Exemplarische Evaluierung von Ansätzen zur Modellierung ereignisorientierter Simulationsszenarien anhand von Petrinetzen und DESMO

Studienarbeit

Universität Hamburg

Fachbereich Informatik

Arbeitsbereich Theoretische Grundlagen der Informatik

Juli 2001

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

Betreuer: Michael Köhler

Inhaltsverzeichnis

Kapitel 1

Einleitung

In dieser Arbeit sollen zwei Ansätze der Simulation anhand eines Beispiels aus dem Bereich der Logistik eines Containerterminals betrachtet und verglichen werden.[1]

Der erste Ansatz ist die Simulation mit dem Framework Desmoj, das im Arbeitsbereich ASI entwickelt wurde. Der zweite Ansatz ist der Einsatz von Petrinetzen, deren Ablauf mit Hilfe des Petrinetzwerkzeugs Renew simuliert werden kann. Das Petrinetzwerkzeug Renew ist ebenfalls eine Universitätsentwicklung, die im Arbeitsbereich TGI entstanden ist.

Das Beispiel, an dem beide Methoden betrachtet werden, ist ein Netz von Warte-/Bediensystemen. Es handelt sich um den Ausschnitt der Lkw-Abfertigung am Containerterminal im Hafen. Da ein vollständiges Containerterminal sehr komplex ist, und das entsprechende Szenario schlecht nachzuvollziehen wäre, wird für den Vergleich der Simulationen ein kleineres Beispiel gewählt: Die Lkw-Abfertigung am Containerterminal bietet ein geeignetes Szenario. Die detaillierte Beschreibung des Szenarios kann in Kapitel 3.1 nachgelesen werden.

Szenario Die Lkw Abfertigung kann man sich etwa so vorstellen: Lkws fahren zum Terminal, weil sie dort einen bestimmten Auftrag ausführen sollen. Der Auftrag lautet entweder, „bringe einen Container auf das Terminal", „hole einen Container vom Terminal ab" oder „bringe einen hin und hole auch gleich wieder einen ab". Wenn ein Lkw mit einem Auftrag am Terminal ankommt, muß er sich anmelden, damit sein Auftrag auf dem Terminal ausgeführt werden kann. Das macht er, indem er seinen Auftrag an die Terminalkontrolle weitergibt. Die Aufträge werden von der Terminalkontrolle dann an sogenannte Vancarrier weitergegeben. Die Vancarrier sind dafür da, Container von den Lkws zu laden und in einen Yard, in dem die Container aufbewahrt werden, zu fahren; oder einen Container aus einem Yard zu holen und einen Lkw damit zu beladen.

Damit die Ladevorgänge auf dem Terminal stattfinden können, dürfen bestimmte Kapazitäten nicht überschritten werden. D.h. ein Terminal kann aus Platzgründen nur eine bestimmte Anzahl an Lkws auf einmal aufnehmen. Die Lkws, die auf dem Terminal keinen Platz mehr bekommen haben, um von den

[1] Die Idee dieser Arbeit ist im Rahmen eines Praktikums im Arbeitsbereich Angewandte und Sozialorientierte Informatik (ASI) entstanden. Realisiert wurde sie im Arbeitsbereich Theoretische Grundlagen der Informatik (TGI).

Vancarriern bedient zu werden, müssen vor dem Terminal in einer Warteschlange warten, bis wieder ein Platz frei geworden ist.

Fragestellung Wenn man simuliert, möchte man eine bestimmte Frage beantworten. Die Frage für dieses Beispiel lautet: „Wie viele Vancarrier braucht man, und wie groß muß die Kapazität der Bedienstation sein, um einen effizienten Ablauf auf dem Terminal zu garantieren?" Welche Anzahl VCs und welche Kapazitäten man braucht, hängt stark von der Anzahl ankommender Lkws ab. Man muß also herausfinden, wie viele Lkws an einem Tag zu solch einem Containerterminal fahren. Hat man eine bestimmte Anzahl an VCs, muß man prüfen, ob diese VCs kurze Wartezeiten haben und schnell neue Lkws bedienen können, oder ob sie möglicherweise, da die Anzahl der VCs zu klein ist, mit dem Bedienen der Lkws nicht hinterher kommen und sich dadurch eine zu lange Wartezeit für die Lkws ergibt.

Eine größere Anzahl an VCs nützt allerdings erst dann etwas, wenn man gleichzeitig die Kapazität der Bedienstation auf dem Terminal anpaßt. Denn die VCs sollen nicht nacheinander, sondern möglichst parallel arbeiten, da sie sonst zu lange bewegungslos auf dem Terminal stehen. Das kann heißen, daß sie bereits einen Auftrag angenommen haben, ihn aber nicht ausführen können, da der entsprechende Lkw wegen der begrenzten Kapazitäten noch nicht bis zur Bedienstation vorgedrungen ist. Das Ziel ist also, das Terminal so zu gestalten, daß es möglichst effizient arbeitet, also die Lkws so schnell wie möglich bedient werden, bei geringen Wartezeiten der VCs.

Analyse Die Frage des optimalen Aufbaus des Terminals kann man rechnerisch nur schwer lösen, da zu viele Größen an der Rechnung beteiligt sind. Um das Problem aber trotzdem analysieren zu können, simuliert man das Szenario, um nach ausreichend vielen Durchläufen des Simulators eine Aussage bezüglich der Fragestellung treffen zu können.

Das oben beschriebene Szenario ist auf zwei Arten simuliert worden. Mit dem Framework Desmoj und mit dem Petrinetzwerkzeug Renew. Beide Werkzeuge haben ihre Vor- und Nachteile, die im Folgenden kurz erläutert werden.

Das Konzept für Desmo ist schon relativ alt, und es wurde früher schon in verschiedenen Programmiersprachen realisiert. Da man im Arbeitsbereich ASI schon sehr viele Erfahrungen mit Desmo gemacht hatte, wurde es auch in Java übersetzt und Desmoj genannt. Desmoj ist ein Framework, daß speziell für die Programmierung von Simulatoren erstellt wurde. Da Desmoj in Java geschrieben ist und deshalb zum Stil der objektorientierten Programmierung paßt, ist es für jeden, der Java programmieren kann oder die objektorientierte Programmierung beherrscht, leicht zu erlernen. Desweiteren bietet Desmoj eine statistische Auswertung, die für die Beantwortung einer bestimmten Fragestellung sehr hilfreich ist.

Ein Nachteil an Desmoj ist, daß es, wenn man noch keine große Erfahrung mit Java hat, sehr kompliziert ist. Man muß die objektorientierte Programmierung gut beherrschen, um den Quellcode lesen zu können und Veränderungen am Simulator vornehmen zu können. Vor allem sind Synchronisationen, die bei der Simulation häufig notwendig sind, in Desmoj kompliziert und sehr feh-

leranfällig.

Renew bietet im Gegensatz dazu die Vorteile, daß der Simulator, also ein Petrinetz, intuitiv und visuell erstellt werden kann. Man baut in Renew ein Modell, das man sieht und verhältnismäßig leicht erfassen kann. Änderungen können sehr leicht vorgenommen werden, und Fehler können recht schnell erkannt und analysiert werden. Außerdem ist ein erstelltes Modell direkt ausführbar, d.h. das Modell ist auch gleichzeitig das Programm, das die Ausführung vornimmt. Die Simulation kann während der Ausführung beobachtet werden. Für das Beispiel heißt das, ein Lkw kann beobachtet werden, während er das Terminal durchläuft und ent- oder beladen wird. Renew bietet eine sehr gute Prozeßorientierung, da jeder Prozeß durch ein eigenes Petrinetz dargestellt wird. Die verschiedenen Netze können sehr einfach synchronisiert werden, wenn das Szenario es verlangt.

Renew ist ebenfalls in Java geschrieben und kann leicht andere Javaklassen einbinden. Ein Nachteil an Renew ist, daß es noch keine statistische Auswertung gibt, und so Aussagen über eine Fragestellung schwerer getroffen werden können. Es ist aber prinzipiell möglich, Renew um diese fehlenden zur Simulation geeigneten Konstrukte zu erweitern.

Gliederung Die Arbeit ist wie folgt gegliedert: Im zweiten Kapitel werden Grundlagen, die für die Simulation wichtig sind, eingeführt. Dazu gehören die Unified Modelling Language, eine Einführung in Petrinetze, stochastische Modelle, Grundlagen zur zeitdiskreten Simulation und die Werkzeuge Desmoj und Renew.

Im dritten Kapitel wird das Szenario des gewählten Beispiels detailliert erläutert; die beiden mit verschiedenen Werkzeugen erstellten Simulationsmodelle werden präsentiert.

Im vierten Kapitel wird eine Auswertung bezüglich der Fragestellung an hand eines stochastischen Modells und eines Simulators vorgenommen. Zusätzlich wird eine Aussage über die gewählten Vorgehensweisen getroffen.

Es folgen eine Zusammenfassung und ein Ausblick.

Kapitel 2

Grundlagen

In dem Kapitel Grundlagen werden die Voraussetzungen für die Simulation mit Petrinetzen und Desmo dargestellt. Das betrifft Grundlagen der Unified Modelling Language, des Umgangs mit Petrinetzen, der Verwendung von stochastischen Modellen, der zeitdiskreten Simulation und der Werkzeuge, die zur Simulation eingesetzt werden.

2.1 Unified Modelling Language

Eine grundlegende Einführung in die Theorie und Praxis der Unified Modelling Language findet sich in vielen Werken der Standardliteratur, s. z.B. [RJB99], [FS98] oder [Bur97].

Nach [RJB99] ist die Unified Modelling Language (kurz: UML) eine Sprache für die Modellierung und Darstellung von Softwaresystemen und Geschäftsprozessen. Es ist möglich, mit der Unified Modelling Language, die mit Hilfe von Worten und Bildern arbeitet, Systeme zu beschreiben.

Es können mit der UML beliebige, verschiedenartige Systeme, wie zum Beispiel Geschäftsprozesse oder Softwaresysteme modelliert werden. Man benutzt als Modellierungswerkzeug UML-Diagramme, die sich aus bestimmten graphischen Elementen zusammensetzen. Diese Elemente sind zum Beispiel Ellipsen, Rauten oder verschiedene Pfeiltypen. Jedes dieser grafischen Elemente hat eine bestimmte Bedeutung, so daß es möglich ist, ein Diagramm, das nach bestimmten Regeln aus den grafischen Elementen zusammengesetzt ist, lesen zu können. Ein einzelnes Diagramm (oder auch mehrere Diagramme zusammen) bilden ein Modell. Es gibt unterschiedliche Diagrammtypen; die bekanntesten unter ihnen sind das Anwendungsfalldiagramm und das Klassendiagramm. In Kapitel 2.1.3 werden sie und andere wichtige Diagrammtypen genauer beschrieben.

UML ist die weltweite Einigung auf eine Modellierungssprache, die von der Objekt Management Group (OMG) standardisiert wurde. Die UML wurde zu großen Teilen aus bestehenden Modellierungssprachen und mit Bezug auf die Praxis entwickelt, d.h. sie baut auf bewährten und weit verbreiteten Ansätzen auf. Es gibt viele Gründe, warum UML als Modellierungssprache gut geeignet ist. Die Vereinheitlichung der Terminologie und die Standardisierung der Notation führen zu einer erheblichen Erleichterung der Verständigung zwischen allen Beteiligten, und der Austausch von Modellen zwischen verschiedenen Gruppen

wird damit erleichtert.

Ein weiterer Grund für ihren Einsatz ist, daß die UML mit ihren Anforderungen wächst. Es ist möglich, mit dem Erstellen einfacher Modelle zu beginnen, man kann aber auch sehr komplexe Sachverhalte im Detail modellieren, da die UML eine mächtige Modellierungssprache ist. Zusätzlich macht die UML es auch möglich, die Grundfunktionalität mittels Stereotypen selbst zu erweitern.

2.1.1 Warum braucht man Modelle?

Modelle dienen der Kommunikation, der Visualisierung und der Überprüfung. Im Kontext von Geschäfts- und IT-Systemen werden oft Modelle erstellt, um existierende oder zukünftige Systeme besser verstehen zu können. Modellieren bedeutet aber immer Hervorheben wesentlicher Eigenschaften und Weglassen unwichtiger Details. Das führt dazu, daß ein Modell nie genau der Realität entspricht. Was wesentlich und was unwichtig ist, kann man nicht allgemein beantworten, das hängt immer davon ab, welches Ziel mit dem Modell verfolgt werden soll. Je mehr Information in einem Modell gezeigt werden soll, desto komplexer und schwieriger wird es. Man kann deshalb versuchen, verschiedene Informationen auf einzelne Modelle zu verteilen. Dadurch werden verschiedene Sichten auf ein zu betrachtendes System abgebildet, die in vielfältiger Weise miteinander verbunden sind. Wenn eine Sicht geändert wird, müssen in der Regel alle anderen Sichten auch geändert bzw. angepaßt werden.

Soll ein IT-System erstellt werden, sind nach [GBB00] drei Modelle erforderlich:

- Das Modell des Geschäftssystems,
- das Modell des IT-Systems und
- das Modell der Systemintegration.

Um einen reibungslosen Einsatz mit Hilfe von IT-Systemen zu gewährleisten, ist es unbedingt erforderlich, das Umfeld von IT-Systemen zu kennen und zu verstehen. Bei der Entwicklung und Integration von IT-Systemen sind daher Analyse und Modellierung von Prozessen sehr wichtige Komponenten. Die meisten IT-Systeme sind heutzutage nicht nur in ein Umfeld eingebettet, sondern auch mit anderen IT-Systemen verbunden. Das führt dazu, daß jedes neue IT-System in zweierlei Hinsicht in seine Zielumgebung passen muß. Zum einen in die Prozeßebene, zum Anderen in die IT-Systemebene. Jedem IT-System müssen Aktivitäten eines Prozesses so zugeordnet sein, daß der gesamte Prozeß mit allen beteiligten Komponenten korrekt und effizient abgewickelt werden kann. Die Kommunikation mit weiteren IT-Systemen, die beteiligt sind, muß reibungslos funktionieren. Dies erfordert semantisch und technisch perfekt passende Schnittstellen.

Um ein angemessenes System zu bauen, ist es wichtig, daß alle Beteiligten das gleiche Verständnis des Systems haben. Die verwendete Terminologie muß von allen verstanden werden. Auftraggeber und Entwickler müssen sich auf gemeinsame Anforderungen einigen, die in einem Modell abgebildet werden, das beide verstehen.

Ein weiterer Grund für den Einsatz von Modellen ist, daß gefällte Entscheidungen, die auch nach Monaten noch nachvollziehbar bleiben müssen, noch nachvollzogen werden können. D.h. es geht um Kommunikation zwischen Beteiligten über die Zeit hinweg. Dies ist ohne Aufzeichnungen, ohne Repräsentation der gefundenen Fakten nicht möglich.

Das Wesentliche an einem Modell ist also, daß die Fakten, die zu einem System gesammelt werden, so repräsentiert werden müssen, daß sie von den Betroffenen verstanden werden können. Das ist in den meisten Fällen durch ein graphisches Modell einfacher als durch eine rein textliche Beschreibung zu erreichen. Ein mehr oder weniger formales Modell ermöglicht es, die gefundenen Fakten bezüglich Vollständigkeit, Widerspruchsfreiheit und Korrektheit zu überprüfen. Durch die klare Darstellung insbesondere von Zusammenhängen wird es möglich, gezielt Fragen zu stellen und zu beantworten.

Es hat großen Einfluß auf das Ergebnis der Modellierung, für wen ein Modell erstellt wird und zu welchem Zweck es benutzt werden soll. Werden diese Punkte nicht ausreichend diskutiert und definiert, besteht die Gefahr, daß Modelle entstehen, die nicht das enthalten, was den Anwender interessiert. Mit anderen Worten, das Hervorheben des Wesentlichen und das Weglassen unwichtiger Details ist nicht sachgemäß durchgeführt worden und das Modell ist dadurch wertlos.

2.1.2 UML-Diagramme

Ein bestimmtes UML-Diagramm entspricht einer Sicht auf das Modell eines Systems. Je nach Diagrammtyp werden dabei verschiedene Aspekte hervorgehoben bzw. weggelassen. Alle unterschiedlichen Sichten zusammen ergeben ein Modell des Systems.

Die meisten UML-Diagramme sind Graphen, d.h. sie bestehen aus Elementen (Knoten), die durch Linien (Kanten) verbunden sind. Um die Diagramme lesen zu können muß man jeweils wissen, welche Arten von Knoten und welche Arten von Kanten erlaubt sind und was deren Bedeutung ist.

Es gibt für jedes Modell eine externe und eine interne Sicht. Die externe Sicht beschreibt die Interaktion mit externen Parteien wie Kunden oder Partnern und stellt das Geschäftssystem oder das IT-System als Blackbox dar. Dem gegenüber beschreibt die interne Sicht die internen Abläufe, Aktivitäten, Beziehungen und Strukturen des Geschäfts- oder IT-Systems. Man taucht quasi in die Blackbox ein und versucht, sie zu beschreiben. Bei der internen Sicht auf das Geschäftssystem spielt der Aufbau der Organisationseinheit eine Rolle: Organisationseinheiten werden in der UML als Pakete dargestellt, die Mitarbeiter, Geschäftsobjekte und weitere Organisationseinheiten enthalten können.

Bei einem IT-System gibt es vier Sichten auf ein Modell: Die Sicht von außen, die strukturelle Sicht, die Ablaufsicht und die Verhaltenssicht. Jeder dieser Sichten sind bestimmte Diagrammtypen zugeordnet. Jede Sicht hebt bestimmte Aspekte hervor und vernachlässigt dafür alle anderen. Alle Sichten zusammen ergeben das vollständige Modell der Funktionalität eines IT-Systems.

- Die Sicht von außen zeigt in Form von UML-Anwendungsfalldiagrammen und Oberflächenprototypen die Anwendungsfälle des IT-Systems. Es wird

ersichtlich, welche Funktionalität das IT-Systems den Anwendern zur Verfügung stellt.

- Die strukturelle Sicht zeigt in Form von UML-Klassendiagramen die fachlich relevanten Klassen des IT-Systems. Es wird ersichtlich, in welchen Strukturen die Informationen im IT-System abgelegt werden.
- Die Verhaltenssicht zeigt in Form von Zustandsdiagrammen das Verhalten der einzelnen Objekte. Es wird ersichtlich, was alles mit einem Objekt geschehen kann, das im IT-System abgelegt ist.
- Die Ablaufsicht zeigt in Form von Sequenz-und Kollaborationsdiagrammen die Abläufe, die bei Mutationen und Abfragen im Inneren des IT-Systems stattfinden. Es wird ersichtlich, was im IT-System vor sich geht, wenn es durch Anwender benutzt wird.

Folgende Diagrammtypen sind nach [GBB00] in der UML zur Modellierung von Systemen üblich.

- Anwendungsfalldiagramme
- Aktivitätsdiagramme
- Sequenzdiagramme
- Klassendiagramme

Es ist aber nicht notwendig, daß alle Diagrammtypen in einem Modell verwendet werden. Die Verwendung von Diagrammtypen hängt davon ab, welche Merkmale des Systems herausgehoben werden sollen.

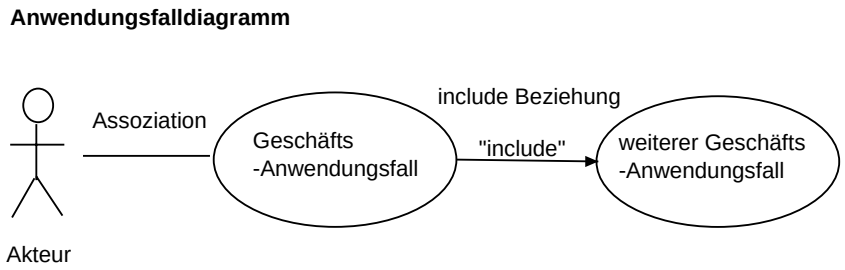

Abbildung 2.1: Das Anwendungsfalldiagramm

In Abbildung 2.1 ist ein *Anwendungsfalldiagramm* abgebildet. Es zeigt Akteure, Geschäfts-Anwendungsfälle und deren Beziehung. In Anwendungsfalldiagrammen sind keine Abläufe beschrieben. Alternative Szenarien bleiben ebenfalls verborgen. Diese Diagramme geben einen guten Überblick über die Funktionalitäten und den Kontext des Systems. Es finden aber keine Aussagen über einen zeitlichen Verlauf statt.

In Abbildung 2.2 ist ein *Aktivitätsdiagramm* zu sehen. Aktivitätsdiagramme beschreiben Abläufe, das sind Geschäftsprozesse des Geschäftssystems oder

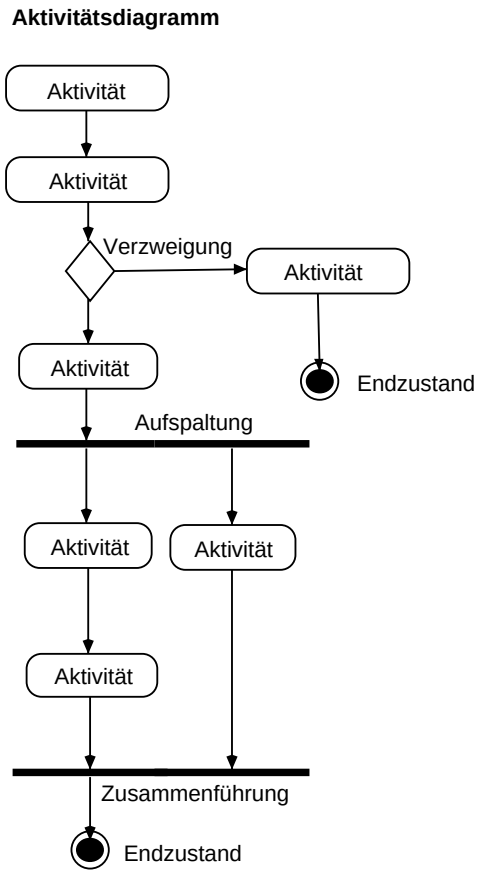

Abbildung 2.2: Das Aktivitätsdiagramm

Abläufe eines IT-Systems. Gegenstand der Beschreibung sind die Interaktionen der Akteure, als die Leistungen, die Kunden und Geschäftspartnern angeboten werden. Außenstehende erkennen anhand von Aktivitätsdiagrammen, wie sie mit dem System interagieren müssen. Sie eignen sich vor allem, um Sequenzen, Alternativen und Parallelen von Abläufen zu beschreiben. Aktivitätsdiagramme können in unterschiedlichen Detaillierungsgraden erstellt werden. Anwendungsfall-Aktivitätsdiagramme zeigen die Abläufe einzelner Anwendungsfälle. Sie dienen der Detaillierung und Präzisierung von Anwendungsfällen: Mit Aktivitätsdiagrammen können die möglichen Alternativen (Szenarien) eines Ablaufs dargestellt werden und verfeinert werden.

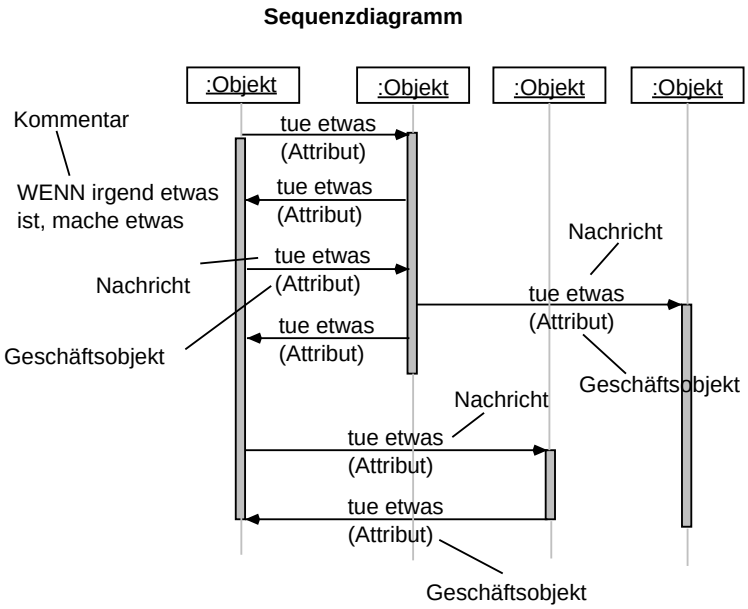

Abbildung 2.3: Das Sequenzdiagramm

Abbildung 2.3 zeigt ein *Sequenzdiagramm*. Sequenzdiagramme beschreiben den zeitlichen Ablauf der Interaktionen in einem Modell. Es werden nicht die gesamten Abläufe mit allen Verzweigungen und Parallelitäten dargestellt, sondern die Nachrichten, die zwischen den Beteiligten in zeitlicher Reihenfolge ausgetauscht werden. Dieses Diagramm ist eine gute Grundlage für den Daten- bzw. Nachrichtenaustausch mit Partnern und Kunden.

Anwendungsfall-Sequenzdiagramme zeigen den Ablauf der Interaktion eines Anwendungsfalls. Sie dienen der Detaillierung und Präzisierung von Geschäfts-Anwendungsfällen, indem sie den Nachrichtenaustausch betonen.

UML-Diagramme von Anwendungsfällen können durch textliche Beschreibungen und erläuternde Abbildungen ergänzt werden.

Ein für die Entwicklung von IT-Systemen sehr wichtiger Diagrammtyp ist das Klassendiagramm, es wird daher etwas ausführlicher erklärt. Abbildung 2.4 zeigt ein typisches *Klassendiagramm*.

Eine Klasse repräsentiert ein fachlich relevantes Konzep: Eine Menge von Personen, Dingen oder Ideen, die in einem IT-System abgebildet werden. Im

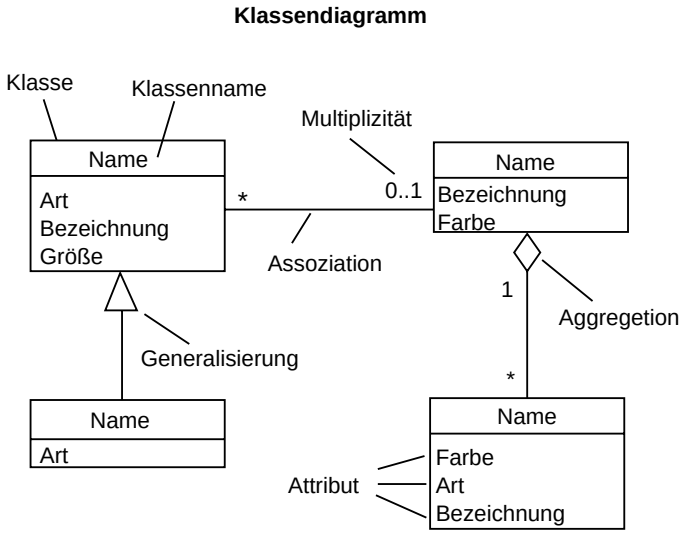

Abbildung 2.4: Das Klassendiagramm

Klassendiagramm können die struckturellen Teile eines Geschäftssystems dargestellt werden, d.h. die Beziehungen, die die einzelnen Mitarbeiter, Geschäftsobjekte und außenstehende Parteien miteinander haben.

- Ein *Attribut* einer Klasse repäsentiert ein Merkmal einer Klasse, das für die Benutzer des IT-Systems interessant ist.
- Die *Generalisierung* ist eine Beziehung zwischen zwei Klassen, einer generellen und einer speziellen Klasse.
- Eine *Assoziation* repräsentiert eine Beziehung zwischen zwei Klassen. Sie besagt, das Objekte der einen Klasse eine Verbindung zu Objekten der anderen Klasse haben, wobei diese Verbindung eine genau definierte fachliche Bedeutung hat. Die Assoziation muss entsprechend beschriftet sein. Die Assoziation gilt in beide Richtungen.
- Die *Multiplizität* erlaubt Aussagen über die Anzahl der Objekte, die an einer Assoziation beteiligt sind.
- Die *Aggregation* ist ein spezieller Fall einer Assoziation mit der Bedeutung „besteht aus". Diese Bedeutung wird durch den Rombus dokumentiert, eine Beschriftung erübrigt sich.

2.2 Petrinetze

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

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.

Ein Petrinetz ist ein bipatiter Graph, der aus Objekten, Ereignissen und Kanten besteht. Petrinetze dienen der Beschreibung verteilter, diskreter und nebenläufiger Systeme. Es wird in Petrinetzen zwischen Zuständen und Zustandsänderungen, in denen im allgemeinen jeweils mehrere Teilzustände geändert werden, unterschieden. 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. Formal kann man sagen:

Definition 1 (Netz) *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: $F \subseteq S \times T + T \times S$.*

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 sagt auch Netzgraph dazu. 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] hat ein Petrinetz u.A. folgende Eigenschaften.

Definition 2 *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 $A \subset S \cup T$ 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 gleich 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 denselben Vor- und denselben 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.

2.2.1 Bedingungs/Ereignis-Netze

Bedingungs/Ereignis-Netze sind nach [Bau90] 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 meistens als ein schwarzer Punkt in der Stelle dargestellt wird. Diese Netze sind also an ihren Stellen entweder mit einer Marke markiert oder unmarkiert. Eine Markierung M eines Netzes N ist eine Teilmenge von $S : M \subset S$

Eine Marke heißt:„Bedingung gilt", keine Marke 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 Markierungen in einem Netz zeigt den momentanen Zustand an, in dem sich ein System gerade befindet.

Transitionen sind auch hier die aktiven Elemente. Sie können, wenn sie aktiviert sind, schalten (feuern), d.h. sie können Markierungen der Eingangstellen entfernen und in den Ausgangsstellen wieder erzeugen. Aktiviert ist eine Transition dann, wenn in ihren Eingangsstellen eine Marke liegt, die die Transition zum Schalten benötigt, und in den Ausgangsstellen keine. Voraussetzung für das Schalten einer Transition ist, daß alle Eingangsstellen und Nebenbedingungen markiert sowie alle Ausgangsstellen unmarkiert sind.

Definition 3 *Eine Transition t ist aktiviert in M, notiert als $M[t\rangle$, falls*

1. *$^\bullet t \subset M$ („genug Marken")*
2. *$t^\bullet \cap M = \emptyset$ („kontaktfrei")*

Eine aktivierte Transition t kann in die Nachfolgemarkierung M' schalten: $M' = (M \setminus {}^\bullet t) \cup t^\bullet$.

Um zu kennzeichnen, von welcher Eingangsstelle zu welcher Ausgangsstelle eine Markierung gefeuert wurde, benötigt man die Kanten, die als Pfeile dargestellt werden. Die Kanten können von einer Stelle zu einer Transition und von der Transition wieder zu einer Stelle gehen. 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 $W(x, y) \leq 1$.

Es gibt vier typische Muster an Schaltfolgen, die in Petrinetzen auftreten können.

1. Bei einem Muster wird das Netz sequentiell ausgeführt. Es feuert eine Transition nach der anderen. Abbildung 2.5 zeigt so ein Verhalten.

2. Im nächsten Muster findet eine Synchronisation statt, d.h. eine Transition vereinigt zwei sonst unabhängige Äste eines Netzes. In Abbildung 2.6 ist so eine Schaltfolge zu sehen. Wichtig ist, das eine Transition erst feuern kann, wenn alle Stellen, die Kanten auf diese eine Transition gezeichnet

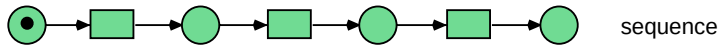

Abbildung 2.5: Eine sequentielle Schaltfolge

haben, sämtliche Markierungen enthalten, die zum Feuern der Transition notwendig sind. Eine Synchronisation wird sozusagen erzwungen, denn sollte eine Stelle noch nicht bereit sein, da noch Marken fehlen, wartet die Transition mit dem feuern, bis wirklich alle Stellen bereit sind.

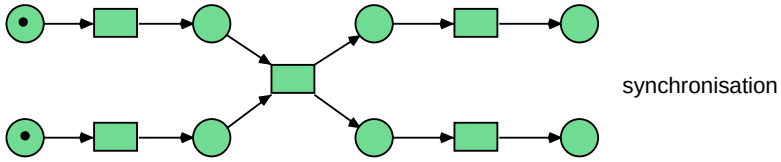

Abbildung 2.6: Eine Synchronisation

3. Ein drittes Muster beschreibt einen Konflikt (Abbildung 2.7). 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 wird, bzw. zuerst feuern wird. Es können nicht mehrere Transitionen gleichzeitig feuern. Unter Konflikt versteht man die nicht nebenläufige gleichzeitige Aktiviertheit von Transitionen.

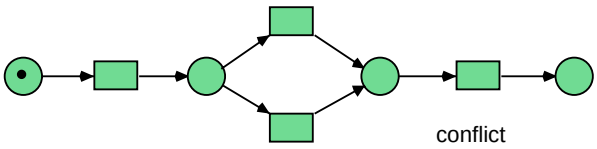

Abbildung 2.7: Eine Konfliktsituation

4. Das letzte Muster beschreibt ein nebenläufiges Verhalten. Von einer Transition gehen Kanten auf mehrere Stellen. Diese Stellen bilden in dem Netz dann jeweils den Anfang eines Astes des Netzes. Abbildung 2.8 zeigt so ein Verhalten. Wenn die Transition feuert, bekommt jede dieser Stellen eine oder mehrere Marken, je nach Beschriftung der Pfeile. Man kann jetzt aber für den weiteren Verlauf des Netzes nicht sagen, ob die folgenden Transitionen der einzelnen Netze gleichzeitig oder in einer beliebigen Reihenfolge schalten. Sollte es später wieder zu einer Synchronisation kommen, und die Transition mit dem Feuern warten muß, kann man erkennen, daß das Schaltverhalten der Transitionen der einzelnen Äste nicht zeitgleich war. Voraussetzung für so eine Aussage ist natürlich, daß die einzelnen Äste gleich aufgebaut sind.

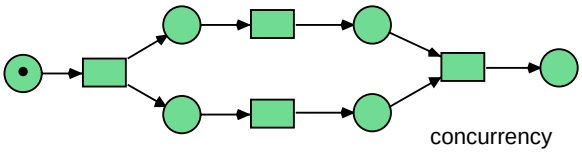

Abbildung 2.8: Ein nebenläufiges Verhalten

2.2.2 Stellen/Transitions-Netze (S/T-Netze)

Stellen/Transitions-Netze sind Netze, in denen die Stellen eine bestimmte Kapazität für Marken haben können. Es ist also 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 in einer Stelle entfernt und auch mehrere Marken in einer anderen Stelle wieder erzeugt werden. Es können aber auch mehrere entfernt und nur eine erzeugt werden, das hängt immer vom Kantengewicht (auch Kantenbewertung genannt) ab. 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. 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] ein S/T-Netz folgendermaßen definiert:

Definition 4 (S/T-Netz) *Ein markiertes, unbeschränktes Stellen / Transitionsnetz – im folgenden kurz: S/T-Netz – ist das Tupel:*

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

- *S ist die Stellenmenge.*
- *T ist die Transitionsmenge, wobei $S \cap T = \emptyset$.*
- *F ist die Flußrelation: $F \subseteq S \times T + T \times S$.*
- *W ist die Kantengewichtungsfunktion $W : S \times T + T \times S \to \mathbb{N}$, wobei $W(x, y) = 0$ gdw. $(x, y) \notin F$*
- *einer Kapazitätsfunktion $K : S \to \mathbb{N} \cup \{\omega\}$. ($K(s) = \omega$ bedeutet: keine endliche Kapazität),*

Das Tupel (N, m_0) beschreibt ein S/T-Netzsystem, wobei $m_0 : S \to \mathbb{N}$ die Anfangsmarkierung des Systems ist und $m_0(s) \leq K(s)$ für alle $s \in S$ gilt.

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.

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

Definition 5 *Es sei $N = (S, T, W, K)$ ein Netz mit endlicher Stellenmenge $S = \{s_1, ..., s_n\}$. Eine Markierung (marking) m für N ist eine Verteilung von Marken auf den Stellen oder formal eine Abbildung von S in die nicht negativen, ganzen Zahlen $\mathbb{N} := \{0, 1, 2, ...\}$:*

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

$m(s_i) \in \mathbb{N}$ gibt die Anzahl der Marken (englisch: tokens) in der Stelle s_i an. Mit M_S oder M bezeichnet man die Menge aller Markierungen über S.

Damit eine Transition eines S/T-Netzes schalten kann, muß es vorher aktiviert sein. Eine Transition $t \in T$ heißt aktiviert unter M, $M[t\rangle$, wenn für alle $s \in {}^\bullet t : M(s) \geq W(s,t)$ und für alle $s \in t^\bullet : M(s) \leq K(s) - W(t,s)$.

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 $M[t\rangle M'$, 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 und wird auch als M_t geschrieben. Die Aktivierungsbedingungen und die Definition der Folgemarkierungen bezeichnet man als Schaltregel.

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

Definition 6 *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 $M_1[t\rangle$, falls $M_1(s) \leq W(s,t)$ 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 $M_1[t\rangle M_2$, 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 $M_1[t\rangle$ bzw. $M_1[t\rangle M_2$.

Für eine Schaltregel gilt: es sei (N, M_0) ein S/T-Netz und $w = t_1, ...t_n \in T^\star$. Man nennt eine Markierung $M'' \in M(N)$ eine Folgemarkierung von M_0 unter w und schreibt $M_0[w\rangle M''$, wenn gilt:

1. $w = \lambda$ und $M'' = M$ oder
2. Es gibt ein $M' \in M(N) : M_0[t_1...t_{n-1}\rangle M'$ und $M'[t_n\rangle M''$.

Man kann also sagen, eine Transition t eines S/T-Netztes ist dann aktivierbar, wenn eine Schaltfolge w mit $M_0[w\rangle$ existiert. Durch wiederholtes Schalten erhält man eine Schaltfolge, bzw. einen seriellen Prozeß.

2.2.3 High-Level Nets (Prädikaten/Transitionsnetze)

Prädikaten/Transitions-Netze erlauben es, Marken individuell zu unterscheiden. Durch sie können komplexere Systeme einfacher beschrieben werden. Prädikaten/Transitions-Netze (Pr/T-Netze) können als Faltung von S/T-Netzen aufgefaßt werden, daher werden sie als höhere Netztypen (high level nets) bezeichnet.

Hat man zum Beispiel ein Netz, in dem mehrere gleiche Subnetze auftreten, kann man das Netz vereinfachen, indem man die Struktur eines Subnetzes nur einmal zeichnet. Man muß allerdings darauf achten, daß man die unterschiedlichen Zustände, in denen sich die einzelnen Subnetze vorher befinden konnten, beibehält. Man muß also für jedes darzustellende Subnetz eine eigene Beschriftung haben. D.h. unterschiedliche Beschriftungen symbolisieren unterschiedliche Datentypen.

In einem höheren Netz kann eine Substruktur durch eine einzige Transition dargestellt werden. Man nennt diese Transition dann Funktionseinheit. In einem S/T-Netz würde der gleiche Prozeß mehrere Stellen und Transitionen in Anspruch nehmen.

Unter einer Funktionseinheit versteht man ein durch Aufgabe und Wirkung abgrenzbares Gebilde, wobei Aufgabe oder Wirkung aus Handlungen bestehen. Hierzu kann etwa die Verarbeitung von Daten ebenso wie die Übertragung oder Speicherung gehören.

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 werden den Marken also Werte zugeordnet, wodurch man sie unterscheiden kann. Dieses zusätzliche Ausdrucksmittel erlaubt es Sachverhalte kompakter auszudrücken und Algorithmen zu modellieren. Marken können aber auch den gleichen Wert haben, d.h. die Markierung einer Stelle ist eine Multimenge. Die Kanten solcher Netze werden mit Variablen, z.B. x und y, bewertet, die diese Individuen als Werte annehmen können. Es kann sich dabei um Zahlen, Strings, Tupel oder andere komplexe Datentypen handeln.

Im einfachsten Fall ändert sich die Datenstruktur beim Feuern einer Transition nicht, prinzipiell ist das aber möglich. Auch in den Schaltregeln muß Bezug auf den Wert der Marken genommen werden, d.h. dem Schalten einer Transition t muß also 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 beschriftet sein. Wenn die Transition feuert, muß jede Variable an einen festen Wert gebunden sein. Die Variablen entscheiden so, welcher Wert (welche Marke) zur nächsten Stelle bewegt wird. Wie bei S/T-Netzen ist t aktiviert, falls die zu entnehmenden Individuen überhaupt vorhanden sind und eventuell vorgeschriebene Kapazitäten nicht überschritten werden. Netze mit solchen Eigenschaften bezeichnet man auch als gefärbte Netze.

Gefärbte Petri–Netze (coloured Petri nets, CPN) werden von Jensen in [Jen92] vorgestellt. Sie sind wie folgt definiert:

Definition 7

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

- 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) entnommen.

2.2.4 Referenznetze

Möchte man die Vorteile eines high-level Netzes nutzen, aber wegen der Übersichtlichkeit nicht alles in einem Netz darstellen, kann man mehrere Netze zeichnen und sie über Kanäle miteinander synchronisieren. Geht man zum Beispiel von zwei Netzen aus, muß eine Transition in beiden Netzen vorkommen, d.h. die Transition muß in beiden Netzen die gleiche Beschriftung haben.

In einem Netz kann man dann an einer Transition das andere Netz durch seinen Namen und die Beschriftung der Transition aufrufen. Solche Notationen nennt man synchrone Kanäle, weil sie die Transition zwingen, synchron zu feuern und den Informationsfluß zu leiten.

Wenn man sprechende Kanalnamen vergibt, kann man viele Synchronisationen auf einmal durchführen, ohne daß das Netz unübersichtlich wird.

Für klassische Petrinetze existieren die Stellen und Transitionen, die gezeichnet wurden, exakt einmal während der Ausführung. Man kann jetzt verallgemeinern und mehrere Instanzen jedes einzelnen Netzes erlauben, damit verschiedene Instanzen ausgeführt werden können. Man erlaubt deshalb Referenzen zu Instanzen eines Netzes und behandelt diese so wie vorher die Marken in einfachen Netzen. Solche Netze nennt man Referenznetze. Ein Netz kann eine oder mehrere Instanzen eines anderen 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 des Pfeils zu dieser Stelle an der Transition vor das *new* geschrieben werden

2.3 Stochastische Modelle

In dieser Arbeit sollen drei wesentliche stochastische Modellierungsansätze vorgestellt werden. Es handelt sich dabei um Markov-Ketten, Warteschlangensysteme und Stochastische Petrinetze.

2.3.1 Markov-Ketten

Die einfachste Form der abhängigen Koppelung und gleichzeitig die wichtigste für die Modellierung mehrstufiger praktischer Probleme ist die sogenannte Markov-Koppelung, benannt nach A.A.Markov, der solche Modelle zu Beginn dieses 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 Markov-Kopplung. Die Folge der Beobachtungen bildet dann einen Markov-Prozeß, im diskreten Fall auch Markov-Kette genannt.

Nach [Hüb96] ist eine Markov-Kette (MK) die Folge der Beobachtungen $X_0, X_1, X_2, ...$ in einem unendlichstufigen Versuch mit Markov-Kopplung und abzählbarer Zustandsmenge I. Die Zufallsvariablen $X_n : \Omega \to I$ beschreiben also den Zustand eines Systems zu den Zeitpunkten $n = 0, 1, 2, ...$

Eine Markov-Kette (X_n) heißt homogen, falls die Übergangswahrscheinlichkeiten (ÜZ-Dichten) $f_n^{n-1}(i;j) = P(X_n = j | X_{n-1} = i)$ für alle Zeitpunkte gleich sind. In diesem Fall schreibt man $p_{ij} := f_n^{n-1}(i;j)$.

Die Matrix $(p_{ij}) := (p_{ij}, i, j \in I)$ heißt Übergangsmatrix, kurz Ü-Matrix. Sie kann auch (abzählbar) unendlich viele Zeilen und Spalten besitzen. Die Zeilensumme ist stets gleich 1. Solche Matritzen nennt man „stochastische Matritzen". Zusätzlich zur Ü-Matrix braucht man noch einen festen Startpunkt $i_0 \in I$ oder für den Fall eines zufälligen Startpunktes noch eine sogenannte Startverteilung.

Definition 8 Markov-Eigenschaft*: Die Wkt.-Vtlg. für den Folgezustand hängt nur vom derzeitigen Zustand ab:*

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

Ein homogener *Markov-Prozeß liegt vor, falls die Übergangswkt. nicht vom Zeitpunkt t abhängt:*

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

Beispiel: Das Machine-repairman-Modell Ein einfaches Beispiel für eine derartig zu analysierende Markovkette ist in Abb. 2.9 dargestellt.

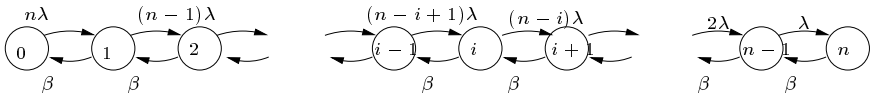

Abbildung 2.9: Machine-repairman-Modell

Jeder Markovprozeß, für den gilt, daß von jedem Zustand jeder Zustand erreichbar ist, heißt irreduzierbar. Jeder irreduzierbarer Markovprozeß besitzt eine konvergierende Zustandsverteilung. Für eine konvergierende Zustandsverteilung, d.h. falls $\lim p_i(t)=\pi_i$ existiert, läßt sich ein Markovprozeß mit Hilfe des Flußgleichgewichts beschreiben:

$$\sum_{j\neq i}\pi_j\lambda_{ji}=\pi_i\sum_{j\neq i}\lambda_{ij}$$

Es ergibt sich für diese konkrete Markov-Kette folgendes Gleichungssystem:

$$\begin{aligned}
p_0 n\lambda &= p_1\beta\\
p_1((n-1)\lambda+\beta) &= p_0 n\lambda + p_2\beta\\
&\vdots\\
p_i((n-i)\lambda+\beta) &= p_0(n-i+1)\lambda + p_{i+1}\beta\\
&\vdots\\
p_n\beta &= p_{n-1}\lambda
\end{aligned}$$

Diese Gleichungen besitzen eine anschauliche Beschreibung: Im Gleichgewicht ist der Fluß aus einem Knoten heraus genauso groß wie der Fluß in den Knoten hinein.

Mit der Abkürzung $\rho=\lambda/\beta$ erhalten wir durch Umformungen:

$$p_i=\frac{n!}{(n-i)!}\rho^i p_0$$

Aufgrund der stochastischen Normierung ergibt sich p_0:

$$1=\sum_0^n p_j=\sum_0^n\frac{n!}{(n-j)!}\rho^j p_0 \Longrightarrow p_0=\left(\sum_0^n\frac{n!}{(n-i)!}\rho^i\right)^{-1}$$

2.3.2 Elementare Wartesysteme (M/M/m/k)

Eine besondere Klasse von Markovprozessen ist als Wartesysteme bekannt. Wartesysteme werden kurz als $(M/M/m/k)$ notiert, wobei das erste M den Ankunftsprozeß beschreibt (hier: Markov, d.h. exponential verteilt), das zweite M den Bedienprozeß, m die Anzahl der Bediener ist und k die Größe des Warteraumes.

Für Exp.-Vtlg mit den Parameter λ bzw. μ ergeben sich folgende Erwartungswerte:

Zwischenankunfszeit A	$E[A] = \frac{1}{\lambda}$
Zugang Z	$E[Z] = \frac{t_2-t_1}{E[A]}\frac{1}{t_2-t_1} = \lambda$
Durchsatz D	$E[D] = \lambda$
Grenzdurchsatz C	$E[C] = \mu$
Bedienzeit B	$E[B] = \frac{1}{\mu} = \frac{1}{E[C]}$

Mit $\rho := \lambda/(m\mu)$ ergibt sich aus der Analyse des Markovsystems:

$$\begin{aligned} p_i &= \frac{(\rho m)^i}{i!}p_0 \qquad i \in \{1..m\} \\ p_i &= \frac{\rho^i m^m}{m!}p_0 \qquad i \in \{m+1..k\} \\ p_0 &= \left(\sum_{j=0}^{m}\frac{(\rho m)^j}{j!} + \sum_{j=m+1}^{k}\frac{(\rho m)^m}{m!}\right)^{-1} \end{aligned}$$

Daraus ergibt sich der Erwartungswert der Füllung:

$$E[F] = \sum_{f=1}^{k} f p_f$$

Ein einfach zu analysierendes System ergibt sich für einen unendlichen Warteraum und einen Bediener, kurz $(M/M/1)$.

Ein Bedienprozeß $(M/M/1)$ Es ergibt sich für $m = 1$, $k \to \infty$ folgende Vereinfachung (unter der Voraussetzung, daß $\rho < 1 \iff \lambda < \mu$):

$$p_0 = \left(\sum_{j=0}^{m}\frac{(\rho m)^j}{j!} + \sum_{j=m+1}^{k}\frac{(\rho m)^m}{m!}\right)^{-1} = \left(\sum_{j=0}^{\infty}\rho^j\right)^{-1} = \left(\frac{1}{1-\rho}\right)^{-1} = 1-\rho$$

Daraus folgt für die p_i

$$p_i = \frac{\rho^i m^m}{m!}p_0 = \rho^i(1-\rho)$$

Der Erwartungswert der Füllung ist dann

$$\begin{aligned} E[F] &= \sum_{i=1}^{k} i p_i = \sum_{i=1}^{\infty} i\rho^i(1-\rho) = (1-\rho)\sum_{i=1}^{\infty} i\rho^i \\ &= (1-\rho)\rho\sum_{i=1}^{\infty} i\rho^{i-1} = (1-\rho)\rho(\sum_{i=1}^{\infty}\rho^i)' = (1-\rho)\frac{\rho}{(1-\rho)^2} = \frac{\rho}{1-\rho} \end{aligned}$$

Daraus ergibt sich dann für den Erwartungswert der Füllung F, der Verweilzeit Y und der Wartezeit W:

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

Geht man von einfachen Wartesystemen über zu Wartenetzen (Netzen von Wartesystemen), so sind auch komplexe Szenarien wie bspw. das hier dargestellte Terminal analytisch zu lösen. Eine Behandlung von Wartesystemen liegt aber außerhalb des Rahmens dieser Arbeit.

2.3.3 Stochastische Petrinetze

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

Seit den 70er Jahren gibt es aber einige veröffentlichte Anregungen zum Integrieren von zeitverbrauchenden Funktionen in Petrinetzen. In [BK96] werden zwei verschiedene Möglichkeiten von Klassifizierungen 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 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 wieder in zwei Klassen,

1. „Preselection models" und

2. „Race models",

geteilt werden.

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 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 vorgesehene Zeitintervall vergangen ist und feuert dann sofort danach. Es ist bei diesem Petrinetztyp allerdings möglich, daß eine Transition nicht mehr aktiviert ist, wenn ein Zeitintervall vergangen ist. Das liegt daran, das die Marken nicht reserviert wurden und andere aktivierte Transitionen, die ein kürzeres Zeitintervall gewartet haben, die Marken verbraucht haben. In so einem Petrinetz können Transitionen also 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", die stochastischen „stochastic Petri Nets".

In [BK96] werden stochastische Petrinetze als ein Tupel $SN = (N, \Lambda)$ beschrieben, bestehend aus einem $S/T - NetzN = (S, T, F, K, W, m_0)$ und einer Menge $\Lambda = (\lambda_1, ..., \lambda_m)$. λ_i ist die markierungsabhängige Übergangsrate der Transition t_i.

Im allgemeinen ist die Feuerungszeit exponentiell verteilt. Die Verteilung der Zufallsvariable F_{χ_i} der Feuerungszeit einer Transition ist durch

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

gegeben.

In Verbindung mit der Tatsache, daß die Möglichkeit der Zustandsänderung unabhängig von der Verweilzeit ist, heißt das, daß ein stochastisches Petrinetz einen Markov-Prozeß beschreibt. Die quantitative Analyse eines stochastischen Petrinetzes kann durchgeführt werden, indem man einfach den korrespondierenden Markov-Prozeß analysiert. Markov-Ketten eines stochastischen Petrinetzes können von dem Erreichbarkeitsgraphen des zugrunde liegenden S/T-Netzes erzeugt werden. Der MK Zustandsraum ist die Erreichbarkeitsmenge $R(N)$ und die Übergangsrate eines Zustands M_i zu einem Zustand M_j ist gegeben durch $q_{ij} = \lambda_k$, die Feuerungsrate der Transition t_k. Wenn mehrere Transitionen von M_i nach M_j feuern, dann ist q_{ij} die Summe der Feuerungsraten der einzelnen Transitionen. Genauso ist $q_{ij} = 0$, wenn keine Transition von M_i nach M_j feuert.

2.4 Simulation

Nach [Pag92] umfaßt die Simulation die Disziplin des Modellentwurfs, der Systemanalyse, des Softwaredesigns und die der Implementation. Zusätzlich stellt sie auch einen interdisziplinären Bezug zum Anwendungsbereich dar.

Um die Simulation verstehen zu können, müssen als erstes die Begriffe "System" und "Modell" aus der Systemtheorie erklärt werden.

2.4.1 System und Modell

Ein System kann man sich als einen speziell ausgewählten Bereich der Realität vorstellen, der unter einer speziellen Fragestellung betrachtet wird. Das System besteht aus mehreren Komponenten, die innerhalb des Systems einen bestimmten Zweck bzw. eine bestimmte Aufgabe erfüllen, indem sie miteinander interagieren. Eine einzelne Komponente ist meistens nicht in der Lage, eine Aufgabe allein zu erfüllen. Oft sind zu betrachtende Systeme sehr komplex und ihre Struktur und ihr Verhalten nicht unmittelbar durchschaubar. Es gibt verschiedene Arten von Systemen, wie zum Beispiel natürliche, technische oder soziale. Systeme können aber auch Ideengebilde von noch nicht realisierten Projekten sein.

Modelle dienen dazu, ein besseres Verständnis für Vorgänge, Wirkungsweisen und Verhalten eines realen Systems zu entwickeln. Dazu ist vorher eine detaillierte Systemanalyse nötig, um die Systemstruktur und die Systemkomponenten zu erkennen. Modelle sind ebenfalls Systeme, die die Elemente und Relationen des Ursprungssystems in veränderter Weise darstellen. Ein Modell stellt das Originalsystem also vereinfacht dar, dadurch wird erst die Untersuchung komplexer Systeme handhabbar. Zu einem System können mehrere verschiedene Modelle erstellt werden, je nach Zielsetzung der Modellstudie und der subjektiven Sichtweise des Modellbildners.

„Modelle sind materielle oder immaterielle Systeme, die andere Systeme so darstellen, daß eine experimentelle Manipulation der abgebildeten Strukturen und Zustände möglich ist." ([Nie77], S.77), zitiert nach [Pag00].

Man kann Modelle nach verschiedenen Gesichtspunkten klassifizieren: Nach Art der Untersuchungsmethode, nach Abbildungsmedium, nach Art der Zustandsübergänge und nach Art des Verwendungszwecks.

Bei den Untersuchungsmethoden unterscheidet man zwischen *analytischen Modellen* und *Simulationsmodellen*. Modelle mit analytischem Lösungsansatz erlauben es, in ein Gleichungssystem, das die vorhandenen Systembeziehungen wiederspiegelt, bestimmte angenommene Werte einzusetzen und in einem geschlossenen Lösungsdurchlauf den zu ermittelnden Systemzustand direkt zu bestimmen. In Simulationsmodellen dagegen wird der Modellzustand Schritt für Schritt fortgeschrieben, d.h. die Zwischenergebnisse der Berechnung besitzen eine Interpretation als Zwischenzustände des Originals. Aus diesem Grund eignen sich Simulationsmodelle besonders zur Veranschaulichung des Systemverhaltens. Analytische Untersuchungen sind außerdem wegen mathematischer Restriktion auf Systeme mit relativ geringer Komplexität begrenzt.

Es gibt verschiedene Arten von Modellen, die nicht notwendigerweise in einem Formalismus dargestellt werden müssen. Abbildungsmedien können sein:

- materielle Modelle, wie zum Beispiel Schiffsmodelle,
- verbale Modelle, das sind umgangssprachliche Beschreibungen,
- grafisch-deskriptive Modelle, wie Flußdiagramme,

- mathematische Modelle, wie Gleichungssysteme und
- grafisch-mathematische Modelle, wie zum Beispiel Petrinetze.

Bei der Klassifikation nach Art der Zustandsübergänge unterteilt man die Modelle zuerst in *statische* und in *dynamische Modelle*. Bei einem statischen Modell treten keine Zustandsänderungen auf. Bei dynamischen Modellen sind die Modellzustände dagegen zeitabhängig.

Die dynamischen Modelle teilt man wiederum in die *kontinuierlichen* und die *diskreten* auf. Die Zustandsvariablen eines kontinuierlichen Modells lassen sich durch stetige Funktionen beschreiben. Die diskreten Modelle ändern dagegen die Werte ihrer Zustandsvariablen sprunghaft zu bestimmten, auf der Zeitachse diskret verteilten Zeitpunkten.

Sowohl die kontinuierlichen als auch die diskreten Modelle teilen sich noch einmal in *deterministische* und *stochastische Modelle* auf. Deterministisch heißt ein Model, wenn seine Reaktion auf eine bestimmte Eingabe, ausgehend von einem bestimmten Zustand, eindeutig festgelegt ist. Wenn dies nicht der Fall ist, d.h. wenn sich Reaktionen des Modells nur durch Wahrscheinlichkeitsverteilungen beschreiben lassen, nennt man es stochastisch.

Eine letzte Klassifizierung von Modellen ist die nach Verwendungszweck. Der Verwendungszweck eines Modells ist durch die Fragestellung und Zielsetzung der Modellstudie festgelegt. Abhängig vom Verwendungszweck werden unterschiedliche Anforderungen an Modelle gestellt. Man unterscheidet

- Erklärungsmodelle,
- Prognosemodelle,
- Gestaltungsmodelle und
- Optimierungsmodelle.

Die Simulation verwendet man also, um Folgen eines Eingriffs in ein System abschätzen oder die Realisierbarkeit eines Projekts testen zu können. Um Erkenntnisse über Systeme zu gewinnen, ist es meist sinnvoll, die Untersuchung nicht am System selbst vorzunehmen, sondern ein Modell zu bilden, das die wesentlichen Eigenschaften eines Systems abbildet. Dafür benötigt man Detailwissen einzelner Objekte des Systems, d.h. man muß deren Eigenschaften kennen. Außerdem braucht man noch Kenntnisse über Wirkungsgefüge, die die einzelnen Objekte miteinander eingehen. 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.

„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" ([Page91],S.7), zitiert nach [Pag00].

Um komplexe Systeme verstehen zu können und handhabbar zu machen, findet eine *Abstraktion* des Realsystems auf das Modell statt. Dadurch zeigt

sich auch, welche Eigenschaften oder Aspekte eines Systems elementar sind und auf jeden Fall in dem Modell berücksichtigt werden müssen, welche unwichtig sind und unbedingt weggelassen werden sollten, und welche, je nach Sicht des Betrachters, berücksichtigt werden können aber nicht müssen. Ob ein Aspekt als wichtig oder unwichtig angesehen wird, hängt zusätzlich von der Fragestellung ab, auf die das System hin untersucht wird. Im allgemeinen werden nur solche Modelle als Simulationsmodelle bezeichnet, die keine analytische Behandlung erlauben, d.h. Simulationsmodelle in diesem Sinne können keiner mathematischen Methode 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 erstrebte Abbildungsgenauigkeit erreicht ist. Simulationsmodelle haben gegenüber analytischen Modellen folgende Vorteile:

- Mit Simulationsmodellen lassen sich im Gegensatz zu analytischen Modellen auch komplexe Systemstrukturen untersuchen, die die Einschränkungen analytischer Verfahren nicht erfüllen.
- 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 aber 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. Z.B. ist für ein Simulationsmodell nur schwer entscheidbar, ob sich das System noch in der „Einschwingphase" befindet oder schon im stabilen Bereich.

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. Deshalb ist es nötig, 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 aussagefähig. Man muß davon ausgehen, daß das Modell fehlerhaft ist. Bei geringen 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 sind. In unserem Beispiel bildet die Abschätzung durch ein sehr einfaches analytisches Modell - siehe Kapitel 4.1 - eine benötigte Plausibilitätskontrolle.

2.4.2 Zeitdiskrete Simulation

Thema dieser Arbeit sind diskrete Simulationsmodelle. Da dieser Modellansatz eine Diskretisierung der Zeit und damit eine bestimmte Betrachtungsweise der Realsysteme voraussetzt, spricht man auch von der Simulation zeitdiskreter Systeme (siehe auch [Pag00]).

Zeitdiskrete Simulationsmodelle treten in vielfältiger Form auf und werden für die unterschiedlichsten Problemstellungen eingesetzt. Zeitdiskrete Simulationen sind häufig zufallsabhängig, es müssen also in so einem Fall statistische Verfahren zur Zufallszahlenerzeugung herangezogen werden. Die Simulation ist also ein Instrument zur Analyse und Modellbildung komplexer Systeme. Die Simulation kann aber nicht nur in realexistierenden Systemen hilfreich sein, sondern auch in geplanten Systemen, indem sie von hypothetischen Systemen die Prognose, Optimierung und Validierung ermöglicht.

Bei *zeitdiskreten Simulationsmodellen* werden die Änderungen des Systemzustandes nur zu bestimmten, diskreten Zeitpunkten betrachtet. Bei einem zeitdiskreten Simulationsmodell wird, durch eine endliche Folge von Zuständen, die zeitliche Entwicklung des Systems betrachtet. Die Zeitpunkte zwischen diesen bestimmten Zuständen 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 des Systems zu schaffen.

Im Gegensatz dazu findet bei *zeitgesteuerten diskreten Modellen* die Neuberechnung des Systems zu bestimmten äquidistanten Zeitpunkten statt. Die zeitgesteuerte diskrete Simulation wird in dieser Arbeit nicht weiter berücksichtigt.

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. Solche Systeme sind zum Beispiel Bediensysteme wie das Containerterminal, das in Kapitel 3.1 dieser Arbeit beschrieben wird. Ein zeitdiskretes Simulationsmodell besteht aus Systemobjekten, die Entitäten genannt werden. Die Komponenten eines Systems werden durch solche Entitäten modelliert. Diese Entitäten stehen in Wechselwirkung zueinander, genau wie die Komponenten eines realexistierenden Modells auch. Eine Entität kann als Objekt der objektorientierten Programmierung aufgefaßt werden, dessen Verhalten im Verlauf einer bestimmten Simulationszeit defi-

niert ist. Eine Entität wird zusätzlich durch ihren Zustand und durch Transformationsregeln charakterisiert. Diese Transformationsregeln verändern den Entitätszustand während der Simulationszeit. Damit eine Zustandstransformation zu einem bestimmte 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. „Eine Entität ist ein Objekt, welches in der Lage ist, sich (aktiv) in der Simulationszeit fortzubewegen." ([SH95], S.9)

2.4.3 Modellierungsansätze in der zeitdiskreten Simulation

In der klassischen diskreten Simulation haben sich zwei Modellierungsansätze entwickelt, die bei der Modellierung der Entitäten eine unterschiedliche Sichtweise einnehmen. Der eine Ansatz ist die prozessorientierte Modellierung, die die am weitesten verbreitete Sichtweise darstellt. Der zweite Modellierungsansatz ist die ereignisorientierte Simulation.

Prozessorientierte Simulation

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 in einem Prozeß abgebildet werden. Während ein Prozeß aktiv ist, kann Simulationszeit vergehen, d.h. eine Entität muß in der Lage sein, während ihr Lebenszyclus durchlaufen wird, die Kontrolle vorübergehend an die Zeitführungsroutine (Scheduler) der Simulation abzugeben, damit sie die Simulationsuhr weiter schalten kann. Aus der Sicht des Prozesses vergeht dann die Zeit, wenn er selber aktiv ist. Die zeitkonsumierenden Aktivitäten eines Prozesses und seine passiven Phasen werden durch inaktive Prozesszustände dargestellt. Es werden also zwei Arten von inaktiven Phasen eines Prozesses unterschieden. Im ersten Fall bildet der Prozeß eine zeitkonsumierende Tätigkeit ab; er ist also konzeptionell aktiv. Aber 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 zum Beispiel 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 aber, daß ein Prozeß nach der Durchführung einer Zustandsänderung noch 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 immer nur ein Prozeß im aktiven Zustand sein kann, gibt es so etwas wie pseudo parallele Abläufe beliebig vieler Prozesse zu einem bestimmten Simulationszeitpunkt. Pseudoparallel sind sie deshalb, da sie nur sequentiell bearbeitet werden können, die Simulationsuhr aber nicht vor geschaltet wird. Daraus folgt, daß auch beliebig viele Entitäten parallel agieren können.

Ereignisorientierte Simulation

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 von einem Ereignis zusammengefaßt. Die zwischen den Ereignissen liegenden Aktivitäten, die diese Zustandsänderungen erst 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 beim prozessorientierten Ansatz müssen beim ereignisorientierten Ansatz auch zuerst die relevanten Systemobjekte und ihre Attribute identifiziert werden. Der Modellierer nimmt aber 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.

2.5 Werkzeuge

Um die schon erwähnten Methoden der Simulation durchführen zu können, braucht man geeignete Werkzeuge. Für die Simulation mit Petrinetzen wird das Petrinetzwerkzeug Renew und für die andere Methode das Simulationsframework Desmoj eingesetzt. Beide Werkzeuge werden im Folgenden beschrieben. Da beide Werkzeuge in Java programmiert sind, gibt es einführend noch ein Kapitel zu Java.

2.5.1 Java

Java ist eine Programmiersprache, die von der Fima Sun ursprünglich für elektronische Geräte konzipiert wurde und heute die wichtigste Programmiersprache im Internet ist.

Eines der wesentlichen Merkmale von Java ist die starke Anlehnung an das Konzept der objektorientierten Programmieung. Die Syntax und viele Konzepte orientieren sich stark an der Sprache C++. Verglichen mit C++ ist Java aber deutlich einfacher aufgebaut. Java kommt vor allem ohne die fehlerträchtigen Pointer aus. Auch das Konzept des Überladens von Operatoren und der mehrfachen Vererbung wurde in Java nicht realisiert. Stattdesen gibt es in Java das Konzept der Interfaces. Eine weitere wichtige Eigenschaft ist die Unabhängigkeit von der Rechnerplattform. Ein gewöhnlicher Java-Compiler erzeugt nämlich keinen Maschinencode für einen bestimmten Prozessor, sondern einen sogenannten Bytecode. Dabei handelt es sich um einen Maschinencode, der für einen virtuellen Prozessor bestimmt ist: Die sogenannte Java Virtual Machine. Der Bytecode wird während der Ausführung eines Java Programms in den prozessorabhängigen Maschinencode übersetzt. Der Übersetzer ist kein Interpreter im herkömmlichen Sinn, da er weniger Aufgaben durchzuführen hat; zum Beispiel entfallen syntaktische Überprüfungen, die bereits bei der Kompilierung durchgeführt wurden. Der Java Bytecode-Übersetzer arbeitet daher schneller als ein konventioneller Interpreter. Die Programme laufen mit entsprechend höherer Geschwindigkeit. Java Programme können auf allen Betriebssystemen laufen, für die eine Java Virtual Machine verfügbar ist.

Das Bytecode-Konzept reicht für zeitkritische Anwendungen allerdings nicht aus, daher werden inzwischen sogenannte Just-In-Time-Compiler eingesetzt, die den Bytecode einmal und dauerhaft in die Maschinensprache des jeweiligen Prozessors übersetzen. Auch beim Einsatz dieser Compiler bleibt Java Hardware unabhängig, da weiter der prozessorunabhängige Bytecode über das Netzwerk verbreitet wird.

Mit Java können beliebige Anwendungen entwickelt werden, sehr einfache bis hin zu sehr komplexen. In Verbindung mit Java lassen sich auch universelle austauschbare Softwarekomponenten entwickeln, die als Java-Beans bezeichnet werden. Ein weiterer wichtiger Aspekt von Java ist die Sicherheit. Da Java Programme nur wenige Zugriffsrechte auf das Betriebssystem haben, gelten sie als recht sicher.

2.5.2 Renew

Renew ([KW98]) ist ein Programm bzw. eine integrierte Entwicklungsumgebung, mit der man sehr leicht und schnell Referenznetze per Mausklick erstellen kann (es ist aber auch möglich UML Diagramme mit Renew zu zeichnen). Es bietet eine graphische, benutzerfreundliche Schnittstelle, um Referenznetze und zusätzliche graphische Elemente zu erstellen. Abbildung 2.10 zeigt die graphische Oberfläche von Renew.

Renew ist im Arbeitsbereich TGI von Olaf Kummer und Frank Wienberg entwickelt worden. Es ist in Java geschrieben und hat dadurch den Vorteil, daß es auf allen gängigen Betriebssystemen lauffähig ist. Außerdem ist es in Renew möglich, jede Javaklasse zu benutzen. Renew wird mit Quellcode geliefert, d.h. seine Algorithmen können erweitert und modifiziert werden. Es ist möglich, schnell spezielle Netzbeschriftungen oder einen komplett neuen Netzformalismus zuzufügen, ohne die Basisstrukturen von Renew zu ändern.

Referenznetze in Renew sind vergleichbar mit Klassen in Java, und Net-

Abbildung 2.10: Renew

zinstanzen sind vergleichbar mit Objekten. D.h. man kann in Renew objektorientiert mit Petrinetzen modellieren. In Renew sind die Referenznetze mit garbage collection für die Netzinstanzen entwickelt worden, was für saubere objektorientierte Programmierung unbedingt notwendig ist. Man kann sehr einfach Aufrufe von Javacode zu Referenznetzen machen, sowie Aufrufe von Referenznetzen zu Javacode.

In Renew sind viele Kantenarten verfügbar, die die Netzformalismen abdecken. Außerdem unterstützt Renew das Konzept der Synchronisationskanäle. Sie bilden einen sehr guten Kommunikationsmechanismus, und sie können als zuverlässiges Abstraktionskonzept verwendet werden.

Auch wenn man Java nicht lernen möchte, kann man Renew sehr gut benutzen. In den meisten Fällen reicht es aus, zu wissen, wie man Zahlen, Strings, Variablen und die einfachsten Operatoren schreibt.

Referenznetze in Renew unterstützen Erweiterungen, die über einfache high-level Petrinetze mit Java-Inschriften hinaus gehen. Referenznetze arbeiten nahtlos mit Java Programmen zusammen und gewinnen dadurch die Fähigkeiten der Java Bibliotheken. Derzeit gibt es allerdings nur sehr rudimentäre Analysewerkzeuge für Renew, d.h. es verläßt sich ganz auf die Simulation, um die Fähigkeiten eines Netzes zu untersuchen.

Die Simulation in Renew eignet sich gut, um die dynamischen und interaktiven Prozesse eines Referenznetzes zu analysieren. In dem in Renew verwendeten Petrinetzformalismus gibt es aber noch keine Notation über Schaltmöglichkeiten oder Schaltprioritäten. Da Renew eine offene Architektur hat, können diese aber noch hinzugefügt werden.

2.5.3 Desmoj

Desmoj ([PCL00]) ist ein Framework für diskrete Ereignissimulation und Modellierung in Java. Es enthält eine Anzahl an Java Packages, die benutzt werden können, um die Entwicklung von Simulationsmodellen zu vereinfachen und zu beschleunigen. Desmoj erlaubt es dem Benutzer, sich auf die Modellierung eines Systems, mit dem er experimentieren möchte zu konzentrieren, da es einige Basiskomponenten der diskreten Modellierung - wie zum Beispiel Warteschlangen - bietet. Außerdem ist es in alle Arten von Programmtypen verankerungsfähig, wie zum Beispiel in Applets, Applications oder Java Beans. Die Ergebnisse einer Simulation mit Desmoj werden als HTML Dateien produziert, die von jedem Webbrowser gelesen werden können.

Desmoj besteht aus fünf Packages,

- desmoj,
- desmoj.dist,
- desmoj.exception,
- desmoj.report und
- desmoj.statistic.

Das Package desmoj bildet eine Schnittstelle zu allen Klassen, die nötig sind, um ein ganz individuelles Modell mit seinen ganz bestimmten Optionen erstellen zu können. Desmoj.dist ist die Schnittstelle zu den Klassen der Zufahlszahlenströme. Desmoj.exception enthält die Schnittstelle zu den Excepions des Frameworks. Desmoj.report unterstützt die Basisschnittstelle, die von allen Klassen, die Nachrichten jeden Typs in Desmoj erhalten können, implementiert wurde. Desmoj.statistic enthält die Schnittstelle zu allen Klassen, die für die statistische Auswertung eines Simulators eine Rolle spielen.

Die Hauptidee eines Frameworks ist, Basisalgorithmen und geeignete Softwarearchitekturen zu unterstützen, um Probleme eines bestimmten Anwendungsgebiets zu lösen. Um eine Anwendung lauffähig zu machen, muß der Benutzer nur einige Klassen oder Schnittstellen aus dem Framework, die die Anwendung fordert, hinzufügen. Der größte Nutzen an Frameworks ist die Wiederverwendbarkeit: Man kann große Code Teile oder eine Architektur in unterschiedlichen Modellen einsetzen. Wird eine Architektur wiederverwendet, stellt das sicher, daß diese getestet und für eine weitere Anwendung geeignet ist. Das schützt vor schlechtem oder falschem Design. Weitere Vorteile der Wiederverwendbarkeit sind die Reduzierung der Entwicklungsarbeit und der Entwicklungszeit. Wiederverwendbare Software ist aber aufwendiger zu implementieren, da ein bestimmter Kontext, in dem eine Klasse benutzt wird nicht vorhergesehen werden kann. Wiederverwendbare Klassen müssen daher mit einigen Einschränkungen entwickelt werden, da man sie sonst nur selten benutzen könnte.

Die Wurzeln dieses Frameworks reichen zurück bis 1985 zu Birtwistles Desmos, ein auf Simula basierendes Framwork für die diskrete Simulation. Ein Nachfolger von Desmos ist das auf Modula-2 basierende Desmo, das 1991 im Arbeitsbereich ASI der Universität Hamburg des Fachbereichs Informatik entwickelt wurde. Es wurde in vielen Computersimulationsseminaren eingesetzt. Desmo ist ein robustes Werkzeug, mit dem es möglich war, einfach Simulationsmodelle zu implementieren. Es gibt außerdem noch eine Reihe Diplomarbeiten über Desmo Frameworks die in Oberon, Smalltalk und C++ geschrieben sind.

Wenn man an Simulationssprachen denkt, erscheint Java auf den ersten Blick nicht unbedingt als die geeignetste Sprache.Kommt es zu „number crunching" laufen Java basierte Simulationen langsamer als C++ oder C basierte. Aber der wichtigste Teil beim Simulieren ist das Bauen von Modellen, die Identifikation seiner Akteure und deren Interaktionen. Dafür eine einfache Programmiersprache zu benutzen, so wie Java, hilft Fehler zu vermeiden, während man das Modell implementiert. Die Hauptaufgabe von Desmoj ist, bei der Umsetzung eines Modells in Javacode, zu unterstützen und Implementationsarbeit abzunehmen. Desmoj ist zwar ursprünglich für die Lehre entwickelt worden,

es faßt aber einige Simulationsmethoden zusammen, die sonst kaum irgendwo gefunden werden, und ist deshalb auch außerhalb der Lehre einsetzbar. Die einzigartige Kombination von ereignis- und prozessorientierter Modellierung, die Möglichkeit, Modelle als Teile komplexerer Modelle wiederzuverwenden, und der reine Java Quellcode machen es zu einem sehr gut nutzbaren Werkzeug der diskreten Modellierung.

Kapitel 3

Simulationsmodell

In diesem Kapitel wird das Beispiel Szenario des Hafenterminals detailliert beschrieben. Danach wird auf die beiden Simulationsmodelle genau eingegangen: Zuerst auf das Simulationsmodell, das aus Petrinetzen besteht und mit Hilfe von Renew erstellt wurde. Danach wird das mit Desmoj erstellte Simulationsmodell vorgestellt.

3.1 Szenario

Das Szenario befaßt sich mit Logistikprozessen eines Containerterminals. Genauer gesagt um den Ausschnitt der Lkw Abfertigung auf dem Containerterminal. Lkws besitzen Anforderungen an das Terminal, nämlich Container bringen oder abliefern zu können. Container werden innerhalb des Terminals von Vancarriern transportiert.

Die Lkw Abfertigung kann man sich etwa so vorstellen: Lkws fahren zum Terminal, weil sie dort einen bestimmten Auftrag ausführen sollen. Der Auftrag kann entweder heißen, „bringe einen Container auf das Terminal", „hole einen Container vom Terminal ab" oder „bringe einen hin und hole auch gleich wieder einen ab". Wenn ein Lkw mit einem Auftrag am Terminal ankommt, muß er sich anmelden, damit sein Auftrag auf dem Terminal ausgeführt werden kann. Das macht er, indem er seinen Auftrag an die Terminalkontrolle weitergibt. Die Aufträge werden von der Terminalkontrolle dann an sogenannte Vancarrier weitergegeben. Die Vancarrier sind dafür da, Container von den Lkws zu laden und in einen Yard, in dem die Container aufbewahrt werden, zu fahren oder einen Container aus einem Yard zu holen und einen Lkw damit zu beladen. Damit die Ladevorgänge auf dem Terminal stattfinden können, dürfen bestimmte Kapazitäten nicht überschritten werden, d.h. ein Terminal kann aus Platzgründen nur eine bestimmte Anzahl an Lkws auf einmal aufnehmen. Die Lkws, die auf dem Terminal keinen Platz mehr bekommen haben, um von den Vancarriern bedient zu werden, müssen vor dem Terminal in einer Warteschlange warten, bis wieder ein Platz frei geworden ist.

3.1.1 Beschreibung

Der Aufbau des Terminals ist in Bild Abbildung 3.1 dargestellt.

Der zu simulierende und analysierende Ablauf läßt sich wie folgt beschreiben.

Ein Lkw fährt mit einem bestimmten Auftrag zum Terminal. Es gibt drei verschiedene Auftragsformen.

1. Er kann einen Container zum Terminal bringen und ihn leer wieder verlassen,

2. er kann leer zum Terminal kommen und mit einem Container beladen werden, den er dann mitnimmt, oder

3. er kann einen Container bringen und einen anderen wieder mitnehmen.

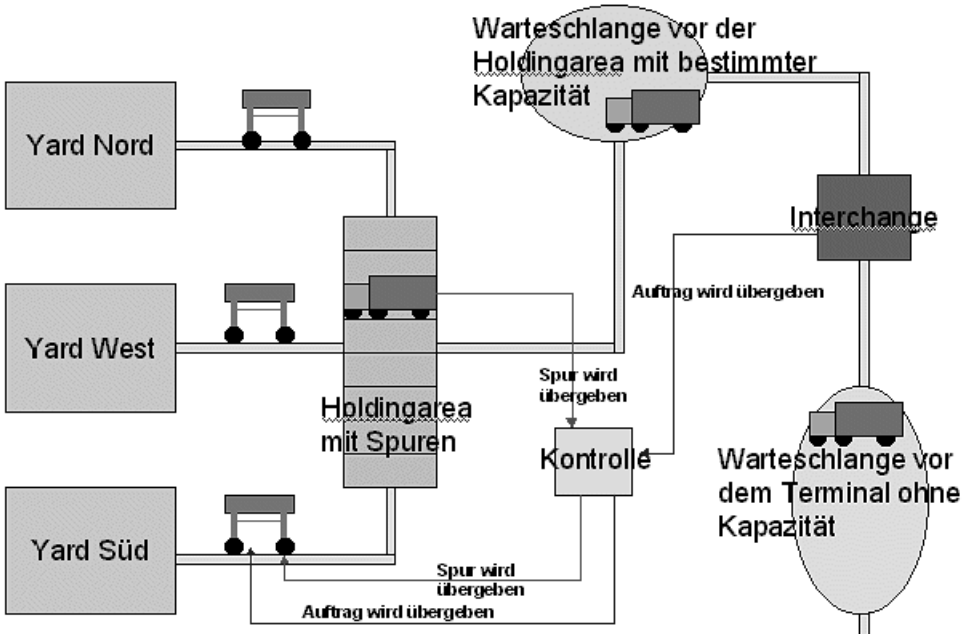

Abbildung 3.1: Das Bild eines Terminals

Das Ent- und Beladen der Lkws geschieht mit Hilfe geeigneter Fahrzeuge, sogenannter Vancarrier (kurz VC). Die Vancarrier können auf dem Terminal nur zwischen dem Yard und den Lkws in der Holdingarea hin und her fahren. Der Yard ist der Ort, an dem die Container gelagert werden. In diesem Beispiel gibt es drei verschiedene Yards, in denen Container gelagert werden können. Jedem Yard sind bestimmte VCs zugeordnet, d.h. ein VC kann nicht zwischen verschiedenen Yards hin und her fahren. Um die Yards unterscheiden zu können, werden sie Nord-, West- und Süd-Yard genannt.

Nun kann ein VC einen ankommenden Lkw nicht einfach so bedienen, da das Containerterminal beschränkte Kapazitäten hat, d.h. es können nicht beliebig viele Lkws auf das Terminal herauf fahren. Das Terminal setzt sich aus verschiedenen Stationen zusammen, die der Lkw durchlaufen muß, bis er von einem VC bedient wird.

Wenn der Lkw am Terminal ankommt, kann er möglicherweise nicht gleich hinein fahren, da die Kapazitäten ausgeschöpft sein könnten. Er muß also warten. Das macht er in einer Warteschlange vor dem Terminal, die im Allgemeinen eine unendliche Kapazität hat.

Bei freigewordender Kapazität, fährt der Lkw ins *Interchange* (IC). Das Interchange ist räumlich als Eingang anzusehen. Es hat dazu noch die Aufgabe, die Abwicklung der Lkw-Aufträge zu koordinieren, d.h. es ist zusätzlich noch eine Kontrolle. Im Interchange übergibt der Lkw seinen Auftrag. Das Interchange gibt den Auftrag an einen VC des entsprechenden Yards weiter. Hat der Lkw beispielsweise einen Entladeauftrag für den Nord Yard und einen Beladeauftrag für den Süd Yard, muß das Interchange den Auftrag an zwei VCs weitergeben. An einen zum Entladen aus dem Nord Yard und an einen zum Beladen aus dem Süd Yard.

Hat der Lkw seinen Auftrag übergeben, muß er zur *Holdingarea* (HO) fahren. Die Holdingarea ist der Ort, an dem die Lkws von den VCs bedient werden, also die Bedienstation. Die Holdingarea besteht aus einer bestimmten Anzahl Spuren, in die die Lkws hinein fahren, um bedient zu werden. Sollte gerade keine Spur frei sein, muß der Lkw in einer Warteschlange vor der Holdingarea warten, bis eine frei geworden ist. Diese Warteschlange hat, da sie sich auf dem Terminal befindet, nur eine begrenzte Kapazität. Sollte diese Kapazität ausgeschöpft sein, werden keine Lkws mehr auf das Terminal gelassen, sie müssen dann in der schon erwähnten Warteschlange vor dem Terminal warten.

Steht ein Lkw nun in einer Spur der Holdingarea, benachrichtigt er die Kontrolle über seine Spur, in der er steht. Die Kontrolle übergibt diese Information dann dem oder den entsprechenden VCs. Der Weg über die Kontrolle ist nötig, da der Lkw nicht weiß, welcher oder welche VCs ihn bedienen.

In der Holdingarea kann es passieren, daß der Lkw eintrifft und noch auf die VCs warten muß. Das ist dann wahrscheinlich, wenn nur sehr wenige VCs zur Verfügung stehen. Die VCs bekommen dann ununterbrochen Aufträge zugeteilt, die sie allmählich abarbeiten. Dadurch kann es natürlich in der Holdingarea zu Verzögerungen kommen.

Es kann aber auch sein, daß der VC schon vor dem Lkw in der Holdingarea angekommen ist. Dann muß der VC auf den Lkw warten. Dieser Fall tritt dann ein, wenn relativ viele VCs zur Verfügung stehen. Sollte der Lkw einen Ent- und Beladeauftrag haben, kann es auch sein, daß der Lkw und der VC, der beladen soll, auf den VC, der entladen soll, warten müssen.

Ist ein Lkw fertig bearbeitet, verläßt er das Terminal, und ein wartender Lkw kann in die frei gewordene Spur hinein fahren. Ein VC, der seinen Auftrag erfüllt hat, meldet sich bei der Kontrolle, damit er für einen nächsten Auftrag herangezogen werden kann.

3.1.2 Fragestellung

Es stellen sich bei diesem Szenario die Fragen, wieviele VCs man braucht und welche Kapazität für die Holdingarea ausreichend ist. Ziel ist es, eine möglichst optimale Auslastung der VCs zu erreichen und die schnelle Abfertigung der ankommenden lkws zu ermöglichen; d.h. die Wartezeit W eines Lkws zu minimieren. Die Parameter, die man verändern kann, sind

1. die Anzahl der VCs und

2. die Anzahl der Spuren.

Man braucht eine ausreichend große Anzahl an VCs, um alle ankommenden Lkws bedienen zu können und um ein langes Warten der Lkws zu vermeiden. Man muß aber gleichzeitig darauf achten, daß man nicht soviele VCs hat, daß es bei ihnen zu oft zu langen Wartezeiten im Yard oder in der HO kommt. Hat man zu wenig VCs, kommen diese mit dem Bearbeiten der Lkw-Aufträge möglicherweise nicht nach, und die Lkws haben zu lange Wartezeiten. Dadurch kann es zu Rückstaus vor dem Terminal kommen.

Eine zweite Größe, die man optimal einstellen möchte, ist die Anzahl der Spuren. Hat man zuwenig Spuren, hilft auch eine große Anzahl VCs nicht, die ankomenden Aufträge schnell abzuarbeiten, da die VCs mit angenommenen Aufträgen, die sie erledigen möchten, herumstehen. Das tun sie, weil die Lkws nicht in die Holdingarea fahren können. Hat man zuviele Spuren, die nicht schnell bedient werden, muß man sich eventuell fragen, ob man den Platz einiger Spuren vielleicht besser anders nutzen könnte.

Ziel ist, herauszufinden, bei welchem Lkw-Aufkommen welche Anzahl von VCs und Spuren optimal ist, damit das Terminal effizient arbeiten kann.

Da diese Fragestellung rechnerisch bzw. analytisch nur schwer zu lösen ist, baut man einen Simulator, der bei der Lösungsfindung hilft. Es gibt mehrere Möglichkeiten, einen Simulator zu erstellen. In dieser Arbeit werden zwei Ansätze zur Simulation vorgestellt.

Die erste Möglichkeit ist, einen Simulator mit Referenznetzen zu bauen. Die zweite Möglichkeit ist, das Simulationsframework Desmoj einzusetzen. Beide Methoden sollen auf unterschiedlichem Wege auf das gleiche Ergebnis kommen. Bei den Petrinetzen steht die Visualisierung im Vordergrund. Das ist wichtig, um zu sehen, ob Abläufe wirklich so ablaufen, wie sie ablaufen sollen. Man kann die Simulation, während sie abläuft, beobachten, d.h. das erstellte Modell ist gleichzeitig das Programm in dem die Simulation abläuft. Mit Hilfe von Renew kann man das Szenarion leicht modellieren, da man die Prozesse intuitiv entwickeln kann. Fehler können schnell erkannt und Veränderungen leicht vorgenommen werden. Renew unterstützt außerdem die Prozeßorientierung, die für dieses Szenario gefordert ist, sehr gut, da jeder Prozeß durch ein eigenes Netz dargestellt werden kann. Zusätzlich ist es in Renew möglich, andere Javaklassen einzubinden, da Renew auch in Java geschrieben ist. Renew hat allerdings keine statistische Auswertung, so daß man Aussagen über ein Szenario nur schwer treffen kann. Es besteht aber grundsätzlich die Möglichkeit, dieses nachzurüsten. Weitere Gründe, weshalb man Petrinetze noch benutzt, sind ihre Ausdrucksfähigkeit und ihre präzise Semantik.

Das Simulationsframework Desmoj ist für statistische Auswertungen gut geeignet, da es genaue statistische Informationen darüber liefert, wie viele Lkws sich zum Beispiel in einer Warteschlange gesammelt haben, oder wie viele VCs wie lange auftragslos herumstanden. Außerdem gibt es einen sehr großen Erfahrungshorizont hinter der Simulation mit Desmoj, da schon sehr viele Simulatoren mit Desmoj programmiert wurden. Desmoj unterstützt die objektorientierte Programmierung, da Desmoj ein Framework in Java ist. Wenn man

gut Java programmieren kann, kann man schnell Simulatoren in Desmoj bauen. Kann man es nicht, ist Desmoj etwas kompliziert, da man Veränderungen am Simulator, wenn man den Quellcode nicht lesen kann, nur schwer vornehmen kann. Im Folgenden werden beide Methoden genauer betrachtet.

3.1.3 Modell des Szenarios

Man kann sich den Containerterminal aus vier Klassen bestehend vorstellen. Aus der Terminal-, der Kontroll-, der Lkw- und der VC-Klasse. Diese vier Klassen bilden die wesentlichen Bestandteile des beschriebenen Containerterminals.

In der Petrinetzsimulation werden diese vier Klassen als einzelne Petrinetze dargestellt, die miteinander interagieren (siehe Kapitel 3.2). In Desmoj bilden sie Klassen, die das Package Terminal bilden (siehe Kapitel 3.3).

Die Klasse Terminal sollte alle strukturellen Bestandteile des Terminals beinhalten. Zum Beispiel sämtliche Warteschlangen, die auf dem Containerterminal zu finden sind, die Holdingarea, die Yards und das Interchange, das räumlich als Eingang auf das Terminal zu betrachten ist.

Die Kontrolle sollte alle Methoden zur Koordination des Ablaufs auf dem Terminal beinhalten wie die Entgegennahme der Lkw-Aufträge und die Verwaltung der freien VCs.

Die Lkw- und die VC-klasse sollten jeweils Methoden enthalten, die ihre Bewegungsmöglichkeiten darstellen wie zum Beispiel Fahrzeiten von einem Ort zum anderen oder Servicezeiten. Die Abbildungen 3.2, 3.3 und 3.4 zeigen den Ablauf der einzelnen Klassen als UML-Diagramm.

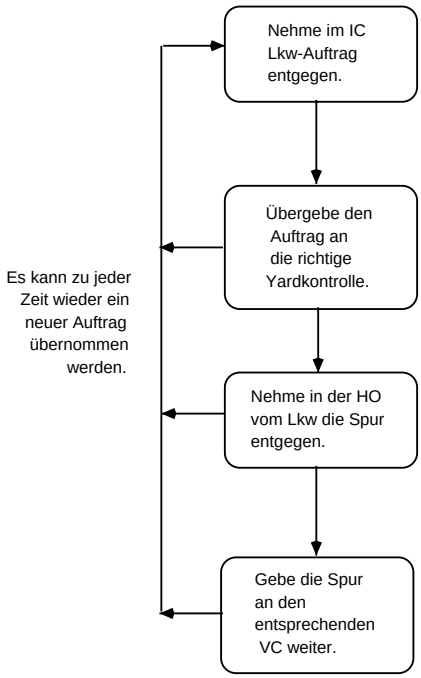

Abbildung 3.2: Der Ablauf der Kontrolle

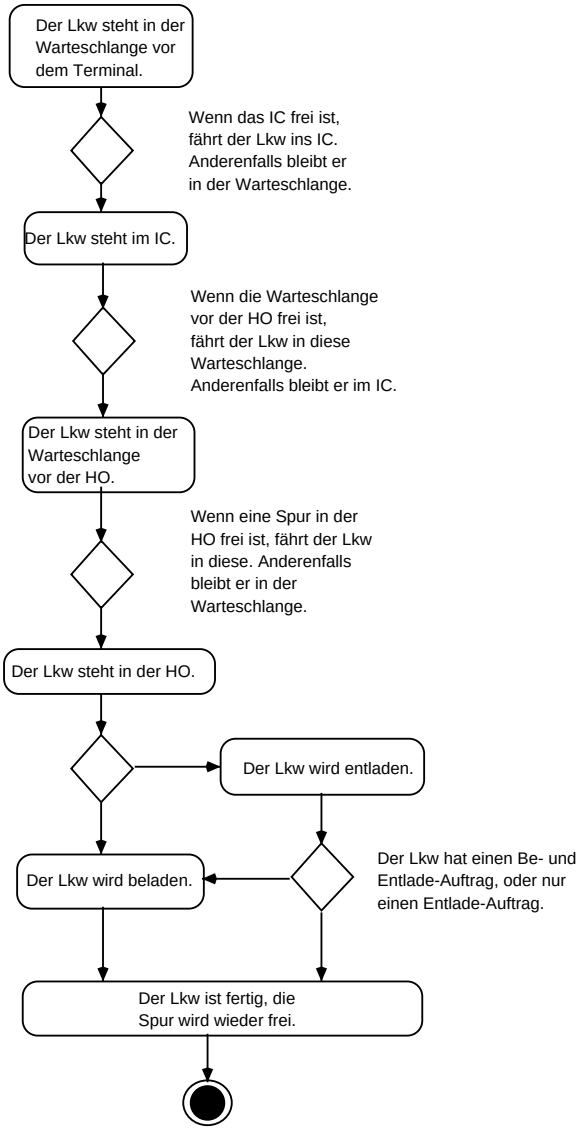

Abbildung 3.3: Der Ablauf des lkw

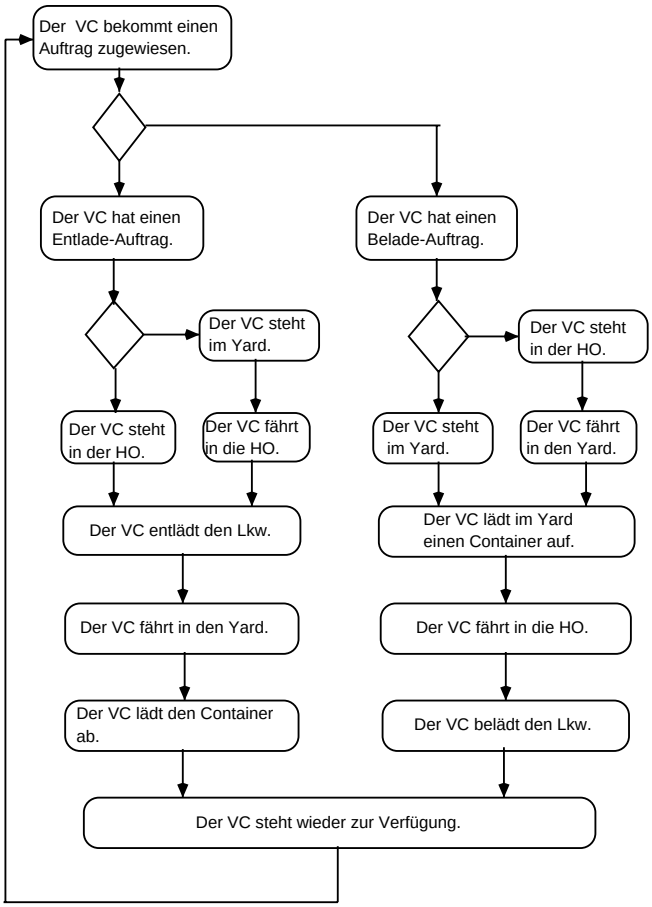

Abbildung 3.4: Der Ablauf des VC

3.2 Petrinetzmodellierung

Um das Containerterminal-Modell in Petrinetzen zu realisieren, sind vier Netze nötig. Es handelt sich hier um Referenznetze (siehe Kapitel 2.2.5).

- Ein Netz für das Terminal, auf dem die einzelnen Stationen abgebildet sind, die entweder ein Lkw oder ein VC durchlaufen.
- Ein Netz für die Kontrolle, in dem die Koordination abgebildet ist.
- Ein Netz für einen Lkw, in dem die möglichen Aufträge abgebildet sind und die Stationen, die er durchläuft.
- Und ein Netz für einen VC, in dem seine Fahrtmöglichkeiten abgebildet sind.

3.2.1 Das Terminal

Man kann sich das Terminalnetz als eine Darstellung vorstellen, auf der alle Straßen, Warteschlangen, Yards und sonstige Bauwerke, die es auf dem Containerterminal gibt, abgebildet sind (siehe Abbildung 3.5).

Ein Lkw, der zum Terminal fährt, stellt sich als erstes in die Warteschlange vor dem Terminal. Für das Petrinetzmodell heißt das, der Lkw befindet sich zuerst in der Stelle Warteschlange vor dem Terminal. Die Orte auf dem Terminal werden im Petrinetz als Stellen dargestellt. Die Möglichkeiten, irgendwohin zu fahren oder eine andere Aktivität auszuführen, werden als Transitionen dargestellt. Die Kanten, bzw. Pfeile kann man sich als Straßen vorstellen.

Ist also die Transition Lkw fährt ins IC aktiviert, kann sie schalten, und der Lkw befindet sich dann in der Stelle Interchange Bereich. Dann ist die Transition Lkw fährt in die Warteschlange aktiviert und kann schalten. Der Lkw kommt dann in die Stelle Warteschlange vor der HO. Schaltet die Transition Lkw fährt in die HO, ist der Lkw in der Stelle Lkw in HO, bereit für VC. Diese Stelle und die Stelle VC in HO, bereit für Lkw, kann man als Holdingarea zusammenfassen. Die Transitionen beladen und entladen sollen hier vorerst nicht weiter berücksichtigt werden.

Betrachtet man das Netz von unten, sieht man die Stellen Yard Nord, Yard Süd und Yard West. Dort werden die Container gelagert. VCs können z.B. von der Stelle Yard Nord aus, wenn die Transition VC fährt in die HO schaltet, in die Stelle VC in HO, bereit für Lkw verschoben werden.

Ein fertig bearbeiteter Lkw kann von der Stelle Lkw in HO, bereit für VC zu der Stelle bearbeitete Lkws verschoben werden, wenn die Transition Lkw raus schaltet.

3.2.2 Das Kontroll-Netz

Zentraler Gegenstand der Kontrolle sind eine Stelle Request Warteschlange und eine Stelle verfügbare VC. Daran erkennt man, daß das Kontroll-Netz die ankommenden Aufträge und die VCs verwaltet. Abbildung 3.6 zeigt das Netz der Kontrolle

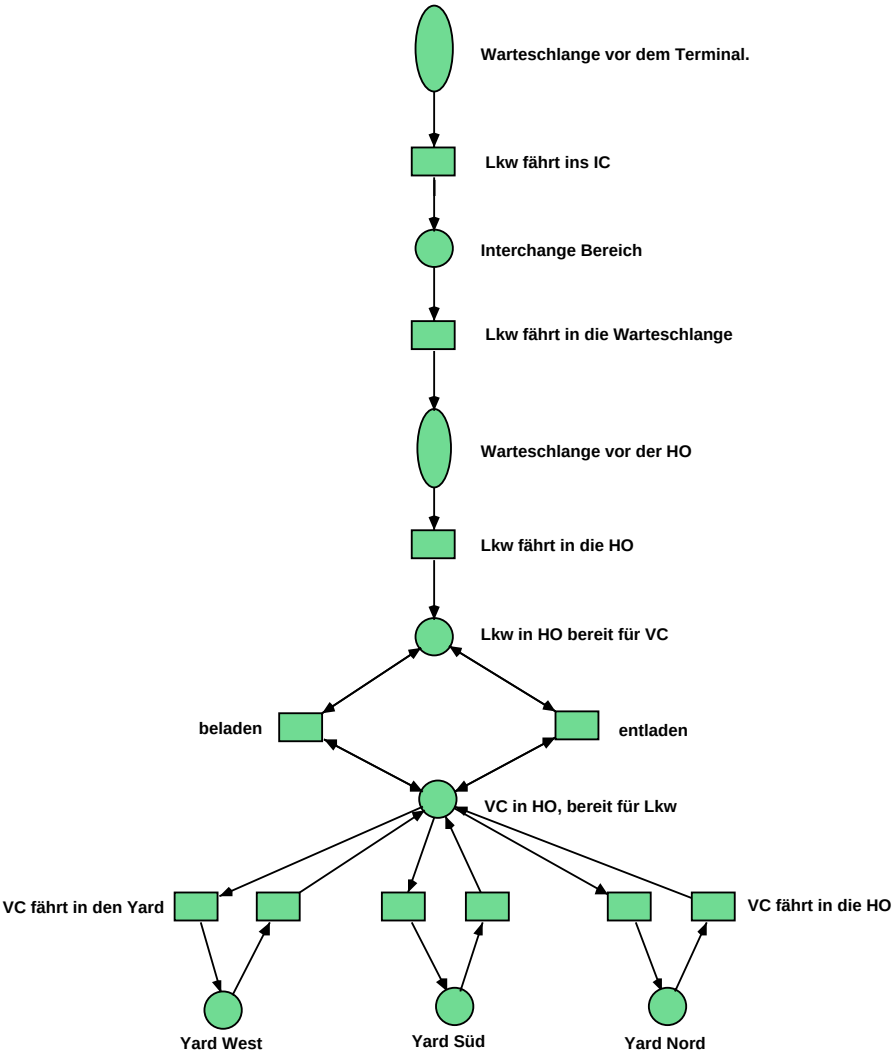

Abbildung 3.5: Das Terminalnetz in vereinfachter Form

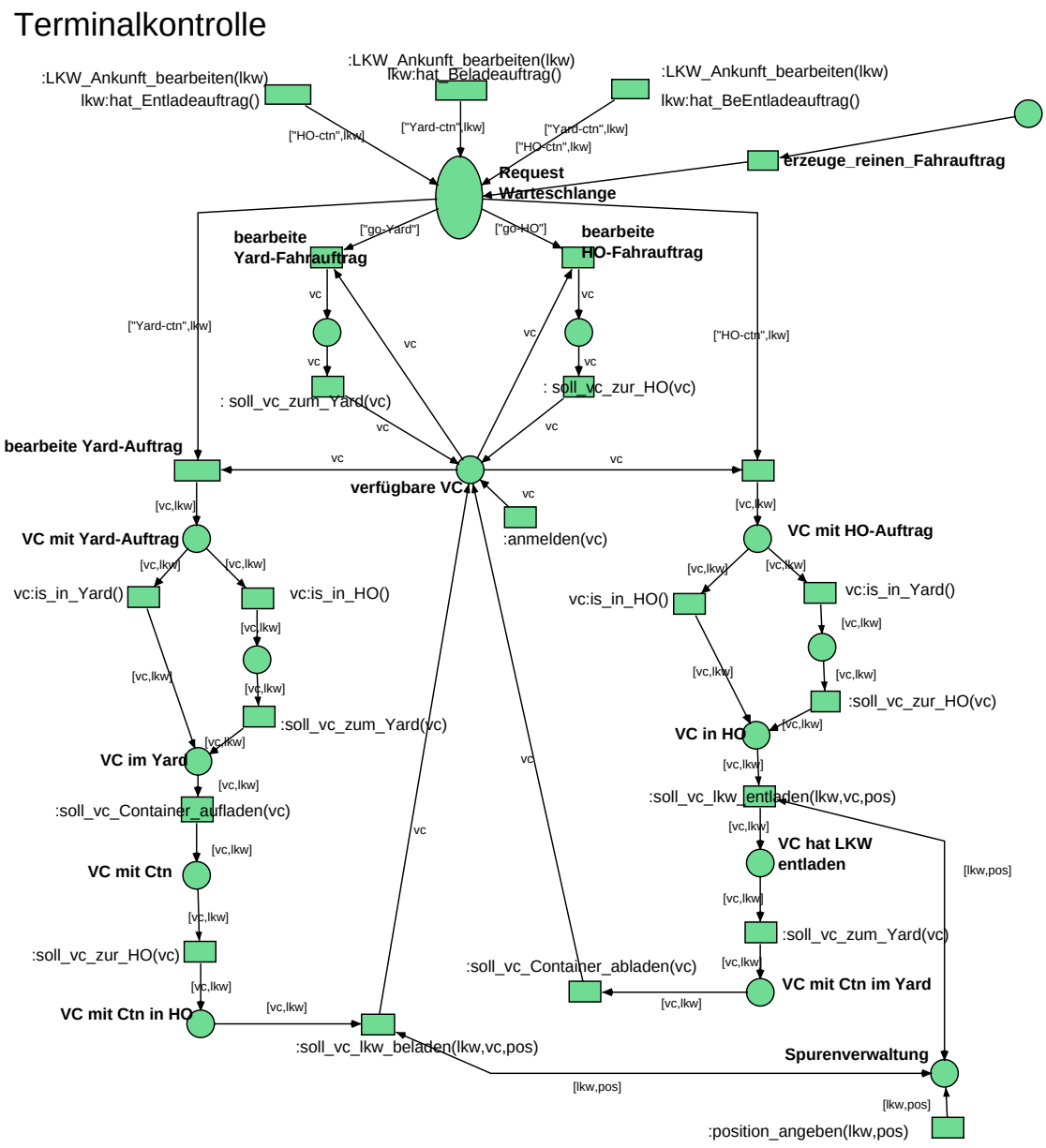

Abbildung 3.6: Das Netz der Kontrolle

Die Aufträge kommen in die Request Warteschlange, wenn die Transitionen :Lkw-Ankunft-bearbeiten(lkw) gefeuert hat. Eine Transitionen :Lkw-Ankunft-bearbeiten(lkw) kann dann feuern, wenn die Transition mit dem Lkw-Netz synchronisiert wird, d.h. sobald ein Lkw vom Terminal einen Auftrag bekommen hat (lkw:hat-Entladeauftrag(), lkw:hat-Beladeauftrag() oder lkw:hat-BeEntlade-auftrag()).

In der Stelle verfügbare VC befinden sich alle VCs, die es im System gibt, d.h. alle VCs, die im Terminal-Netz erzeugt wurden. Beim Erzeugen eines VCs hat die Transition :anmelden(vc) gefeuert und damit jeden VC auf die Stelle verfügbare VCs gelegt.

Jenachdem um was für eine Art Auftrag es sich handelt, kann die Transition bearbeite Yard-Auftrag oder die Transition bearbeite HO-auftrag feuern. Soll der Lkw entladen werden, handelt es sich um einen HO-Auftrag, da ein leerer VC dafür in der HO sein muß. Soll der Lkw beladen werden, handelt es sich um einen Yard-Auftrag, da der VC einen Container aus dem Yard zum Lkw holen muß.

In dieser Beschreibung der Kontrolle wird davon ausgegangen, daß der Lkw einen Beladeauftrag hat. Für den VC heißt das, er hat einen Yard-Auftrag. Es muß die Transition bearbeiteYard-Auftrag feuern. Sie ist dann aktiviert und kann feuern, wenn der Auftrag aus der Request Warteschlange und ein VC aus der Stelle verfügbare VC entnommen werden.

Hat die Transition gefeuert, befindet sich die Kontrolle in der Bearbeitung an der Stelle VC mit Yard-Auftrag. An dieser Stelle muß geprüft werden, ob sich der VC im Yard oder in der HO befindet. Dazu muß das Netz mit dem VC-Netz synchronisiert werden (vc:is-in-Yard() oder vc:is-in-HO()). Es kann dann entweder die 'Transition VC ist im Yard oder die Transition VC ist in der HO feuern. Jenachdem wo sich der VC befindet. Sollte sich der VC in der HO befinden, muß jetzt die Transition :soll-zum-Yard(vc) feuern, damit der VC in den Yard fährt.

An der Stelle VC im Yard kann dann die Transition :soll-vc-container-aufladen(vc) feuern. Sämtliche darauf folgenden Transitionen und Stellen korrespondieren zu denen des VC-Netzes (siehe VC Netz). Der VC steht jetzt mit dem Container beladen im Yard und möchte zur HO fahren. Feuert die Transition :soll-zur-HO(vc), ist die Kontrolle in ihrer Bearbeitung an der Stelle VC mit Container in HO angekommen. Als nächstes muß der Lkw beladen werden. Die Transition :soll-vc-Lkw-beladen(lkw,vc,pos) kann aber nur dann feuern, wenn ihr zusätzlich zu dem Lkw und dem VC noch die richtige Spur aus der Stelle Spurenverwaltung als Attribut übergeben wird, die dann aber sofort wieder zurückgelegt wird. Hat diese Transition gefeuert, trägt sich der VC wieder in die Stelle verfügbare VC ein.

Sollte der VC einen HO-Auftrag haben, arbeitet die Kontrolle auf die gleiche Weise, nur das der VC evtl. in die HO fahren muß.

In diesem Netz sind auch die reinen Fahraufträge abgebildet. Es handelt sich dabei um Aufträge, die nicht dem Ent-oder Beladen dienen, sondern nur dem Standortwechsel. Es gibt die Transition erzeuge-reinen-Fahrauftrag, durch die diese Aufträge in die Request Warteschlange eingetragen werden. Wird ein solcher Auftrag wieder aus der Warteschlange herausgeholt, muß auch wieder ein VC aus der Stelle verfügbare VCs genommen werden, und es können entweder

die Transition bearbeite Yard-Fahrauftrag oder bearbeite HO-Fahrauftrag feuern. Jenachdem um was für einen Auftrag es sich handelt, kann dann entweder die Transition :soll-vc-zum-Yard(vc) oder die Transition :soll-vc-zur-HO(vc) feuern.

Man kann an den Transitionen erkennen, daß dieses Netz eine Koordinationsfunktion hat, da fast alle Transitionen mit dem Wort soll anfangen. Vergleicht man dieses Netz mit den anderen drei Netzen,sieht man, daß, wenn es zu Synchronisationen kommt, immer eine soll-Transition des Kontroll-Netzes mit synchronisiert wird. Es wird vor dem Feuern einer Transition also gefragt, ob irgend etwas Bestimmtes tatsächlich passieren soll. Das dient zusätzlich der Überprüfung, ob auch alle Transitionen der beteiligten Netze aktiviert sind, und eine Transition auch wirklich feuern kann.

3.2.3 Das Lkw-Netz

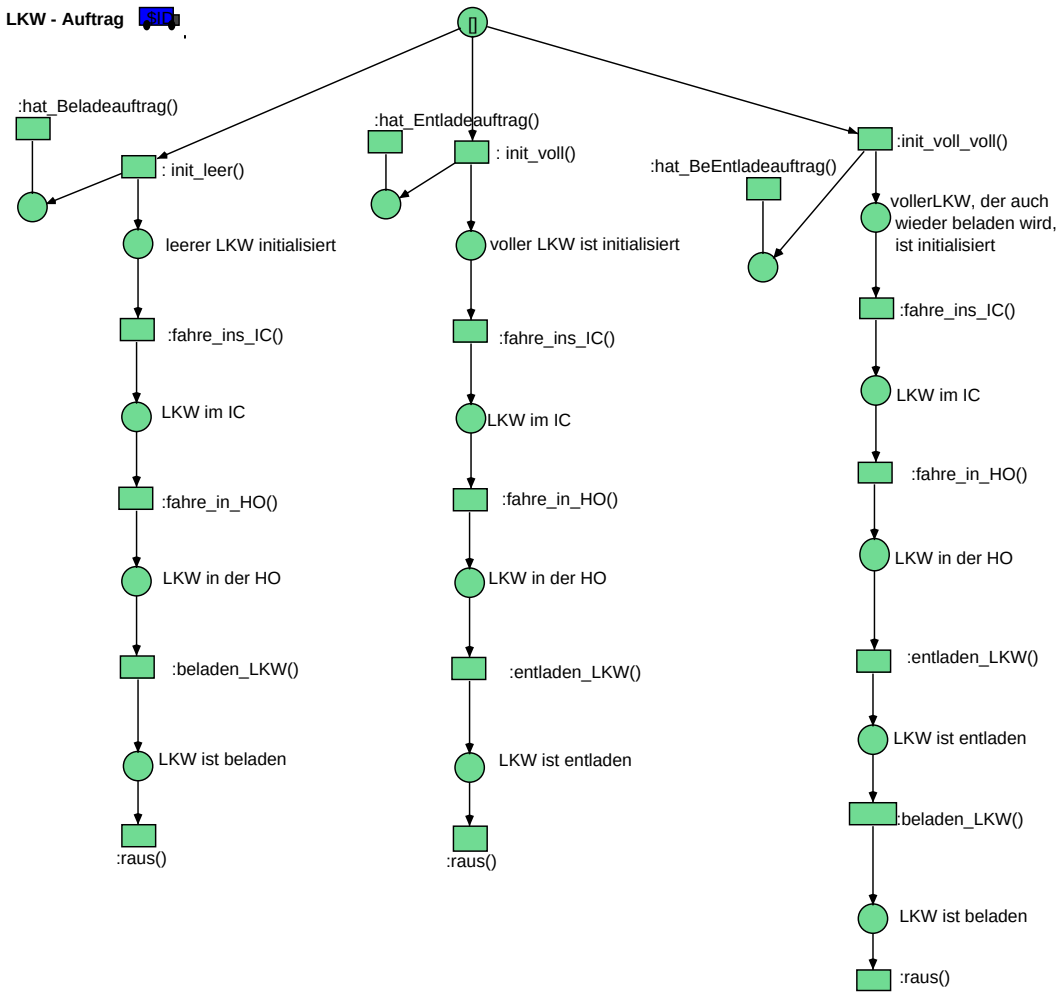

Abbildung 3.7: Das Netz eines Lkw

Das Petrinetz des Lkws zeigt, daß ein Lkw genau einen von drei möglichen Aufträgen haben kann, wie in Abbildung 3.7 zu sehen ist: entweder einen Bela-

deauftrag, einen Entladeauftrag oder einen Ent- und Beladeauftrag. Wird ein neuer Lkw initialisiert, bekommt er per Zufall einen der drei Aufträge zugeteilt. Im Netz passiert das in der Stelle Auftrag, in der ein Token liegt, welches den Auftrag darstellt. Eine der drei Transitionen :init-leer(), :init-voll() oder :init-voll-voll() kann dann feuern und bringt den Lkw dadurch in einen der drei Zustände leerer Lkw initialisiert, voller Lkw initialisiert oder voller Lkw, der auch wieder beladen wird, ist initialisiert.

Das Netz wird anhand des Auftragstyps entladen und wieder beladen beschrieben, da dieser Typ die beiden anderen enthält.

Betrachtet man die Stelle voller Lkw, der auch wieder beladen wird, ist initialisiert, kann die Transition :fahre-ins-IC() feuern. Der Lkw befindet sich jetzt auf der Stelle Lkw im IC. Danach feuert die Transition :fahre-in-die-HO(). Danach ist der Lkw in der HO.

Man kann in diesem Netz genau sehen, welche Stationen der Lkw zurücklegen muß. Es sind zu einem großen Teil die gleichen wie im oberen Teil des Terminal-Netzes. Die Transitionen sind bis zu dieser Stelle mit den entsprechenden des Terminal-Netzes synchronisiert.

Wenn der Lkw in der HO steht kann er entladen werden, d.h. die Transition :entladen-Lkw() ist aktiviert und kann feuern.

Hätte der Lkw nur einen Entladeauftrag, könnte er das System mit der Transition :raus() jetzt verlassen. Da er aber in diesem Beispiel noch beladen werden muß, feuert die Transition :belade-Lkw(), wenn der Lkw entladen ist. Die Transitionen :entladen-Lkw() und :beladen-Lkw() werden mit dem VC und der Kontrolle synchronisiert.

In der Stelle Lkw ist beladen kann der Lkw das System mit der Transition :raus() verlassen.

3.2.4 Das Vancarrier-Netz

Das Petrinetz eines VCs bildet genau die Bewegungsmöglchkeiten ab, die ein VC hat (vgl. Abbildung 3.8). Sämtliche Transitionen in diesem Netz werden mit dem Netz der Kontrolle synchronisiert. Die Transitionen beladen-Lkw() und :entladen-Lkw() werden zusätzlich noch mit dem Netz des Lkws synchronisiert.

Ein VC kann ohne Auftrag in der Holdingarea oder im Yard stehen (VC steht in der HO ohne Auftrag und VC steht im Yard ohne Auftrag).

Betrachtet man die Stelle VC steht in HO ohne Auftrag kann man sehen, daß er jetzt entweder die Möglichkeit hat, einen Lkw zu entladen oder zum Yard zu fahren (:entladen-Lkw() und :fahre-zum-Yard()).

Geht man davon aus, daß er einen Lkw entladen hat, kommt der VC auf die Stelle VC steht mit Container beladen in der HO und will in den Yard fahren. Von dieser Stelle aus kann nur eine Transition feuern (:fahre-zum-Yard()), die den VC auf die Stelle VC steht mit Container beladen im Yard, will Container abladen bringt. Auch von dieser Stelle kann nur eine Transition feuern (:lade-Container-ab()), die den VC auf die Stelle VC steht im Yard ohne Auftrag bringt.

Den eben beschriebenen Durchlauf könnte man Entladezyklus nennen, da der VC in der HO einen Lkw entlädt, den Container in den Yard bringt, ihn dort ablädt und jetzt im Yard steht.

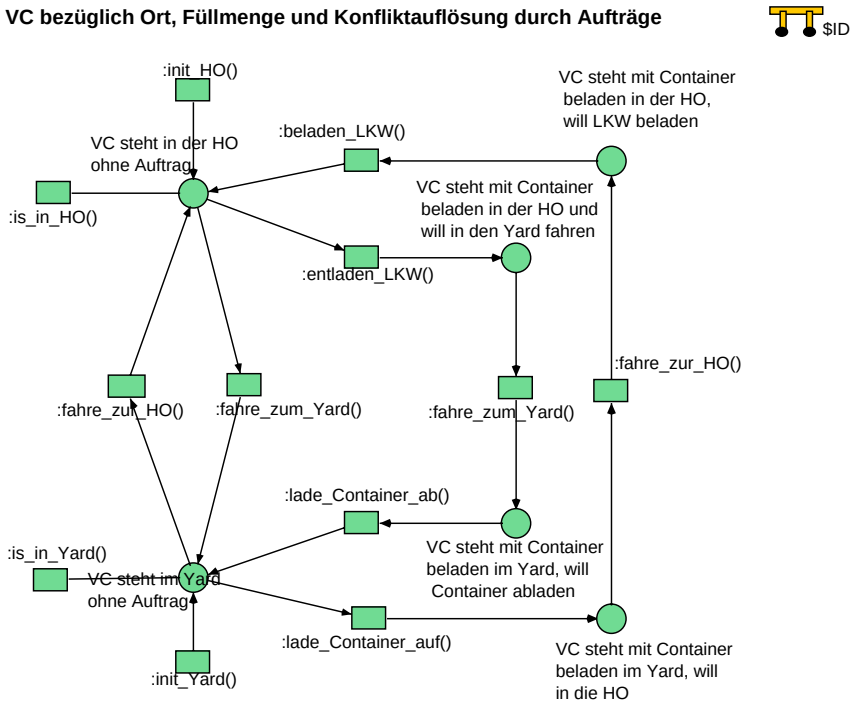

Abbildung 3.8: Das Netz eines Vancarrier

Genauso muß es auch einen Beladezyklus geben, bei dem der VC aus dem Yard mit einem Container beladen in die HO fährt, dort einen Lkw belädt und dann in der HO steht: Geht man von der Stelle VC steht im Yard ohne Auftrag aus, kann man den Beladezyklus beschreiben, indem man die Transition :lade-Container-auf() feuert. Der VC kommt dann auf die Stelle VC steht mit Container beladen im Yard, will in die HO. Von dieser Stelle aus läßt man die Transition :fahre-zur-HO() feuern, damit der VC auf die Stelle VC steht mit Container beladen in der HO, will beladen kommt. Dann kann die Transition :beladen-Lkw() feuern, damit der VC auf die Stelle VC steht in der HO ohne Auftrag kommt, und der Beladezyklus abgeschlossen ist.

Neben dem Entlade- und Beladezyklus hat der VC noch die Möglichkeit, Leerfahrten zu machen, d.h. er kann ohne Container von der HO in den Yard oder vom Yard in die HO fahren. Dazu muß von der Stelle VC steht in der HO ohne Auftrag die Transition :fahre-zum-Yard() feuern und von der Stelle VC steht im Yard ohne Auftrag die Transition :fahre-zur-HO().

Die beiden Transitionen init-Yard() und init-HO() dienen dazu, VCs entweder in der HO oder im Yard zu initialisieren. Die beiden Transitionen is-in-HO() und :is-in-Yard() dienen dazu, abfragen zu können, wo ein VC sich gerade aufhält.

3.2.5 Das Terminal-Netz

Im Terminal-Netz werden die Kontrolle, die Lkws und die VCs erzeugt und miteinander synchronisiert, wie in Abbildung 3.9 zu sehen ist (aus Übersichtlichkeitsgründen ist nur ein Yard in der Abbildung zu sehen).

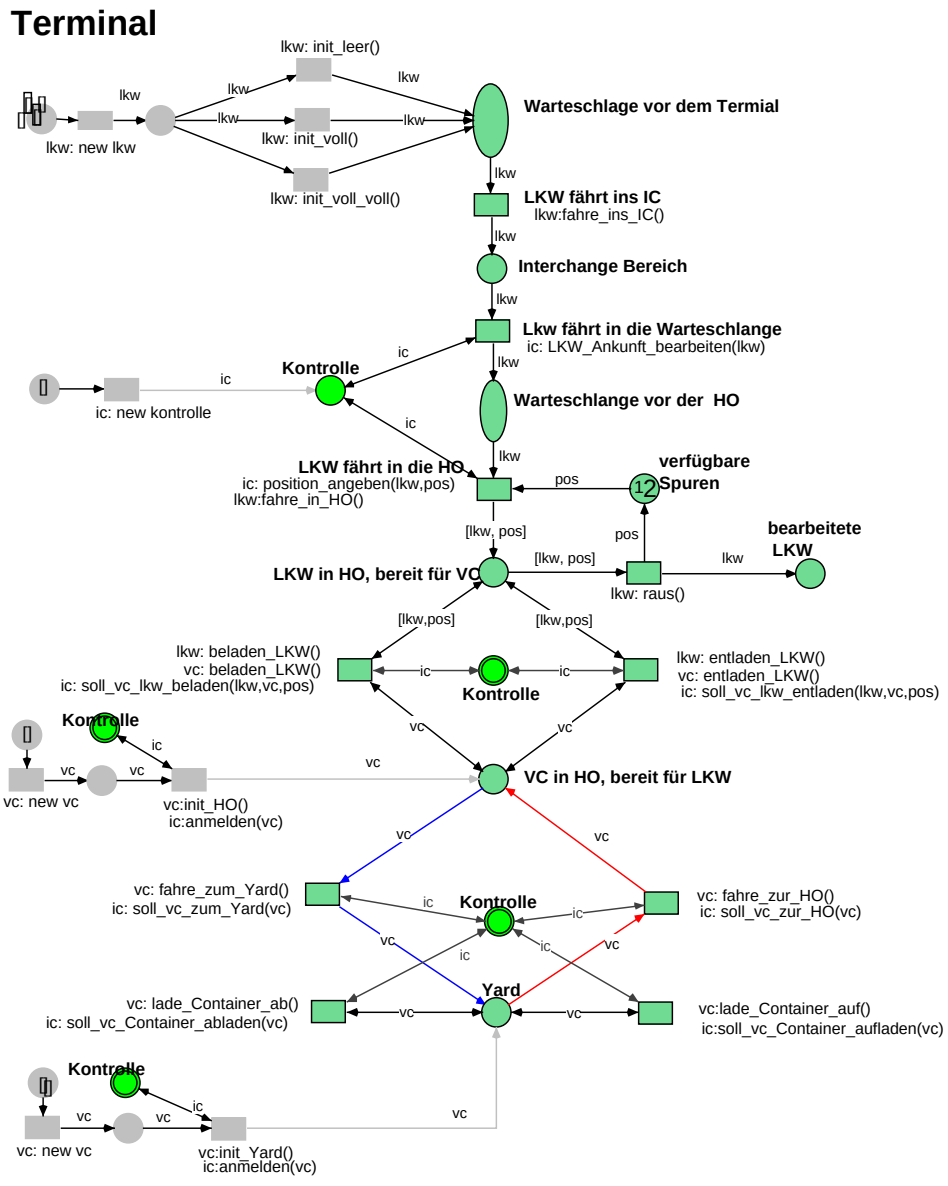

Abbildung 3.9: Das Terminal-Netz

Die Simulation startet, indem mit lkw: new Lkw neue Lkw Referenzen erzeugt werden. Jedem neuen Lkw Objekt wird per Zufall ein Auftragstyp zugewiesen. Entweder :init-leer(), init-voll() oder init-voll-voll() (siehe dazu auch das Lkw-Netz). Ist einem Lkw per Zufall ein Auftragstyp zugeordnet worden, befindet er sich auf der Stelle Warteschlange vor dem Terminal. Um ins Interchange (IC) zu kommen, muß die Transition :Lkw fährt ins IC feuern. Initial werden die Kontrolle und alle VCs erzeugt. Das Kontroll-Objekt wird nur beim ersten Durchlauf generiert, da es die ganze Simulationszeit über erhalten bleibt.

Im Interchange kann die Transition Lkw fährt in die Warteschlange feuern. Ist der Lkw auf der Stelle Warteschlange vor der HO kann die Transition Lkw fährt in die HO feuern. Die drei letzt genanten Transitionen, also Lkw fährt ins IC, Lkw fährt in die Warteschlange und Lkw fährt in die HO müssen mit dem Lkw und/oder dem Kontroll-Netz synchronisiert werden, damit die Simulation, an der mehr Netze als nur das Terminalnetz beteiligt sind, fehlerfrei ablaufen kann.

An der Transition Lkw fährt ins IC muß zusätzlich noch lkw:fahre-ins-IC() und ic:Lkw-Ankunft-bearbeiten(lkw) stehen, d.h. in dem Lkw-Netz muß die Transition fahre-ins-IC() schalten und in der Kontrolle die Transition Lkw-ankunft-bearbeiten(). Sollte im Lkw-Netz eine Transition nicht schalten können, dann darf sie es auch nicht in dem Terminal-Netz. Der Ablauf der Simulation wird unterbrochen und erst dann wieder fortgesetzt, wenn die Transitionen der beteiligten Netze aktiviert sind.

An der Transition Lkw fährt in die HO wird mit dem Lkw-Netz (lkw:fahre-in-HO()) und mit der Kontrolle (ic:position-angeben(lkw,pos)) synchronisiert. Zusätlich wird noch aus der Stelle verfügbare Spuren ein Token, das eine Spur darstellt, abgezogen. Der Kontrolle wird die Spur mit dem entsprechenden Lkw als Attribut übergeben.

Der Lkw und die Spur sind dann auf der Stelle Lkw in HO, bereit für VC. Bis zu dieser Stelle entspricht das Terminal-Netz fast dem des Lkws, da es bis zu dieser Stelle synchron mit dem Lifecycle eines Lkws ist. (siehe Lkw-Netz).

In der HO werden VCs initialisiert und bei der Kontrolle angemeldet. Das passiert an den Transitionen vc:new VC und vc:init-HO() und an der Transition ic:anmelden(vc). Es gibt also im Yard und in der HO zur Verfügung stehende VCs .

Wenn man das Terminal-Netz jetzt von der unteren Seite betrachtet, sieht man, daß dort die Bereiche abgebildet sind, die für einen VC eine Rolle spielen. Es werden zum Beispiel neue VC Objekte im Yard erzeugt (vc:new VC). Diese neuen VC Objekte werden durch eine Transition in dem Terminal-Netz initialisiert (vc:init-Yard()) und bei der Kontrolle als zur Verfügung stehende VCs angemeldet (ic:anmelden(vc)).

Steht ein Lkw, der entladen werden soll, in der HO, ist es günstig, auf einen VC zurückzugreifen, der ebenfalls in der HO steht. Soll ein Lkw beladen werden, sollte man einen VC beauftragen, der im Yard steht. Soll ein in der HO stehender Lkw beladen werden, muß ein VC dafür herangezogen werden (siehe hierzu auch das Kontroll-Netz). Wenn der dafür herangezogene VC im Yard steht, kann die Transition Container aufladen feuern. An dieser Transition synchronisiert sich das Terminal-Netz mit dem VC-Netz (vc:lade-container-auf()) und mit dem Kontroll-Netz (ic:soll-vc-Container-aufladen(vc)), da wieder der fehler-

freie Ablauf sichergestellt werden muß: d.h. es wird überprüft, ob der VC und die Kontrolle wirklich beide bereit sind, d.h. ihre entsprechenden Transitionen aktiviert sind. Die Transition fahre zur HO, die sich auch mit dem VC - und dem Kontroll-Netz synchronisiert (vc:fahre-zur-HO(), ic: soll-vc-zur-HO(vc)), ist zu diesem Zeitpunkt auch aktiviert und kann feuern.

Der VC ist jetzt auf der Stelle VC in HO, bereit für Lkw angekommen. Da jetzt sowohl der Lkw als auch der VC bereit sind zu beladen, kann die Transition beladen gefeuert werden. An dieser Transition synchronisieren sich das Lkw -, das VC- und das Kontroll-Netz (lkw:beladen-Lkw(), vc:beladen-Lkw(), ic:soll-vc-Lkw-beladen(lkw,vc,pos)). Nur wenn in allen drei Objekten die entsprechenden Transitionen aktiviert sind, kann die Transition beladen feuern.

Wenn der Vorgang des Beladens abgeschlossen ist, bleibt der VC in der HO, d.h. er bleibt auf der Stelle VC in HO bereit für Lkw. Die Transition Terminal verlassen, die mit dem Lkw-Netz synchronisiert ist (lkw:raus()), ist nach dem Beladen aktiviert und kann jetzt feuern. Dadurch verläßt der Lkw das Terminal. Durch das Feuern dieser Transition wird auch das vorher entnommene Spur-Token wieder zurückgelegt.

Soll ein in der HO stehender Lkw entladen werden, sollte ein VC herangezogen werden, der schon in der HO bzw. auf der Stelle VC in HO, bereit für Lkw steht. Es kann die Transition entladen feuern, wenn das Lkw -, das VC – und das Kontroll-Netz bereit sind (lkw:entladen-Lkw(), vc:entladen-Lkw(), ic:soll-vc-Lkw-entladen(lkw,vc,pos)). Im Terminal kann danach die Transition fahre zum Yard feuern, die mit dem VC- und dem Kontroll-Netz synchronisiert wird (vc:fahre-zum-Yard(), ic:soll-vc-zum-Yard(vc)). Wenn der VC dann auf der Stelle Yard angekommen ist, ist die Transition Container abladen aktiviert und kann feuern. Auch sie wird mit dem VC-Netz und dem Kontroll-Netz synchronisiert (vc:lade-Container-ab(), ic:soll-vc-Container -abladen(vc)).

Der Lkw kann, wenn er einen Entladeauftrag hat, das Terminal sofort nach dem Entladen verlassen. Er muß nicht warten, bis der VC seinen Auftrag erledigt hat. D.h die Transition Terminal verlassen kann feuern, wärend der VC noch auf der Stelle VC in HO, bereit für Lkw steht und wartet, daß die Transition VC fährt in den Yard feuert.

Genauso kann ein VC mit einem Auftrag schon in die HO fahren, auch wenn sein Lkw noch nicht da ist. Er wartet dann solange in der HO. Falls der Lkw einen Ent- und Beladeauftrag hat, kann es passieren, daß der VC, der beladen möchte, früher in der HO eintrifft, als der, der entladen möchte. Der VC, der beladen soll, wartet dann solange, bis der andere seinen Auftrag erledigt hat. Der Lkw verläßt das Terminal erst dann, wenn er wieder beladen ist.

Hat man einen Entladeauftrag und keinen VC in der HO, muß die Kontrolle dafür sorgen, das ein VC leer, d.h. ohne Container, in die HO fährt. Sollte kein VC im Yard sein, wenn man einen im Yard benötigt, muß die Kontrolle einen VC leer aus der HO in den Yard fahren lassen.

3.3 Desmoj Modell

Dieses Kapitel befaßt sich mit der Simulationsmethode mit Desmoj. Um die Vorgehensweise verstehen zu können, werden als erstes einige Klassendiagram-

me erklärt. Es handelt sich dabei um die in Desmoj zur Verfügung stehenden Klassen, die für dieses Modell benutzt worden sind, die Klassen, die für das Modell neu geschrieben worden sind, und um die Vererbungsbeziehungen der Klassen. Im Anschluß beschäftigt sich dieses Kapitel mit den zum Terminal gehörenden Klassen und Auszügen aus dem Quellcode. Im Anhang findet man den vollständigen Quellcode.

3.3.1 Klassendiagramme

Im Desmoj Modell werden das Terminal, die Kontrolle, der Lkw und der VC, wie bei dem Petrinetzmodell auch, von einander gekapselt. Es gibt für jedes Konstrukt eine eigene Klasse, mit den dazugehörigen Variablen, Konstanten und Methoden. Außerdem enthält jede dieser Klassen, außer der Terminal Klasse, einen eigenen Lifecycle. In dem Lifecycle wird der jeweilige Prozeß abgebildet, den ein bestimmtes Konstrukt durchläuft. Der Lifecycle enthält alle Methoden, die nötig sind, um einen Prozeß zu realisieren. Zum Beispiel wäre der Lifecycle eines Lkws der Prozeß von der Generierung des Lkws bis zur Beendigung seines Auftrages. Es sind bei dem Desmoj Modell aber noch weitere Klassen nötig, um das Szenario zu realisieren. Diese zusätzlichen Klassen sind nötig, da einigen Desmoj Klassen nur bestimmte Typen übergeben werden können. Sie dienen also im Wesentlichen der Typumwandlung. Im folgenden werden die Klassendiagramme besprochen. Im Anhang befindet sich der vollständige Quellcode.

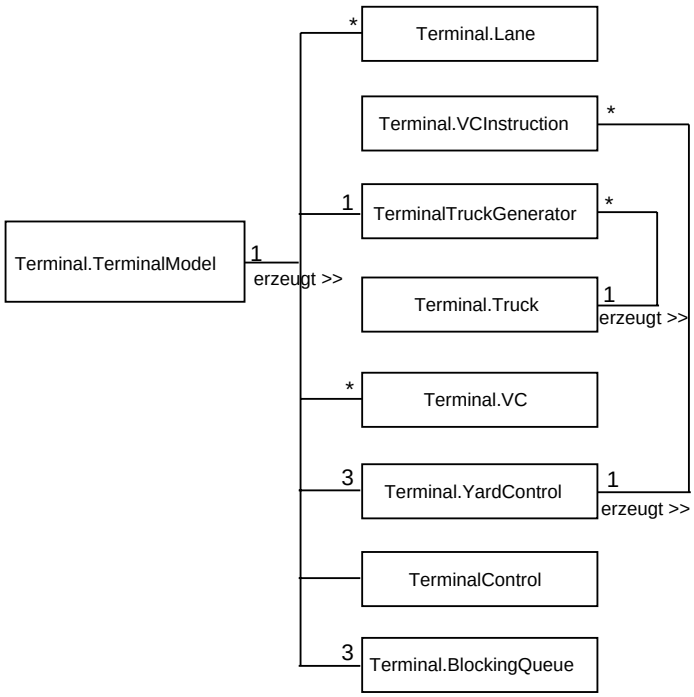

Abbildung 3.10: Das Klassendiagramm, der vom Package Terminal ereugten Klassen

Das Package, zu dem dieses Modell gehört, heißt Terminal. Zu diesem Packa-

ge Terminal gehören die in Abbildung 3.10 zu sehenden Klassen. Die Klasse Terminal erzeugt Objekte der Klassen Lane, TruckGenerator, VC, YardControl, Control und BlockingQueue. Das TruckGenerator-Objekt ist für die Erzeugung neuer Lkws (bzw. Lkw-Objekte) zuständig. Jedes YardControl-Objekt erzeugt VCInstruction-Objekte. Ein VCInstruction-Objekt stellt einen Auftrag für einen VC dar. Dieses Objekt enthält also Angaben darüber, ob ein VC entladen oder beladen soll. Alle Lane-Objekte zusammen bilden die Holdingarea, d.h. ein Lane-Objekt ist eine Spur, in der ein Lkw bedient werden kann. Da in dem gewählten Beispiel von drei Yards ausgegangen wird, wurden auch drei YardControl-Objekte erzeugt. Sie sind für die Verwaltung der VCs eines ihnen zugewiesenen Yards zuständig, d.h. sie nehmen VC Aufträge an und geben jeweils einen an einen VC weiter, der sich bei der YardControl als bereit gemeldet hat. Das Control-Objekt ist dazu da, alle Abläufe, die auf dem Terminal stattfinden, zu koordinieren. Die Klasse BlockingQueue sorgt dafür, daß die Lkws bei ihren Fahrten von einer Warteschlang in die nächste wirklich den Platz bekommen, der für sie frei geworden ist, auch wenn sie von einem anderen Lkw überholt werden.

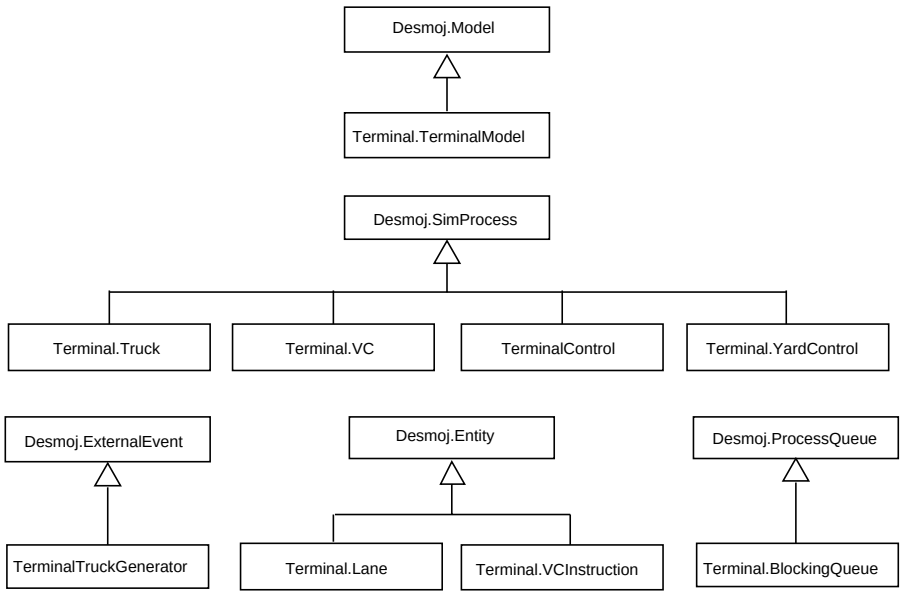

Abbildung 3.11: Das Klassendiagramm, der Vererbung im Package Terminal

Abbildung 3.11 zeigt die Klassen, von denen die im Modell benutzten Klassen erben. Die Klasse TerminalModel erbt von der Klasse-Model, die in dem Desmoj Package desmoj enthalten ist. Die Model Klasse ist dafür da, die Referenzen zu den statischen Modellkomponenten aufzunehmen, die zum Modell gehören. Das sind zum Beispiel Verteilungen oder statische Zähler. Außerdem ist die Terminal Klasse die Klasse, die die main-Methode enthält.

Die Klassen Lane und VCInstruction erben beide von der Klasse Desmoj.Entity. Diese Klasse stellt die Superklasse für alle Entitäten, da die in einem Modell enthalten sind. Entitäten werden zusammen mit kompatiblen Ereignissen zu einem bestimmten Simulationszeitpunkt in die Ereignisliste eingetragen. En-

titäten kapseln üblicherweise alle Daten einer Modellentität, die aus Sicht des Entwicklers relevant sind. Ereignisse (engl. Events) können diese Daten manipulieren und somit den Modellzustand verändern, sobald der Zeitpunkt, für den ein solches Ereignis vorgemerkt wurde, erreicht ist.

Die Klasse TruckGenerator erbt von der Klasse ExternalEvent. ExternalEvent stellt die Basis für benutzerdefinierte Ereignisse dar. Benutzerdefinierte Ereignisse werden definiert, indem Klassen von ExternalEvent erben. Um neue externe Ereignisse zu generieren, ist es notwendig, neue Objekte dieser Klasse zu definieren.

Die vier Klassen Truck, VC, YardControl und Control erben jeweils von der Klasse SimProcess. SimProcess beschreibt Entitäten mit besonderen Eigenschaften – insbesondere besitzen SimProcesse einen eigenen Lifecycle.

Simprocess-Objekte könne sowohl ereignis- als auch prozeßorientiert verwendet werden. Subclasses müssen die lifeCicle() Methode überschreiben, um ihr individuelles Verhalten zu beschreiben.

Die Klasse BlockingQueue erbt von der Klasse ProcessQueue. Die ProcessQueue ist eine Warteschlange, die SimProcess-Objekte, wie zum Beispiel einen Lkw, aufnehmen kann.

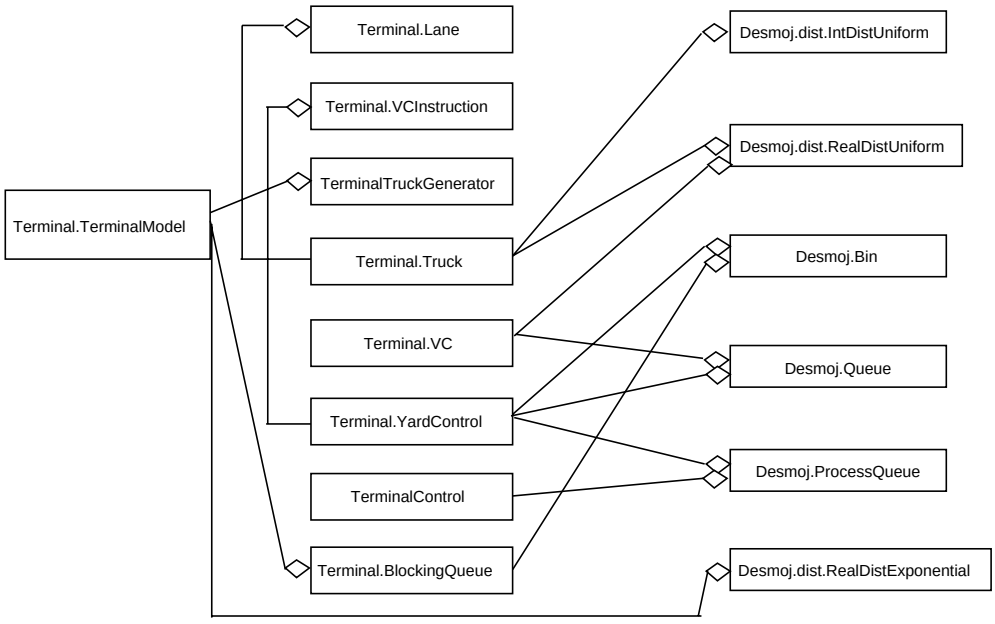

Abbildung 3.12: Das Klassendiagramm, der vom Package Terminal benutzten Klassen

In Abbildung 3.12 wird gezeigt, welche Klasse welche Klassen benutzt. Die Klasse TerminalModel benutzt die Klasse TruckGenerator, da es die Aufgabe des Terminals ist, für neue Lkws zu sorgen. Außerdem benutzt sie die Klasse BlockingQueue, die eine Warteschlange darstellt, und die Klasse Desmoj.dist.RealDistExponential, die einen Zufallszahlenstrom exponential verteilter rationaler Zahlen darstellt. Die BlockingQueues sollen genau die Warteschlangen representieren, die man auf dem Terminal auch tatsächlich findet. Zum Beispiel die Warteschlange vor der Holdingarea. Die BlockingQueue ist in der

Lage, Objekte von Typ Entität und SimProcess aufzunehmen. Der Lkw ist vom Typ SimProcess, d.h. er kann sich in die Warteschlange einreihen. Die Exponentialverteilung ist für die Ankunft der neuen Lkws geeignet. Es wird per Zufall entschieden, welchen zeitlichen Abstand die Lkws zur durchschnittlichen Ankunftszeit bekommen.

Die Klasse Truck benutzt die Klassen Desmoj.dist.IntDistUniform und Desmoj.dist.RealDistUniform. Hierbei handelt es sich ebenfalls um Zufallszahlenströme, allerdings um gleichverteilte, einmal mit ganzen Zahlen und einmal mit rationalen Zahlen. Mit der Gleichverteilung der ganzen Zahlen wird per Zufall ausgewählt, welche Art von Auftrag einem Lkw zugewiesen wird. Die Aufträge sind durchnummerierte Konstanten. Mit dem anderen Zufallszahlenstrom, sollen die Fahrzeiten eines Lkws zwischen zwei Standorten auf dem Terminal beschrieben werden.

Der VC muß auch Fahrzeiten berücksichtigen, und benutzt deshalb ebenfalls die Klasse Desmoj.dist.RealDistUniform. Und er benutzt die Klasse Desmoj.Queue. Diese Klasse stellt eine Warteschlange dar, die mit Entitäten gefüllt werden kannn. Der VC speichert in dieser Warteschlange seine Aufträge (VCInstruction), die von Typ Entität sind. Die Klasse YardControl benutzt die schon besprochenen Klassen Desmoj.Queue und Desmoj.ProcessQueue um Aufträge und freie VCs zu speichern. Zusätzlich benutzt sie noch die Klasse Desmoj.Bin. Diese Klasse ist die Implementation eines Semaphors, das die Auftragsanzahl verwaltet. In diesem Beispiel werden VC-Aufträge verwaltet. Die Klasse BlockingQueue benutzt ebenfalls die Klasse Desmoj.Bin. Sie kombiniert die Funktionalität einer Queue mit der eines Bins, um die freien Plätze der Warteschlangen auf dem Terminal zu verwalten.

Die Klasse Control benutzt ausschließlich die Klasse Desmoj.ProcessQueue. Ein Control-Objekt muß, da es der Koordination dient, sehr viele Zustände speichern. Es muß speichern, welcher Lkw welchen Auftrag hat und in welche Spur er sich einreihen wird, bzw. er sich eingereiht hat. Und es muß speichern, an welchen VC welcher Auftrag übergeben wurde, damit es dem VC die Spur, in der der entsprechende Lkw steht, später übergeben kann. All diese Informationen verwaltet ein Control-Objekt in verschiedenen ProcessQueues.

3.3.2 Die Klasse TerminalModel

Man kann in dem Klassendiagramm der Klasse TerminalModel alle Variablen und Methoden sehen, die für diese Klasse nötig sind: Sämtliche auf dem Terminal befindlichen Orte, wie die Warteschlange vor dem Terminal, das IC, die Warteschlange vor der Holdingarea und eine Warteschlange für bearbeitete Lkws, findet man in dieser Klasse als BlockingQueue oder Desmoj.ProcessQueue wieder. Zusätzlich kann man in dieser Klasse angeben, wie viele VCs es pro Yard geben soll und wie viele Spuren die Holdingarea haben soll. Die Holdingarea selbst wird durch eine Desmoj.Queue dargestellt, in der die Spuren enthalten sind. Mit der Methode description() kann man das Modell kurz beschreiben. DoInitialSchedules() sorgt dafür, daß die Entitäten oder Ereignisse, die für ein Modell gebraucht werden, mit den definierten Einstellungen gestartet werden. Die Methode init() macht die Initialisierungsarbeit für das Modell. Die Klasse TerminalModel hat keinen Lifecycle (vgl. Abbildung 3.13). Vergleiche auch die

folgende Darstellung mit den entsprechenden Ablaufdiagrammen aus Abbildung 3.2 bis 3.4.

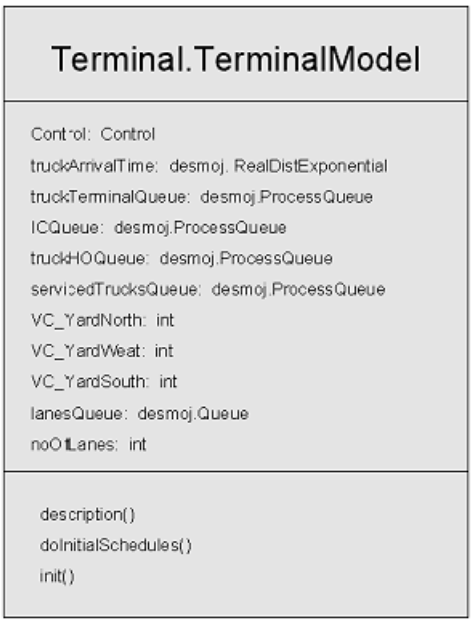

Abbildung 3.13: Das Klassendiagramm des Terminals

3.3.3 Die Klasse Control

Die Klasse Control dient der Organisation des Terminals. Es gibt drei verschiedene Desmoj.ProcessQueues, in denen die Kontrolle speichert, welcher VC oder welcher Lkw in der Holdingarea steht. Es kann sein, daß ein Lkw auf einen VC warten muß, daß ein VC auf einen Lkw warten muß, oder daß ein VC, der beladen will, und ein Lkw auf einen VC, der entladen soll, warten müssen. Es gibt also eine Queue für wartende Lkws, für wartende VCs und für wartende belade VCs. Die Methode serviceTruckarrival() übergibt die Lkw-Aufträge an die entsprechenden Yard Kontrollen. Die Methode tellPosition() übergibt einem VC die Spur, in der sich sein entsprechender Lkw befindet. Der Lifecycle der Kontrolle wird an einer bestimmten Stelle in zwei Methoden aufgeteilt. Jenachdem, wer zuerst an der Holdingarea ankommt, der VC oder der Lkw, wird entweder die Methode handleArrivedTruck() oder handleArrivedVC() aufgerufen. Das wird gemacht, da der Ablauf des Lifecycles, je nach Ankunftsreihenfolge, unterschiedlich fortgesetzt wird (vgl. Abbildung 3.14, 3.15 und 3.2).

3.3.4 Die Klasse YardControl

Die Klasse YardControl ist eine relativ kleine Klasse mit wenigen Variablen und nur einer Methode. Das liegt daran, daß diese Klasse nicht mehr zu tun hat, als die von der Kontrolle übergebenen VC-Aufträge zu speichern und zu gegebener Zeit an die VCs zu übergeben (handoverInstruction()). Die Aufträge werden in der instructionsQueue gespeichert, die vom Typ Desmoj.Queue ist.

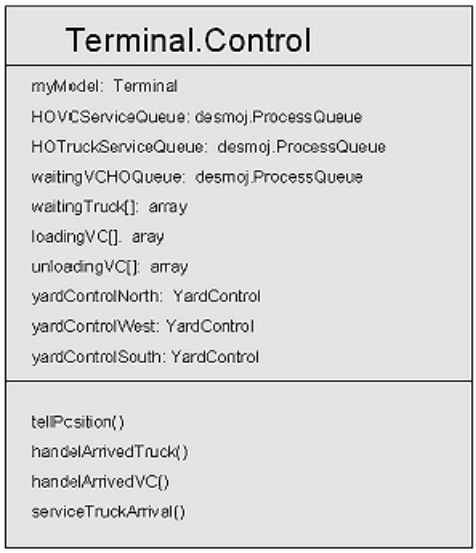

Abbildung 3.14: Das Klassendiagramm der Kontrolle

```
public void lifeCycle() {
    while(true) {
         this.passivate();
         // An der HO ist jemand angekommen
         // Wer hat uns aktiviert: der Lkw oder der VC?
         if(!HOTruckServiceQueue.isEmpty())  { // OK: ein Lkw
             handleArrivedTruck();
         } else  { // OK: VC hat uns geweckt
             handleArrivedVC();
         }
    }
}
```

Abbildung 3.15: Lebenszyklus der Kontrolle

Die zur Verfügung stehenden VCs werden in der idleVCQueue, vom Typ Desmoj.ProcessQueue, gespeichert. Diese Klasse hat einen Lifecycle, der genau das eben Beschriebene durchführt (vgl. Abbildung 3.16 und 3.17).

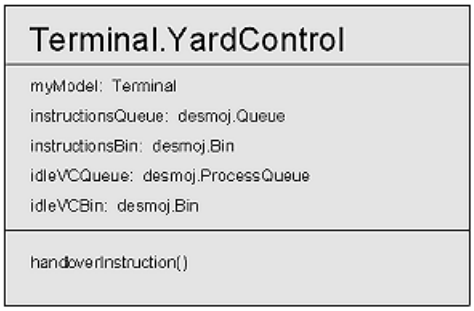

Abbildung 3.16: Das Klassendiagramm der Yardkontrolle

```
public void lifeCycle() {
    VC vc;
    VCInstruction instruction;
    while(true) {
        instructionsBin.deliver(1); // Blockiere, bis Auftrag da
        instruction = (VCInstruction) instructionsQueue.first();
        instructionsQueue.remove(instruction);
        // Vergebe Auftrag an unbeschäftigten VC
        idleVCBin.deliver(1); // Warte, bis einer da
        vc = (VC) idleVCQueue.first();
        idleVCQueue.remove(vc);
        vc.instructionsQueue.insert(instruction);
        vc.activateAfter(this);
    }
}
```

Abbildung 3.17: Lebenszyklus der Yard-Kontrolle

3.3.5 Die Klasse Truck

In der Klasse Truck findet man eine ganze Reihe Zufallszahlenströme, die die verschiedenen Fahrtzeiten zwischen zwei Orten auf dem Terminal darstellen. Ein Zufallszahlenstrom ist für die Auswahl des Lkw-Auftrags vorgesehen. Es gibt 15 verschiedene Auftragstypen (siehe Quellcode), d.h. es wird für jeden Lkw eine Zahl zwischen eins und 15 zufällig ausgewählt, hinter der sich ein ganz bestimmter Auftrag verbirgt.

Für einen Lkw, der einen Ent- und Beladeauftrag hat, ist die Variable firstTaskCompleted interessant. Sie zeigt, ob ein Lkw in der Holdingarea schon beladen werden kann, oder ob er erst noch entladen werden muß. In der Variblen Lane speichert der Lkw, in welche Spur er in der Holdingarea gefahren ist.

Bei den Methoden findet man viele, die dem Lkw vorgeben, was er als nächstes tun soll. GoToHO() sorgt zum Beispiel dafür, das der Lkw in die Holdingarea fährt. Zu diesen Methoden gibt es die passenden get Methoden. Diese Methoden sorgen dafür, daß die Zeitverbräuche zwischen zwei Orten oder Aktivitäten auch tatsächlich eingehalten werden. Zum Beispiel wird in der Methode goToHO() die Methode getDrivingTimeHOQueueHO() aufgerufen, die dafür sorgt, das die Simulationsuhr weitergeschaltet wird, also Fahrtzeit vergeht. Im Lifecycle kann man erkennen, welche Stationen der Lkw nacheinander durchlaufen muß, was er dabei tun muß und mit welchen Objekten er dabei kommunizieren, bzw. sich synchronisieren muß. Zum Schluß kann er bedient werden und sich dann in die Warteschlang der fertig bearbeiteten Lkws einreihen (vgl. Abbildung 3.18 und 3.19).

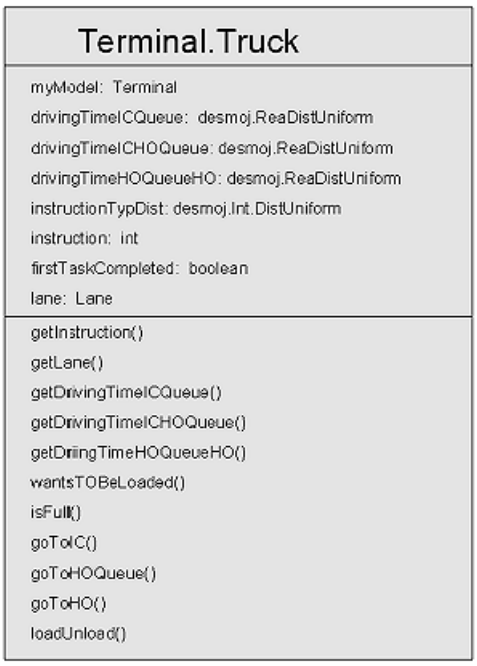

Abbildung 3.18: Das Klassendiagramm eines Lkw

3.3.6 Die Klasse VC

Bei der VC Klasse kann man anhand der Namen der Variablen und Methoden gut erkennen, welchen Zweck sie haben. Um Fahrzeiten darzustellen gibt es auch hier Zufallszahlenströme. Ein VC kann sich entweder im Yard oder in der Holdingarea aufhalten. Er muß deshalb speichern, wo er sich gerade befindet. Das tut er mit der Variable isInHO, die vom Typ Boolean ist. Welchen Auftrag der VC hat, speichert er mit der Variablen instruction, vom Typ VCInstruction. Diesen Auftrag speichert er dann in der InstructionsQueue vom Typ Desmoj.Queue. Die Methoden sind durch ihre Namen selbsterklärend. Erwähnenswert sind allerdings die Methoden loadCycle() und unloadCycle(). Ein VC muß sich, wenn er einen Beladeauftrag hat, anders verhalten, als wenn er einen Entladeauftrag hat. Darum teilt sich der Lifecycle an einer bestimmten

```
public void lifeCycle() {
    // Lkw-Zustand: "Lkw initialisiert"
    // Ankunft am Terminal:
    myModel.truckTerminalQueue.insert(this);
    goToIC();
    // Lkw-Zustand ist nun: "Lkw im IC"
    // und "ubergebe Kontrolle Auftragsbeschreibung
    myModel.control.serviceTruckArrival(this);
    goToHOQueue();
    goToHO();
    // Lkw-Zustand ist nun: "Lkw in der HO"
    // teile Kontrolle Spur mit
    myModel.control.tellPosition(this, lane);
    loadUnload();
    // FERTIG!
    myModel.servicedTrucksQueue.insert(this);
}
```

Abbildung 3.19: Lebenszyklus des Lkw

Stelle in die Methoden loadCycle() und unloadCycle auf (vgl. Abbildung 3.20 und 3.21).

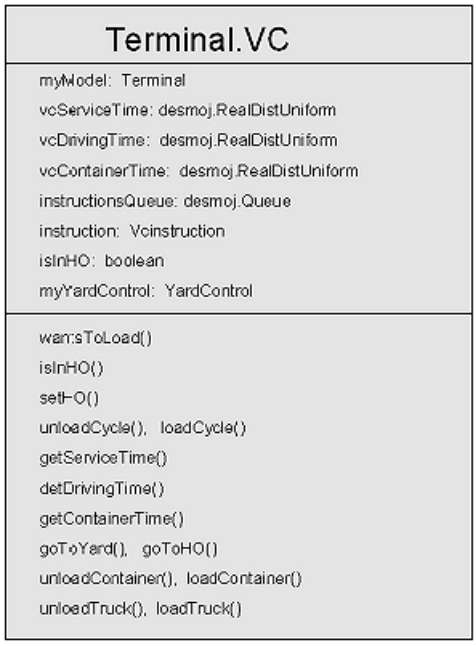

Abbildung 3.20: Das Klassendiagramm eines VC

```
public void lifeCycle() {
   while(true) {
       // Queue enthält 0 oder 1 Auftrag
       if(instructionsQueue.isEmpty()) { //leer
           // Sich in seinem Yard als unbeschäftigt melden
           myYardControl.idleVCQueue.insert(this);
           myYardControl.idleVCBin.store(1);
           // Nix zu tun, Yard-Kontrolle soll uns wecken.
           passivate();
       }
       // Yard-Kontrolle hat uns einen Auftrag gegeben.
       instruction = (VCInstruction) instructionsQueue.first();
       instructionsQueue.remove(instruction);
       // dies ist mein zu bearbeitender Lkw
       switch(instruction.typ) {
           case VC.UNLOAD:
                unloadCycle(instruction.truck);
                break;
           case VC.LOAD:
                loadCycle(instruction.truck);
                           break;
           default: break;
       }
    }
}
```

Abbildung 3.21: Lebenszyklus des Vancarrier

```
public void unloadCycle(Truck truck)
{
        if(!isInHO()) {
                gotoHO();
        }

        // bei der Kontrolle vormerken
        myModel.control.HOVCServiceQueue.insert(this);
        // und sie wecken
        myModel.control.activateAfter(this);
        // Lass die Kontrolle mal machen
        this.passivate();

        // LKW ist jetzt da und will entladen werden:
        unloadTruck();
        truck.activateAfter(this);

        gotoYard();
        unloadContainer();
}

public void loadCycle(Truck truck) {
        if(isInHO()) {
                gotoYard();
        }
        loadContainer();
        gotoHO();

        myModel.control.HOVCServiceQueue.insert(this);
        // und sie wecken
        myModel.control.activateAfter(this);
        this.passivate();

        // LKW ist jetzt da und will beladen werden:
        loadTruck();
        truck.activateAfter(this);
}
```

Abbildung 3.22: Lebenszyklus des Vancarrier, fortgesetzt

Kapitel 4

Auswertung

In diesem Kapitel werden die Auswertung durch das stochastische Modell, die Auswertung bezüglich der Fragestellung und die Auswertung der Vorgehensweise vorgenommen. Das stochastische Modell gibt Aufschluß über die Anzahl der VCs, die man theoretisch braucht, um das System im Gleichgewicht zu halten. Es werden dabei obere und untere Grenzen der Bedienzeiten für die Lkws eingesetzt, um eine Mindestanzahl und eine ausreichende Anzahl an VCs angeben zu können. Das nächste Kapitel erklärt die Simulationsergebnisse, die mit dem Simulator, der in Desmoj erstellt wurde, erzielt worden sind. Danach werden Renew und Desmoj bezüglich ihrer Handhabung zur Simulation miteinander verglichen.

4.1 Abschätzung durch stochastisches Markovmodell

Betrachten wir eine sehr große Abschätzung des Szenarios durch ein analytisches Modell: ein Wartesystem, genau: M/M/1.

Mit Hilfe der Formeln aus Kapitel 2.3.2 läßt sich ausrechnen, wieviele VCs immer im Einsatz sein müssen und wieviele VCs ausreichend sind um das Wartesystem im Gleichgewicht zu halten. Es wird die Formel für ein M/M/1-System verwendet. Die Daten der Rechnung stammen aus dem Quellcode im Anhang. Es kommt alle 5 Minuten ein Lkw am Terminal an, es gibt 6 Spuren und die Bedienzeit der VCs liegt zwischen 11 und 68 Minuten.

Zwischenankunfszeit A	$E[A] = \frac{1}{\lambda} = 5$ Minuten pro Lkw
Zugang Z	$E[Z] = \lambda = 0,2$ Lkws pro Minute
Durchsatz D	$E[D] = \lambda = 0,2$ Lkws pro Minute und
Grenzdurchsatz C	$E[C] = \mu = \frac{1}{11}$ Minimum und $\frac{1}{68}$ Maximum
Bedienzeit B	$E[B]_{min} = \frac{1}{\mu} = 11$ min. (Minimum)
	$E[B]_{max} = \frac{1}{\mu} = 68$ min. (Maximum)
Füllung F	$E[F] = \frac{\rho}{1-\rho}$

Setzen wir $E[F]$ mit der Anzahl der Spuren l gleich, so ergibt sich: $E[F] = l$

$$
\begin{array}{rrcl}
\Leftrightarrow & \rho & = & l - \rho l \Leftrightarrow \rho = \frac{l}{1+l} \\
 & \rho & = & \frac{l}{1+l} \leq 1 \\
 & \rho & = & \frac{\lambda}{\mu} \\
 & \mu & = & m\mu_0 \\
\Leftrightarrow & \rho & = & \frac{\lambda}{m\mu_0} = \frac{l}{1+l} \\
\Leftrightarrow & m & = & \frac{l+1}{l}\frac{\lambda}{\mu_0}
\end{array}
$$

m ist die Anzahl der Bediener, d.h. die Anzahl der benötigten VCs. Um eine Abschätzung machen zu können, wieviele VCs man braucht, wird einmal mit der minimalen Bedienzeit und einmal mit der maximalen Bedienzeit gerechnet. Die beiden errechneten Werte bilden dann die unterste und die oberste Schranke.

$$m_{min} = \frac{l+1}{l}\frac{\lambda}{\mu_{min}} = \frac{7}{6} * \frac{1/5}{1/11} = \frac{7*11}{5*6} = 2,566$$
$$m_{max} = \frac{l+1}{l}\frac{\lambda}{\mu_{max}} = \frac{7}{6} * \frac{1/5}{1/68} = \frac{7*68}{5*6} = 15,86$$

In dem behandelten Szenario wird von drei Yards ausgegangen, d.h., daß für jeden Yard im besten Fall nur ein VC im Einsatz sein muß, und im schlechtesten Fall mindestens sechs VCs pro Yard vorhanden sein müssen, um das System im Gleichgewicht halten zu können. Da diese Rechnung stark vereinfacht wurde, sind die beiden errechneten Werte keine genauen Angaben. Sie dienen lediglich zur Abschätzung der Größenordnung.

4.2 Auswertung bezüglich der Fragestellung

Die Fragestellung lautete, wieviele VCs und wieviele Spuren sind nötig, um alle auf dem Terminal ankommenden Lkws möglichst schnell zu bedienen, ohne dabei zuviele VCs auftragslos auf dem Yard stehen zu lassen.

Da sich diese Arbeit mit dem Vergleich zweier Simulationswerkzeuge beschäftigt, wird auf die Auswertung bezüglich der Realität eines Lkw-Terminals kein Wert gelegt. Es geht vielmehr um das Prinzip der Auswertung. Es wird bei dieser Auswertung auch nur das mit Desmoj erstellte Modell berücksichtigt, da es in Renew noch keine vergleichbare stochastische Auswertung gibt.

In dieser Auswertung soll festgestellt werden, wieviele Lkws bei einer konstanten Anzahl an Spuren und verschiedenen Anzahlen an VCs in einer bestimmten Zeit bedient werden können. Es wird deshalb eine konstante Anzahl an Spuren genommen, da die Veränderungen einer Simulation besser wahrgenommem werden, wenn sich nur ein Parameter verändert, d.h. wenn sich nur die Anzahl der VCs ändert.

Es soll alle fünf Minuten ein Lkw das Terminal erreichen, die Anzahl der Spuren beträgt sechs und die Anzahl der VCs geht von eins bis neun. Die Simulationszeit soll 1000 Minuten betragen.

In den Abbildungen 4.1 und 4.2 sieht man, daß sieben VCs pro Yard die größte Anzahl an Lkws bedienen können. Hat man weniger VCs, dann können nur weniger Lkws bearbeitet werden. Hat man aber mehr als sieben VCs, dann müßten eigentlich noch mehr Lkws bedient werden können. Es werden

aber weniger bedient. Das liegt daran, daß der Verwaltungsaufwand für eine steigende Anzahl an VCs auch größer wird. Bei sieben VCs liegt demnach wahrscheinlich das Optimum zwischen der Dauer der Verwaltung der VCs und der Auslastung der VCs.

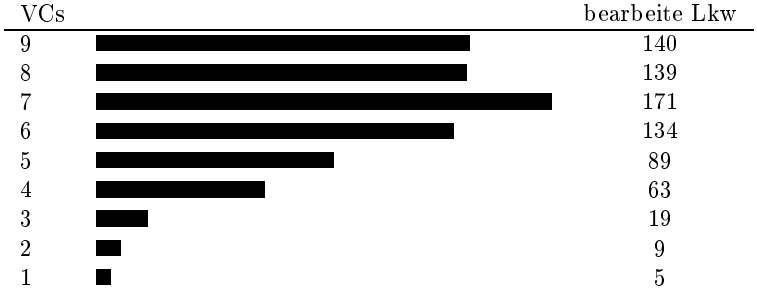

Abbildung 4.1: Anzahl der bearbeiteten Lkws, bei unterschiedlicher Anzahl VCs

Um solche Diagramme erstellen zu können, muß man sich die statistische Auswertung von Desmoj angucken. Es wird bei jedem Simulationsdurchgang eine HTML-Datei produziert, die Aufschluß über den Simulationsdurchgang gibt. In der statistischen Auswertung für dieses Szenario sind alle Warteschlangen berücksichtigt, wie zum Beispiel u.a. das Interchange, die Idle VC Queues, die YCinstuctionQ (Yardcontrol instruction queue) und die serviced truck Queue. Alle Warteschlangen zeigen eine durchschnittliche, eine maximale und eine momentane Füllung an. Außerdem zeigen sie zusätzlich noch durchschnittliche Wartezeiten und das Warte-Prinzip (hier immer „fifo" (first in first out)) an. Die Abbildungen 4.3, 4.4 und 4.5 zeigen so eine statistische Auswertung.

4.3 Evaluierung der gewählten Vorgehensweise

Man kann an den beiden vorgestellten Modellen sehen, daß es möglich ist, das gleiche Szenario mit ganz unterschiedlichen Methoden zu simulieren. Man kann aber nicht sagen, welche Methode besser oder schlechter ist als die andere, da sie beide ganz unterschiedlich arbeiten. Man kann aber Aussagen darüber treffen, wo die Methoden jeweils ihre Stärken und ihre Schwächen haben.

Das Petrinetz Modell hat den großen Vorteil, daß es nur aus den erwähnten vier Netzen Terminal, Kontrolle, Lkw und VC besteht, und keine weiteren Netze nötig sind um das Szenario zu simulieren. Im Desmoj Modell benötigt man zusätzlich zu den Klassen Terminal, Kontrolle, Lkw und VC noch einige weitere Klassen zur Unterstützung. Man braucht zum Beispiel eine YardControl-, eine Lane- und eine VCInstruction-Klasse, um die verschiedenen Vorgänge auf dem Terminal korrekt simulieren zu können. Diese zusätzlichen Klassen haben die Aufgabe, die Synchronisation der einzelnen Objekte, die im Desmoj Modell deutlich aufwendiger ist als im Petrinetz Modell, besser durchführen zu können. Im Petrinetz Modell reicht es, eine Transition entsprechend zu beschriften, damit die Synchronisation von Objekten korrekt arbeitet. Man kann also sagen, daß das Petrinetz Modell, was die Synchronisation von Objekten

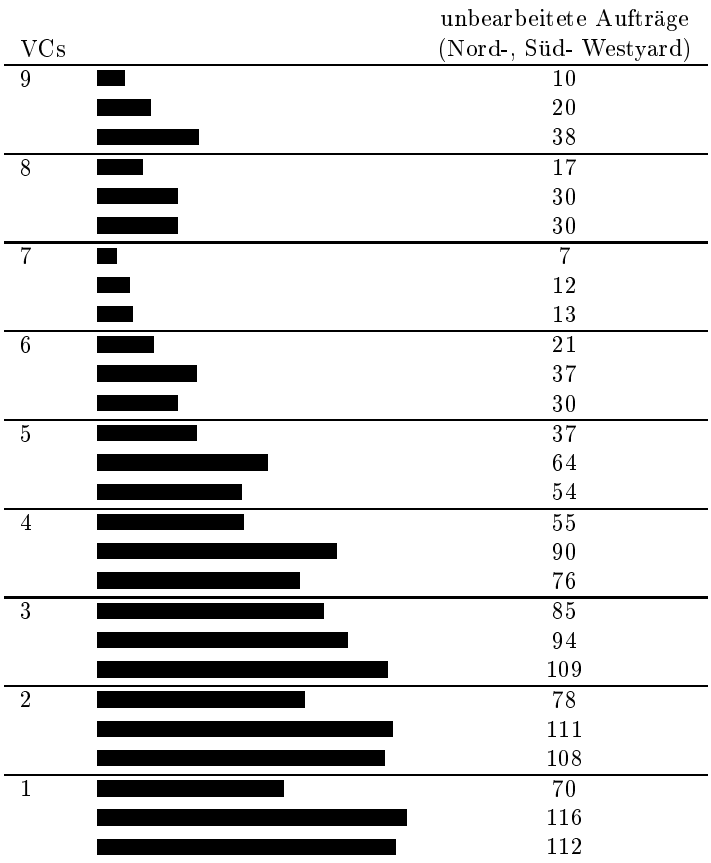

Abbildung 4.2: Anzahl der Aufträge, die von den Yard-Kontrollen noch an keinem VC übergeben wurden.

Title	Obs	Qmax	Qnow	Qavg.	Zeros	avg.Wait	refus.
Trucks outside terminal	199	1	0	0.0	199	0.0	0
Interchange	198	1	0	0.0	198	0.0	0
Trucks in front of the HO	70	127	127	41.44392	70	0.0	0
Serviced trucks	0	64	64	49.61118	0	0.0	0
VC arriving in the HO	120	1	0	0.0	120	0.0	0
Service-requests in HO	108	1	0	0.0	108	0.0	0
VC waiting for LKW	76	16	16	12.48543	58	17.56502	0
Idle VC Queue	39	6	0	1.61887	5	41.5094	0
Idle VC Queue	44	6	0	0.64862	20	14.74129	0
Idle VC Queue	37	6	0	1.46993	10	39.72777	0
Empty lanes	70	6	0	1.11993	0	15.99894	0
YC instructionsQ	40	48	48	12.68874	38	0.02902	0
YC instructionsQ	45	73	73	23.95178	31	2.68369	0
YC instructionsQ	38	78	78	24.70186	29	7.53695	0

Abbildung 4.3: Statistische Auswertung bei sechs VCs

Title	Obs	Qmax	Qnow	Qavg.	Zeros	avg.Wait	refus.
Trucks outside terminal	199	1	0	0.0	199	0.0	0
Interchange	198	1	0	0.0	198	0.0	0
Trucks in front of the HO	109	88	88	22.2746	109	0.0	0
Serviced trucks	0	103	103	69.17036	0	0.0	0
VC arriving in the HO	191	1	0	0.0	191	0.0	0
Service-requests in HO	176	1	0	0.0	176	0.0	0
VC waiting for LKW	126	19	19	13.36803	92	19.49626	0
Idle VC Queue	47	7	0	2.5656	3	54.58726	0
Idle VC Queue	66	7	0	1.63758	26	24.81177	0
Idle VC Queue	78	7	0	1.25162	19	16.0464	0
Empty lanes	109	6	0	1.48349	0	13.60998	0
YC instructionsQ	48	47	47	9.70808	46	0.45145	0
YC instructionsQ	67	46	46	9.4839	47	6.94001	0
YC instructionsQ	79	44	44	9.6685	70	0.55771	0

Abbildung 4.4: Statistische Auswertung bei sieben VCs

Title	Obs	Qmax	Qnow	Qavg.	Zeros	avg.Wait	refus.
Trucks outside terminal	199	1	0	0.0	199	0.0	0
Interchange	198	1	0	0.0	198	0.0	0
Trucks in front of the HO	114	83	83	20.29982	114	0.0	0
Serviced trucks	0	108	108	71.39209	0	0.0	0
VC arriving in the HO	196	1	0	0.0	196	0.0	0
Service-requests in HO	178	1	0	0.0	178	0.0	0
VC waiting for LKW	119	23	23	15.56189	81	29.69657	0
Idle VC Queue	53	8	0	2.25531	17	42.553	0
Idle VC Queue	82	8	0	2.7774	6	33.87079	0
Idle VC Queue	61	8	0	2.62998	11	43.11445	0
Empty lanes	114	6	0	1.7192	0	15.08071	0
YC instructionsQ	54	35	35	6.6246	42	2.74547	0
YC instructionsQ	83	38	38	7.66002	80	1.22074	0
YC instructionsQ	62	54	54	11.30056	52	4.56455	0

Abbildung 4.5: Statistische Auswertung bei acht VCs

betrifft, deutlich einfacher zu handhaben ist als das Desmoj Modell.

Ein weiterer Vorteil der Petrinetze gegenüber dem Desmoj Modell ist der benötigte Zeitaufwand, um einen Simulator fertigzustellen. Da man den Ablauf eines Szenarios im Petrinetz intuitiv darstellen kann, ist der Zeitaufwand zur Fertigstellung eher gering, im Vergleich zu einem Modell das mit Desmoj erstellt wird. Das liegt daran, daß es für die Simulation mit Desmoj sehr wichtig ist, gut Java programmieren zu können. Der Quellcode von Desmoj ist nicht intuitiv erfaßbar, und es ist nicht einfach Fehler zu finden oder Änderungen durchzuführen, ohne daß andere Fehler auftreten können. Im Petrinetz Modell können sehr einfach Veränderungen durchgeführt werden. Man braucht zum Beispiel nur einige neue Transitionen und Stellen, die dann einen anderen Ablauf darstellen können. Außerdem ist es möglich, ein Petrinetz Modell, das mit Renew erstellt worden ist, bei der Simulation zu beobachten. diese Möglichkeit ist sehr nützlich bei der Fehlersuche, da man einen Fehler in einem Ablauf, den man beobachtet, schneller finden kann, als bei einer Ausgabe von Zahlen oder einer einfachen Fehlermeldung direkt nach einer Simulation.

Simulatoren, die mit Hilfe von Desmoj entwickelt werden, haben aber den großen Vorteil, daß sie über eine gute statistische Auswertung verfügen. Nach einem Simulationsdurchgang mit Desmoj bekommt man eine Auswertung, die einem genaustens Aufschluß darüber gibt, wie zum Beispiel Warteschlangen zu bestimmten Zeitpunkten gefüllt waren. Zu fast jeder von Desmoj vorgegebenen Klasse, die man zur Simulation einsetzen kann, gibt es diese Auswertung. Im Vergleich zu Petrinetzen ist Desmoj bezüglich der Ergebnisauswertung weit überlegen. In Petrinetzmodellen, die mit Renew erstellt worden sind, ist es derzeit noch nicht möglich, eine Auswertung zu machen, die über die Aussage, daß ein Simulator richtig konstruiert ist, hinaus geht. Es gibt aber Werkzeuge, die mit Perinetzen simulieren und eine stochastische Auswertung integriert haben. Auf diese Werkzeuge wird in dieser Arbeit aber nicht eingegangen.

Ein weiterer großer Vorteil von Desmoj gegenüber Petrinetzen, die mit Renew erstellt worden sind, ist, daß es in Desmoj wesentlich mehr verschiedene Warteschlangentypen und Typen von Zufallszahlenströmen gibt, die für eine gute, realistische Simulation wichtig sind. Die verschiedenen Warteschlangentypen arbeiten z.B. nach dem „fifo"- oder „lifo" Prinzip und können bestimmte Kapazitäten haben. Die Zufallszahlenströme sind für unterschiedliche Zeitverbräuche nützlich, die ein in der Simulation verwendetes Objekt haben kann. Es gibt in Desmoj Zufallszahlenströme mit Exponentialverteilung, Normalverteilung und Gleichverteilung. Unter Renew ist es zwar möglich, eine Stelle mit einer Kapazität zu versehen, es ist aber noch nicht möglich Warteschlangen zu realisieren. Um in Renew Zeitverbräuche zu simulieren, ist es derzeit möglich, zeitbehaftete Transitionen einzusetzen, bei denen nur eine bestimmte Zeit verbraucht wird.

Man kann abschließend sagen, das die Simulation mit Petrinetzen in der Handhabung der Simulation mit Desmoj überlegen ist. Betrachtet man die Auswertung einer Simulation, ist Desmoj den Petrinetzen überlegen. Es wäre also nützlich ein Simulationswerkzeug zu verwenden, das auf Petrinetzen basiert und eine umfangreiche statistische Auswertung besitzt.

Kapitel 5

Zusammenfassung und Ausblick

In dieser Arbeit wurden zwei unterschiedliche Werkzeuge, die zur Simulation eingesetzt werden können, miteinander verglichen. Es handelt sich zum einen um die Simulation mit dem Petrinetzwerkzeug Renew und zum anderen um die Simulation mit dem Framework Desmoj. Der Vergleich wurde an einem Beispiel eines Bedien-/Wartesystems aus dem Bereich der Logistik vorgenommen. Hier geht es dabei um den Bereich der Lkw-Abfertigung eines Containerterminals.

Das Terminal erreichen Lkws, die einen bestimmten Auftrag zu erfüllen haben. Entweder müssen sie einen Container abliefern, einen Container vom Terminal abholen oder einen bringen und einen abholen. Damit die Lkws bedient werden können, gibt es auf dem Terminal sogenannte Vancarrier (VC), die Container transportieren und Lkws be- und entladen. Die Container lagern in Yards, von denen es in diesem Beispiel drei gibt. Jedem Yard sind ganz bestimmte VCs zugeordnet. Hat ein Lkw einen Entlade-Auftrag für einen bestimmten Yard, dann muß ein VC aus dem richtigen Yard kommen, den Lkw entladen und den Container in seinen Yard fahren und dort abstellen. Das Terminal hat nur eine begrenzte Kapazität, d.h. es können nicht beliebig viele Lkws auf das Terminal hinauffahren. Damit die Kapazitäten nicht überschritten werden und die Koordination auf dem Terminal funktioniert, bedarf es einer Kontrolle, die den Ablauf regelt.

Die Frage bei diesem Simulationsbeispiel lautet: Wieviele VCs werden gebraucht und wie groß müssen die Kapazitäten des Terminals sein, damit in einer bestimmten Zeit möglichst viele Lkws bedient werden können.

Da es in dieser Arbeit um den Vergleich zweier Simulationswerkzeuge geht, ist die Auswertung der Fragestellung etwas vereinfacht worden. Die Kapazitäten bleiben konstant und es wird nur bezüglich der Anzahl der benötigten VCs optimiert.

Das beschriebene Szenario ist mit beiden Simulationswerkzeugen erstellt worden, wobei sich gezeigt hat, daß beide ganz unterschiedliche Vor- und Nachteile haben. Es hat sich bei der Auswertung der Vorgehensweise gezeigt, daß das Erstellen des Simulators mit Petrinetzen wesentlich einfacher und schneller geht als mit Desmoj. Das ist darauf zurückzuführen, daß Petrinetze intuitiv erstellt werden können. Um einen Simulator mit Desmoj zu erstellen, muß man

das Framework gut kennen oder sich darin einarbeiten, was einen nicht unbeträchtlichen Zeitaufwand bedeuten kann. Selbst wenn man das Framework kennt, ist es zusätzlich noch wichtig, gut in Java programmieren zu können. Außerdem wurde festgestellt, daß Fehler in einem Petrinetz schneller erkannt werden können als in Java-Code.

Bei der Betrachtung der Auswertung bezüglich der Fagestellung der beiden Simulatoren hat sich herausgestellt, daß Desmoj dem Werkzeug Renew überlegen ist. Es ist mit Renew zwar möglich, ein Szenario schnell zu erstellen und zu simulieren, aber es gibt in Renew keine statistische Auswertung. Man kann deshalb, obwohl man einen sehr guten Simulator hat, keine Aussagen bezüglich der Fragestellung machen. Der Simulator, der mit Desmoj erstellt wurde, ist im Gegensatz dazu in der Lage, solche Aussagen zu treffen, denn Desmoj verfügt über eine aussagekräftige statistische Auswertung. Zusätzlich verfügt Desmoj über eine recht große Anzahl an verschiedenen Warteschlangen und Zufallszahlenstöme, die für die Simulation sehr nützlich sind.

Es wäre also optimal, wenn man ein Simulationswerkzeug eingesetzt werden könnte, das einfach zu bedienen ist und eine umfangreiche statistische Auswertung besitzt. Es gibt eine ganze Reihe unterschiedlicher Simulationswerkzeuge, sowohl im Bereich der Petrinetze als auch im Bereich der Frameworks, die auf diese Eigenschaften hin getestet werden müßten. Dazu wäre es notwendig, diese Werkzeuge zu evaluieren und in Klassen bezüglich ihrer Fähigkeiten einzuteilen, um dann Aussagen darüber machen zu können, welche Werkzeuge für den Einsatz geeignet sind und welche nicht.

Literaturverzeichnis

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

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

[Bur97] Rainer Burkhardt. *UML - Unified Modelling Language: objektorientierte Modellierung für die Praxis.* Addison-Wesley, 1997.

[FS98] Martin Fowler and Kendall Scott. *UML - konzentriert: die Standardobjektmodellierungssprache anwenden.* Addison-Wesley, 1998.

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

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

[Jen92] K. Jensen. *Coloured Petri nets, Basic Methods, Analysis Methods and Practical Use*, volume 1 of *EATCS monographs on theoretical computer science.* Springer-Verlag, 1992.

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

[KW98] Olaf Kummer and Frank Wienberg. *Reference net workshop (Renew).* Universität Hamburg, http://www.renew.de, 1998.

[Nie77] G. Niemeyer. *Kybernetische Systeme und Modelltheorie, System Dynamics.* Franz Vahlen, 1977.

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

[Pag00] B. Page. *Objektorientierte Simulation in Java.* Libri, 2000.

[PCL00] B. Page, S. Claassen, and T. Lechler. *The desmoj Homepage.* Universität Hamburg, http://www.desmoj.de, 2000.

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

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The unified modeling language reference manual: The definitive reference to the UML from the original designers.* Addison-Wesley object technology series. Addison-Wesley, Reading, Mass., 1999.

[SH95] Otto Spaniol and Simon Hoff. *Ereignisorientierte Simulation: Konzepte und Systemrealisierung.* Internationale Thompson Pupl., 1995.

Abbildungsverzeichnis

Anhang A

Quelltexte

A.1 Klasse Terminal

```
package terminal;

import desmoj.*;
import desmoj.dist.*;

/**  This class is the "main" class of the terminal model.
   * @author: Frauke Strümpel
  */
public class Terminal extends Model
{
        /** Random stream used to draw an arrival time for the next truck.       */
        private desmoj.dist.RealDistExponential truckArrivalTime;

        /** A waiting queue object is used to represent the parking area for
         * the trucks in front of the terminal.      */
        protected BlockingQueue truckTerminalQueue;

        /**A waiting queue object is used to represent the interchange.        */
        protected BlockingQueue ICQueue;

        /** A waiting queue object is used to represent the parking area for
         * the trucks in front of the Holdingarea.        */
        public BlockingQueue truckHOQueue;

        /** Servied trucks will go in there. */
        public desmoj.ProcessQueue servicedTrucksQueue;

        //Anzahl der VCs im Yard North
        protected int VC_YardNorth;

        //Anzahl der VCs im Yard West
        protected int VC_YardWest;

        //Anzahl der VCs im Yard South
        protected int VC_YardSouth;

    /**
      * A waiting queue object is used
      * to represent the free lanes in the holdingarea.
      */
    protected desmoj.Queue lanesQueue;

        protected desmoj.Bin lanes;
        /**
        Anzahl der Spuren
        */
        public int noOfLanes;

        /**
         Die Kontrolle des Terminals
```

```
        */
        public Control control;


/**
 * terminal_model constructor.
 *
 * @param owner desmoj.Model
 * @param modelname java.lang.String
 * @param showInReport boolean
 * @param showInTrace boolean
 * @param noOfLanes
 * @param VC_YardNorth int
 * @param VC_YardWest int
 * @param VC_YardSouth int
 */
public Terminal(Model owner, String modelName,
                        boolean showInReport, boolean showInTrace,
                        int noOfLanes, int VC_YardNorth,
                        int VC_YardWest, int VC_YardSouth)
{
        super(owner, modelName, showInReport, showInTrace);
        this.noOfLanes  = noOfLanes;
        this.VC_YardNorth = VC_YardNorth;
        this.VC_YardWest = VC_YardWest;
        this.VC_YardSouth = VC_YardSouth;
}

/** Describes what the model does */
public String description() {
        return   "This model is a service station model located at a "+
         "container harbour. Conatinertrucks will arrive and "+
         "require the loading of a container. A Vancarrier (VC) is "+
         "on duty and will head off to find the required container "+
         "in the storage. He will then deliver the container to the "+
         "truck. The truck then leaves the area."+
         "In case the VC is busy, the truck waits "+
         "for his turn on the parking-lot."+
         "If the VC is idle, it waits on his own parking spot for the "+
         "truck to come. "+
        "Parameters: "+noOfLanes+" lanes in HO,"  +
        VC_YardNorth + " VCs in north yard," +
        VC_YardWest + " VCs in west yard, and " +
        VC_YardSouth +"VCs in south yard.";

}


public void doInitialSchedules() {
        VC vc;
        int i;
        YardControl yardControl;
        Lane lane;

        //create a truck spring
        TruckGenerator firstarrival
                     = new TruckGenerator(this,"TruckArrival", false);
        firstarrival.schedule(new SimTime(getTruckArrivalTime()));

        //create the  control
        control = new Control (this, "Control", false);
        control.activate(new SimTime(0.0));
        // create all VCs

        yardControl = new YardControl (this,"yardControlNorth",false);
        control.yardControlNorth = yardControl;
        yardControl.activate(new SimTime(0.0));
        for (i = 1; i <= VC_YardNorth; i++) {
                vc = new VC(this, "VC (north)", false, yardControl);
                vc.activate(new SimTime(0.0));
        }
```

```
        yardControl = new YardControl (this,"yardControlWest",false);
        control.yardControlWest = yardControl;
        yardControl.activate(new SimTime(0.0));
        for (i = 1; i <= VC_YardWest; i++) {
                vc = new VC(this, "VC (west)", false, yardControl);
                vc.activate(new SimTime(0.0));
        }

        yardControl = new YardControl (this,"yardControlSouth",false);
        control.yardControlSouth = yardControl;
        yardControl.activate(new SimTime(0.0));
        for (i = 1; i <= VC_YardSouth; i++) {
                vc = new VC(this, "VC (south)", false, yardControl);
                vc.activate(new SimTime(0.0));
        }

        // create all lanes
        for (i = 0; i < noOfLanes; i++) {
                lane = new Lane(this, "Lane", false, i);
                lanesQueue.insert(lane);
        }
}



/** Returns a sample out of the random stream used to measure
 *  the next truck arrival time.  */
public double getTruckArrivalTime() {
        return truckArrivalTime.sample();
}



public void init()
{
        //initalizing the truckArrivalTimeStream
        //mean time in minutes between arrival of trucks
        truckArrivalTime= new
            RealDistExponential(this, "TruckArrivalTimeStream", 5.0, true, false);

        truckTerminalQueue = new
             BlockingQueue(this, "Trucks outside the terminal",true, false);

        ICQueue = new BlockingQueue(this, "Interchange", true, false);

        truckHOQueue = new
               BlockingQueue(this, "Trucks in front of the HO",true, false);

        lanes = new Bin(this, "Bin of free lanes", noOfLanes, true, false);
        lanesQueue = new Queue(this, "Empty lanes",true, false);

        servicedTrucksQueue = new
             ProcessQueue(this, "Serviced trucks",true, false);
}


/**
 * Starts the application.
 * @param args  : is an array of command-line arguments
 */
public static void main(java.lang.String[] args) {
    for(int noOfVC = 1; noOfVC < 2; noOfVC++) {
          Experiment terminal_Experiment = new Experiment("ftm"+noOfVC);
          Terminal terminal_Model = new Terminal(null, "Frauke's Terminal Model",
                         true, true, 6, noOfVC, noOfVC, noOfVC);

          terminal_Model.connectToExperiment(terminal_Experiment);
          terminal_Experiment.tracePeriod( new SimTime(0.0), new SimTime(90) );
          terminal_Experiment.debugPeriod( new SimTime(0.0), new SimTime(90) );
          terminal_Experiment.stop(new SimTime(1000.0));
```

```
          terminal_Experiment.start();
    // --> now the simulation is running until it reaches its ending criteria
    // ...
    // <-- after reaching ending criteria, the main thread returns here
          terminal_Experiment.report();
          // stop all threads still alive and close all output files
          terminal_Experiment.finish();
        }
}
}
```

A.2 Klasse Control

```
package terminal;

import desmoj.*;

/**
Control manages the aariving VCs and the arriving trucks in the HO.
Truck request are deligated to Yard controlers.
*/
public class Control extends SimProcess {
        private Terminal myModel;

       // VCs arriving at HO register here:
    protected desmoj.ProcessQueue HOVCServiceQueue;

       // Trucks arriving at the HO register here:
    protected desmoj.ProcessQueue HOTruckServiceQueue;

        /** VCs waiting for the truck wait here */
        private desmoj.ProcessQueue waitingVCHOQueue;

        // array for trucks in service
        Truck waitingTruck[];

        // array for VCs, that want to load a truck
        VC loadingVC[];

        // array for VCs, that want to unload a truck
        VC unloadingVC[];

        public YardControl yardControlSouth;
        public YardControl yardControlWest;
        public YardControl yardControlNorth;

public Control(Model owner, String name, boolean showInTrace) {
        super(owner, name, showInTrace);

        myModel = (Terminal) owner;
        int noOfLanes = myModel.noOfLanes;

        //initalizing the HOVCserviceQueue
        HOVCServiceQueue = new
            ProcessQueue(myModel, "VC arriving in the HO", true, false);

        //initalizing the HOTruckserviceQueue
        HOTruckServiceQueue = new  ProcessQueue(
             myModel, "Service-requests of trucks in the HO", true, false);

        waitingVCHOQueue = new
            ProcessQueue(myModel, "VC waiting for LKW", true, false);

        waitingTruck = new Truck[noOfLanes];
        loadingVC = new VC[noOfLanes];
        unloadingVC = new VC[noOfLanes];
}
```

```
public void tellPosition(Truck truck, Lane lane) {
        // der Truck parkt jetzt in seiner Spur in der HO:
        waitingTruck[lane.no] = truck;

        // Suche nach seinem VC:
        VC vc = (VC) waitingVCHOQueue.first();  // first() == null if q is empty
        while(vc != null) {
                // will dieser VC den truck bedienen?
                if(vc.getTruck() == truck) {
                        // Jetzt kann der VC in die Spur des LKW fahren
                        if(vc.wantsToLoad()) {
                                loadingVC[lane.no] = vc;
                        } else {
                                unloadingVC[lane.no] = vc;
                        }
                        //  weiter suchen - es kann ja noch einen 2. VC geben.
                }
                // gehe zum nachfolgende VC:
                vc = (VC) waitingVCHOQueue.succ(vc);
               // succ(vc) == null if no successor
        }
}


public void lifeCycle()
{
        while(true) {
                this.passivate();

                // Back alive
                // An der HO ist jemand angekommen
                // Wer hat uns aktiviert: der LKW oder der VC?
                if(!HOTruckServiceQueue.isEmpty()) { // OK: ein LKW
                        handleArrivedTruck();
                }
                else { // OK: VC hat uns geweckt
                        handleArrivedVC();
                }
        }
}


private void handleArrivedTruck() {
        VC vc;
        Truck truck = (Truck) HOTruckServiceQueue.first();
        HOTruckServiceQueue.remove(truck);
        Lane lane = truck.getLane();


        if(truck.wantsToBeLoaded()) {
                // Ist sein VC schon da?
                if(loadingVC[lane.no] != null) {
                        // Ja
                        vc = loadingVC[lane.no];
                        loadingVC[lane.no] = null;
                        vc.activateAfter(this);
                        // Die sollen jetzt mal beladen
                } else {
                        // VC ist noch unterwegs, LKW bleibt deaktiviert
                        waitingTruck[lane.no] = truck;
                }
        } else { // LKW will entladen werden
                // Ist sein VC schon da?
                if(unloadingVC[lane.no] != null) { // Ja
                        vc = unloadingVC[lane.no];
                        unloadingVC[lane.no] = null;
                        vc.activateAfter(this);
                // Die sollen jetzt mal beladen
                } else {
                        // VC ist noch unterwegs, LKW bleibt deaktiviert
```

```
                        waitingTruck[lane.no] = truck;
                }
        }
}


private void handleArrivedVC()
{
        // VC hat sich passiviert und wartet auf unser OK
        VC vc = (VC) HOVCServiceQueue.first();
        HOVCServiceQueue.remove(vc);
        Truck truck = vc.getTruck();


        // erstmal vc in diese Queue eintragen
        waitingVCHOQueue.insert(vc);

        // Ist sein LKW schon da? Wenn ja, ist er angemeldet in einer Spur.
        int laneNo = 0;
        boolean truckFound = false;
        while(laneNo < myModel.noOfLanes && !truckFound) {
                truckFound = (waitingTruck[laneNo] == truck);
                if(!truckFound) {
                        laneNo = laneNo + 1;
                }
        }

        if(truckFound) { //LKW ist da
                waitingVCHOQueue.remove(vc);
                // VC kann in die entsprechende Spur fahren
                if(vc.wantsToLoad()) {
                        if(truck.wantsToBeLoaded()) {
                                vc.activateAfter(this);
                                // Die sollen jetzt mal beladen
                        } else {
                                // LKW will erst entladen werden
                                loadingVC[laneNo] = vc;
                        }
                }
                else { // VC will entladen, sollte immer OK sein.
                        vc.activateAfter(this);
                // Die sollen jetzt mal beladen
                }
        }
        else {
                // LKW ist noch unterwegs.
                // VC bleibt in der waitingVCHOQueue.
        }
}


/**
Der Kontrolle wird der LKW "truck".
übergeben, um seinen Auftrag zu bearbeiten.
*/
public void serviceTruckArrival(Truck truck)
{
  int truckInstruction = truck.getInstruction();

  switch(truckInstruction) {
  case Truck.loadNorth:
          yardControlNorth.handoverInstruction(truck, VC.LOAD);
          break;
  case Truck.unloadNorth:
          yardControlWest.handoverInstruction(truck, VC.UNLOAD);
          break;
  case Truck.unloadNorthloadNorth:
          yardControlNorth.handoverInstruction(truck, VC.UNLOAD);
          yardControlNorth.handoverInstruction(truck, VC.LOAD);
          break;
  case Truck.loadWest:
```

```
          yardControlWest.handoverInstruction(truck, VC.LOAD);
          break;
  case Truck.unloadWest:
          yardControlWest.handoverInstruction(truck, VC.UNLOAD);
          break;
  case Truck.unloadWestloadWest:
          yardControlWest.handoverInstruction(truck, VC.UNLOAD);
          yardControlWest.handoverInstruction(truck, VC.LOAD);
          break;

  case Truck.loadSouth:
          yardControlSouth.handoverInstruction(truck, VC.LOAD);
          break;
  case Truck.unloadSouth:
          yardControlSouth.handoverInstruction(truck, VC.UNLOAD);
          break;
  case Truck.unloadSouthloadSouth:
          yardControlSouth.handoverInstruction(truck, VC.UNLOAD);
          yardControlSouth.handoverInstruction(truck, VC.LOAD);
          break;

  case Truck.unloadNorthloadWest:
          yardControlNorth.handoverInstruction(truck, VC.UNLOAD);
          yardControlWest.handoverInstruction(truck, VC.LOAD);
          break;
  case Truck.unloadNorthloadSouth:
          yardControlNorth.handoverInstruction(truck, VC.UNLOAD);
          yardControlSouth.handoverInstruction(truck, VC.LOAD);
          break;
  case Truck.unloadWestloadNorth:
          yardControlWest.handoverInstruction(truck, VC.UNLOAD);
          yardControlNorth.handoverInstruction(truck, VC.LOAD);
          break;
  case Truck.unloadWestloadSouth:
          yardControlWest.handoverInstruction(truck, VC.UNLOAD);
          yardControlSouth.handoverInstruction(truck, VC.LOAD);
          break;
  case Truck.unloadSouthloadNorth:
          yardControlSouth.handoverInstruction(truck, VC.UNLOAD);
          yardControlNorth.handoverInstruction(truck, VC.LOAD);
          break;
  case Truck.unloadSouthloadWest:
          yardControlSouth.handoverInstruction(truck, VC.UNLOAD);
          yardControlWest.handoverInstruction(truck, VC.LOAD);
          break;
  }
}

}
```

A.3 Klasse Truck

```
package terminal;

import desmoj.*;
import desmoj.dist.*;

public class Truck extends SimProcess {

        private Terminal myModel;
        // Random stream used to draw a driving time to the IC.
        private desmoj.dist.RealDistUniform drivingTimeICQueue;

        // Random stream: driving time from the IC to the truckHoQueue.
        private desmoj.dist.RealDistUniform drivingTimeICHOQueue;

        // Random stream: driving time from HO-queue to the Holdingarea.
        private desmoj.dist.RealDistUniform drivingTimeHOQueueHO;

        /**
```

```
        * Random stream used to draw the truck instrucktion.
        */
        private desmoj.dist.IntDistUniform instructionTypDist;

        private int instruction;

        // zeigt an, ob der 1. Teilauftrag schon bearbeitet wurde
        private boolean  firstTaskCompleted = false;

        // our lane in HO
        private Lane lane;


    public static final int  loadNorth = 1;
    public static final int  unloadNorth = 2;
    public static final int  unloadNorthloadNorth = 3;
    public static final int  loadWest = 4;
    public static final int  unloadWest = 5;
    public static final int  unloadWestloadWest = 6;
    public static final int  loadSouth = 7;
    public static final int  unloadSouth = 8;
    public static final int  unloadSouthloadSouth = 9;
    public static final int  unloadNorthloadWest = 10;
    public static final int  unloadNorthloadSouth = 11;
    public static final int  unloadWestloadNorth = 12;
    public static final int  unloadWestloadSouth = 13;
    public static final int  unloadSouthloadNorth = 14;
    public static final int  unloadSouthloadWest = 15;


/**
 * Constructor of the truck process
 */
public Truck(Model owner, String name, boolean showInTrace) {

        super(owner, name, showInTrace);

        myModel = (Terminal) owner;

        //initalizing the ICQueueDrivingTimeStream
        drivingTimeICQueue = new RealDistUniform(
               myModel, "ICQueueDrivingTimeStream", 1.0, 4.0, false, false);

        //initalizing the truckHoQueueDrivingTimeStream
        drivingTimeICHOQueue = new RealDistUniform(
             myModel, "TruckHoQueueDrivingTimeStream", 2.0, 5.0, false, false);

        //initalizing the truckHoDrivingTimeStream
        drivingTimeHOQueueHO = new RealDistUniform(
             myModel, "TruckHoDrivingTimeStream", 1.0, 3.0, false, false);

        //initalizing the truck instruction
        instructionTypDist = new IntDistUniform(
               myModel, "instructionTyp",1, 15, false, false);

        // initialsing the truck instruction
        instruction = (int) instructionTypDist.sample();
}

/**         Returns the instruction of the truck.*/
public int getInstruction() {
        return instruction;
}

/**         Returns the lane of the truck.*/
public Lane getLane() {
        return lane;
}

/**        Returns driving time from IC to the HO queue */
public double getDrivingTimeICQueue()  {
```

```
        return drivingTimeICQueue.sample();
}

/**        Returns driving time from the HO queue to the HO*/
public double getDrivingTimeICHOQueue() {
        return drivingTimeICHOQueue.sample();
}

/**        Returns driving time from outside to the IC*/
public double getDrivingTimeHOQueueHO() {
        return drivingTimeHOQueueHO.sample();
}

public boolean wantsToBeLoaded()
{
        boolean result = false;

        switch(instruction) {
        case loadNorth:
        case loadWest:
        case loadSouth:
                result = true;
                break;
        case unloadNorth:
        case unloadWest:
        case unloadSouth:
                 result = false;
                 break;
        case unloadNorthloadNorth:
        case unloadWestloadWest:
        case unloadSouthloadSouth:
        case unloadNorthloadWest:
        case unloadNorthloadSouth:
        case unloadWestloadNorth:
        case unloadWestloadSouth:
        case unloadSouthloadNorth:
        case unloadSouthloadWest:
             result = firstTaskCompleted;
             break;
        }
        return result;
}



/**
 * This lifeCycle() describes what the truck does when it
 * becomes activated by DESMO-J
 */
public void lifeCycle()
{
        // LKW-Zustand: "LKW initialisiert"
        // Ankunft am Terminal:
        myModel.truckTerminalQueue.insert(this);

        goToIC();
        // LKW-Zustand ist nun: "LKW im IC"

        // und übergebe Kontrolle Auftragsbeschreibung
        myModel.control.serviceTruckArrival(this);

        goToHOQueue();

        goToHO();
        // LKW-Zustand ist nun: "LKW in der HO"

         // teile Kontrolle Spur mit
        myModel.control.tellPosition(this, lane);

        loadUnload();
        // FERTIG!
```

```
        myModel.servicedTrucksQueue.insert(this);
}


private void driveTo(BlockingQueue fromQ,
                  BlockingQueue toQ, SimTime drivingTime)
{
        toQ.deliver(1);

        //  toQ must be free now.
        fromQ.remove(this);
        fromQ.store(1);

        // fahre ...
        hold(drivingTime);

        // da
        toQ.insert(this);
}


private void goToIC()
{
        driveTo(myModel.truckTerminalQueue, myModel.ICQueue,
                     new SimTime(getDrivingTimeICQueue()));
}


private void goToHOQueue()
{
        driveTo(myModel.ICQueue, myModel.truckHOQueue,
                     new SimTime(getDrivingTimeICHOQueue()));
}


private void goToHO()
{
        // Testen der Transition "der LKW fährt in die HO"
        myModel.lanes.deliver(1);
        // Spur steht jetzt zur Verfügung.
        lane = (Lane) myModel.lanesQueue.first();
        myModel.lanesQueue.remove(lane);

        // fahre in die HO:
        myModel.truckHOQueue.remove(this);
        myModel.truckHOQueue.store(1);

        // fahre ...
        hold(new SimTime(getDrivingTimeHOQueueHO()));
}

private void loadUnload()
{
        switch(instruction) {   // Lass Dich bedienen...
        case loadNorth:
        case loadWest:
        case loadSouth:
                // uns bei der Kontrolle vormerken
                myModel.control.HOTruckServiceQueue.insert(this);
                // warte auf das Beladen:
                myModel.control.activateAfter(this);
                // Kontrolle synchronisiert
                this.passivate(); // Auf VC warten
                // Back alive!
                firstTaskCompleted = true;
                // OK, VC ist fertig.
        break;
        case unloadNorth:
        case unloadWest:
        case unloadSouth:
```

```
                // uns bei der Kontrolle vormerken
                myModel.control.HOTruckServiceQueue.insert(this);
                // warte auf das Entladen:
                myModel.control.activateAfter(this);
                this.passivate();// Auf VC warten
                // Back alive!
                firstTaskCompleted = true;
                // OK, VC ist fertig.
        break;
        case unloadNorthloadNorth:
        case unloadWestloadWest:
        case unloadSouthloadSouth:
        case unloadNorthloadWest:
        case unloadNorthloadSouth:
        case unloadWestloadNorth:
        case unloadWestloadSouth:
        case unloadSouthloadNorth:
        case unloadSouthloadWest:
                // uns bei der Kontrolle vormerken
                myModel.control.HOTruckServiceQueue.insert(this);
                // warte auf das Entladen:
                myModel.control.activateAfter(this);
                this.passivate();       // Auf VC warten
                // Back alive.

                firstTaskCompleted = true;

                // warte auf das Beladen:
                // uns bei der Kontrolle vormerken
                myModel.control.HOTruckServiceQueue.insert(this);
                myModel.control.activateAfter(this);
                this.passivate();// Auf VC warten
                // OK, VC ist fertig.
        break;
        }

        // Spur wieder freigeben
        myModel.lanesQueue.insert(lane);
        myModel.lanes.store(1);
}


}
```

A.4 Klasse VC

```
package terminal;

import desmoj.*;
import desmoj.dist.*;

public class VC extends SimProcess
{
        private Terminal myModel;

        //Random stream: service time to load or unload the truck.
        private desmoj.dist.RealDistUniform vcServiceTime;

       // Random stream: driving time for a VC from the Yard to the HO or back.
        private desmoj.dist.RealDistUniform vcDrivingTime;

        // Random stream:  time for loading or unloading a container.
        private desmoj.dist.RealDistUniform vcContainerTime;

        // A waiting queue object is used to represent die auftraege eines VCs
        public desmoj.Queue instructionsQueue;

        /**
         dies ist mein zu bearbeitender Auftrag
        */
```

```
        private VCInstruction instruction;

        /** wo bin ich? */
        private boolean isInHO;

        /** ist fuer mich zuständig */
        private YardControl myYardControl;

        public static final int LOAD = 0;
        public static final int UNLOAD = 1;



/**
 * This method constructs a new VC
 *
 * @param owner desmoj.Model     the associated model
 * @param name java.lang.String  of the VC
 * @param yard int
 * @param showInTrace boolean    show in trace file or not show in trace
 */
public VC(Model owner, String name,
                boolean showInTrace, YardControl yardControl) {
        super(owner, name, showInTrace);

        myModel = (Terminal) owner;
        this.myYardControl = yardControl;

        //initalizing the serviceTimeStream
        vcServiceTime= new RealDistUniform(
                  myModel, "vcServiceTimeStream", 5.0, 10.0, false, false);

        //initalizing the vcDrivingTimeStream
        vcDrivingTime= new RealDistUniform(
                 myModel, "VCDrivingTimeStream", 3.0, 8.0, false, false);

        //initalizing the vcContainerTimeStream
        vcContainerTime= new RealDistUniform(
                 myModel, "VCContainerTimeStream", 3.0, 8.0, false, false);

    //initalizing the auftragsQueue
        instructionsQueue= new Queue(
                myModel, "VC instructions queue", false, false);
}


public Truck getTruck() {
        return instruction.truck;
}

public boolean wantsToLoad() {
        return (instruction.typ == VC.LOAD);
}

public boolean isInHO() {
        return isInHO;
}

public void setHO(boolean isInHO) {
        this.isInHO = isInHO;
}


public void lifeCycle()
{
        while(true) {
                // Queue enthält 0 oder 1 Auftrag
                if(instructionsQueue.isEmpty()) { //leer
                        // Sich in seinem Yard als unbeschäftigt melden
                        myYardControl.idleVCQueue.insert(this);
                        myYardControl.idleVCBin.store(1);
```

```
                        // Nix zu tun, Yard-Kontrolle soll uns wecken.
                        passivate();
                }

                // Yard-Kontrolle hat uns einen Auftrag gegeben.
                instruction = (VCInstruction) instructionsQueue.first();
                instructionsQueue.remove(instruction);

                // dies ist mein zu bearbeitender LKW
                switch(instruction.typ) {
                case VC.UNLOAD:
                        unloadCycle(instruction.truck); break;
                case VC.LOAD:
                        loadCycle(instruction.truck); break;
                default: break;
                }
        }
}

public void unloadCycle(Truck truck)
{
        if(!isInHO()) {
                gotoHO();
        }

        // bei der Kontrolle vormerken
        myModel.control.HOVCServiceQueue.insert(this);
        // und sie wecken
        myModel.control.activateAfter(this);
        // Lass die Kontrolle mal machen
        this.passivate();

        // OK, LKW ist jetzt auch da und will auch entladen werden:
        unloadTruck();
        truck.activateAfter(this);

        gotoYard();
        unloadContainer();
}


public void loadCycle(Truck truck) {
        if(isInHO()) {
                gotoYard();
        }
        loadContainer();
        gotoHO();

        myModel.control.HOVCServiceQueue.insert(this);
        // und sie wecken
        myModel.control.activateAfter(this);
        this.passivate();

        // OK, LKW ist jetzt auch da und will auch beladen werden:
        loadTruck();
        truck.activateAfter(this);
}


private double getServiceTime() {
        return vcServiceTime.sample();
}

private double getDrivingTime() {
        return vcDrivingTime.sample();
}

private double getContainerTime() {
        return vcContainerTime.sample();
}
```

```
private void gotoYard() {
        hold(new SimTime(getDrivingTime()));
        setHO(false);
}

private void gotoHO() {
        hold(new SimTime(getDrivingTime()));
        setHO(true);
}

private void unloadContainer() {
        hold(new SimTime(getContainerTime()));
}

private void loadContainer() {
        hold(new SimTime(getContainerTime()));
}

private void unloadTruck() {
        hold(new SimTime(getServiceTime()));
}

private void loadTruck() {
        hold(new SimTime(getServiceTime()));
}

}
```

A.5 Klasse YardControl

```
package terminal;

import desmoj.*;

public class YardControl extends SimProcess {
        private Terminal myModel;

        private desmoj.Queue instructionsQueue;
        private desmoj.Bin instructionsBin;

        public desmoj.ProcessQueue idleVCQueue;
        public desmoj.Bin idleVCBin;

public YardControl(Model owner, String name, boolean showInTrace) {
        super(owner, name, showInTrace);
        myModel = (Terminal) owner;

        //initalizing the auftragsQueue
        instructionsQueue = new Queue(myModel, "YC instructionsQ",true, false);

        //initalizing the idleVCQueue
        idleVCQueue = new ProcessQueue(myModel, "Idle VC Queue",true, false);

        //initalizing the instructionsBin
        instructionsBin = new Bin(myModel, "instruction Bin", 0, true, false);

        idleVCBin = new Bin(myModel, "Idle VC bin", 0, true, false);
}

public void handoverInstruction(Truck truck, int instructionNo)
{
        VCInstruction instruction = new VCInstruction(
              myModel, "VC instruction", false, truck, instructionNo);
        instructionsQueue.insert(instruction);
        instructionsBin.store(1);
}

public void lifeCycle() {
        VC vc;
```

```
        VCInstruction instruction;
        boolean ok;

        while(true) {
                instructionsBin.deliver(1); // Blockiere, bis Auftrag da

                instruction = (VCInstruction) instructionsQueue.first();
                instructionsQueue.remove(instruction);

                // Vergebe Auftrag an unbeschäftigten VC
                idleVCBin.deliver(1); // Warte, bis einer da
                vc = (VC) idleVCQueue.first();
                idleVCQueue.remove(vc);

                ok = vc.instructionsQueue.insert(instruction);
                vc.activateAfter(this);
        }
}

}
```

A.6 Klasse BlockingQueue

```
package terminal;

import desmoj.*;

public class BlockingQueue extends ProcessQueue {
        private Bin freePlaces;
        private boolean bounded;

        public BlockingQueue(Model owner, String modelName,
                                boolean showInReport, boolean showInTrace) {
                        super(owner, modelName, showInReport, showInTrace);
                bounded = false;
        }

        public BlockingQueue(Model owner, String modelName,
                    int capacity, boolean showInReport, boolean showInTrace) {
                super(owner, modelName, showInReport, showInTrace);
                freePlaces = new Bin(owner, "Number of free places",
                 capacity, true, false);
                bounded = true;
        }

        public void deliver(int i)
        {
                if(bounded) {freePlaces.deliver(i);} // Platz muss da sein
        }

        public void store(int i)
        {
                if(bounded) {freePlaces.store(i);} // Platz zurück
        }
}
```

A.7 Klasse Lane

```
package terminal;

import desmoj.*;

public class Lane extends Entity {
        public int no;

        public Lane(Model owner, String modelName,
                            boolean showInTrace, int laneNo) {
                super(owner, modelName, showInTrace);
                this.no = laneNo;
```

```
        }
}
```

A.8 Klasse TruckGenerator

```
package terminal;

import desmoj.*;

public class TruckGenerator extends ExternalEvent {

  private Terminal myModel;

/**
 * TruckGenerator constructor comment.
 * @param arg1 desmoj.Model
 * @param arg2 java.lang.String
 * @param arg3 boolean
 */
public TruckGenerator(Model owner,
                String name, boolean showInReport) {
        super(owner, name, showInReport);

        myModel = (Terminal) owner;
}

public void eventRoutine() {
   Truck newTruck = new Truck(myModel, "Truck", true);

   //now let the newly created truck roll on the parking-lot
   newTruck.activateAfter(this);
   this.schedule(new SimTime(myModel.getTruckArrivalTime()));
}

}
```

A.9 Klasse VCInstruction

```
package terminal;

import desmoj.*;

public class VCInstruction extends Entity {
        public int typ;
        public Truck truck;

        public VCInstruction (Model owner, String modelName,
                        boolean showInTrace, Truck truck, int instructionTyp) {
                super(owner, modelName, showInTrace);
                this.typ = instructionTyp;
                this.truck = truck;
        }
}
```