                    "set ylabel \"filling\"\n"+
                    "plot \" "+key+".dat\" with steps\n";
               schreiben(key+".plt",header);
               schreiben(key+".dat",((ColoredTransitionEvent)te).dumpFillings());
               try {String [] cmd = {"/bin/sh", "-c", "gnuplot-persist <"+key+".plt"};
               Runtime.getRuntime().exec(cmd);
            }
               catch (Exception e) {
                   System.out.println("das war nix...");
               }          }       if (true) { //dwells vs fillings
               String header = "#!/usr/local/bin/gnuplot -persist\n"+
                   "set title \"Dwell times vs fillings for: "+key+"\"\n"+
                   "set xlabel \"filling\"\n"+
                   "set ylabel \"mean dwell time\"\n"+
                   "plot \" "+key+"2.dat\" with boxes\n";
               schreiben(key+"2.plt",header);
               schreiben(key+"2.dat",((ColoredTransitionEvent)te).dumpDwellVsFilling());
               try {String [] cmd = {"/bin/sh", "-c", "gnuplot-persist < "+key+"2.plt"};
               Runtime.getRuntime().exec(cmd);
            }
               catch (Exception e) {
                   System.out.println("das war nix...");
               }
           }
           System.out.println("End of report--------\n\n\n");
       }
   }

   //realtime
   public void simulationStop() {
       simulationStop(new Long(System.currentTimeMillis()).doubleValue());
   }

   public String toString() {
       return transEvents.toString();
   }

   private void schreiben(String datei, String zuSchreiben) {
       File f;
       FileWriter out = null;
       try {
           f = new File(datei);
           out = new FileWriter(f);
           out.write(zuSchreiben,0,zuSchreiben.length());
           out.flush();
       } catch (IOException e) {
           System.out.println(e.getClass().getName()+": "
                              +e.getMessage());
       }
       finally { try {if (out != null) out.close(); } catch
           (IOException e) {} }
   }

}// Auswertung
```

```
/**
 * ColoredTransitionEvent.java
 * * internalTime is a relative time measure:
 *    - subtract starting time (relativeTime == true)
 *    - normalize (divide time values by timeDivisor)
 * * the additional logging mechanisms may be switched off:
 *    - log the fillings
 *    - log the dwell times versus the filling
 *   if switched off, the dumping produces empty strings
 * @author <a href="mailto: "Heiko Roelke</a>
 * @version
 */

import java.awt.geom.Point2D;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;

public class ColoredTransitionEvent {

    private double startingTime = 0.0;

    //properties
    private static final boolean relativeTime = true; //subtract start time
    private static final double timeDivisor = 1.0; //divide time values by me
    private double internalTime(double t) {
        if (relativeTime)
            return (t - startingTime) / timeDivisor;
        else
            return t / timeDivisor;
    }
    private static final boolean logFilling = true;
    private static final boolean logDwellPerFilling = true;

    //values to store
    private Hashtable inTimes = new Hashtable();
    private long inCounter = 0;  //input
    private long outCounter = 0; //throughput
    private long maxFill = 0;
    private long currentFill = 0;
    private Vector dwellTimes = new Vector();

    //used if specified (see properties)
    private Vector fillings = new Vector();
    private Hashtable dwellPerFilling = new Hashtable();

    //assuming empty system at start time
    public ColoredTransitionEvent() {
        if (logFilling)
            fillings.add(new Point2D.Double(0.0,0.0));
    }

    public ColoredTransitionEvent(double time) {
        startingTime = time;
        if (logFilling)
            fillings.add(new Point2D.Double(internalTime(time),0.0));
    }
```

```
public void in(Object o, double t) {
    double time = internalTime(t);
    if (inTimes.containsKey(o)) {
        throw new RuntimeException("Duplicate entry in statistical system.\n"+
                                   "Was trying to add: "+o);
    }
    inTimes.put(o, new Double(time));
    inCounter++;
    currentFill++;
    if (currentFill > maxFill)
        maxFill = currentFill;
    if (logFilling)
        fillings.add(new Point2D.Double(time,currentFill));
}

public void out(Object o, double oT) {
    double outTime = internalTime(oT);
    outCounter++;
    currentFill--;
    double inTime = ((Double)inTimes.remove(o)).doubleValue();
    double dwellTime = outTime - inTime;
    dwellTimes.add(new Double(dwellTime));
    if (logFilling)
        fillings.add(new Point2D.Double(outTime,currentFill));
    if (logDwellPerFilling) {
        Long fill = new Long(currentFill + 1);
        if (!dwellPerFilling.containsKey(fill))
            dwellPerFilling.put(fill,new Vector());
        ((Vector)dwellPerFilling.get(fill)).add(new Double(dwellTime));
    }
}

//black token case:
//  assuming FIFO behaviour, removing the first element of the Hashtable

public void out(double outTime) {
    out(inTimes.keys().nextElement(),outTime);
}

/** ********************************************************************
 *  Analysing / Output
 */

public String analyse(double sT) {
    double startTime = internalTime(startingTime);
    double stopTime = internalTime(sT);
    StringBuffer sb = new StringBuffer("Statistical report.\n\n");
    sb.append("Starting time: "+startTime+"\n");
    sb.append("Stopping time: "+stopTime+"\n");
    double simTime = stopTime - startTime;
    sb.append("Simulation time: "+simTime+"\n\n");

    if (currentFill != 0)
    sb.append("SYSTEM NOT EMPTY!\n\n");
    sb.append("Total input: "+inCounter+"\n");
    sb.append("Total throughput: "+outCounter+"\n");
```

```
        sb.append("Maximal fill: "+maxFill+"\n\n");

        double meanInput = inCounter / simTime;
        double meanThrough = outCounter / simTime;
        sb.append("Mean input: "+meanInput+"\n");
        sb.append("Mean throughput: "+meanThrough+"\n\n");

        //dwell time
        double timeCounter = 0;
        for (int i=0;i<dwellTimes.size();i++) {
            double dTime = ((Double)dwellTimes.get(i)).doubleValue();
            timeCounter += dTime;
        }

        double meanDwellTime = timeCounter / outCounter;
        sb.append("Mean dwell time: "+meanDwellTime+"\n");
        //approximate variance
        double varCount = 0.0;
        for (int i=0;i<dwellTimes.size();i++) {
            double distance = ((Double)dwellTimes.get(i)).doubleValue() - meanDwellTime;
            varCount += (distance * distance);
        }

        double approxDwellVariance = varCount / (dwellTimes.size() - 1);
        sb.append("Mean dwell time approximated variance: "+approxDwellVariance+"\n");
        sb.append("Mean dwell time approximated deviation: "
                  +Math.sqrt(approxDwellVariance)+"\n\n");

        //mean fill
        double meanfill = meanThrough * meanDwellTime;
        sb.append("Mean fill: "+meanfill+"\n\n");
        return sb.toString();
    }

    public String analyseHTML(String title, double sT) {
        double startTime = internalTime(startingTime);
        double stopTime = internalTime(sT);
        StringBuffer sb = new StringBuffer
            ("<html>\n<head>\n<title>"+
             "\nStatistical report for:"+title+"</title>\n\n"+
             "<body>\n<table cellspacing=\"10\" cellpadding=\"0\""+
             " border=\"1\">\n<tr><th></th><th></th></tr>");
        sb.append("<tr><td>Starting time</td><td>"+startTime+"</td></tr>\n");
        sb.append("<tr><td>Stopping time</td><td>"+stopTime+"</td></tr>\n");
        double simTime = stopTime - startTime;
        sb.append("<tr><td>Simulation time</td><td>"+simTime+"</td></tr>\n\n");
        //if (currentFill != 0)
        //    sb.append("SYSTEM NOT EMPTY!\n\n");
        sb.append("<tr><td>Total input</td><td>"+inCounter+"</td></tr>\n");
        sb.append("<tr><td>Total throughput</td><td>"+outCounter+"</td></tr>\n");
        sb.append("<tr><td>Maximal fill</td><td>"+maxFill+"</td></tr>\n\n");
        double meanInput = inCounter / simTime;
        double meanThrough = outCounter / simTime;
        sb.append("<tr><td>Mean input</td><td>"+meanInput+"</td></tr>\n");
        sb.append("<tr><td>Mean throughput</td><td>"+meanThrough+"</td></tr>\n\n");
        //dwell time
        double timeCounter = 0;
```

```
    for (int i=0;i<dwellTimes.size();i++) {
        double dTime = ((Double)dwellTimes.get(i)).doubleValue();
        timeCounter += dTime;
    }

    double meanDwellTime = timeCounter / outCounter;
    sb.append("<tr><td>Mean dwell time</td><td>"+meanDwellTime+"</td></tr>\n");
    //approximate variance
    double varCount = 0.0;
    for (int i=0;i<dwellTimes.size();i++) {
        double distance = ((Double)dwellTimes.get(i)).doubleValue() - meanDwellTime;
        varCount += (distance * distance);
    }

    double approxDwellVariance = varCount / (dwellTimes.size() - 1);
    sb.append("<tr><td>Mean dwell time approximated variance</td><td>"
               +approxDwellVariance+"</td></tr>\n");
    sb.append("<tr><td>Mean dwell time approximated deviation</td><td>"
              +Math.sqrt(approxDwellVariance)+"</td></tr>\n");
    //mean fill
    double meanfill = meanThrough * meanDwellTime;
    sb.append("<tr><td>Mean fill</td><td>"+meanfill+"</td></tr>\n");
    sb.append("</table>\n</body>\n</html>\n");
    return sb.toString();
}

public String dumpFillings() {
    StringBuffer sb = new StringBuffer();
    Enumeration allPoints = fillings.elements();
    while(allPoints.hasMoreElements()){
        Point2D.Double p = (Point2D.Double)allPoints.nextElement();
        sb.append(p.getX()+" "+p.getY()+"\n");
    }
    return sb.toString();
}

public String dumpDwellVsFilling() {
    StringBuffer sb = new StringBuffer();
    Enumeration allData = dwellPerFilling.keys();
    while(allData.hasMoreElements()){
        Long key = (Long)allData.nextElement();
        Enumeration dwellEnum = ((Vector)dwellPerFilling.get(key)).elements();
        long count = 0;
        double dwellSum = 0.0;
        while(dwellEnum.hasMoreElements()){
            count++;
            dwellSum += ((Double)dwellEnum.nextElement()).doubleValue();
        }
        sb.append(key.longValue()+" "+(dwellSum / count)+"\n");
    }
    return sb.toString();
}

public void report(double stopTime) {
    //wegen doppeltem internem Aufruf hier nicht umwandeln!
    System.out.println(analyse(stopTime));
}
```

```
    public String reportHTML(String title, double stopTime) {
        //wegen doppeltem internem Aufruf hier nicht umwandeln!
        return (analyseHTML(title, stopTime));
    }
}
```

C Abbildungsverzeichnis

D Literaturverzeichnis

[ACR00] Agha, G., F. De Cindio und G. Rozenberg (Herausgeber): *Concurrent Object-Oriented Programming and Petri Nets*, Band 2001 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[Bau90] Baumgarten, B.: *Petri-Netze - Grundlagen und Anwendungen*. BI Wissenschaftsverlag, 1990.

[BB95] Biberstein, Olivier und Didier Buchs: *Structured Algebraic Nets with Object-Orientation*. Technischer Bericht, University of Geneva and Swiss International Institute of Technology, 1995.

[BK96] Bause, F. und P. F. Kritzinger: *Stochastic Petri Nets*. Vieweg, 1996.

[Bla97] Blaschek, G.: *Informatik Handbuch*, Kapitel Objektorientierte Programmierung. Carl Hanser Verlag, 1997.

[BM93] Becker, Ulrich und Daniel Moldt: *Object-Oriented Concepts for Coloured Petri nets*. In: *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics*, Seiten 279–286, Oktober 1993.

[Bor76] Borovkov, A.A.: *Stochastic Processes in Queueing Theoriy*. Springer-Verlag, 1976.

[Cab02] Cabac, Lawrence: *Evaluation und beispielhafte Erweiterung einer referenznetzbasierten Agentenumgebung*. Studienarbeit, Universität Hamburg, Fachbereich Informatik, 2002.

[Car03] Carl, Timo: *Ein Viewer für Mulan-Plattformen*. Studienarbeit, Universität Hamburg, Fachbereich Informatik, 2003.

[CMR03] Cabac, Lawrence, Daniel Moldt und Heiko Rölke: *A Proposal for Structuring Petri Net-Based Agent Interaction Protocols*. In: Aalst, W. v. d. und E. Best (Herausgeber): *Proceedings of the International Conference on Application and Theory of Petri Nets 2003*, Band 2679 der Reihe *Lecture Notes in Computer Science*, Seiten 102–120. Springer-Verlag, 2003.

[Fer01] Ferber, Jaques: *Multiagentensysteme*. Addison-Wesley, 2001.

[Fis73] Fishman, George S.: *Concepts and Methods in Discrete Event Digital Simulation*. John Wiley & Sons, Inc, 1973.

[Fis78] Fishman, George S.: *Principles of Discrete Event Simulation*. John Wiley & Sons, Inc, 1978.

[Fis01] Fishman, George S.: *Discrete Event Simulation: Modelling, Programming and Analysing*. Springer-Verlag, 2001.

[GBB00] Grässle, P., H. Baumann und P. Baumann: *UML - Projektorientiert*. Galileo Press GmbH, 2000.

[Gen97] Genesereth, M. R.: *An Agent-Based Framework for Interoperability*. In: Bradshaw, J.M. (Herausgeber): *Software Agents*, Band 15, Seiten 317–346. AAAI Press, 1997.

[GK94] Genesereth, M. und S. Ketchpel: *Software Agents*. Communications of the ACM 37 No.7, Seiten 48–53, 1994.

[GV02] Girault, Claude und Rüdiger Valk: *Petri Nets for System Engineering – A Guide to Modeling, Verification, and Applications*. Springer-Verlag, 2002.

[HLuK78] Heller, W., H. Lindenberg und M. Nuske und K.Schriever: *Stochastische Systeme*. Walter de Gruyter, 1978.

[HU79] Hopcroft, John E. und Jeffrey D. Ullman: *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[Hüb96] Hübner, Gerhard: *Stochastik*. Vieweg, 1996.

[Jen92] Jensen, K.: *Coloured Petri Nets, Basic Methods, Analysis Methods and Practical Use*, Band 1 der Reihe *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1992.

[JV87] Jessen, E. und R. Valk: *Rechensysteme – Grundlagen der Modellbildung*. Springer-Verlag, 1987.

[JW95] Jennings, Nicolas R. und Michael Wooldridge: *Intelligent Agents: Theory and Practice*. Knowledge Engeneering Review, 1995.

[KMR01] Köhler, Michael, Daniel Moldt und Heiko Rölke: *Modeling the Behaviour of Petri Net Agents*. In: Colom, J. M. und M. Koutny (Herausgeber): *Proceedings of the 22st Conference on Application and Theory of Petri Nets*, Band 2075 der Reihe *Lecture Notes in Computer Science*, Seiten 224–241. Springer-Verlag, 2001.

[KMR02] Köhler, Michael, Daniel Moldt und Heiko Rölke: *Liveness Preserving Composition of Behaviour Protocols for Petri Net Agents*. Technischer Bericht, Universität Hamburg, FB Informatik, 2002.

[KR03] Köhler, Michael und Heiko Rölke: *Concurrency for Mobile Object Net Systems*. Fundamenta Informaticae, 54(2-3), 2003.

[Kum98] Kummer, Olaf: *Simulating Synchronous Channels and Net Instances*. In: Desel, J., P. Kemper, E. Kindler und A. Oberweis (Herausgeber): *Forschungsbericht Nr. 694: 5. Workshop Algorithmen*

und Werkzeuge für Petrinetze, Seiten 73–78. Universität Dortmund, Fachbereich Informatik, 1998.

[Kum99] KUMMER, OLAF: *A Petri Net View on Synchronous Channels.* Petri Net Newsletter, 56:7–11, 1999.

[Kum01] KUMMER, OLAF: *Introduction to Petri nets and Reference nets.* Sozionik aktuell, 1, 2001.

[Kum02] KUMMER, OLAF: *Referenznetze.* Logos-Verlag, 2002.

[KW03] KUMMER, OLAF und FRANK WIENBERG: *Reference net workshop (Renew).* Universität Hamburg, http://www.renew.de, 2003.

[KWD] KUMMER, OLAF, FRANK WIENBERG und MICHAEL DUVIGNEAU: *Renew User Guide.*

[Käm90] KÄMPER, SABINE: *Pegros: Ein Konzept zur Entwicklung eines graphischen, objektorientierten Modellbildungs- und Simulationswerkzeug auf der Basis von Petrinetzen.* Dissertation, Universität Hamburg, FB Informatik, Vogt-Kölln Str. 30, 22527 Hamburg, 1990.

[Lak95a] LAKOS, CHARLES: *From coloured Petri Nets to Object Petri Nets.* In: *Proceeding of the 16th International Conference on Application and Theory of Petri Nets*, Lecture Notes in Computer Science, Seiten 278–297, Berlin, June 1995. Springer-Verlag.

[Lak95b] LAKOS, CHARLES: *The Object Orientation of Object Petri Nets.* In: *Workshop on Object Oriented Programming and Models of Concurrency.*, 1995.

[Mar95] MARSAN, M.AJMONE: *An Introduction to Generalized Stochastic Petri Nets.* In: TURIN, UNIVERSITÄT (Herausgeber): *Introductory Tutorials, within the 16th International Conference on Application and Theory of Petri Nets*, 1995.

[MBC+95] MARSAN, M.A., G. BALBO, G. CONTE, S. DONATELLI und G. FRANCESCHINIS: *Modelling With Generalized Stochastic Petri Nets.* Wiley, 1995.

[MR00] MOLDT, DANIEL und HEIKO RÖLKE: *Verhaltensmodellierung von Petrinetzagenten.* In: *Workshop Visuelle Verhaltensmodellierung verteilter und nebenläufiger Systeme. VVVNS'00*, 2000.

[NBBC66] NAYLER, T. H., J.L. BALINTFY, D.S. BUDICK und KONG CHU: *Computer Simulation Techniques.* John Wiley & Sons, Inc., 1966.

[Neu77] NEUMANN, KLAUS: *Operations Research Verfahren Band II.* Carl Hanser Verlag, 1977.

[Nie77] NIEMEYER, G.: *Kybernetische Systeme und Modelltheorie.* Franz Vahlen, 1977.

[Pag92] Page, B.: *Diskrete Simulation.* Springer-Verlag, 1992.

[PBH+88] Page, B., R. Blöckow, A. Heymann, R. Kadler und H.Liebert: *Simulation und moderne Programmiersprachen.* Springer-Verlag, 1988.

[Pet62] Petri, Carl Adam: *Kommunikation mit Automaten.* Dissertation, Universität Bonn, Rheinisch Westfälisches Institut für instumentelle Mathematik, 1962.

[PLC00] Page, B., T. Lechler und S. Claassen: *Objektorientierte Simulation in Java.* Libri, 2000.

[Pri84] Pritsker, Alan B.: *Simulation and SLAM II.* John Wiley & Sons, Inc., 1984.

[RD98] Roberts, C. A. und Y. M. Dessouky: *An Overview of Object-Oriented Simulation.* Simulation, 70(7):359–368, 1998.

[Rei85] Reisig, Wolfgang: *Petri Nets: An Introduction.* Springer-Verlag, 1985.

[Röl99] Rölke, Heiko: *Modellierung und Implementation eines Multiagentensystems auf der Basis von Referenznetzen.* Diplomarbeit, Universität Hamburg, 1999.

[RR98] Reisig, Wolfgang und Grzegorz Rozenberg: *Lectures on Petri Nets 1: Basic Models.* Springer-Verlag, 1998.

[SH95] Spaniol, Otto und Simon Hoff: *Ereignisorientierte Simulation: Konzepte und Systemrealisierung.* International Thompson Puplications., 1995.

[Str01] Strümpel, Frauke: *Exemplarische Evaluierung von Ansätzen zur Modellierung ereignisorientierter Simulationsszenarien anhand von Petrinetzen und DESMO.* Studienarbeit, Universität Hamburg, Fachbereich Informatik, 2001.

[Tan95] Tanenbaum, Andrew S.: *Moderne Betriebssysteme.* Hanser, 1995.

[Val98] Valk, Rüdiger: *Petri Nets as Token Objects: An Introduction to Elementary Object Nets.* In: Desel, Jörg und Manuel Silva (Herausgeber): *Application and Theory of Petri Nets*, Band 1420 der Reihe *LNCS*, Seiten 1–25, Juni 1998.

[Val00a] Valk, Rüdiger: *Concurrency in Communicating Object Petri Nets.* In: Agha, G., F. De Cindio und G. Rozenberg (Herausgeber): *Concurrent Object-Oriented Programming and Petri Nets*, Lecture Notes in Computer Science, Berlin, 2000. Springer-Verlag.

[Val00b] Valk, Rüdiger: *Relating different semantics for object Petri nets.* Technischer Bericht FBI-HH-B-226/00, Universität Hamburg, FB Informatik, 2000.

[Val00c] VALK, RÜDIGER: *Mobile and Distributed Objects versus Central Referencing.* In: J. GRABOWSKI, S. HEYMER (Herausgeber): *ITG-Fachgespräch 2000 Lübeck, Formale Beschreibungstechniken für Verteilte Systeme*, Seiten 7 – 27, 2000.

[vdA95] AALST, W. M. P. VAN DER: *A class of Petri nets for modeling and analyzing business processes.* Technischer Bericht 95, Eindhoven University of Technology Computing Science, 1995.

[vSGH99] STORCH, HANS VON, STEFAN GUESS und MARTIN HEIMAN: *Das Klimasystem und seine Modellierung.* Springer, 1999.

[Wat96] WATT, DAVID A.: *Programmiersprachen.* Carl Hanser Verlag, 1996.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der verwendeten Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen hat. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Hamburg, den 19.11.2003

Frauke Strümpel