Inhaltsverzeichnis

Verwendete Begriffe und Abkürzungen

BA:
Bedienanlage. Sie beinhaltet eine Anzahl von Kanälen zur Bedienung von Forderungen.

Bedienknoten:
Logische Zusammenfassung von Warteraum und Bedienanlage. Als Synonym wird "Knoten" verwendet.

Bedienungsnetz:
Zu einem Bedienungsnetz gehören mindestens zwei Bedienknoten. Als Synonym existiert der Begriff Warteschlangennetz.

Bedienungssystem:
Ein Bedienungssystem (oder Bediensystem) kann Bedieneinrichtung, Warteraum, Quelle und Senke enthalten. Als Synonym wird der Begriff Warteschlangensystem verwendet.

Bedienungstheorie:
Teilgebiet der Mathematik, welches Methoden zur analytischen Lösung von Bedienungssystemen und -netzen zur Verfügung stellt. Völlig gleichberechtigt wird der Begriff Warteschlangentheorie verwendet.

BNETD:
Bedienungs**net**ze der TU **D**resden

CUA:
Common **U**ser **A**ccess - Standard für Programmoberflächen.

Deadlock:
Aus engl. "to reach deadlock": in eine Sackgasse geraten. Aus der Petri-Netz-Theorie entlehnter Begriff, der hier eine anormale Situation während des Simulationslaufes kennzeichnen soll. Synonyme sind die Begriffe Verklemmung, Kreisblockierung, unaufhebbare Blockierung.

Forderung:
Dynamische Elemente eines Bedienungsmodelles. Synonym werden die Begriffe Auftrag, Job und Werkstück verwendet.

MS-DOS:
Micro**s**oft **D**isc **O**perating **S**ystem.

SAA:
System **A**pplication **A**rchitecture.

WS:
Warteschlange, W. bilden sich in Warteräumen.

Warteraum:
Einrichtung mit einer bestimmten Kapazität zur Aufnahme einer Warteschlange.

Warteschlangennetz:
siehe Bedienungsnetz.

Warteschlangenstrategien:
W. kennzeichnen die Art und Weise der Entnahme von Forderungen aus einer Warteschlange. Wichtige Vertreter sind die Strategien FIFO (First In First Out) und LIFO (Last In First Out).

Warteschlangensystem:
siehe Bedienungssystem.

Warteschlangentheorie:
siehe Bedienungstheorie.

Verzeichnis der Abbildungen und Tabellen

ABBILDUNGEN:

TABELLEN

1. Einleitung

1.1. Das Projekt BNETD

Im Rahmen dieser Einleitung soll sowohl auf die "historische" Entwicklung des Programmsystems BNETD als auch auf die Stellung der Simulation innerhalb dieses Projektes eingegangen werden.
Das Projekt BNETD ("Bedienungsnetze der TU Dresden") wurde in den Jahren 1982/83 an der TU Dresden entwickelt. Diese erste Version lief unter dem Betriebssystem OS auf ESER-Rechnern, bestand aus einem Satz von Werkzeugen zur analytischen Lösung von Warteschlangenmodellen.
Ende der achtziger Jahre wurde als Nachfolger der ursprünglichen BNETD-Version das Programm DIMPES entwickelt. DIMPES, unter MS-DOS mit Turbo-Pascal programmiert, erreichte seinen geplanten Funktionsumfang nicht. Eine Simulationskomponente wurde nicht integriert, weiterhin bestanden erhebliche Mängel in Bezug auf Laufsicherheit und Nutzerfreundlichkeit.
Im Jahre 1993 wurde das Projekt BNETD wieder aufgenommen. Die zu entwickelnde neue Version war Gegenstand mehrerer Beleg- und Diplomarbeiten und hat ein Programmsystem mit kommerziellem Anspruch zur analytischen und simulativen Lösung von Bediensystemen als Ziel. Auf die wesentlichen Eigenschaften beschränkt, kann die zweite Version von BNETD wie folgt charakterisiert werden:

- Komplexes Programmpaket mit integrierter Analyse- und Simulationskomponente,
- Zeitgemäße und standardisierte pseudografische Nutzerschnittstelle,
- Grafische Modelleingabe, -anzeige und Ergebnisrepräsentation.

Gegenstand der hier vorgestellten Diplomarbeit ist die Entwicklung einer Klassenbibliothek zur simulativen Lösung von bedienungstheoretischen und verwandten Modellen, die Integration einer mit dieser Bibliothek erstellten speziellen Simulationskomponente in das Programmsystem BNETD sowie die Anpassung der BNETD-Oberfläche an die neu entstandenen Anforderungen.

Für die Unterstützung dieser Arbeit soll an dieser Stelle Herrn Prof. K. Irmscher, Herrn Dr. Cl.-J. Hamann und Frau Dipl.-Ing. D. Dietrich gedankt werden.

1.2. Stellung der Simulation innerhalb von BNETD

Hauptthema dieser Arbeit ist die Erstellung einer Komponente zur ereignisorientierten diskreten Simulation, welche die Lösung von Aufgabenstellungen aus dem Bereich der Bedienungstheorie ermöglichen soll. Hauptanwendungsgebiet des Programmsystems soll die Leistungsbewertung von Rechnersystemen sein. Gesteigerter Wert soll dabei auf das Kriterium der Universalität gelegt werden, das heißt, die zu entwickelnde Klassenhierarchie soll gleichfalls auf abgewandelte Modellstrukturen anwendbar sein - solche, deren Modellierung innerhalb der BNETD-Oberfläche nicht vorgesehen ist. Dabei soll jedoch keinesfalls die zweite wesentliche Anforderung außer acht gelassen werden - die Erstellung eines praktikablen Programmsystems BNETD, welches mittels der so integrierten Simulationskomponente die Bewertung von bedienungstheoretischen Modellen ermöglicht - insbesondere solchen, an denen die analytische Lösung teilweise scheitert, wie:

- Systeme mit begrenzten Warteschlangenkapazitäten,
- Systeme, in denen Blockierungen auftreten,
- Systeme, deren Verteilungen von Zwischenankunftszeit oder Bedienzeit nicht exponentiell sind,
- Systeme, deren Warteschlangenstrategien nicht FIFO sind,
- Instabile Systeme - die Stabilitätsbedingung ist für eine oder mehrere Bedienanlagen nicht erfüllt.

Bei der Implementation soll weiterhin eine möglichst portable Lösung geschaffen werden, Anmerkungen dazu werden im Kapitel 4 gemacht.
Den zweiten Schwerpunkt der vorliegenden Arbeit stellt die Anpassung an die bestehenden Programmkomponenten von BNETD dar. Hierzu ist es erforderlich, die Ergebnisschnittstelle der Oberfläche festzulegen, unter derer Nutzung die Ergebnisrepräsentation erfolgt - siehe dazu Kapitel 6.
Bevor im folgenden Kapitel auf Grundlagen der Simulation eingegangen wird, soll als Abschluß dieser Einleitung ein kurzer Überblick zu Warteschlangenmodellen gegeben werden.

1.3. Warteschlangenmodelle

Die Warteschlangentheorie (oder Bedienungstheorie) findet wesentliche Anwendungen in folgenden Bereichen:

- Bereich Wirtschaftswissenschaften: ökonomische Optimierungsprobleme (zum Beispiel Logistik und Lagerhaltung). Bedienungstheoretische Fragestellungen sind unter anderem Gegenstand des Operation Research;
- Bereich Informatik: Leistungsbewertung von Rechensystemen. Ein Rechensystem wird in einem Warteschlangennetz als System unabhängiger Knoten dargestellt. Je nach Art des Netzes existieren Verfahren zur analytischen Ermittlung relevanter Bewertungsgrößen. Für die analytische Lösung existieren eine Anzahl von Berechnungsalgorithmen. Erwähnt seien die Mittelwertanalyse für geschlossene Netze, das Verfahren von Marie, die Dekompositionsmethode von Kühn, die SCAT-Methode, der Bard-Schweitzer-Algorithmus, der Faltungsalgorithmus von Buzen, Chandy, Herzog und Woo, die RTP-Methode, die Summationsmethode und die Maximum-Entropie-Methode.

GRUNDBEGRIFFE

Die Grundstruktur eines Warteschlangenmodells (ein alternativer Begriff lautet Bedienungsmodell) kann wie folgt dargestellt werden (siehe Abbildung nächste Seite):

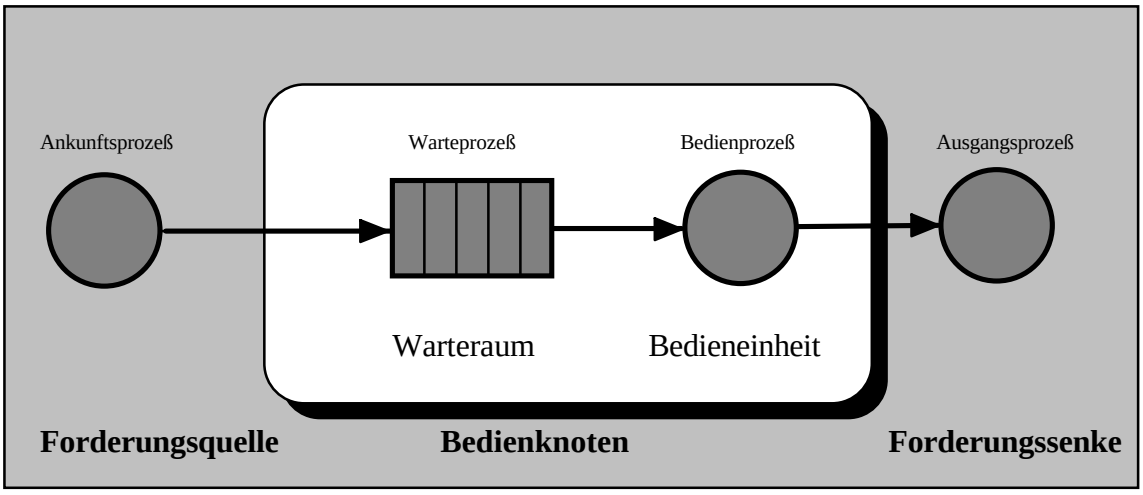

Abbildung 1-1: Struktur eines Warteschlangensystems

Zunächst sollen Eigenschaften eines einzelnen Knotens (Bedienknoten) beschrieben werden. Ein Bedienknoten besteht aus einem Warteraum mit einer bestimmten Kapazität und einer Bedieneinrichtung mit einer bestimmten Anzahl Bedienkanäle. Eine verbreitete Möglichkeit zur Charakterisierung eines Bedienknotens stellt die Notation von KENDALL dar:

KENDALLsche Notation: **A/B/k/r**

Dabei bedeuten die einzelnen Parameter:
- A: Beschreibung des Ankunftsstromes (Ankunftsprozeß),
- B: Beschreibung des Bedienstromes (Bedienprozeß),
- k: Anzahl der (identischen) Bedienkanäle der Bedieneinheit,
- r: Kapazität des Warteraumes.

Für die Beschreibung von Ankunfts- und Bedienstrom (Parameter A und B) ist folgende Symbolik üblich:
- M: Exponentialverteilung;
- E_i: Erlangverteilung der Ordnung i;
- H_i: Hyperexponentialverteilung der Ordnung i;
- C_i: Cox-Verteilung der Ordnung i;
- D: Deterministische Verteilung und
- G: Allgemeine Verteilung.

Für die Beschreibung der Ankunftsrate wird das Symbol λ genutzt, für die Beschreibung der Bedienrate das Symbol μ.

Die Klassifizierung nach k ergibt folgende Bezeichnungen:
-für k=1: Single Server;
-für k=∞: Infinite Server;
-für k>1: Multiple Server.

In Abhängigkeit von R bezeichnet man das durch den Bedienknoten gegebene System als:
-Verlustsystem für R=0;
-Wartesystem für R=∞;
-gemischtes Warte-Verlustsystem für alle R mit $0 < R < \infty$.

Oft wird der Parameter R nicht angegeben, in diesem Falle wird vom "Default-Wert" R=∞ ausgegangen.

Die Warteschlange kann nach verschiedenen Auswahlstrategien abgearbeitet werden:

- FCFS oder FIFO - Abarbeitung in der Reihenfolge der Ankunft;
- LCFS oder LIFO - Abarbeitung in umgekehrter Ankunftsreihenfolge;
- Round Robin (RR) - Bedienung nach festen Zeitscheiben. Dabei muß eine Forderung mitunter mehrmals wieder in die Warteschlangen eingereiht werden (Verdrängung);
- Processor Sharing (PS) - Round Robin mit unendlich kleiner Zeitscheibe;
- Random - zufällige Auswahl aus der Warteschlange;
- Priority - Auswahl nach statischen Prioritäten der Forderungen. Bei gleichen Prioritäten (eine Forderungsklasse) erfolgt die Auswahl nach FIFO.

Einige Warteschlangendisziplinen können preemptiv (verdrängend) arbeiten, das heißt, eine ankommende Forderung verdrängt die gerade in Bearbeitung befindliche Forderung. Preemptivität kann für die Warteschlangendisziplinen LIFO und Priority definiert sein. Die Warteschlangendisziplin kann der eigentlichen KENDALLschen Notation angefügt sein, um einen Knoten knapp und präzise zu beschreiben, ein Beispiel wäre die folgende Notation:
M/M/1 - FIFO.

LEISTUNGSGRÖßEN

Wie oben erwähnt, ist das Ziel der Behandlung eines solchen Warteschlangenmodells die Ermittlung von Leistungsgrößen, welche zu Aussagen über das Verhalten des realen Modells führen. Dabei spielen in der Regel nur Ergebnisse im stationären Zustand eine Rolle. Stationärer Zustand heißt, daß alle Einschwingvorgänge abgeklungen sein müssen. Das System befindet sich im statistischen Gleichgewicht. Notwendige Voraussetzung für ein statistisches Gleichgewicht ist, daß im gesamten Modell die Bedingung Auslastung < 1 erfüllt sein muß.
Wichtige Leistungsgrößen (hier ist gleichfalls der alternative Begriff Bewertungsgrößen anzutreffen) sind:

- Zustandswahrscheinlichkeiten p(k): Zustandswahrscheinlichkeiten (Wahrscheinlichkeit dafür, daß sich k Forderungen im Wartesystem befinden) sind die grundlegende Leistungsgröße, aus ihnen lassen sich alle anderen wichtigen Größen ableiten. Mit der Berechnung von Zustandswahrscheinlichkeiten ist ein wesentlicher Unterschied zwischen analytischer und simulativer Lösung von Warteschlangenmodellen gegeben, Zustandswahrscheinlichkeiten werden simulativ nicht ermittelt.
- Auslastung ρ: Die Auslastung gibt den Teil der Gesamtzeit an, in der die Bedieneinheit belegt ist. Die Auslastung ist wie folgt definiert:

$$\text{Auslastung } \rho = \frac{\lambda}{k\mu} \quad .$$

- Durchsatz D: Der Durchsatz gibt an, wieviele Forderungen pro Zeiteinheit fertig bedient werden. Bei erfüllter Stabilitätsbedingung ist dieser damit gleich der Anzahl der in das System hineingehenden Forderungen λ, in Verlustsystemen in Abhängigkeit von der Verlustwahrscheinlichkeit geringer.
- Wartezeit t_W: Die Wartezeit gibt an, welche Zeitspanne eine Forderung vor ihrer Bearbeitung in der Warteschlange verbringt. Eine solche Wartezeit ist eine Zufallsgröße, als Leistungsgrößen werden daher stets Mittelwerte angegeben. Es soll daher die Konvention getroffen werden, daß die Symbolik t_W (ebenso wie t_V, n_V, t_b, n_b, n_W - siehe unten) immer **einen Mittelwert** bezeichnen soll.
- Warteschlangenlänge n_W: Hiermit wird die mittlere Anzahl Aufträge in der Warteschlange bezeichnet.
- Bedienzeit t_b: Die mittlere Bedienzeit t_b ist normalerweise eine Eingangsgröße, eignet sich jedoch innerhalb einer simulativen Lösung eines Warteschlangenmodells gleichfalls als Leistungsgröße. Nichttriviale Abläufe (Verlustsysteme, Blockierungen) können damit leichter verifiziert werden.

- Anzahl genutzter Kanäle n_b: Die mittlere Anzahl genutzter Kanäle steht in unmittelbarem Zusammenhang mit der Auslastung eines Knotens: $n_b=\rho k$.
- Verweilzeit t_V: Als mittlere Verweilzeit (auch mittlere Antwortzeit) t_V wird die Zeitspanne bezeichnet, die eine Forderung im System zubringt. Für einen einfachen Knoten gilt: $t_V=t_W+t_b$. Bemerkung: Diese Berechnung gilt für Systeme, in denen keine Blockierungen auftreten. Eine Erweiterung dieser Berechnung wird im Rahmen dieser Arbeit später eingeführt.
- Anzahl der Aufträge im System n_V: Die mittlere Anzahl Aufträge im System oder auch Füllung bezeichnet die Anzahl Forderungen, die sich im Mittel im gesamten System befinden. Diese Größe ist nur für offene und gemischte Systeme interessant, für geschlossene Systeme bleibt die Füllung konstant. Eine einfache Berechnung für die Füllung eines Knotens ist mit $n_V=n_W+n_b$ gegeben. Hierbei gilt ebenfalls die obige Bemerkung.

Eine wichtige Rolle bei der analytischen Lösung eines Warteschlangensystems spielt das
Gesetz von Little:

$$n_v=t_v*\lambda \text{ und } n_w=t_w*\lambda$$

WEITERE BEGRIFFE

Eine weitere grundlegende Unterscheidung von Warteschlangenmodellen ist die Frage nach Abgeschlossenheit von der Außenwelt. Hier können offene, geschlossene und gemischte Systeme unterschieden werden.

Offene Warteschlangensysteme:

Ein Warteschlangensystem heißt offen, wenn (sämtliche) Forderungen von außen in das System eintreten können (Forderungsquelle) und es auch wieder verlassen (Forderungssenke) können. Die Füllung des Systems variiert.

Geschlossene Warteschlangensysteme:

Ein Warteschlangensystem heißt geschlossen, wenn die Anzahl der Forderungen im System konstant bleibt. Es besteht kein Kontakt zur "Außenwelt".

Gemischte Warteschlangensysteme:

Warteschlangensysteme können *mehrklassig* sein, das heißt, es existieren verschiedene Klassen von Forderungen. Verschiedene Forderungsklassen können sich voneinander durch Prioritäten, benötigten Ankunfts- und Bedienzeiten und verschiedenen "Wegen" durch das Modell unterscheiden. Möglich sind auch sogenannte Klassenwechsel - eine Forderung kann ihre Zugehörigkeit zu einer Forderungsklasse zugunsten einer anderen aufheben.

Ein gemischtes Warteschlangensystem ist dadurch gekennzeichnet, daß sowohl Forderungsklassen mit Kontakt zur Außenwelt als auch Forderungsklassen ohne Kontakt zur Außenwelt existieren.

Reale Systeme lassen sich in ihrer Komplexität nur selten als (einfache) Warteschlangensysteme darstellen - meist erfolgt die Modellierung als Warteschlangennetz. Ein Netz ist dadurch gekennzeichnet, daß zum System mindestens zwei Knoten gehören und zwischen den Knoten Übergänge stattfinden. Der Weg der Forderungen durch ein Warteschlangennetz wird durch Übergangswahrscheinlichkeiten beschrieben. Für Warteschlangennetze können weitere Leistungsgrößen ermittelt werden, die sich auf das gesamte System beziehen. Im einzelnen können dies Gesamtdurchsatz, Gesamtauslastung, Gesamtfüllung und Verweilzeit im Gesamtsystem sein. Weiterhin können durch Vergleich der Leistungsgrößen pro Knoten bestimmte Maximal-

und Minimalwerte erkannt werden, jenes führt zu Aussagen wie der Festlegung der "Schwachstelle" im Netz (Knoten mit maximaler Auslastung).

2. Grundlagen der Simulation

2.1. Begriff der Simulation

Der Begriff der Simulation ist allgemein dadurch gekennzeichnet, daß, anstelle der direkten Untersuchung eines realen Systems durch Messungen, ein Modell experimentell untersucht wird. Durch diese Vorgehensweise sind (mit bestimmten Einschränkungen) Rückschlüsse über das Verhalten des realen Systems möglich.

Der Erkenntnisgewinn durch Experimente kann im einzelnen bestehen aus:

- Messungen am Modell
- Beobachtungen am Modell
- Analytische Ergebnisfindung, zum Beispiel numerische Lösung von Differentialgleichungen bei zeitkontinuierlichen Modellen
- Statistikführung und Auswertung mittels wahrscheinlichkeitstheoretischer und statistischer Methoden im Falle zeitdiskreter Modelle

Dabei kommt der Modellbildung eine wichtige Aufgabe zu, nur eine möglichst realitätsnahe Modellierung kann in der Folge zuverlässige Erkenntnisse über das zu untersuchende System hervorbringen.

2.2. Modell und Modellbildungszyklus

"Modelle sind materielle oder immaterielle Systeme, die andere Systeme so darstellen, daß eine experimentelle Manipulation der abgebildeten Strukturen und Zustände möglich ist." [Page 91]

Die Bildung eines Modells aus einem realen System ist kein deterministischer Prozeß, die Bildung eines Modells muß je nach gewünschtem Zweck einen Kompromiß zwischen unumgänglicher Vereinfachung und exakter Abbildung des Systems beinhalten. Daraus ergibt sich, daß zu jeder realen Fragestellung sehr viele verschiedene Möglichkeiten existieren, ein Modell zu bilden, experimentell zu lösen und die gewonnenen Erkenntnisse zur Beantwortung der ursprünglichen Fragestellung einzusetzen:

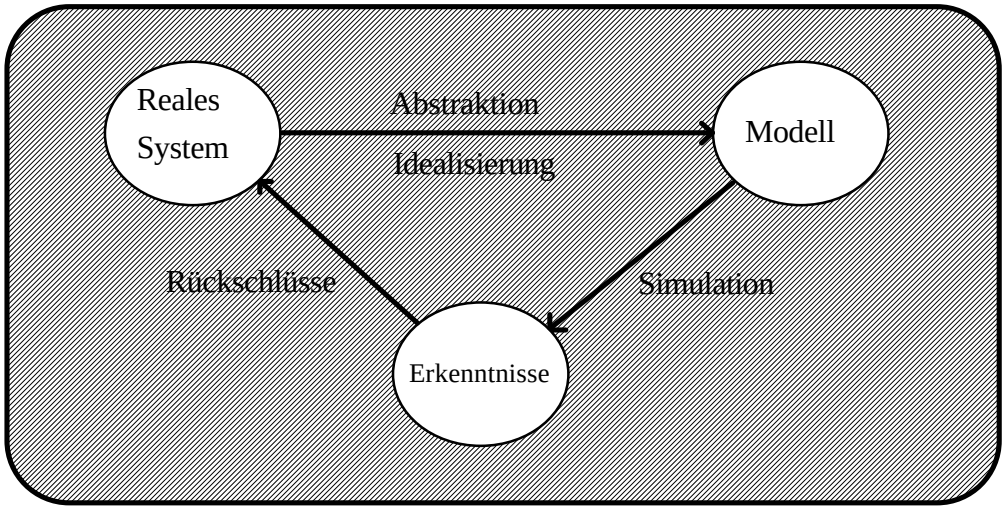

Abbildung 2-1: Modellbildungszyklus

Im folgenden wird eine Klassifikation möglicher Modelle erläutert, die sich an der Darstellung in [Page 91] orientiert:

Mögliche Kriterien für eine Klassifizierung von Modellen können sein:

- Art der Untersuchungsmethode (analytisch / simulativ)
- Abbildungsmedium
- Art der Zustandsübergänge
- Verwendungszweck

KLASSIFIZIERUNG NACH DER ART DER UNTERSUCHUNGSMETHODE

Unter der Voraussetzung, daß ein formales Modell mit der Absicht gebildet wurde, Erkenntnisse über ein reales System zu gewinnen, kann nach der Art der Lösung unterschieden werden zwischen:

- analytischen Modellen und
- simulativen Modellen.

Analytische Modelle erlauben eine exakte Lösung mittels Algorithmen, während Simulationsmodelle das Systemverhalten schrittweise nachzuvollziehen suchen.
Als wesentliche Vorteile simulativer Modelle und der damit forcierten simulativen Lösung sind zu nennen:

- Untersuchung von Systemen mit höherer Komplexität ist möglich, damit verbunden ist oft eine realitätsnäherere Modellierung und
- Simulation kann eine anschaulichere Art der Lösung bieten, da das Systemverhalten schrittweise nachvollzogen wird - Animation bzw. grafische Nachbildung des Verhaltens realer Systeme wird damit möglich (Bekannte Anwendungen hierzu sind die Simulation von physikalischen oder astronomischen Vorgängen).

Im Gegenzug existieren folgende Nachteile:

- Das Auffinden der exakten oder optimalen Lösung ist nicht gesichert und
- Computersimulation erfordert im Normalfall mehr Rechenzeit und Speicherplatz als die alternative Anwendung analytischer Methoden - dieser Nachteil sollte jedoch in der Perspektive an Gewicht verlieren.

Anzumerken sei, daß eine klare Trennung von formalen Modellen, die simulativ gelöst und solchen, die analytisch gelöst werden können, nicht existiert - ein Großteil der analytischen Modelle können auch simulativ gelöst werden. Gleichfalls gilt, daß die analytische Lösung von Modellen, die zur simulativen Lösung bestimmt sind, durchaus möglich sein kann. In der folgenden Abbildung ist dieser Sachverhalt skizziert:

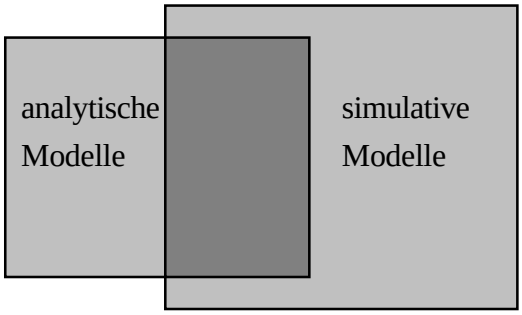

Abbildung 2-2:Mengendarstellung von analytischen und simulativen Modellen

KLASSIFIZIERUNG NACH ABBILDUNGSMEDIUM

Anhand folgender Beispiele soll die Klassifikation nach dem Abbildungsmedium erläutert werden (in Anlehnung an [Page 91]):

- materielles Modell: Häusermodelle für die Stadtplanung
- verbales Modell: umgangssprachliche Beschreibung eines realen Systems
- grafisch-deskriptives Modell: Skizzen, Block- und Flußdiagramme
- mathematisches Modell: Gleichungssystem, bedienungstheoretische Modellbeschreibungen
- grafisch-mathematisches Modell: Petri-Netz
- formales Modell: mathematisches oder grafisch-mathematisches Modell
- informales Modell: verbales oder grafisch-deskriptives Modell
- immatrielles Modell: informales oder formales Modell
- allgemeines Modell: materielles oder immatrielles Modell

Das Programmsystem BNETD bearbeitet in dieser Klassifikation mathematische Modelle.

KLASSIFIKATION NACH DER ART DER ZUSTANDSÜBERGÄNGE

Eine wesentliche Klassifikationsmethode ist die Klassifikation nach Art der Zustandsübergänge. Folgende Abbildung gibt einen Überblick über mögliche Modellklassen:

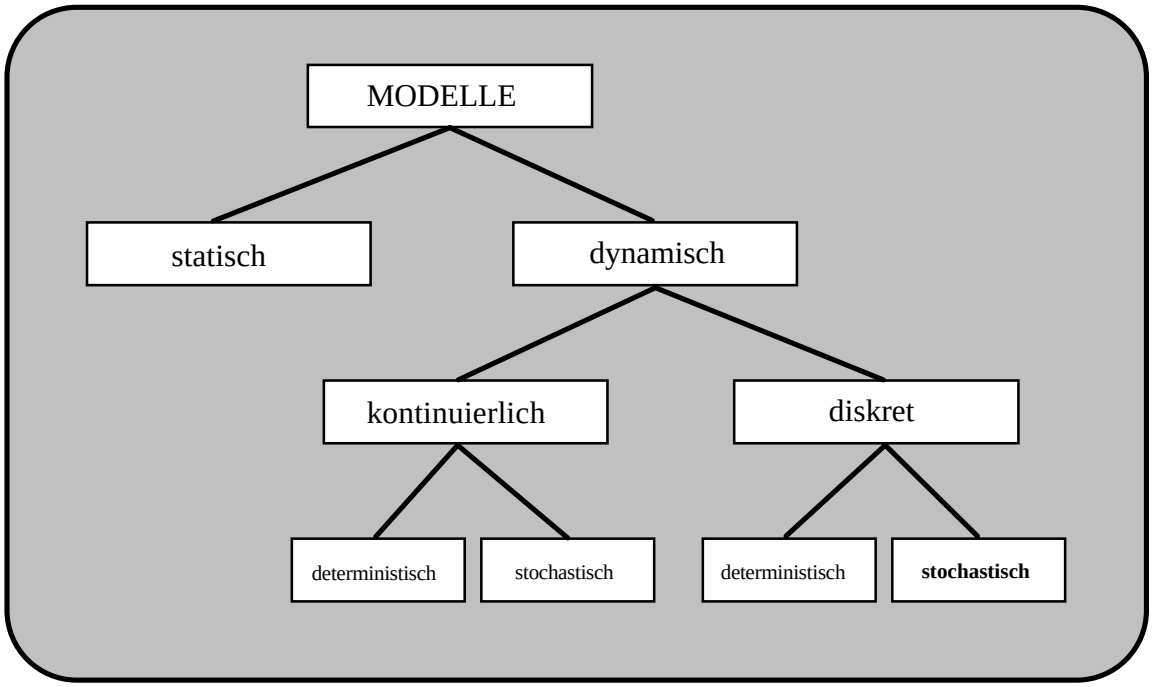

Abbildung 2-3: Klassifizierung nach Art der Zustandsübergänge nach [Page 91]

Unter statischen Modellen versteht man Modelle, deren Zustandsvariablen sich nicht ändern. Dagegen sind dynamische Modelle durch Zustandsänderungen in Abhängigkeit von der Zeit beschrieben.
Die Zustandsvariablen eines kontinuierlichen Modells ändern sich stetig, im Gegensatz dazu erfolgen Zustandsänderungen der Zustandsvariablen eines diskreten Modells zu bestimmten Zeitpunkten.
Modelle, deren Zustandsänderung eindeutig durch den vorhergehenden Zustand und einen bestimmten Eingangswert festgelegt ist, heißen deterministisch. Anderenfalls heißt das Modell stochastisch.

Gegenstand dieser Arbeit ist die simulative Behandlung von diskreten, stochastischen Modellen.

KLASSIFIKATION NACH VERWENDUNGSZWECK

Wie im Abschnitt 2.2. erwähnt, beinhaltet der Prozeß der Modellbildung eine subjektive Komponente. Bei der Modellbildung sollte stets der Verwendungszweck in Betracht gezogen werden, damit sind notwendige Abstraktionen und Idealisierungen auf den gewünschten Verwendungszweck abstimmbar.
[Page 91] unterscheidet folgende Modellklassen:

- Erklärungsmodelle
- Prognosemodelle
- Gestaltungsmodelle
- Optimierungsmodelle

Die Modellierung und Lösung in BNETD ist an keinen Verwendungszweck gebunden.

2.3. Potential und Probleme der simulativen Lösung

Das Programmsystem BNETD bietet für viele bedienungstheoretische Modelle alternative Lösungswege an, die simulative und die analytische Lösung. Daher soll an dieser Stelle eine Gegenüberstellung von Potential und möglichen Problemen der simulativen Lösung erfolgen (siehe auch Darstellung in [Page 91]).
In der folgenden Tabelle sind wesentliche Vor- und Nachteile zusammengefaßt:

Potential	**Probleme**
Erfassung der Systemkomplexität	Realitätsferne
Erhöhtes Systemverständnis	Gleichsetzung von Modell und Realsystem
Alternative zu Realexperimenten	Mangelnde Transparenz
Entscheidungshilfe	Datenmangel

Tabelle 2-1: Potential und Probleme des simulativen Ansatzes

Im weiteren sollen diese Potentiale und Probleme kurz dargestellt werden:

ERFASSUNG DER SYSTEMKOMPLEXITÄT

Die Simulation ermöglicht eine schrittweise Verfolgung (engl. "Tracing") der Zustandsänderungen ausgewählter Komponenten eines komplexen Modells. Durch diese separaten Betrachtungen ist es möglich, ein besseres Verständnis für das Verhalten eines Gesamtsystems zu erlangen. Insbesondere die simultane grafische Darstellung des Simulationsablaufes (Animation) kann das Verstehen der Abläufe in einem Modell drastisch erhöhen - setzt allerdings leistungsfähige Hardware voraus.

ERHÖHTES SYSTEMVERSTÄNDNIS

Durch den ersten Schritt, der zur simulativen Behandlung eines realen Systems nötig ist - der Bildung eines Simulationsmodelles - , wird der Modellentwickler zur Beschäftigung mit dem in das Modell einfließende Wissen gezwungen und kann dadurch zu einem Erkenntnisgewinn bezüglich des realen Systems kommen. Weiterhin gilt die oben getroffene Aussage die Animation betreffend auch hier.

ALTERNATIVE ZU EXPERIMENTEN AM REALEN SYSTEM

Aus verschiedenen Gründen können Experimente am realen System oft nicht durchgeführt werden - etwa technische, finanzielle oder ökologische Unmöglichkeit. Somit sind Fälle denkbar, in denen Computersimulation die einzige Möglichkeit darstellt, zu Erkenntnissen über das reale System zu gelangen.

ENTSCHEIDUNGSHILFE

Die aus der simulativen Lösung gewonnenen Erkenntnisse können als Entscheidungshilfe bei der Erstellung oder Veränderung eines realen Systems dienen. Sie erlauben die Abschätzung der Auswirkungen geplanter Änderungen am realen System und unterstützen die Strategiefindung für den Umgang mit realen Systemen.

REALITÄTSFERNE

Ein mögliches Problem bei der Modellbildung stellt eine zu starke Vereinfachung der Beziehungen des Realsystems dar - gleichfalls könnten komplexe Beziehungen eines realen Modells falsch interpretiert werden. Ein weiteres Problem der Modellbildung besteht darin, daß qualitative Einflüsse im realen System schwer zu quantifizieren sind, womit eine weitere Fehlerquelle entsteht.

GLEICHSETZUNG VON MODELL UND REALSYSTEM

Bei der Interpretation der durch Modellierung und Simulation gewonnenen Erkenntnisse besteht die Gefahr der Überbewertung der gewonnenen Fakten. Eine Objektivität der auf das Realsystem übertragenen Erkenntnisse kann nicht garantiert werden - sie sollten nur als Entscheidungshilfe verwendet werden.

MANGELNDE TRANSPARENZ

In der Regel kann nicht gewährleistet werden, daß alle Schritte, die zum Zustandekommen der Erkenntnisse und Folgerungen führten, transparent sind (und damit nachprüfbar).

DATENMANGEL

Wenn die schwere Beschaffbarkeit von Eingangsdaten durch Schätzungen bei der Modellbildung kompensiert wird, können in der Folge Verfälschungen auftreten, die den Wert der gewonnenen Erkenntnisse mindert.

2.4. Konzepte der Simulation

In diesem Abschnitt sollen Unterschiede zwischen den wichtigsten Konzepten der Simulation beschrieben werden.
Wie oben dargestellt, ist unter einem dynamischen Modell ein Modell zu verstehen, dessen Zustandsvariablen sich in zeitlicher Abhängigkeit befinden. Dynamische Modelle unterscheiden sich nun durch kontinuierliche oder diskrete Änderung des Systemzustandes als Funktion der Zeit. An dieser Stelle können die zwei Hauptkonzepte der Simulation voneinander getrennt werden:

- Zeitkontinuierliche Simulation (oder auch "dynamische Simulation") - eine Lösung von zeitkontinuierlichen Simulationsmodellen erfolgt über die numerische Integration der das Modell beschreibenden Differentialgleichungen. Es besteht ebenfalls die Möglichkeit, durch Diskretisierung eines kontinuierlichen Modelles ($\Rightarrow$ quasikontinuierliches Modell) eine Lösung durch zeitdiskrete Simulation zu erzielen. Ein Beispiel ist das Räuber-Beute-Modell.

- Zeitdiskrete Simulation - Vorgänge, die zeitdiskret ablaufen, eignen sich zum schrittweisen Nachvollziehen durch einen Digitalrechner. Sie entsprechen dem intuitiven gedanklichen Bild der Simulation. Erkenntnisse werden dabei aus einer während der Simulation geführten Statistik gewonnen. Auf verschiedene Modellierungsansätze innerhalb der zeitdiskreten Simulation wird im folgenden eingegangen.

ANSÄTZE INNERHALB DER ZEITDISKRETEN SIMULATION

Die zeitdiskrete Simulation läßt sich in mehrere Modellierungsstile unterteilen, die vor allem durch populäre Simulationssprachen wie *GPSS* und *Simula* geprägt wurden. In [Page 91] werden vier konzeptuelle Sichtweisen angegeben:

- ereignisorientiert
- transaktionsorientiert (GPSS)
- aktivitätsorientiert (Simula)
- prozeßorientiert oder vorgangsorientiert

Die beiden wichtigsten Konzepte sind der ereignisorientierte und der prozeßorientierte Ansatz. Auf transaktionsorientierten und aktivitätsorientierten Ansatz soll nicht weiter eingegangen werden, hier sei auf [Page 91] verwiesen.
Das ereignisorientierte Konzept ist dadurch gekennzeichnet, daß Zustandsänderungen im Modell durch Ereignisse ausgelöst werden. Diese geschehen dabei konzeptuell zeitverzugslos, da zeitlich ausgedehnte Vorgänge auf eine Folge von Ereignissen reduziert werden. Die Prozesse im Modell (zum Beispiel die Bearbeitung eines Jobs) werden dabei nur durch Anfangs- und Endezeitpunkt beschrieben. Die Steuerung des Ablaufes der Simulation erfolgt dabei mit Hilfe einer Ereignisliste, aus der das jeweils nächste Ereignis entnommen wird und so eine entsprechende Ereignisroutine ausgelöst werden kann. Die Simulationsuhr wird auf den Aktivierungszeitpunkt des gerade entnommenen Ereignisses vorgestellt. Die ereignisorientierte Sichtweise ist der populärste Ansatz zur Simulation zeitdiskreter Modelle, nicht zuletzt durch maßgeschneiderte Unterstützung durch objektorientierte Modellierungstechniken und Programmiersprachen.
Das prozeßorientierte Konzept faßt auf ein Objekt (zum Beispiel eine Bedienstation) bezogene Aktivitäten zu einem Prozeß zusammmen. Ein solcher Prozeß kann sich in einem aktiven oder passiven Zustand befinden. Ein Prozeß in seiner aktiven Phase hat dabei starke Ähnlichkeit mit in einer in Bearbeitung befindlichen Ereignisroutine, auch hier wird keine Simulationszeit verbraucht - im Sinn einer Nichtfortschreibung der Simulationszeit. Passive Phasen eines Prozesses verbrauchen hingegen Simulationszeit (Bearbeitung eines Jobs). Operationen über einem solchen Prozeß sind die Passivierung und die Aktivierung bzw. Reaktivierung. Diese Operationen können durch den Prozeß selbst, häufiger jedoch doch durch andere Prozesse erfolgen. Eine solche Operation kann als Ereignis aufgefaßt werden, dann funktioniert die Ablaufsteuerung ähnlich der Ablaufsteuerung des ereignisorientierten Konzepts mittels einer Ereignisliste. Ein wichtiger Unterschied zum ereignisorientierten Konzept ist, daß ein Prozeß nach seiner Reaktivierung nicht notwendigerweise von Beginn an abgearbeitet wird, sondern vielmehr von der Stelle ausgehend, an der der Prozeß deaktiviert wurde. Dieses Konzept macht die Abspeicherung des Prozeßstatus notwendig. Zusammenfassend kann gesagt werden, daß der prozeßorientierte Ansatz methodische Vorteile bei der Modellierung paralleler Vorgänge bietet, jedoch ist durch die komplexere Ablaufsteuerung gegenüber dem ereignisorientierten Ansatz mit Effizienzproblemen zu rechnen.
Abschließend sei die Hierarchie der verschiedenen Simulationskonzepte in der folgenden Abbildung skizziert:

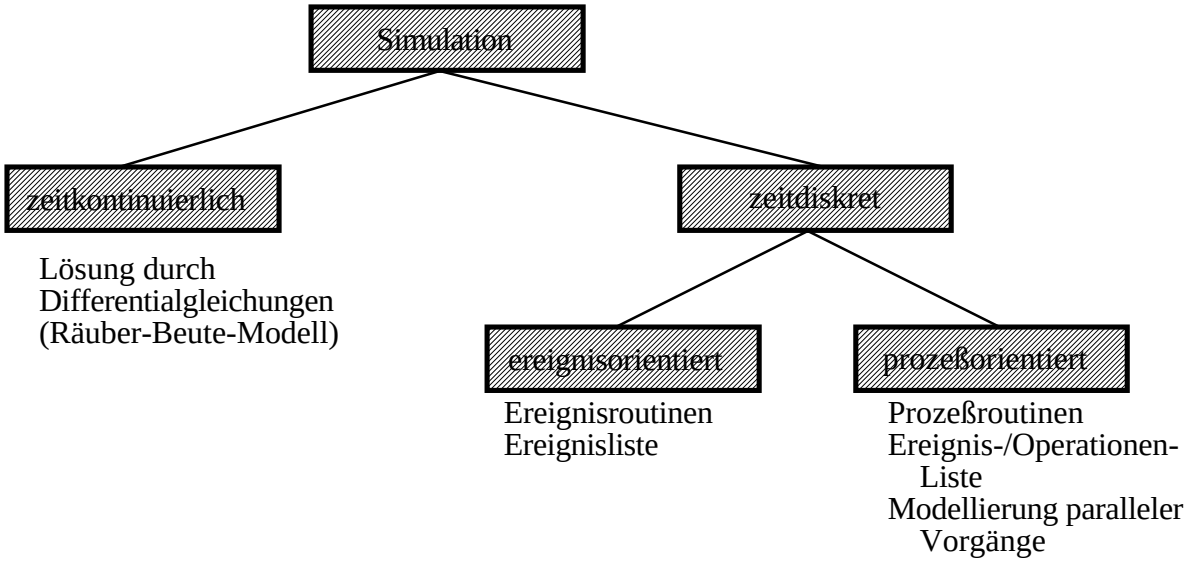

Abbildung 2-4: Hierarchie wesentlicher Simulationskonzepte

Die in dieser Arbeit entwickelte Simulationskomponente folgt dabei dem ereignisorientierten Modellierungsstil.
Im folgenden Abschnitt werden die Elemente eines ereignisorientierten zeitdiskreten Simulationsmodelles betrachtet.

2.5. Elemente ereignisorientierter zeitdiskreter Simulationsmodelle

Der ereignisorientierte Ansatz ermöglicht zunächst eine klare Trennung zwischen statischen und dynamischen Komponenten des Modells. Unter *statischen Komponenten* versteht man dabei diejenigen Elemente, die während des Simulationslaufes permanent vorhanden sind. Bei der Simulation eines bedienungstheoretischen Modells wären dies Bedienanlagen, Warteschlangen, Quellen, Senken, Verteilerobjekte, Ablaufsteuerung, Ereignisliste, Simulationsuhr und Statistikobjekte jeweils einschließlich zugehöriger Funktionen. Dagegen versteht man unter *dynamischen Komponenten* diejenigen Bestandteile, die während des Simulationslaufes erzeugt und vernichtet werden können. Hierzu zählen Ereignisse und Forderungen (Werkstücke, Aufträge, Kunden). Dynamische Komponenten verhalten sich bezüglich des Speicherbedarfs der Simulation kritisch - die Speicheranforderungen eines Simulationslaufes sind für das jeweilige Modell unterschiedlich und mitunter schwer vorhersehbar. Bei der Simulation eines offenen Warteschlangensystems, in welchem die Stabilitätsbedingung nicht erfüllt ist, wächst die Anzahl der im System befindlichen Forderungen, über größere Zeitabschnitte betrachtet, monoton an und führt bei entsprechender Simulationsdauer zur Erschöpfung des vorhandenen Speichers.
Ereignisse als grundlegende Bestandteile einer ereignisorientierten Simulation lösen Ereignisroutinen und damit Zustandsänderungen des Systems aus. Sie beinhalten stets einen Parameter, der ihren Eintritts- bzw. Bearbeitungszeitpunkt festlegt (Ereigniszeitpunkt). Ein Ereigniszeitpunkt kann dabei jeder beliebige diskrete Wert innerhalb des Wertebereichs des Digitalrechners sein. Beispielereignisse sind das Erzeugen einer Forderung, das Weiterleiten einer Forderung, das Ende der Bearbeitung einer Forderung. Ereignisse werden von Ereignisroutinen erzeugt. Folgende Unterscheidung von Ereignissen ist sinnvoll:

- Ereignisse, deren Ereigniszeitpunkt mit dem Zeitpunkt der Erzeugung des Ereignisses identisch ist und somit *Signale* darstellen. Ein Beispiel für ein Signal ist das Ereignis: "Aufhebung einer Blockierung".
- Ereignisse, deren Ereigniszeitpunkt mit dem Erzeugungszeitpunkt nicht identisch ist. Ein solches Ereignis ist beispielsweise das Ereignis "Ende der Bearbeitung einer Forderung".

Ereignisroutinen als Elemente ereignisorientierter Simulation legen die Reaktion auf bestimmte Ereignisse fest. In diesen wird die eigentliche "Arbeit" verrichtet - der Aufgabenbereich einer Ereignisroutine umfaßt:

- Erzeugung neuer Ereignisse und Veranlassung der Einfügung in die Ereignisliste
- Erzeugen, Löschen und Modifizieren von Forderungen
- Aktualisieren der Statistik

Die Steuerung der Simulation erfolgt durch die *Ablaufsteuerung (Simulatorkern)*. Diese arbeitet nach folgendem Prinzip, in einer Metasprache formuliert:

```
WHILE ( Simulationszeit nicht abgelaufen  UND kein Fehler aufgetreten )
    DO {
        Hole nächstes Ereignis aus der Ereignisliste ;
        Stelle Simulationsuhr auf den Eintrittszeitpunkt des Ereignisses ;
        Ermittle zugehörige Ereignisroutine und rufe diese auf ;
        Wenn  kein Fehler auftrat, lösche Ereignis aus der Ereignisliste;
      }
```

Die dabei verwendete *Ereignisliste* beinhaltet dabei alle noch zu bearbeitenden Ereignisse, die zum aktuellen Zeitpunkt bekannt sind. Es existiert eine Funktion, die in der Lage ist, das nächste zu verarbeitende Ereignis aus der Liste herauszusuchen, daher trägt die ereignisorientierte Simulation auch die Bezeichnung "next-event-simulation". Diese Funktion muß über einen Prioritätsmechanismus verfügen, der im Falle gleicher Ereigniszeitpunkte eine deterministische Entscheidung trifft, da eine Sequentialisierung einer solchen Parallelität notwendig ist.
Die *Simulationsuhr* gibt die aktuelle Modellzeit an. Sie wird nach Entnahme eines Ereignisses auf den Ereigniszeitpunkt gestellt. Die Zeit zwischen zwei aufeinanderfolgenden Ereignissen kann damit beliebig groß sein. Eine solche Simulationsuhr ist keine Uhr im wörtlichen Sinne, sondern lediglich eine Variable, die zur Ablaufsteuerung und Statistikführung in der ereignisorientierten Simulation benötigt wird.
Weiterhin sind Funktionen zur Initialisierung, Statistikführung und Deinitialisierung Bestandteile einer ereignisorientierten Simulation.

2.6. Zufallszahlen und Zufallszahlengeneratoren

2.6.1. Die Rolle der Zufallszahlen in der Simulation

Zufallszahlen sind für die zeitdiskrete Simulation grundlegend. Durch die Erzeugung von gleichverteilten Zufallszahlen im Intervall [0,1) und anschließender Transformation in weitere praktisch relevante Verteilungen (wie Exponentialverteilung und Normalverteilung) wird die Voraussetzung für die Modellierung realer Abläufe geschaffen.
Zufallszahlen können im Ergebnis eines zufälligen Versuchs erzeugt werden.

Beispiel:

Versuche mit einem idealen Würfel haben gleichverteilte diskrete Zufallszahlen x mit $x \in \{1,2,3,4,5,6\}$ *zur Folge. Die auf diese Weise erzeugten Zufallszahlen genügen dem Kriterium der Unabhängigkeit von den vorhergehenden Versuchen und sind somit echt zufällig.*

Echte Zufallszahlen können durch einen (zehnseitigen) Würfel, das Werfen einer Münze, die Nutzung von Zufallszahlentafeln oder Beobachten physikalischer Ereignisse erhalten werden. Aus Effizienzgründen ist für die Computersimulation eine algorithmische Erzeugung von Zufallszahlen nötig. Diese stößt allerdings auf ein Problem, da jeder geschlossene Algorithmus früher oder später die erzeugte Sequenz von Zufallszahlen wiederholen muß. Ein Umgehen

dieses Problemes ist durch Ausnutzen von zufälligen Ereignissen innerhalb des Rechners (Registerbelegungen, Systemzeit) prinzipiell möglich, praktisch ist es jedoch sehr schwierig, mit diesen Methoden zufriedenstellende Ergebnisse zu erzielen.

Im folgenden Abschnitt wird gezeigt, daß algorithmisch erzeugte "Zufallszahlen" - *Pseudozufallszahlen* - unter bestimmten Bedingungen ein ausreichender Ersatz für echte Zufallszahlen sein können.

2.6.2. Erzeugung von Pseudozufallszahlen

Im vorangegangenen Abschnitt wurde erwähnt, daß Pseudozufallszahlen eine periodische Folge bilden. Man kann jedoch davon ausgehen, daß bei ausreichend großer Periodenlänge Pseudozufallszahlen eine gute praktische Näherung für Zufallszahlen bilden. Eine solche Folge von Pseudozufallszahlen muß folgende Bedingungen erfüllen:

A. Gleichverteilung im Intervall [0,1)
B. Stochastische Unabhängigkeit der einzelnen Pseudozufallszahlen
C. Periodenlänge sollte möglichst groß sein
D. Erzeugung der Zufallszahlenfolge einfach und schnell

Es existieren statistische Verfahren, um die Einhaltung insbesondere der Forderungen A. (Anpassungstest) und B. (Unabhängigkeitstest) zu prüfen. Auf die Testung des verwendeten Zufallszahlengenerators wird im Kapitel 5 eingegangen. Insbesondere für lineare Kongruenzgeneratoren sind die zu wählenden Parameter theoretisch ausgiebig erforscht worden, siehe dazu [Dotzauer 87] und [Langendörfer 92].

Zur computerunterstützten Erzeugung von in [0,1) gleichverteilten Pseudozufallszahlen kann man sich der folgenden Pseudozufallszahlengeneratoren bedienen:

METHODE DER QUADRATMITTEN

Die Methode der Quadratmitten wurde 1946 von John von Neumann entwickelt. Ausgehend von einer Zahl x_0 mit 2n Stellen ($n \in N$) wird eine Zahl h mit 4n Stellen ermittelt. Fehlende führende Ziffern werden dabei mit 0 aufgefüllt. Die mittleren 2n Stellen von h ergeben die nächste Pseudozufallszahl x_{i+1} der Folge. Es zeigte sich, daß mit diesem Ansatz nur eine geringe Periodenlänge erreicht werden kann.

TAUSWORTHE-GENERATOR

Tausworthe-Generatoren erzeugen Folgen von Pseudozufallszahlen nach folgender Vorschrift [Kleijnen 92]:

$$b_i = \left(\sum_{j=1}^{q} c_j b_{i-j}\right) \bmod 2$$

mit $c_q=1$ und $c_j \in \{0,1\}$ für j=1,2,..,q-1 und mindestens einem c_j ungleich 0. Tausworthe-Generatoren sind unabhängig von der Wortgröße des Rechners und führen nur primitive Maschinenbefehle aus (Die Anweisung modulo 2 entspricht einer Boolschen Exklusiv-Oder-Operation). Ein solcher Generator erzeugt Folgen von Bits, welche nach verschiedenen Vorschriften zu Pseudozufallszahlen zusammengesetzt werden können.

LINEARER KONGRUENZEN-GENERATOR

Diese Methode wurde zuerst von D. Lehmer vorgeschlagen. Die Verwendung von linearen Kongruenzgeneratoren stellt die heute häufigste Vorgehensweise der Erzeugung von Pseudozufallszahlen dar. Diese erfolgt nach der Vorschrift

$$y_n = (a * y_{n-1} + b) \bmod m \quad \text{mit } a,b,m \in N.$$

Ausgehend von einem Startwert y_0 ungleich Null wird hier eine zyklische Folge von natürlichen Pseudozufallszahlen im Intervall [0,m) mit der maximalen Periode m erzeugt. Dabei heißen Generatoren mit b=0 *multiplikativer linearer Kongruenzgenerator*, andernfalls *gemischter linearer Kongruenzgenerator*. Die Erweiterung um die additative Konstante b wurde von D.Lehmer zunächst nicht vorgesehen. Sie wurde eingeführt, da ein multiplikativer linearer Kongruenzgenerator die maximale Periode m nicht erreichen kann. Durch Division durch m ergeben sich aus y_n in [0,1) gleichverteilte Pseudozufallszahlen.
Besondere Bedeutung kommt der Wahl der Parameter a und m zu. Ausführliche Betrachtungen sind in [Dotzauer 87] und [Langendörfer 92] zu finden. Für die Bestimmung des Moduls m können folgende Richtlinien gegeben werden:

- Der Modul sollte möglichst groß sein, da er sich maßgeblich für den Maximalwert der Periodenlänge verantwortlich zeigt und
- Der Modul sollte aus Effizienzgründen eine Potenz des verwendeten Zahlensystems sein.

Bei entsprechender Wahl von a, b und m kann eine maximale Periodenlänge von m für gemischte lineare Kongruenzgeneratoren erreicht werden [Dotzauer 87][Langendörfer 92].
Wichtiges Kriterium für die Wahl eines a ist der Autokorrelationskoffizient ρ_1. Dieser ist ein Maß für die Abhängigkeit zweier Größen. Hierbei bedeutet $\rho_1=0$ völlige Unabhängigkeit,$\rho_1=1$ völlige Abhängigkeit zweier (aufeinanderfolgender) Pseudozufallszahlen. In [Kleijnen 92] und [Langendörfer 92] ist eine Schranke für ρ_1 als Funktion von a, b und m angegeben:
$|\rho 1| < (1/a) - (6b/am)(1-b/m) + a/m$.
Hieraus resultiert die Empfehlung der Wahl von a als ungefähr der Quadratwurzel aus m. Eine weitere Empfehlung ist die Wahl von a gemäß dem *Goldenen Schnitt* [Langendörfer 92].

Für die Implementation des Basiszufallszahlengenerators der Simulationskomponente von BNETD wurde ein multiplikativer linearer Kongruenzengenerator mit folgenden gebräuchlichen Parametern verwendet:
$a = 16807 = 7^5$, b = 0 und $m = 2147483647 = 2^{31}-1$.
Die durchgeführten Tests (siehe Kapitel 5) gaben keinen Anlaß, einen alternativen Generator zu verwenden.

MEHRFACHER LINEARER KONGRUENZGENERATOR

Aufbauend auf einfachen linearen Kongruenzgeneratoren bilden mehrfache lineare Kongruenzgeneratoren eine Reihe von Pseudozufallszahlen wie folgt:
$$\mathbf{y_n=(a_1*y_{n-1}+a_2*y_{n-2}+...+a_k*y_{n-k} + b) \bmod m}.$$ (k-facher linearer Kongruenzgenerator)
Durch Division durch m erhält man in [0,1) gleichverteilte Zufallszahlen.
Die Motivation für die Anwendung mehrfacher linearer Kongruenzgeneratoren besteht in der Idee, ein "höheres Maß an Zufall" innerhalb von n-Tupeln aufeinanderfolgender Pseudozufallszahlen zu erzielen. Dem steht ein höherer Rechenaufwand gegenüber. Der Verzicht auf den Multiplikator a_i ergibt die folgende Klasse von Generatoren:

ADDITIVE GENERATOREN

Diese Generatoren sind aufgrund ihrer einfachen Struktur die bei weitem schnellsten Generatoren. Nach [Langendörfer 92] heißt eine Funktion

$$\mathbf{y_i=(y_{i-h} + y_{i-k}) \bmod m} \quad \text{mit } y_i \in N,\ y_i<m$$

und den Startwerten $y_0,...,y_{i-1}$ ein additiver linearer Kongruenzgenerator. Ein spezieller additiver Generator ist der "Fibonacci-Generator", der jedoch praktisch unbrauchbare Werte liefert. Dagegen konnten dem bereits 1958 von G.J.Mitchell und D.P.Moore vorgeschlagenen Generator mit h=24, k=55 und m gerade keine Nachteile bei empirischen Tests nachgewiesen werden

[Langendörfer 92]. Dieser Generator verfügt über eine extrem lange Periode, die größer als 2^{55} ist.

GEMISCHTE GENERATOREN

In dem Fall, daß ein Generator die an ihn gestellten Anforderungen teilweise nicht erfüllt, besteht die Möglichkeit, durch Kombination zweier verschiedener Generatoren ein besseres Gesamtresultat zu erreichen. Auch andere Manipulationen (Permutationen innerhalb einer Folge, Kombination mehrerer Folgen) sind denkbar. Der größte Nachteil besteht darin, daß die erreichten Testergebnisse schwer theoretisch zu untermauern sind.

2.6.3. Transformation

Für die meisten Anwendungen ist eine gleichverteilte Zufallszahl in [0,1) nicht ausreichend, sie muß in andere Verteilungen transformiert werden. Dazu existieren verschiedene Möglichkeiten:

- Methode der inversen Verteilungsfunktion
- Sinus-/Kosinustransformation (Polarmethode)
- Verwerfungsmethode
- Kompositionsmethode

Es sollen hier Methoden zur Transformation von Zufallszahlen in stetige Verteilungen vorgestellt werden, wie sie auch innerhalb des entwickelten Simulationspaketes zur Anwendung kommen.
Innerhalb dieses Abschnittes wird in der Folge für eine Zufallszahl aus [0,1) abkürzend R_0 geschrieben.

ERZEUGUNG EINER BELIEBIGEN GLEICHVERTEILUNG

Begonnen werden soll mit der einfachsten Transformation, der Erzeugung einer Gleichverteilung in [a,b). Diese erfolgt durch die Normierung
$R_{GV[a,b]} = (b-a)R_0 + a$.

ERZEUGUNG EINER EXPONENTIALVERTEILUNG

Unter Zuhilfenahme der Methode der inversen Verteilungsfunktion kann die Transformation einfach durchgeführt werden:
Aus $R_0 = 1-e^{-\lambda x}$
ergibt sich durch Umstellen:

$$x = -1/\lambda \ln(1-R_0)$$

oder weiter vereinfacht:

$$x = -1/\lambda \ln(R_0)$$

Die Zufallsgröße x ist exponentialverteilt mit dem Erwartungswert $1/\lambda$.

ERZEUGUNG EINER ERLANGVERTEILUNG

Die Erlangverteilung ist die k-fache Faltung der Exponentialverteilung, ihre Erzeugung hängt mit der Bildung einer exponentialverteilten Zufallsgröße zusammen. Innerhalb des Simulationspaketes für BNETD wurde folgende Funktion zur Realisierung einer Erlangverteilung der Ordnung k mit dem Erwartungswert $1/\lambda$ implementiert:

$$R_{ERLANG(\lambda,k)} = \sum_{i=0}^{k-1} \frac{-1}{k\lambda} \ln(R_i)$$

ERZEUGUNG EINER NORMALVERTEILUNG

Eine Methode der Transformation in [0,1) gleichverteilter Zufallszahlen in eine Normalverteilung stellt die Sinus-Kosinus-Transformation dar. Außerdem existieren Approximationsmethoden, eine ist in [Frank 93] angegeben. Nach entsprechender Prüfung der verschiedenen Methoden wurde für die Implementation eine in [Dotzauer 92] beschriebene Praktik verwandt, deren Test sehr gute Werte ergab:

Wiederhole
{ Hilfsgröße $h_1 := 2R_0-1$;
Hilfsgröße $h_2 := 2R_1-1$;
Summe $S := h_1^2+h_2^2$;
}
Bis S größer oder gleich 1 ist;

$$R_{NV} = h_1 \sqrt{\frac{-2\ln(S)}{S}}$$

Auf diese Weise kann eine standardnormalverteilte Zufallgröße erhalten werden.
Durch Multiplikation mit der Standardabweichung σ und Addition des Erwartungswertes μ erhält man die gewünschte $N(\mu,\sigma^2)$-verteilte Zufallsgröße.

Im folgenden Kapitel soll dargestellt werden, welche Konsequenzen ein objektorientierter Entwurf für die ereignisorientierte Modellierung nach sich zieht.

3. Objektorientierter Ansatz

Der objektorientierte Ansatz ist die natürliche Herangehensweise des Software Engineering: Die in der Realität existierenden Objekte werden als Softwareobjekte modelliert. Er ist damit in seiner Grundidee sehr einfach. Anstelle des herkömmlichen prozeduralen, oft Top-Down orientierten Entwurfs erfolgt hierbei eine Konzentration auf die Daten eines Softwareprodukts.
Die konkrete Umsetzung wird im Kapitel 4 beschrieben, zunächst soll auf Motivation und Grundlagen eingegangen werden (Abschnitt 3.1). Eine ausführliche Darstellung enthält [Meyer 90], an welcher der folgende Überblick angelehnt ist.
Der Abschnitt 3.2. stellt die zur Implementation benutzte Sprache C++ vor.

3.1. Objektorientierter Entwurf

3.1.1. Motivation

[Meyer 90] bezeichnet die objektorientierte Herangehensweise als "Prachtstraße zu Softwareentwurf und -realisierung". Als Grundidee und Motivation ist natürlich die Verbesserung der Qualität der Software zu sehen, insbesondere von Kriterien wie Modularität, Erweiterbarkeit, Wiederverwendbarkeit, Korrektheit, Robustheit und Kompatibilität.

Dabei spielt auch der Bereich der Softwarewartung innerhalb des Softwarelebenszyklus eine große Rolle - Schätzungen zufolge (nach [Meyer 90]) fallen in diesem Sektor 70% der Gesamtkosten an. Dabei setzen sich die Wartungskosten im wesentlichen aus Änderungen in Benutzeranforderungen (circa 40%) und Änderungen in Datenformaten (circa 20%) zusammen. Ein gutes Beispiel aus jüngeren Vergangenheit ist der enorme Kostenaufwand, den die Einführung der neuen Postleitzahlen in Deutschland verursachte. Mit der Einhaltung von Softwarequalitätskriterien kann dieser Änderungs- und Erweiterungsaufwand wesentlich reduziert werden.

3.1.2. Softwarequalität

Wie bereits erwähnt, bietet die objektorientierte Herangehensweise ein großes Potential zur Verbesserung der Qualität von Softwareprodukten. In diesem Abschnitt soll mit der Erläuterung der Qualitätskriterien für den Softwareentwurf eine Zielbestimmung gegeben werden.
Qualitätskriterien können zunächst in zwei grundsätzliche Klassen eingeteilt werden - in **innere** und **äußere Kriterien**. Die folgende Tabelle gibt einen Überblick:

Innere Kriterien	***Äußere Kriterien***
Modularität	*Korrektheit*
Lesbarkeit	*Robustheit*
Quelltextdokumentation	*Erweiterbarkeit*
	Wiederverwendbarkeit
	Kompatibilität
	Effizienz
	Portabilität
	Verifizierbarkeit
	Integrität
	Benutzerfreundlichkeit

Tabelle 3-1: Softwarequalitätskriterien

Als innere Kriterien sind diejenigen Faktoren aufgeführt, die dem Nutzer des Programmes im allgemeinen verborgen bleiben. Innere Faktoren sind jedoch der Schlüssel für die Erfüllung der äußeren Qualitätskriterien.
Äußere Kriterien sind die für die Qualität einer Software entscheidenden Faktoren.
Die kursiv dargestellten Qualitätskriterien sind dabei diejenigen Kriterien, die der objektorientierte Ansatz im Vergleich zu anderen Entwurfsmethoden - wie der prozedurale Top-Down-Entwurf oder die datenflußorientierte Vorgehensweise von DeMarco - zu verbessern trachtet. Diese Kriterien werden von [Meyer 90] als Schlüsselkriterien für den objektorientierten Entwurf bezeichnet und im folgenden vorgestellt (Definitionen nach [Meyer 90]).

KORREKTHEIT

Definition:

> Korrektheit ist die Fähigkeit von Softwareprodukten, ihre Aufgaben exakt zu erfüllen, wie sie durch Anforderungen und Spezifikationen definiert sind.

Die Korrektheit ist das primäre Kriterium für Softwarequalität, welches unbedingt erfüllt sein muß.

ROBUSTHEIT

Definition:

> Robustheit heißt die Fähigkeit von Softwaresystemen, auch unter außergewöhnlichen Bedingungen zu funktionieren.

Robustheit beschreibt das Verhalten des Softwareproduktes unter Bedingungen, die nicht in der Spezifikation beschrieben sind.
Das häufig gebrauchte Qualitätskriterium "Zuverlässigkeit" wird geeignet als Obermenge von Korrektheit und Robustheit aufgefaßt.

ERWEITERBARKEIT

Definition:

> Erweiterbarkeit bezeichnet die Leichtigkeit, mit der Softwareprodukte an Spezifikationsänderungen angepaßt werden können.

Das Problem der Änderung bestehender Programme wächst mit ihrer Größe. Im Extremfall ähnelt ein komplexes Softwaresystem einem Kartenhaus, welches beim Herausnehmen einer Komponente instabil zu werden droht. Ungeeignet sind auf jeden Fall Entwürfe mit einer zentralen Architektur, in denen viele wechselseitige Abhängigkeiten bestehen (zum Beispiel erzeugen globale Variablen solche Abhängigkeiten). Zur Verbesserung der Erweiterbarkeit können folgende Prinzipien angegeben werden:

a) Einfachheit des Entwurfs und
b) Dezentralisierung mit in hohem Grad autonomen Modulen.

WIEDERVERWENDBARKEIT

Definition:

> Die Wiederverwendbarkeit von Softwareprodukten ist die Eigenschaft, ganz oder teilweise für neue Anwendungen wiederverwendet werden zu können.

Die Wiederverwendbarkeit ist ein grundlegendes Problem der Softwareherstellung, ein ständiges "Neuerfinden des Rades" kann keine Lösung sein. Die Erfüllung des Kriteriums der Wie-

derverwendbarkeit würde weiterhin einen positiven Einfluß auf andere Kriterien des Softwareentwurfs haben (Wiederverwendung eines hochqualitativen Moduls!). Im Abschnitt 3.1.4. wird auf Wiederverwendbarkeit weiter eingegangen.

KOMPATIBILITÄT

Definition:

> Kompatibilität ist das Maß der Leichtigkeit, mit der Softwareprodukte mit anderen verbunden werden können.

Das Kriterium der Kompatibilität ist Voraussetzung für das Zusammenfügen einzelner Programmodule, die miteinander kommunizieren. Der Weg zur Erfüllung dieses Kriteriums kann bestehen in der Nutzung standardisierter Dateiformate, Datenstrukturen, Benutzerschnittstellen und Protokolle.
Die Lösungsidee für dieses Problem besteht in der objektorientierten Herangehensweise in der Definition abstrakter Datentypen (Abschnitt 3.1.5.).

Zu den weiteren Qualitätskriterien soll angemerkt werden:

- Portabilität ist das Maß der Leichtigkeit, mit der Softwareprodukte auf verschiedene Hardware- und Softwareumgebungen übertragen werden können;
- Verifizierbarkeit ist das Maß der Leichtigkeit, mit der Testdaten und Prozeduren zur Fehlererkennung und -verfolgung während der Test- und Nutzungsphase erzeugt werden können.
- Integrität ist die Fähigkeit des Softwaresystems, seine verschiedenen Komponenten gegen unberechtigte Zugriffe und Veränderungen zu schützen.
- Benutzerfreundlichkeit ist die Leichtigkeit, mit der die Benutzung von Softwaresystemen, ihre Bedienung, das Bereitstellen von Eingabedaten, die Auswertung der Ergebnisse und das Wiederaufsetzen nach Bedienfehlern erlernt werden kann und die Ermöglichung einer effizienten Benutzung (Vermeidung unnötiger Nutzerinteraktion).

Bei der Erfüllung dieser Kriterien müssen oft Kompromisse eingegangen werden, da sich manche Kriterien in Konkurrenz zueinander befinden - ein Beispiel sind Effizienz und Portabilität. Daraus folgt, daß der Begriff Softwarequalität als Kompromiß zwischen einer Menge verschiedener Ziele (Qualitätskriterien) betrachtet werden sollte.

3.1.3. Modularität

Modularität als Schlagwort des Software Engineering wird im herkömmlichen Sinn verstanden als Entwicklung von Programmen, die aus kleinen Teilstücken (Unterprogrammen) bestehen. Hier soll Modularität als wesentlich komplexeres Problem aufgefaßt werden, so muß ein Modul autonom, in sich geschlossen und in einer robusten Gesamtarchitektur organisiert sein. Zur Bewertung von Entwurfsmethoden bezüglich Modularität können fünf Kriterien definiert werden:

- *Modulare Zerlegbarkeit*: Modulare Zerlegbarkeit heißt, daß ein Problem in Teilprobleme zerlegt werden kann, deren Lösungen voneinander getrennt erarbeitet werden können - sie können verschiedenen Programmierern zur Bearbeitung übergeben werden. Die Top-Down-Entwurfsmethode unterstützt das Kriterium der modularen Zerlegbarkeit.
- *Modulare Kombinierbarkeit*: Das Kriterium der modularen Kombinierbarkeit ist erfüllt, wenn Module wie ein Baukastensystem frei zur Herstellung neuer Soft-

wareprodukte zusammengesetzt werden können. Dieses Kriterium wird durch eine Bottom-Up-Herangehensweise gefördert - ein Beispiel sind Software-Bibliotheken.

- *Modulare Verständlichkeit*: Ein Modul soll modular verständlich heißen, wenn der menschliche Leser den Quelltext eines Moduls "versteht", ohne vorher andere Module "verstanden" zu haben müssen. Dieses Kriterium spielt eine besondere Rolle bei der Softwarewartung.
- *Modulare Stetigkeit*: Wenn eine kleine Änderung in der Problemspezifikation nur eine Änderung in einem oder wenigen Modulen erforderlich macht und dabei nicht die Beziehungen (Schnittstellen) zwischen den Modulen beeinflußt, gilt das Kriterium der modularen Stetigkeit als erfüllt. Der Begriff "Stetigkeit" wird verwendet, um eine Analogie zum mathematischen Begriff der stetigen Funktion aufzuzeigen: Eine stetige Funktion hat die Eigenschaft, daß "kleine" Änderungen am Argument eine "kleine" Änderung des Funktionsergebnisses bewirken. Ein Beispiel aus der Programmierpraxis sind symbolische Konstanten (Schlüsselwort *const* in C/C++).
- *Modulare Geschütztheit*: Das Kriterium modulare Geschütztheit gilt dann als erfüllt, wenn Laufzeitfehler sich lediglich auf ein oder wenige Module auswirken. Ein Beispiel ist die Validierung von Benutzereingaben bereits im Eingabemodul, ein Gegenbeispiel eine Ausnahmebehandlung (Exception Handling), die auf Fehler mit etwa "goto Fehlermodul" reagiert.

Zum Zweck der Erreichung der oben angeführten Modularitäts-Kriterien werden fünf Prinzipien angegeben:

- *Sprachliche Moduleinheiten*: Module sollten syntaktisch abgeschlossene Einheiten sein, das heißt, sie sollten sich - wenn implementiert - getrennt übersetzen lassen. Das heißt für den Softwareentwurf, daß die Gegebenheiten der später zu nutzenden Programmiersprache beachtet werden müssen.
- *Wenige Schnittstellen*: Jeder Modul sollte mit möglichst wenig anderen kommunizieren. Strukturen der Modulverbindungen können beispielsweise sternförmig oder ringförmig sein. Der Top-Down-Entwurf ergibt stets zentrale Architekturen, der objektorientierte Ansatz läßt dieses offen.
- *Schmale Schnittstellen*: Der Informationsaustausch zwischen zwei Modulen sollte sich auf das Wesentliche beschränken. In Sprachen wie C oder Pascal erweitern globale Variablen die Schnittstelle zwischen den Modulen implizit.
- *Explizite Schnittstellen*: Wenn zwei Objekte eine Schnittstelle zueinander besitzen, dann muß diese Tatsache leicht aus dem Programmtext erkennbar sein.
- *Geheimnisprinzip*: Das Geheimnisprinzip erzwingt eine Trennung von Schnittstelle (Funktionalität) und Implementierung und ermöglicht somit eine Änderung der Implementierung bei unveränderter Funktionalität.

Während der Entwicklung eines umfangreichen Softwaresystem stößt man auf ein weiteres Problem der Modularisierung, in [Meyer 90] als **Offen-Geschlossen-Prinzip** bezeichnet. Zur Begriffsbildung:
Ein Modul soll offen heißen, wenn der Modul für Erweiterungen zur Verfügung steht. Normalerweise sind dieses Module, deren Entwicklung noch nicht abgeschlossen ist, da für Datenstrukturen oder Funktionen zukünftige Änderungen nicht ausgeschlossen sind.
Dagegen heißt ein Modul geschlossen, wenn seine Entwicklung als abgeschlossen betrachtet wird. Geschlossene Module können in Bibliotheken eingefügt werden, allgemein also zur Benutzung durch andere freigegeben werden.
Das Problem besteht nun darin, daß das endgültige Abschließen eines Moduls niemals garantiert werden kann. Es können immer wieder Änderungen an bereits freigegebenen Programmodulen notwendig werden, was die Verwaltung eines komplexen Projektes erschwert. Teilweise Unterstützung bieten vom UNIX-Make-Utility abgeleitete Software-Engineering-Werkzeuge,

die eine Abhängigkeitsprüfung durchführen und so genau die Programmteile neu übersetzen, die von Änderungen betroffen sind.
Klassische Ansätze bieten keinen Weg, um für einen Modul gleichzeitig Eigenschaften sowohl eines offenen als auch eines geschlossenen Moduls zu bewahren. Der Lösungsansatz der objektorientierten Methode bietet als Lösung für dieses Problem die Vererbung an (Abschnitt 3.1.8.).

3.1.4. Wiederverwendbarkeit

Programmieren ist eine sehr wiederholungsintensive Tätigkeit. Das Ziel der Wiederverwendbarkeit besteht darin, Software ähnlich wie Hardware aus fertigen Bausteinen zusammenzusetzen und so Zeit und Kosten zu sparen. Für die Realisierung dieses Zieles gibt es jedoch einige Hindernisse:

a) "Technische" Hindernisse: Oft soll prinzipiell das gleiche programmiert werden (etwa Suchalgorithmen), oft unterscheidet sich die neue Anforderung im Detail von der bereits vorhandenen Lösung (etwa im Typ der zu suchenden Elemente), die so nicht ohne weiteres übernommen werden kann.
b) Andere Hindernisse: Es können organisatorische Schwierigkeiten auftreten wie fehlendes Wissen um die Existenz eines solchen Bausteins, Schwierigkeiten bei der Beschaffung oder zu hohe Kosten. Ebenfalls treten psychologische Schwierigkeiten auf - ein extern entwickelter Baustein muß gegenüber Eigenentwicklungen entscheidende Vorteile haben. Nur ungern verläßt man sich auf das fehlerfreie Funktionieren "fremder Bausteine".

Einfache Ansätze für die Gewährleistung von Wiederverwendbarkeit sind:

- Wiederverwendung von Quellcode ist an Universitäten und in der Industrie in kleinem Maßstab üblich. Diese Technik verwirklicht unter anderem nicht das Geheimnisprinzip.
- Personal-Wiederverwendbarkeit ist in der Industrie üblich. Durch den Einsatz von den gleichen Softwareentwicklern wird Know-How-Verlust vermieden, zusätzlich werden Erfahrungen in neue Projekte übernommen.
- Wiederverwendbarkeit von Entwürfen erreicht die Wiederverwendung von Modulmustern für bestimmte Problemklassen.

Diese Herangehensweisen sind in vielen Anwendungsfällen nützlich, aber nicht hinreichend, um umfassende Wiederverwendbarkeit von Softwareprodukten zu gewährleisten. Um zu Modulstrukturen zu kommen, die brauchbare wiederverwendbare Bausteine ergeben, müssen folgende fünf Probleme gelöst werden:

- Typ-Variation: Module sollen auf verschiedene Datentypen anwendbar sein.
- Datenstruktur- und Algorithmenvariation: Module sollten im Hinblick auf zu behandelnde Datenstrukuren flexibel sein, und Module sollten verschiedene Algorithmen zur Behandlung der Datenstrukturen ermöglichen.
- Abhängigkeiten von Routinen: Routinen eines Moduls zur Behandlung von Daten stehen im Zusammenhang mit anderen Routinen über dieser Datenstruktur.
- Darstellungsunabhängigkeit: Durch den Nutzer des Moduls muß der Aufruf des Moduls möglich sein, ohne die konkrete Implementierung zu kennen.
- Gemeinsamkeiten zwischen Implementierungsgruppen: Die Behandlung verschiedener Datentypen im Modul sollte effizient erfolgen, indem Gemeinsamkeiten zwischen ihnen ausgenutzt werden.

Ein klassischer Lösungsansatz zur Gewährleistung der Wiederverwendbarkeit ist die Bildung von Routinenbibliotheken. Unter den Begriff Routine fallen Begriffe wie Funktion, Prozedur,

Subroutine, Unterprogramm. Wenn ein solcher Modul der Forderung nach Datenstrukturvariation Rechnung tragen soll, kann dieses mit einer Menge von *case*-Anweisungen erreicht werden, was eine Änderung und Neuübersetzung der ganzen Routine zur Folge hat.
Eine bessere Lösung wird von Modula-2 und Ada angeboten. Hier ist die Definition von Paketen (in Ada "package") möglich. Dadurch ist es möglich, miteinander zusammenhängende Elemente wie Typen, Variablen, Routinen zu sammeln und zu kapseln.
Eine höhere Flexibilität kann erreicht werden durch das Überlagern (bzw. Überladen) von Namen. Ein Beispiel sind überladene Funktionen und Operatoren in C++.
In gleichem Maße ist Generizität zur Erhöhung der Flexibilität geeignet. Sie ist dadurch gekennzeichnet, daß man anstelle eines Datentypes ein Modul mit einem formalen Parameter definiert. Bei der Nutzung eines solchen Moduls ist anstelle des formalen Parameters ein konkreter Datentyp anzugeben. Diese Möglichkeiten werden in C++ mit der Definition von Templates (Schablonen) unterstützt.

3.1.5. Schritte zur "Objektorientiertheit"

In diesem Abschnitt soll gezeigt werden, auf welche Weise man zu Strukturen kommt, die insbesondere die Erreichung der Ziele Wiederverwendbarkeit und Erweiterbarkeit unterstützen. Die grundlegende Entscheidung, welche bei dem Entwurf einer Softwarearchitektur zu fällen ist, besteht darin festzulegen, ob die entstehende Struktur von Funktionen oder von Daten abgeleitet werden soll. Herkömmliche Entwurfsmethoden gehen von einer funktions-orientierten Strukturfindung aus, der objektorientierte Ansatz setzt oberste Priorität auf Daten (Objekte).

FUNKTIONEN- ODER DATENORIENTIERTER ENTWURF ?

Zur Beantwortung der Frage "Soll die Struktur des Softwaresystems von den Daten oder den Funktionen ausgehend entworfen werden?" soll das im Abschnitt 3.1.3. vorgestellte Stetigkeitsprinzip herangezogen werden. In [Meyer 90] wird dargestellt, daß innerhalb des Softwarelebenszyklus eines Softwareprodukts Funktionen die flüchtigen, Daten jedoch die über einen längeren Zeitraum stabilen Elemente darstellen. Von Daten abgeleitete Architekturen erfüllen das Prinzip der Stetigkeit (und somit Erweiterbarkeit) entscheidend besser als von Funktionen abgeleitete Architekturen. Damit ist ein Hauptargument für den objektorientierten Ansatz gefunden. Weitere Punkte, die für datenorientierte Entwurfsmethoden sprechen, sind Verträglichkeit und Wiederverwendbarkeit.
Als Gegenbeispiel sollen Probleme der Top-Down-Entwurfsmethode erwähnt werden:

- Top-Down-Entwurf unterstützt Weiterentwicklungen schlecht, solche Architekturen müssen bei Änderungen der Anforderungen umfangreichen Strukturänderungen unterzogen werden;
- Top-Down-Architekturen sind auf der obersten Ebene durch eine einzige Funktion gekennzeichnet, reale Systeme besitzen keine solche "Spitze";
- Die Beschreibung von relevanten Datenstrukturen wird bei funktionsorientierter Vorgehensweise über das gesamte System verstreut;
- Der Top-Down-Entwurf behindert die Wiederverwendung der Softwarebausteine.

Als Fazit kann festgehalten werden, daß die Top-Down-Methode nützlich für die Entwicklung von kleineren Programmen ist, jedoch bei komplexen Softwaresystemen erhebliche Risiken bezüglich der Qualitätskriterien in sich birgt.

OBJEKTORIENTIERTER ENTWURF

Objektorientierter Entwurf erfordert eine völlig andere Herangehensweise - der Entwerfer modelliert zunächst die Datenstrukturen des Systems, die Spezifizierung der höchsten Funktion des Systems wird aufgeschoben.
Die grundlegende Modellierungstechnik des objektorientierten Ansatzes - das Abbilden realer Objekte in Softwareobjekte - liegt insbesondere auf dem Gebiet der Simulation nahe. Objekt-

orientierte Sprachen wie Simula 67 und auch zunächst C++ wurden für die Implementierung diskreter ereignisorientierter Simulationsmodelle entworfen.
Wesentliche Begriffe der Abbildungstechnik realer Objekte in Softwareobjekte sind die Begriffe: abstrakter Datentyp, Klasse, Objekt. Diese sollen beschrieben werden:

Abstrakter Datentyp
Die Beschreibung realer Objekte erfolgt mit Hilfe der Theorie der abstrakten Datentypen. Ein abstrakter Datentyp hat folgende Anforderungen zu erfüllen: Die Beschreibung muß vollständig, genau und eindeutig sein, gleichzeitig jedoch Möglichkeiten zur Erweiterbarkeit und Wiederverwendbarkeit offenhalten (Vermeidung von Überspezifikation). Ein abstrakter Datentyp wird nicht durch seine technische Realisierung beschrieben, sondern vielmehr durch die verfügbaren Dienste.
Eine formale Spezifikation abstrakter Datentypen beinhaltet (nach [Meyer 90]) Typen, Funktionen, Vorbedingungen und Axiome. Abstrakte Datentypen können generisch sein, was eine höhere Flexibilität zur Folge hat.
Idealerweise kann ein so spezifizierter Datentyp alle wesentlichen Eigenschaften von Datenstrukturen wiedergeben und nicht mehr als das.

Klasse
Eine Klasse ist die Implementation eines abstrakten Datentyps. Durch Klassen werden Mengen von Datenstrukturen mit gleichen Eigenschaften charakterisiert. Klassen sind statisch. Klassen können formale generische Parameter haben.

Objekt
Ein Objekt ist ein konkretes Element einer Klasse. Objekte werden durch Instantiierung von einer Klasse erzeugt. Objekte sind dynamisch, sie können zur Laufzeit entstehen und vernichtet werden.

Damit kann eine Definition für den objektorientierten Entwurf angegeben werden:
Definition (nach [Meyer 90]):

> Objektorientierter Entwurf ist die Entwicklung von Softwaresystemen als strukturierte Sammlungen von Implementierungen abstrakter Datentypen.

Abschließend sollen in diesem Abschnitt Stufen bzw. Ebenen der Objektorientiertheit angegeben werden, eine wirklich objektorientierte Sprache sollte alle sieben Ebenen ermöglichen (Die zur Realisierung des BNETD-Projektes genutzte Sprache C++ unterstützt alle sieben Ebenen):

- Ebene 1: Objektbasierte modulare Struktur
- Ebene 2: Datenabstraktion
- Ebene 3: Automatische Speicherplatzverwaltung
- Ebene 4: Klassen
- Ebene 5: Vererbung
- Ebene 6: Polymorphismus und dynamisches Binden
- Ebene 7: Mehrfachvererbung

3.1.6. Zusicherungen und Ausnahmen

Um die am Beginn des Kapitels eingeführten Qualitätskriterien zu erfüllen, sind weitere Techniken erforderlich. Insbesondere soll hier das Augenmerk auf *Korrektheit* der entstehenden Software gerichtet werden. Zwei wesentliche Konzepte hierzu sind das "Programmieren durch Vertrag", welches sogenannte Zusicherungen als Voraussetzungen definiert, und das Konzept der Ausnahmebehandlung (Exception Handling).

ZUSICHERUNGEN

Das Konzept des Programmierens durch Vertrag bedeutet, daß die Nutzung der Dienste einer Klasse an bestimmte Vereinbarungen gebunden ist. Solche Vereinbarungen oder Zusicherungen sind syntaktisch gesehen Boolsche Ausdrücke. Wichtige Formen solcher Zusicherungen sind Vor- und Nachbedingungen zu Klassenroutinen. Vorbedingungen drücken dabei Eigenschaften aus, die vor Aufruf der Routine gelten müssen, um einen korrekten Ablauf zu sichern. Nachbedingungen sind Ausdrücke, deren Erfüllung nach Beendigung der Routine garantiert ist. Wenn bei Aufruf der Routine die Vorbedingung nicht erfüllt ist, dann wird die Erfüllung der Nachbedingung nicht garantiert.
Die Idee der Zusicherungen besteht darin festzulegen, wo und in welcher Weise Eingangsdaten einer Funktion geprüft werden. Damit wird die Gefahr vermieden, daß Eingangsgrößen nicht oder mehrfach validiert werden. Zusätzlich werden die Routinen, die die schließliche Verarbeitung der Daten durchführen, einfacher und klarer. Zusicherungen bilden die Grundlage für den Ausnahmemechanismus.

AUSNAHMEN

Ausnahmen werden erzeugt, wenn während der Laufzeit die Verletzung von Zusicherungen festgestellt wird oder von der Hardware eine abnorme Situation signalisiert wird. Für die Behandlung von Ausnahmen kommen zwei Strategien infrage. Die erste mögliche Lösung besteht darin, die Ausführung der Routine definiert zu beenden und der aufrufenden Funktion das Fehlverhalten zu melden. Die zweite Strategie besteht in der erneuten Durchführung der gescheiterten Operation (unter geänderter Vorbedingung). Eine schlechte Lösung wäre eine Ausnahmebehandlung, die mit "goto"-Anweisungen arbeitet (siehe Bemerkungen zur Modularität).

3.1.7. Entwurf von Klassenschnittstellen

Während der Arbeit an einem komplexen Softwareprojekt erfährt man schnell die Bedeutung der Festlegung von Klassenschnittstellen. Der objektorientierte Ansatz bietet alle Möglichkeiten, auch dieses Problem zufriedenstellend zu lösen.

Folgende Richtlinien erscheinen sinnvoll:

- Klassen sind durch ihre Schnittstelle, und nicht durch ihre Implementation charakterisiert;
- Schnittstellen sollten einfach und in sich geschlossen sein;
- Die Entwicklung der Schnittstelle ist ein evolutionärer Prozeß;
- Gute Schnittstellen erhält man häufig durch das Auffassen der Datenstrukturen als deterministischen abstrakten Automaten (Berücksichtigung des inneren Zustandes);
- Um der bereits skizzierten Idee zu genügen, daß eine Schnittstelle so schmal wie möglich sein sollte, bietet sich die Festlegung eines Zugriffsschutzes an (siehe Schlüsselwörter public, private und protected in C++).

3.1.8. Vererbung

Vererbung als eines der wichtigsten Schlagwörter objektorientierter Herangehensweisen beinhaltet die Definition neuer Klassen als Erweiterung, Spezialisierung oder Kombination bereits bestehender Klassen. Ohne das Konzept der Vererbung bietet der objektorientierte Ansatz die Voraussetzung für Modularität und die Einhaltung des Geheimnisprinzips. Die Vererbung dagegen kann die Ziele der Wiederverwendbarkeit und Erweiterbarkeit zu erreichen helfen.
Im Abschnitt 3.1.4. wurde das Offen-Geschlossen-Prinzip vorgestellt, es enthält das Problem, daß Module sowohl offen als auch geschlossen sein sollten. Die Vererbung bietet dafür die Lösung. Eine Klasse kann einerseits zur Benutzung zur Verfügung stehen (Eigenschaft

"geschlossen"), dieses wird durch Instantiierung erreicht. Andererseits steht eine Klasse für Erweiterungen zur Verfügung (Eigenschaft "offen"), jenes erfolgt durch Vererbung.
Ein weiterer wichtiger Aspekt, der vorrangig auf das Kriterium der Erweiterbarkeit zielt, ist das Konzept des Polymorphismus. Polymorphismus ist die Fähigkeit, verschiedene Formen anzunehmen. In der objektorientierten Programmierung sagt Polymorphismus aus, daß eine Größe zur Laufzeit auf Realisierungen verschiedener Klassen verweisen kann. Das Laufzeitsystem ist in der Lage, die durch den aktuellen Typ der Größe bestimmte Operation anzuwenden. Diese Fähigkeit ist als dynamisches Binden bekannt. In C++ erfolgt die Realisierung dieser Eigenschaft durch das Konzept der virtuellen Funktionen.
Mehrfachvererbung ist ebenfalls eine leistungsfähige Eigenschaft objektorientierter Systeme. Dadurch wird der Klassenentwickler nicht gezwungen, von zwei wesentlichen Elternklassen eine Elternklasse wegzulassen. In der Programmierpraxis existieren viele nützliche Anwendungsfälle für Mehrfachvererbung, beispielsweise das Erweitern von Klassen um Debug-, Stream- oder Animationsfunktionen durch "Hinzuerben" einer Klasse, die diese Dienste zur Verfügung stellt. Auftretende Schwierigkeiten können Namenskonflikte oder ein indirektes mehrfaches Erben gleicher Eigenschaften sein. Diese Probleme werden je nach Programmiersprache in verschiedener Weise gelöst, in C++ existiert dazu die Möglichkeit der Definition virtueller Basisklassen.
Das Geheimnisprinzip kann bei der Vererbung in folgender Weise umgesetzt werden: Es bestehen Möglichkeiten, Merkmale der Elternklasse in der abgeleiteten Klasse zu verbergen oder öffentlich zu machen.

3.2. Die C++ - Programmiersprache

Die Programmiersprache C++ wurde von Bjarne Stroustrup Anfang der achtziger Jahre an den AT&T Bell Laboratories als Nachfolger der Programmiersprache C entwickelt. Obwohl objektorientierte Programmierung kein neues Konzept ist (denn die Wurzeln der objektorientierten Programmierung liegen in den Sprachen Simula67 und Smalltalk), fand die objektorientierte Herangehensweise mit der zunehmenden Leistungsfähigkeit verfügbarer C++-Compiler immer mehr Verbreitung. C++ erweitert C um die Möglichkeit der Definition und des Einsatzes abstrakter Datentypen, Möglichkeiten der objektorientierten Programmierung und weiterer kleinerer Verbesserungen. Die Sprache wurde so entworfen, daß der Benutzer neue Typen definieren kann, welche genauso einfach und zweckmäßig zu verwenden sind wie standardmäßige Typen. C++ ist eine Obermenge von C und ermöglicht damit die Nutzung von C-spezifischen Bibliotheken und Know-How. Seit 1990 gibt es ein ANSI-C++ -Komitee, das heißt, die Sprache C++ ist mittlerweile standardisiert und verfügbare Compiler unter verschiedenen Systemen sind weitgehend kompatibel.
Eine Einordnung von C++ in die Welt der Programmierung könnte so aussehen:

PROZEDURALE PROGRAMMIERSPRACHEN	PROGRAMMIERSPRACHEN MIT ABSTRAKTEN DATENTYPEN	OBJEKTORIENTIERTE PROGRAMMIERSPRACHEN
Fortran	Ada	Simula
C	Modula-2	Smalltalk
		C++

Tabelle 3-2: Einordnung von C++

Im Gegensatz zu beispielsweise *Smalltalk* bezeichnet man C++ als *hybride* objektorientierte Sprache, da nach wie vor Möglichkeiten der prozeduralen Programmierung existieren.
Ausgewählte wichtige Möglichkeiten von C++ werden nun beschrieben:

KLASSEN

Klassen sind Implementationen abstrakter Datentypen. Sie bieten dem C++ - Programmierer die Möglichkeit, die Sprache praktisch um neue Datentypen zu erweitern. Klassen in C++ beinhalten Daten (Membervariablen) und Funktionen (Memberfunktionen) über diese Daten. Sie ermöglichen die Trennung von Schnittstelle und Implementation mit Hilfe der Schlüsselwörter *public*, *private* und *protected*. Zur garantierten Initialisierung von Instanzen einer Klasse, den Objekten, existieren spezielle Memberfunktionen, sogenannte Konstruktoren. Die Speicherfreigabe von Klassenobjekten, die ihren Bezugsrahmen verlassen, wird gewährleistet, indem das Laufzeitsystem eine weitere spezielle Memberfunktion aufruft, den Destruktor.

TEMPLATES

Templates (Klassenschablonen) sind die C++-Umsetzung der im Abschnitt 3.1. besprochenen Forderung nach Typ-Variation. Der formale generische Parameter wird bei Bildung einer Instanz der Klasse durch einen konkreten Typ ersetzt. Eine Klasse kann auch mehrere generische Parameter enthalten.

VERERBUNG

C++ verfügt über die notwendigen Mechanismen zur Vererbung. Der Informationsschutz in Ableitungen kann wiederum mit den Schlüsselwörtern *public*, *private* und *protected* beeinflußt werden.
C++ unterstützt ebenfalls Mehrfachvererbung.

VIRTUELLE FUNKTIONEN

Virtuelle Funktionen sind spezielle Memberfunktionen, die über eine Referenz oder einen Zeiger einer public-Basisklasse aufgerufen werden. Sie werden zur Laufzeit dynamisch mit einer Klasse verbunden. Das Konzept der virtuellen Funktionen stellt die C++-spezifische Umsetzung von Polymorphismus und dynamischem Binden dar.

STREAMS

Ein Stream ist ein abstrakter Datentyp mit einer theoretisch unbegrenzten Zahl von Argumenten. Streams stellen eine Verallgemeinerung des Dateikonzeptes auf alle Geräte dar. Durch das Überladen der Operatoren << und >> können bequeme und universelle Eingabe- und Ausgabemöglichkeiten für Standard- und Nichtstandard-Datentypen auf alle Geräte genutzt werden.

In neuen Versionen der C++-Compiler existieren Möglichkeiten zur Ausnahmebehandlung, die der für dieses Projekt verwendete Compiler noch nicht bieten konnte. C++ enthält ebenfalls die

für den Softwareentwurf nützliche Möglichkeit der Definition von abstrakten Memberfunktionen und abstrakten Klassen. In der Klassenbibliothek, die Borland zu seinen Compilern mitliefert, ist eine auf Makros beruhende Umsetzung des Prinzips der Zusicherungen (Preconditions und Checks) zu finden. Kleinere Verbesserungen von C++ gegenüber C betreffen eine strengere Typprüfung, die Definition von effizienten inline-Funktionen und die Möglichkeit der Definition von Default-Argumenten.

Im nächsten Kapitel wird die Entwicklung der Simulationskomponente von BNETD beschrieben.

4. Entwicklung des Simulationspaketes

4.1. Zielsetzung

Das Ziel dieser Arbeit ist es, eine Simulationskomponente für das Programmsystem BNETD zu schaffen. Unter Beachtung der in den vorhergehenden Kapiteln angeführten Anforderungen wurde für dieses Projekt nachstehende Abfolge konzipiert, welche die zeitliche Reihenfolge der Bearbeitung des Vorhabens berücksichtigt:

1. Entwurf einer Bibliothek von Klassen, die für die Simulation nützlich sind
2. Implementation der Klassenbibliothek
3. Testung der Klassenbibliothek
4. Entwurf und Implementation der Simulationskomponente unter Nutzung der bereitgestellten Klassen
5. Integration der Simulationskomponente in das Programmsystem BNETD
6. Testung der integrierten Version

Mit diesem Konzept soll vor allem ein Kompromiß zwischen zwei Zielen erreicht werden:

Universalität $\Leftrightarrow$ *Spezialisierte, angepaßte Lösung für BNETD*

Zur Erläuterung der einzelnen konzeptuellen Schritte:

Zu 1.)
Der gewählte objektorientierte Ansatz legt nahe, zunächst durch Definition geeigneter Klassen den Grundstein für eine spätere spezielle Lösung zu legen. Insbesondere wird dadurch den in Kapitel 3 besprochenen Software-Qualitätskriterien besser genügt. Der Entwurf einer Bibliothek impliziert stets eine Verfügbarkeit zur Wiederverwendung.
Der wichtigste Grund ist jedoch, daß die "Bausteine", die in dieser Bibliothek enthalten sind, ein größeres Leistungspotential beinhalten als für die Realisierung in BNETD erforderlich ist. Ein Beispiel sind zwei oder mehr Knoten mit einer gemeinsamen Warteschlange. Derartige Modellstrukturen brauchen für eine Realisierung in BNETD nicht zur Verfügung stehen, da in BNETD behandelte Warteschlangenmodelle über Knoten mit genau einer Warteschlange verfügen. Für die in der Bibliothek existierenden Klassen ist eine erweiterte Funktionalität natürlich wünschenswert.

Zu 2.)
Die Realisierung der Klassenbibliothek wird später detailliert erläutert. Hierbei handelte es sich um den aufwendigsten und abstrakten Teil der Arbeit. Sämtliche Klassen werden zunächst entsprechend dem Entwurf implementiert - es besteht keine einfache Testmöglichkeit, bis die Bibliothek einen hohen Grad an Vollständigkeit erreicht hat. Daher werden Probleme, die auf einem schlechten Entwurf beruhen, erst sehr spät erkannt. Funktionen, die die Fehlererkennung und -verfolgung fördern, werden in diesem konzeptuellen Stadium dringend benötigt - Beispiele sind sogenannte Debug-Ausschriften (in Abhängigkeit von gesetzten Präprozessor-Konstanten) oder die virtuelle Funktion print, mit der jedes Objekt über seinen aktuellen Zustand berichtet.

Zu 3.)
Die Testung der Klassenbibliothek erfolgt in der Weise, daß aus den vorhandenen Klassen bereits Simulationsmodelle erstellt werden. Danach kann unter Nutzung der print-Funktionen der Klassen (dabei insbesondere der dazugehörigen Statistikklassen) die Funktion verifiziert

werden. Bei dieser Vorgehensweise ist es günstig, wenn man sich auf vorhandene korrekte Lösungen der erstellten Modelle stützen kann. Bei Erkennung fehlerhaften Verhaltens wird zu Schritt 2 übergegangen und der Fehler korrigiert. Ab hier geschehen die Schritte 2 und 3 parallel.

Zu 4.)
Die Erstellung einer spezialisierten Simulationskomponente hat den Vorteil, daß nur bestimmte Modelle simulativ zu lösen sind (Umfang und Einschränkungen der Modellbildung sind im Kapitel 6 beschrieben). Dadurch entstehen Möglichkeiten, die Effizienz bezüglich Laufzeit und Platzbedarf zu verbessern, indem bestimmte Charakteristika der "BNETD-Modelle" ausgenutzt werden. Die Einschätzung der Laufzeiteffizienz konnte mit einem Profiler durchgeführt werden; auf effiziente Speicherplatznutzung muß geachtet werden, weil das Programmsystem BNETD unter MS-DOS überwiegend mit 640kByte Hauptspeicher auskommen muß.
Die Implementation der Simulationskomponente von BNETD beinhaltet hauptsächlich die Redefinition zum Simulatorkern gehörender Funktionen:

A. Verifizierung: Prüfung, ob das (von der BNETD-Oberfläche übergebene) Modell korrekt ist,
B. Initialisierung: Erstellung eines Simulationsmodelles aus dem gegebenen Modell. Alle statischen Modellelemente (Quellen, Warteschlangen, ...) werden erzeugt, ein Ereignis Simulationsende wird in die Ereignisliste eingetragen, die Simulation wird angestoßen.
C. Erweiterung der Fähigkeiten der Klassen der Bibliothek um Reaktionen und Statistiken bei auftretenden Blockierungen. Besonders hervorzuheben ist der Entwurf und die Implementation eines Algorithmus zur Erkennung von (unaufhebbaren) Blockierungen (im weiteren mit Deadlocks bezeichnet).
D. Deinitialisierung: Übertragen der gewonnenen Ergebnisse in die Ergebnisschnittstelle, Gewährleistung der Speicherfreigabe.

Zu 5.)
Die Integration der Simulationskomponente in das Programmsystem erfordert zunächst die Definition der Ergebnisschnittstelle (siehe Kapitel 6). Weiterhin sind in diesem Schritt enthalten: Anpassung der Oberfläche (Befehlsverwaltung, Menüstruktur und Hilfefunktion), Implementation von Funktionen zur Ergebnisrepräsentation und - ähnlich der Einbindung des Analysemoduls - eine Funktion, die den definierten Abbruch der Simulation gewährleistet.

zu 6.)
Die Testung der bereits integrierten Version erfolgt praktisch analog der Nutzung von BNETD. Möglichkeiten zum Vergleich erhaltener Ergebnisse bieten andere Simulationssysteme oder - wesentlich komfortabler - die integrierte Analysefunktion.

Im Zuge dieser schrittweisen Entwicklung des Systems BNETD entstanden mehr oder weniger als Randprodukte eine Kommandozeilenversion des Simulators und zwei kleinere Programme zur Testung des Basiszufallszahlengenerators und der abgeleiteten Generatoren. Diese werden am Ende des Kapitels vorgestellt.

4.2. Entwurf einer Klassenhierarchie

Die objektorientierte Herangehensweise legt größtes Gewicht auf den richtigen Entwurf der Klassen. Ein guter Entwurf kann wesentlich zur Erfüllung der in Abschnitt 3 angeführten Qualitätsmerkmale beitragen, dagegen kann ein schlechter Entwurf schnell in eine Sackgasse führen.

Der Erfinder der Sprache C++, Bjarne Stroustrup, gibt als Richtlinien folgende Punkte an:

a. Wenn "etwas" als separate Idee vorstellbar ist, sollte es zu einer Klasse gemacht werden.
b. Wenn "etwas" als separater Gegenstand vorstellbar ist, sollte es zu einem Objekt einer Klasse gemacht werden.
c. Wenn zwei Klassen eine wichtige Gemeinsamkeit haben, sollte diese Gemeinsamkeit Grundlage für eine Basisklasse werden.

Das typische Vorgehen soll am Beispiel der Klassenhierarchie der statischen Simulationsobjekte vorgestellt werden.
Ein erster Schritt ist in jedem Fall die Frage: Was soll zu einer Klasse gemacht werden? Die Antwort darauf ist einfach: Alles, was in der realen Welt ein separates Objekt darstellt, wird in der Simulation Mitglied bzw. Instanz einer Klasse. Auf diese Weise können einige Kandidaten für Klassen bereits bestimmt werden: Dies sind Klassen für Warteschlangen, Bedienanlagen, Quellen, Senken, Zufallszahlengeneratoren und Forderungen.

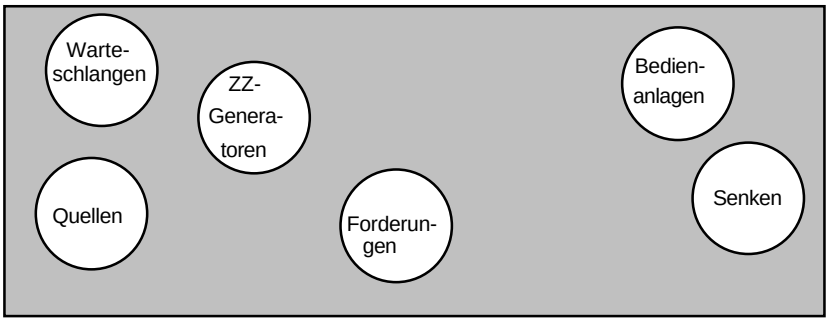

Abbildung 4-1: Verschiedene "reale" Objekte

Eine Klasse faßt gemeinsame Eigenschaften ihrer Objekte zusammen, eine Basisklasse faßt gemeinsame Eigenschaften der abgeleiteten Klassen zusammen. Die zweite Frage heißt damit: Welche der obigen Klassen haben (wesentliche) gemeinsame Eigenschaften? Die Antwort kann in folgender Form erfolgen: Warteschlangen, Bedienanlagen, Quellen, Senken sind statische Simulationsobjekte. Zudem ist ihnen gemeinsam, daß sie die sichtbaren Elemente einer Darstellung eines Warteschlangensystems sind (siehe auch Abschnitt 1.3.) und sie eine Statistikführung in irgendeiner Form beinhalten. Forderungen sind dagegen dynamische Elemente einer Simulation, sie werden erzeugt und vernichtet, sie können nicht zu einer Basisklasse statischer Simulationsobjekte gehören. Zufallszahlengeneratoren sind zwar statische Modellelemente, aber kein Objekt von der Bedeutung einer Warteschlange (nicht "sichtbar", keine Notwendigkeit zur Statistikführung). Zufallszahlengeneratoren bilden eine eigene Klassenhierarchie. Alle statischen Modellelemente, die einen Zufallszahlengenerator benötigen, erhalten diesen als Klassenmitglied. Beispiele sind Warteschlangen, die nach dem Random-Prinzip arbeiten, Quellen (Festlegung der Zwischenankunftszeit), Bedienanlagen (Festlegung der Bedienzeit) und Objekte, die für die Verteilung der Forderungen in einem Netz verantwortlich sind (Verzweigung nach Wahrscheinlichkeiten). Im Ergebnis dieser Betrachtungen sieht die Klassenhierarchie für statische Simulationsobjekte nun wie folgt aus:

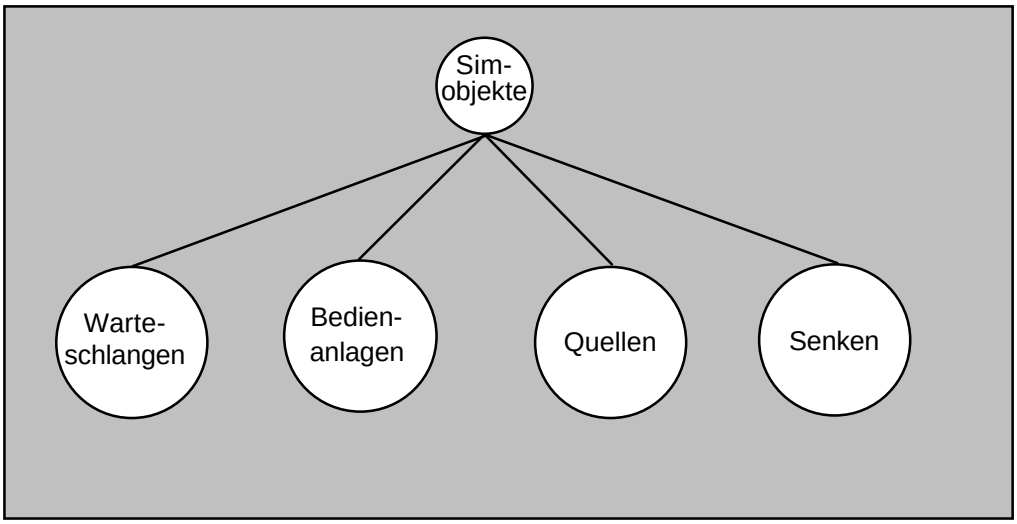

Abbildung 4-2: Ansatz für eine Klassenhierarchie statischer Simulationsobjekte

Der größere Umfang der die abgeleiteten Klassen kennzeichnenden Kreise symbolisiert eine gegenüber der Basisklasse höhere Funktionalität. Die abstrakte Basisklasse für statische Simulationsobjekte, SimulationsObjekt, definiert wie bereits angedeutet Elemente, die alle abgeleiteten Klassen gemeinsam haben werden. Einige Funktionen, welche die Klasse Simulations-Objekt definiert, sind abstrakt, sie können nicht benutzt werden. Gerade diese abstrakten Funktionen, die die Bildung eines Objektes dieser Klasse verhindern, spielen für den Entwurf eine wichtige Rolle. Alle abgeleiteten Klassen müssen diese abstrakten Funktionen überschreiben. Die Klasse Simulationsobjekt ist wie folgt deklariert:

```
class SimulationsObjekt
{
public:
        SimulationsObjekt();
        virtual ~SimulationsObjekt(){number--;};
        void setPredecessor(SimulationsObjekt *P) { prev = P ; };
        void setSuccessor(SimulationsObjekt *P) { next = P ; };
        TYPE_OF_SIMOBJECT getTyp() { return Typ; };
        virtual void print();
        virtual BOOLEAN handleMessage(TMessage&) = 0;
        virtual SPACEQUESTION_ANSWER is_Space() = 0;
        unsigned int getID() { return ID; } ;
protected:
        static unsigned int number;
        unsigned int ID;
        TYPE_OF_SIMOBJECT Typ;
        SimulationsObjekt *next;
        SimulationsObjekt *prev;
        void MessageDone(TMessage&);
};
```

Diese Basisklasse definiert bereits einen Identifikator für das Objekt, ein Datenelement, welches den Typ des Simulationsobjekt aufnehmen kann, Zeiger auf ein nächstes und ein vorhergehendes Simulationsobjekt in der späteren Modellstruktur und grundlegende Funktionen, die von allen Simulationsobjekten benötigt werden. Die abstrakte Funktion handleMessage spielt in den abgeleiteten Klassen eine besondere Rolle, sie bestimmt die Reaktion auf Nachrichten (übermittelte Ereignisse) und somit das Verhalten des Simulationsobjektes.

Der dritte Schritt beim Entwurf der Hierarchie der Simulationsobjekte besteht in einer Erweiterung und Spezifizierung der Klassen.

Beim Betrachten der Objekthierarchie (Abbildung 4-2) gewinnt man eine Vorstellung, worin das Aufgabenspektrum der einzelnen Klassen besteht. Eine Quelle erzeugt neue Forderungsobjekte, eine Warteschlange kann diese puffern usw. Es fehlen jedoch Objekte, die für das Weiterleiten der Forderungen verantwortlich sind.
Zudem ist in der Basisklasse SimulationsObjekt für jedes Objekt zunächst nur ein Nachfolger und ein Vorgänger vorgesehen. Es muß jedoch möglich sein, daß Objekte mehrere Nachfolger und/oder mehrere Vorgänger besitzen dürfen. Die Kenntnis der Vorgänger ist aus folgendem Grund erwünscht: Man stelle sich eine Struktur vor, in der eine Warteschlange zu einem Zeitpunkt t_1 völlig ausgelastet ist, ihre aktuelle Länge ist gleich der maximalen Kapazität. In dieser Situation führt eine Anfrage eines Vorgängers, ob eine Forderung übernommen werden kann, zu einer Absage. Dieser Vorgänger ist nun blockiert und geht in einen passiven Zustand über. Zu einem späteren Zeitpunkt t_2 gibt die Warteschlange eine Forderung an einen dahinterliegenden Knoten ab und ist nun wieder bereit, eine Forderung aufzunehmen. Diese Warteschlange hat nun die Verantwortung, den blockierten Vorgänger darauf aufmerksam zu machen, daß Kapazität zur Übernahme nunmehr vorhanden ist. Eine Benachrichtigung des Vorgängers ist aber nur möglich, wenn dieser auch bekannt ist - will man nicht alle Simulationsobjekte benachrichtigen.
Weiterhin ist die Kenntnis von Vorgängern und Nachfolgern nicht nur für die Behandlung von Blockierungen von großem Vorteil, sondern auch für Statistikführung und "normalen" Simulationsablauf.
Der einfachste Lösungsansatz besteht darin, in ein Objekt "Bedienanlage" eine Funktion einzubauen, die für die Weiterleitung der Forderungen (etwa nach Wahrscheinlichkeiten) verantwortlich ist. Dieser Ansatz hat mehrere Nachteile. Erstens müssen weitere Daten - wie eine Liste der Nachfolger - in jedem Objekt "Bedienanlage" verwaltet werden, dieses entspricht nicht dem Kriterium der Modularität. Zweitens müßten derartige Verzweigungsfunktionen in allen Klassen enthalten sein, die mehrere Nachfolger haben können. Drittens wäre damit das Problem mit mehreren Vorgängern nicht gelöst. Viertes lassen sich damit nur Modellstrukturen mit eingeschränkter Komplexität behandeln. Dieser Lösungsansatz verfügt jedoch auch über Vorteile: Erstens sind dadurch nur wirkliche existierende Objekte (Warteschlangen, Quellen, ...) als "SimulationsObjekt" erfaßt und zweitens kann eine Modellstruktur effizienter behandelt werden, wenn keine zusätzlichen "künstlichen" Objekte existieren.
Wie oben angedeutet, beruht der durchgeführte Entwurf auf der Einbindung solcher "künstlichen" Objekte. Dazu wird eine Klasse TRouter eingeführt, die Fähigkeiten zur Verwaltung mehrerer Nachfolger oder Vorgänger besitzt und Forderungen weiterleiten kann. Diese Klasse wird später genau beschrieben. Ein wichtiger Vorteil sei jedoch erwähnt: Solche *Verteilerobjekte können mit anderen Simulationsobjekten kombiniert werden*, so daß ein breiteres Spektrum realer Systeme modelliert werden kann.

Der zweite Teil des dritten Schrittes zum Entwurf der Klassenhierarchie ist eine Verfeinerung der bestehenden Klassen.
Ausgehend von der (Basis-)Klasse "Warteschlange" können spezialisierte Typen von Warteschlangen abgeleitet werden, wie FIFO-, LIFO-, Random- und Priority-Warteschlangen. Ebenso haben auch die Verteilerobjekte spezialisierte Nachkommen, die Klassen TSplit und TUnify. In der nachstehenden Abbildung ist das Ergebnis dargestellt:

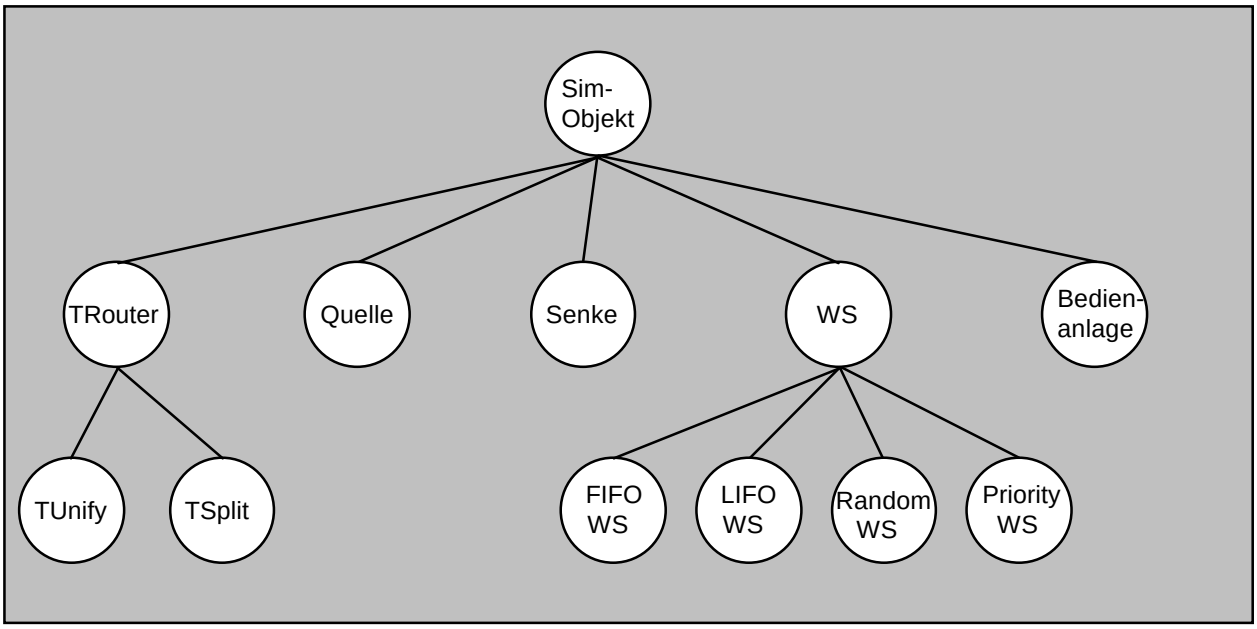

Abbildung 4-3: Klassenhierarchie statischer Simulationsobjekte

Eine vollständige Darstellung der Hierarchien der Klassen ist im Anhang zu finden.
Eine solche Klassenhierarchie ist im Laufe der weiteren Entwicklung des Projektes Änderungen und vor allen Erweiterungen unterworfen, diese werden durch den objektorientierten Ansatz geradezu herausgefordert. Ob die erstellte Klassenhierarchie den Anforderungen entspricht, zeigt die Anwendung. Mögliche Änderungen betreffen beispielsweise das Einfügen einer abstrakten Klasse Außenwelt, um die Gemeinsamkeiten zwischen Quelle und Senke zu betonen. Weiterhin könnte die Klasse Bedienanlage die Basis für eine Hierarchie von Bedienanlagen bieten, welche die Unterschiede (Anzahl der Kanäle, Verteilung der Bedienzeit) auch in der Rangordnung grafisch deutlich macht.

Die Hierarchie der statischen Simulationsobjekte ist die umfangreichste Klassenhierarchie des Simulationspaketes. Weitere Klassen sind die Hierarchie der Zufallszahlengeneratoren, Listenklassen, Statistik- und Ergebnisklassen sowie die Klasse TSimulator, die den Kern der Simulation darstellt.

Die verwendeten Klassen werden im Abschnitt 4.4. detailliert beschrieben.

4.3. Arbeitsweise der Simulation

4.3.1. Vorbemerkungen

Wie bereits im Abschnitt 2.5. beschrieben, arbeitet die innerste Schleife einer diskreten ereignisorientierten Simulation auf der Grundlage von in einer Ereignisliste vorhandenen Ereignissen.
Durch die Kapselung der Daten und Autonomie der Simulationsobjekte, wie sie durch den objektorientierten Ansatz ermöglicht wird, hat die Ablaufsteuerung - im folgenden mit Simulatorkern bezeichnet - vor allem die Aufgabe, Ereignisse an die betreffenden Objekte weiterzuleiten und wieder entgegenzunehmen. Dieser Ansatz betont in starkem Maße die Kommunikation zwischen den Objekten der Simulation.
Der Simulatorkern beinhaltet die notwendige Ereignisliste, eine Liste der statischen Objekte der Simulation, die Simulationsuhr und Routinen über diesen Daten zur Steuerung der Simulation, Initialisierung und Deinitialisierung des Simulationsmodelles. Die folgende Abbildung versucht, diesen Zusammenhang wiederzugeben:

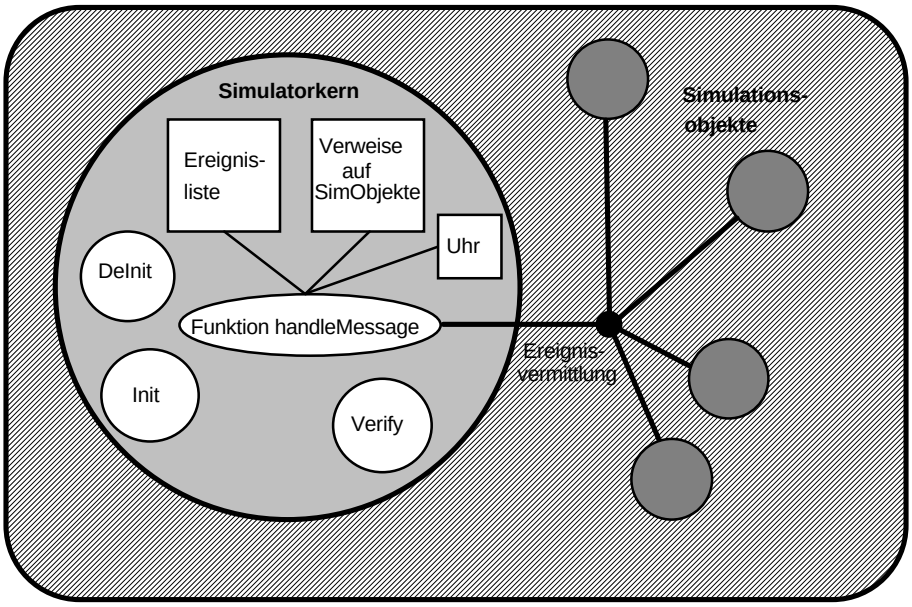

Abbildung 4-4: Arbeitsweise der Simulation (vereinfacht)

Gegenstand des Abschnittes 4.3.2. ist die Erläuterung von Funktion und Implementation des Simulatorkernes.

4.3.2. Der Simulatorkern - die Klasse TSimulator

Wie bereits erwähnt, realisiert der Simulatorkern die Ablaufsteuerung der Simulation und ist daher von grundlegender Bedeutung. Zugleich bieten die Elementfunktionen des Simulatorkernobjektes die Schnittstelle zur Nutzung der Simulationsklassen. Der Simulatorkern ist durch die Klasse TSimulator definiert, die wie folgt deklariert ist:

```
class TSimulator
{
public:     TSimulator();
                ~TSimulator();
                BOOLEAN Init(long MAXREALTIME);
                BOOLEAN Run();
                BOOLEAN Done(Result*);
                void sendMessage(TMessage&);
                float CurTime();
                void UpDateBlockMatrix(unsigned Mode,unsigned Sender,
                                       unsigned Receiver);
private:
                TEventList<TEvent>*  EventList;
                SimulationsObjekt **ModellElementArray;
                DLCheck *DL;
                BOOLEAN CreateEntity4CQN(unsigned char Klasse);
                BOOLEAN NewServiceNode(int ModellElementIndex,int AktNode);
                BOOLEAN NewQueue(int ModellElementIndex,int AktNode);
                BOOLEAN NewQuelle(int NextFreeIndex,int AktNode);
                BOOLEAN InitDynObjects();
                BOOLEAN InitDLVerwaltung();
                BOOLEAN handleMessage();
                void handleSystemMessage(TMessage&);
};
```

Die Datenelemente von TSimulator erfüllen folgende Funktion:

Datum	Erläuterung
EventList	Ereignisliste, beinhaltet alle zum aktuellen Zeitpunkt bekannten Ereignisse, die noch nicht vermittelt wurden. Außerdem verwaltet die Klasse TEventList die Simulationsuhr und stellt entsprechende Funktionen bereit.
ModellElement-Array	Dieser Vektor beinhaltet Zeiger auf alle statischen Simulationsobjekte. Er wird bei der Initialisierung erzeugt.
DL	Objekt, welches der Blockierungserkennung und -verwaltung dient.

Tabelle 4-1: Datenelemente von TSimulator

Folgende Element- oder Memberfunktionen in TSimulator spielen eine besonders wichtige Rolle:

Elementfunktion	Erläuterung
TSimulator	Konstruktor: Erzeugen der Objekte Ereignisliste, ModellElementArray, Blockierungsverwaltung
~TSimulator	Destruktor: Freigabe des durch die Datenelemente belegten Speichers.
Init	Grundlegende Funktion zur Benutzung des Simulationspaketes. Die Funktion Init trägt die Verantwortung dafür, aus einem gegebenen mathematischen Modell eine Struktur zu erstellen, die die simulative Lösung ermöglicht. Die Funktion Init erzeugt sämtliche statischen Simulationsobjekte und überträgt die entsprechenden Parameter. Sie initialisiert die Ereignisliste mit einer maximalen Modell-Simulationsdauer. Durch Aufruf der Funktion Verify wird das Modell validiert.
Run	Die Funktion Run führt die Simulation durch Aufruf der Funktion TSimulator::handleMessage durch.
handleMessage	Hauptschleife der Simulation. Ihre Grundfunktion wurde bereits im Abschnitt 2.5. dargestellt. Auf sie wird unten detaillierter eingegangen.
Done	Die Funktion Done ist für die Übertragung der Simulationsergebnisse in eine Ergebnisschnittstelle verantwortlich und - in Zusammenarbeit mit dem Destruktor - für die Speicherfreigabe.
sendMessage	Funktion, die es den Simulationsobjekten ermöglicht, Ereignisse in die Ereignisliste einzutragen.
CurTime	Liefert die aktuelle Modellzeit.
handleSystem-Message	Ermöglicht die Behandlung von Ereignissen, für die der Simulatorkern selbst zuständig ist.
Verify	Prüft die Korrektheit eines zu modellierenden mathematischen Modelles.

Tabelle 4-2: Wichtige Elementfunktion von TSimulator

Weitere Elementfunktionen betreffen die Behandlung von Blockierungen oder stellen Hilfsfunktionen für den Modellaufbau dar.
Im weiteren soll auf die Funktion TSimulator::handleMessage genauer eingegangen werden. Die Grundschleife - Entnehmen des nächsten Ereignisses aus der Ereignisliste, Weiterstellen

der Simulationszeit und Vermittlung des Ereignisses - wird solange durchgeführt, wie keine der nachstehenden Bedingungen erfüllt ist:

a. Auftreten einer unaufhebbaren Blockierung (Deadlock): Die Simulation wird beendet, die bis zum Auftreten des Deadlocks vorliegenden Ergebnisse können verwendet werden.
b. Das letzte Ereignis konnte nicht vermittelt werden.
c. Das Objekt, welches das Ereignis behandelte sollte, meldet einen Fehler.
d. Die vorgegebene maximale Modellzeit wurde erreicht (Das aktuelle Ereignis ist das Ereignis Simulationsende).
e. Die vorgegebene maximale reale Zeit wurde erreicht.
f. Es wurde eine externe Unterbrechung signalisiert.

Die Funktion von TSimulator als Schnittstellenklasse zur Nutzung des Simulationspaketes wird später in diesem Kapitel beschrieben. Im nächsten Abschnitt soll der Bedeutung der Kommunikation in einer Struktur autonomer Objekte Rechnung getragen werden, es folgt eine Erläuterung der Begriffe Nachricht und Ereignis in diesem Zusammenhang.

4.3.3. Ereignisse und Nachrichten

Der Begriff des Ereignisses wurde im Abschnitt 2.5. bereits umrissen: Ereignisse lösen Ereignisroutinen aus und bewirken somit Zustandsänderungen. Sie sind gekennzeichnet durch den Zeitpunkt ihres Eintretens und durch einen Code, der den Typ des Ereignisses bezeichnet (etwa Ankommen einer Forderung im System zum Modellzeitpunkt 3,56). Zumeist bewirkt ein Ereignis eine Zustandsänderung in genau einem statischen Simulationsobjekt und muß jenem Simulationsobjekt zugeordnet werden. Aus diesem Grund wird ein Vermittlungsmechanismus der Ereignisse notwendig. In einer Welt autonomer Objekte, wie sie der objektorientierte Ansatz erzeugt, ist demzufolge Kommunikation notwendig, um Ereignisse (und andere dynamische Simulationsobjekte - Forderungen) zu übermitteln. Dieser Tatsache ist der Begriff der Nachricht besser angemessen. Nachrichten beinhalten Ereignisse im herkömmlichen Sinn und können als Einheit von Ereignis plus notwendiger Daten zur Vermittlung betrachtet werden. Eine Nachricht ist wie folgt deklariert:

```
class TMessage
{
public:
        TMessage(unsigned int S_ID,unsigned int R_ID,MessageCode M,float MsgTime =  0.0);
        ~TMessage();

        unsigned int SenderID;
        unsigned int ReceiverID;
        MessageWhat what;
        MessageCode Msg;
        kunde* K;
        float Vermittlungszeit;
        unsigned char Parameter1;
        float Parameter2;
};
```

Eine Nachricht beinhaltet die Parameter:

1. SenderID: Identifikator des statischen Simulationsobjektes, welches die Nachricht absendet.
2. ReceiverID: Identifikator des Objektes, welches die Nachricht erhält. Ein Identifikator von Null richtet die Nachricht an den Simulatorkern (Systemmessage). Derartige Nachrichten lösen schließlich die Ereignisroutine TSimulator::handleSystemMessage aus.

3. what: Das Datenelement what kennzeichnet den Status der Nachricht. Mögliche Werte sind:
   a. *Msg_Done* - Die Nachricht wurde erfolgreich bearbeitet.
   b. *Msg_Signal* - Es handelt sich um eine Nachricht, die ein Ereignis enthält, dessen Erzeugungsdatum mit dem Eintrittszeitpunkt identisch ist. Ein Beispiel sind sogenannte Ready-Ereignisse (siehe unten), mit deren Hilfe ein Simulationsobjekt seinem Vorgänger die Bereitschaft zum Übernehmen einer Forderung signalisiert. Ein solche Nachricht wird sofort vermittelt.
   c. *Msg_Event* - Hier ist eine Nachricht definiert, die kein Signal ist. Beispielsweise wird von einem Objekt, welches für die Bedienung von Forderungen verantwortlich ist, zum Zeitpunkt t_1 eine Nachricht abgesandt, die diesem Objekt zum Zeitpunkt t_2 das Ereignis vermittelt: "Behandlung der Forderung K ist beendet".
4. Msg: Dieses Datenelement enthält die Beschreibung des mit der Nachricht verbundenen Ereignisses und definiert damit sämtliche Arten von Ereignissen, die in der Simulation möglich sind.
   Beim Entwurf des Simulationspaketes stellte sich daher die grundlegende Frage "Welche Typen von Ereignissen sind in der Simulation nötig?". Es wurde angestrebt, nur möglichst wenige verschiedene Ereignisstypen zu definieren. Unmittelbar notwendig waren Ereignistypen, die das Erzeugen, Bearbeiten und Bewegen von Forderungen symbolisieren. Zur Realisation der Blockierungsbehandlung wurden weitere zwei Ereignistypen eingeführt. Eines der letzteren und das Ereignis "Simulationsende" sind jedoch Ereignisse, die lediglich den Simulatorkern betreffen. Damit konnte die Anzahl der definierten Ereignistypen insgesamt im angestrebten Rahmen gehalten werden. Auf diesen Aspekt wurde beim Entwurf gesteigerter Wert gelegt, da sich von einer geringen Anzahl Ereignisse eine höhere Effizienz, Wartbarkeit, Änderbarkeit und eine einfachere Struktur versprochen werden kann.
   Aus diesem Grund wurde eine wichtige Information, die jedes Simulationsobjekt (mit Ausnahme der Senke) benötigt, nicht auf dem Wege der Kommunikation mittels Nachrichten realisiert. Diese Information betrifft die Frage "Ist im gewünschten Nachfolgerobjekt Platz für die zu vermittelnde Forderung vorhanden?". Die (negative) Antwort auf diese Frage bildet den Anlaß für den Abschnitt 4.5., welcher sich mit Blockierungen während des Simulationslaufes beschäftigt. Wie sollen jedoch Frage und Antwort zwischen den Objekten übermittelt werden, ohne den Nachfolger zu kennen? Die Lösung besteht in der Nutzung von Datenelementen und Funktionen, die bereits in der Klasse SimulationsObjekt definiert wurden, der Basisklasse für statische Modellelemente. Unter Nutzung des Zeigers next kann die Frage an den Nachfolger direkt gestellt werden. Eine entsprechende Lösung sieht so aus:

   ```
   Bearbeitung der Forderung wurde beendet
   if (next->is_Space()==YES)
   Weitervermittlung der Forderung
   ```

   Durch diese Möglichkeit konnte das Nachrichtenaufkommen in einem recht geringen Rahmen gehalten werden, da für jede Bewegung einer Forderung von einem Objekt zu seinem Nachfolger mindestens zwei Nachrichten weniger verarbeitet werden müssen. Anderenfalls besteht die Gefahr, daß die Ereignisliste während der Simulation sehr viele Signale enthält, die Zahl der "zeitverbrauchenden" Nachrichten dagegen relativ gering wäre. Zahlreiche Überlegungen und Versuche haben für diesen direkten Ansatz keine Nachteile in der Funktionalität ergeben, so daß abschließend die Effizienzsteigerung zu verzeichnen bleibt.
   An dieser Stelle sollen die Typen möglicher Ereignisse aufgeführt werden, die durch diese Betrachtungen während des Entwurfs gewonnen wurden:

a. *Create_Entity*: Derartige Ereignisse werden von Quellen-Objekten erzeugt und bearbeitet.
b. *Entity_Leaves*: Ein solches Ereignis kennzeichnet den Übergang einer Forderung von einem Simulationsobjekt (welches die Nachricht erzeugt) zum Nachfolger (Empfänger der Nachricht). Nachrichten, die dieses Ereignis beinhalten, sind Signale.
c. *Entity_Bearbeitet*: Dieses Ereignis wird ausschließlich von Bedienanlagen-Objekten erzeugt und bearbeitet.
d. *Ready*: Signal, welches dem Vorgängerobjekt anzeigt, daß Kapazität zur Übernahme einer Forderung vorhanden ist. Auslöser einer solchen Nachricht durch ein Objekt ist das Freiwerden von Kapazität durch Weitergeben einer Forderung. In der spezialisierten Simulationskomponente von BNETD werden Ready-Signale nur gesandt, wenn das Objekt vorher maximal ausgelastet war.
e. *Blockierungstest*: Eine Nachricht an das System in der speziellen Simulationskomponente von BNETD, welche unter bestimmten Umständen (siehe Abschnitt 4.5.) die Prüfung der Modellstruktur auf Blockierungen veranlaßt.
f. *Simulationsende*: Dieses Ereignis, welches bei der Initialisierung der Ereignisliste in selbige eingetragen wird, veranlaßt den Simulatorkern, die Simulation zu beenden.

5. Forderung(Kunde): Bestimmte Nachrichten benötigen einen Verweis auf eine Forderung. Dementsprechend kann vom absendenden Objekt der Parameter K der Nachricht mit einem Verweis auf eine Forderung belegt werden.
6. Vermittlungszeit: Die Vermittlungszeit der Nachricht entspricht dem Eintrittszeitpunkt eines Ereignisses.
7. Parameter 1 und 2: Um der Notwendigkeit der Übermittlung zusätzlicher Informationen, deren Art nicht feststeht, Rechnung zu tragen, ist in einem Nachrichtenobjekt Platz reserviert. Beispielsweise muß eine Quelle in mehrklassigen Systemen wissen, welcher Klasse eine zu erzeugende Forderung angehören soll.

Eine Nachricht durchläuft folgende Stationen:

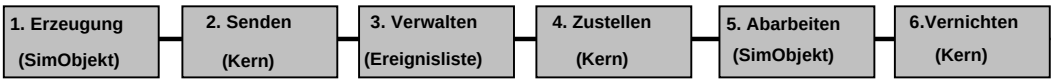

Abbildung 4-5: "Lebenslauf" von Nachrichten

ERZEUGUNG DER NACHRICHT

Nachrichten werden während der Simulation als Reaktion auf Nachrichten erzeugt. Dies geschieht in den Ereignisroutinen der (statischen) Simulationsobjekte, den virtuellen Elementfunktionen handleMessage. Zur Erzeugung von Nachrichten wird der Konstruktor der Klasse TMessage aufgerufen.
Zwei Beispiele sollen den Sachverhalt illustrieren. Das Beispiel a) stammt aus einem Objekt der Klasse Bedienanlage. Es erzeugt eine Nachricht des Inhalts "In 'Time' Zeiteinheiten ist die Bedienung der Forderung abgeschlossen". Es handelt sich hier um einen Sonderfall einer Nachricht, eine Eigenstimulanznachricht. Sender und Empfänger sind identisch. Im Beispiel b) wird das Vorgängerobjekt benachrichtigt, daß das Senderobjekt bereit zur Übernahme einer Forderung ist. Dabei wird keine Übermittlungszeit angegeben, die Nachricht ist ein Signal und wird sofort vermittelt.

a)

```
Time=create_Service_Time(OldMessage.Forderung->get_klasse());
TMessage* Answer =
                    new TMessage(getID(),getID(),ENTITY_BEARBEITET,Time);
                    Answer->set_Entity(*(OldMessage.Forderung));      // Verweis auf die Forderung
```

b)
```
TMessage* ReadyMessage =
                    new TMessage(ID,prev->getID(),READY);
```

SENDEN VON NACHRICHTEN

Für das Senden von Nachrichten ist die Funktion TSimulator::sendMessage verantwortlich. Sie wird in den Ereignisroutinen der Objekte in folgender Form aufgerufen:

```
Simulator->sendMessage(*ReadyMessage);
```

Diese Funktion veranlaßt die Einordnung der Nachricht in die Ereignisliste. Um dem Namen Ereignisliste gerecht zu werden, heißt die Klasse, die tatsächlich in die Liste eingeordnet wird, TEvent. TEvent beinhaltet die Klasse TMessage, erweitert um einen Zeiger auf das nächste Ereignis. Die Funktion TSimulator::sendMessage erzeugt mittels der Nachricht ein neues Ereignis und ruft anschließend die Funktion des Ereignislisten-Objektes auf, welche für das Einordnen des Ereignisses zuständig ist:

```
void TSimulator::sendMessage(TMessage& M)
{
    TEvent* E = new TEvent(M);
    EventList->put(E);
}
```

VERWALTEN DER NACHRICHTEN

Die Verwaltung von Nachrichten geschieht mit Hilfe der Ereignisliste im Simulatorkern. Diese stellt die Funktionen zum Ein- und Ausketten ihrer Elemente zur Verfügung. Das Objekt Ereignisliste wird im nächsten Abschnitt beschrieben.

ZUSTELLUNG DER NACHRICHTEN

Für die Zustellung der Nachrichten ist ebenfalls der Simulatorkern verantwortlich. Wie bereits erwähnt, holt die Funktion TSimulator::handleMessage das nächste Ereignis aus der Ereignisliste. Die Vermittlung geschieht mittels des im Simulatorkern geführten Vektors ModellElementArray. Hier sind alle statischen Simulationsobjekte enthalten. Stimmt die Identifikationsnummer des Empfängers der Nachricht mit der Identifikationsnummer eines dort geführten Simulationsobjektes überein, so wird die Nachricht an dieses Objekt vermittelt. Realisiert wird die Vermittlung durch den Aufruf der Funktion handleMessage des entsprechenden Simulationsobjektes. Der zugehörige Quelltext-Ausschnitt aus der Funktion TSimulator::handleMessage soll dies unterstreichen.

```
for (int i=0; ModellElementArray[i] != 0; i++)
                    {
                    if (ModellElementArray[i]->getID()==
                        Event->Message.ReceiverID)
                        {
                        if (!(ModellElementArray[i]->handleMessage(Event->Message)))
                            // Fehler aufgetreten?
                            // ansonsten fortfahren
```

ABARBEITEN DER NACHRICHT

Durch den Aufruf der Funktion SimulationsObjekt::handleMessage wird die Abarbeitung der Nachricht (bzw. des in ihr enthaltenen Ereignisses) in Gang gesetzt. An dieser Stelle bewirkt

die Nachricht eine Zustandsänderung im abarbeitenden Objekt. Auch können hier neue Nachrichten erzeugt werden, womit der Zyklus erneut beginnt.
Ist im Empfängerobjekt eine Reaktion auf diese Nachricht vorgesehen und konnte diese erfolgreich durchgeführt werden, wird die Nachricht als erledigt markiert. Jenes wird von der Funktion MessageDone erledigt, die in der Basisklasse SimulationsObjekt definiert ist.

```
void SimulationsObjekt::MessageDone(TMessage& M)
{ M.what=Msg_Done ; } ;
```

VERNICHTEN DER NACHRICHT

Das Vernichten von Nachrichten ist Aufgabe des Simulatorkernes. Wie aus dem Quelltext zur Nachrichtenzustellung ersichtlich ist, ruft der Simulatorkern die entsprechende handleMessage-Funktion zwecks Abarbeitung der Nachricht auf. Anschließend kann durch eine Prüfung, ob der Status der Nachricht "Erledigt" bedeutet, die fehlerfreie Abarbeitung der Ereignisroutine verifiziert werden. Ist die Nachricht erledigt, wird sie gelöscht und die nächste Nachricht kann aus der Ereignisliste entnommen werden. Wurde die Nachricht nicht als erledigt markiert, so muß die Simulation beendet werden.

4.3.4. Ereignisliste

Die Ereignisliste ist für eine diskrete ereignisorientierte Simulation unverzichtbar. Der Mechanismus des Ein- und Austragens von Ereignissen aus dieser Liste macht einen wesentlichen Bestandteil der Laufzeit einer Simulation aus. Die Ereignisliste ist ein Nachkomme der Klasse TList und wie folgt deklariert:

```
template <class Eventtyp>
class TEventList : private TList<Eventtyp>
{
public:
        TEventList();
        ~TEventList();
        void print();
        float get_CurrentTime()   { return CurrentTime; } ;
        long get_NumberOfEventsHandled() { return NumberOfEventsHandled; };
        Eventtyp* NextEvent();
        BOOLEAN InitFinalEvent(long MAXSIMTIME=100l);
        virtual BOOLEAN put(Eventtyp*);
private:
        float CurrentTime;
        long NumberOfEventsHandled;
};
```

Die wesentlichen Anforderungen an die Ereignisliste bestehen darin,

a. die Ereignisse speichern zu können,
b. Funktionen zum Einketten und Ausketten bereitzustellen und
c. Ereignisse ihrer Priorität nach sortieren zu können.

Anforderungen a. und b. werden bereits durch die von TList ererbte Funktionalität gewährleistet. TList unterstützt die Initialisierung und Deinitialisierung einer einfachen Listenstruktur und stellt die Funktionen *put* und *get* zur Verfügung, die Datenelemente ein- und ausketten können.
Aus der Bezeichnung "next-event-(oriented-)simulation" ist zu erkennen, daß allein diese Möglichkeiten nicht ausreichen, es muß ein Prioritätsmechanismus für die Ereignisse integriert werden. Diese Anforderung wurde realisiert, indem die virtuelle Funktion put zum Einfügen von Ereignissen überschrieben wurde. Die Möglichkeiten von C++ nutzend, kann die Priorität

zweier Ereignisse einfach mit dem Operator < verglichen werden, nachdem dieser zuvor für die enthaltenen Nachrichten entsprechend überladen wurde :

```
// Wenn die Message M1 vor der Message M2 vermittelt werden muß,
// liefert M1 < M2 true.

inline int operator < (const TMessage& M1,const TMessage& M2)
{ return((M1.Vermittlungszeit < M2.Vermittlungszeit)
  ||((M1.Vermittlungszeit==M2.Vermittlungszeit)&&(M1.Msg<M2.Msg)));
};
```

Das erste Prioritätskriterium ist dabei natürlich die Vermittlungszeit, ist diese gleich, so werden die Inhalte der Nachrichten verglichen. Die Funktion TEventList::put gewährleistet, daß zu jedem Zeitpunkt eine sortierte Ereignisliste vorliegt, mit dem nächsten eintretenden Ereignis (der nächsten zu vermittelnden Nachricht) als Kopf der Liste. Wenn die Anzahl der Elemente in der Ereignisliste mit x und die Anzahl der notwendigen Operationen für den Vergleich zweier Listenelemente mit n bezeichnet werden, so sind für das Einfügen eines neuen Elements im Mittel nx/2 Schritte notwendig. Das unterstreicht die Nützlichkeit eines geringen Nachrichtenaufkommens für die Abarbeitungsgeschwindigkeit. Untersuchungen ergaben, daß während eines Simulationslaufes etwa 30% der Laufzeit für die Abarbeitung der Funktion TEventList::put aufgewendet werden.
Die asynchrone Simulationsuhr ist ein Mitglied der Ereignisliste. Diese Einteilung erscheint sinnvoll, da die Funktionen der Ereignisliste mit der Simulationsuhr eng kooperieren. Bevor ein Element in die Ereignisliste eingefügt wird, ist der Vermittlungszeitpunkt der einzufügenden Nachricht eine relative Größe (für Signale wäre der Vermittlungszeitpunkt 0). Erst an dieser Stelle wird die aktuelle Modellzeit zur Vermittlungszeit der Nachricht addiert und somit wird eine absolute Größe erhalten. Die Operation des Auskettens des nächsten Ereignisses veranlaßt das Weiterstellen der Simulationsuhr und ist die einzige Schreiboperation für letztere.
Erwähnung verdient weiterhin der Destruktor TEventList::~TEventList(), welcher durch Löschung der enthaltenen Elemente die korrekte Speicherfreigabe nach einem Simulationslauf bewirkt.

4.3.5. Struktur von BNETD-Modellen

Mit dem Entwurf und der Implementation der Klassen von Simulationsobjekten wurde das Ziel verfolgt, möglichst vielseitige und leistungsfähige Klassen zu bilden. In der Anwendung soll es möglich sein, Objekte dieser Klassen in beliebiger Weise zu kombinieren. Mit den zur Verfügung stehenden Bausteinen Warteschlange, Bedienanlage, Verteiler und Vereiniger, Quelle und Senke sollen gleichfalls Modellstrukturen erstellt werden können, die nicht der klassischen Struktur eines Warteschlangennetzes entsprechen. Vorstellbar sind mehrere Warteschlangen oder Bedienanlagen in Reihe plaziert, Bedienknoten ohne oder mit mehreren Warteschlangen und anderes mehr. Selbstverständlich bestehen auch bei solchen Modellen Restriktionen, so ist eine Quelle als Nachfolgerobjekt oder eine Senke als Vorgängerobjekt nicht erlaubt.
Bei der Bildung eines Modelles aus den definierten Bausteinen sind die Eigenheiten der Objekte zu beachten, die im Abschnitt 4.4. beschrieben sind. Ein Beispiel wäre die Plazierung zweier Bedienanlagen unmittelbar hintereinander - sollte die Nachfolgerbedienanlage keinen Platz haben, eine von der Vorgängerbedienanlage gesendete Forderung zu übernehmen, tritt ein Fehler (oder zumindest ein Verlust der Forderung) auf. Um Blockierungen zu handhaben, sind die Klassen Verteiler (*TSplit*) und Vereiniger (*TUnify*) definiert worden.

Die unter Nutzung der Simulationsbibliothek gewonnene spezielle Lösung für BNETD weist eine weit einfachere Struktur auf: Es kann davon ausgegangen werden, daß zu einem Bedienknoten genau eine Warteschlange und genau eine Bedienanlage gehört. Um den Anforderungen der Blockierungsbehandlung und der durch mehrfache Vorgänger und Nachfolger aufge-

worfenen Probleme zu begegnen, gehört zu jedem Bedienknoten ein Vereiniger- und ein Verteilerobjekt. Die Struktur eines Knotens eines Warteschlangennetzes sieht in der Umsetzung der Simulationskomponente von BNETD wie folgt aus:

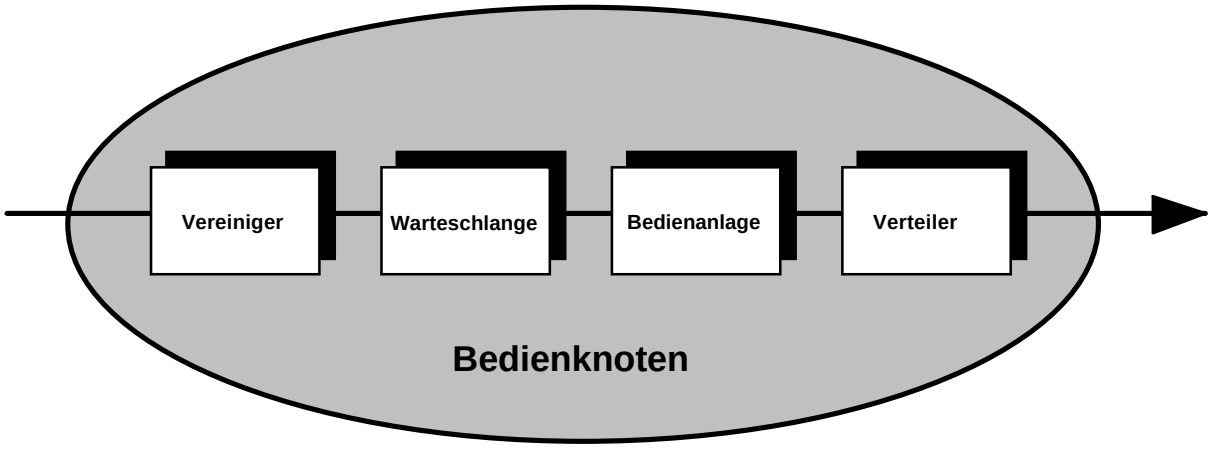

Abbildung 4-6: Modellierung eines Bedienknotens durch die Simulationskomponente

Die Umsetzung der durch das Programmsystem BNETD beschriebenen klassischen Warteschlangenmodelle in dieser Weise als Simulationsmodell soll mit "BNETD-Modell" bezeichnet werden. Es ist gekennzeichnet durch die Bestandteile Vereiniger, Warteschlange, Bedienanlage und Verteiler für **jeden** Bedienknoten.
Durch die Festlegung einer solchen Struktur kann die Leistungsfähigkeit der speziellen Simulationskomponente gegenüber einer allgemeinen aus der Simulationsbibliothek gewonnenen Struktur erhöht werden. Das Vorhandensein einer solchen BNETD-Struktur ermöglicht die etwas effizientere Abarbeitung der Simulation, die Erkennung von unaufhebbaren Blockierungen und die Führung einer ausführlicheren Statistik Blockierungen betreffend.

4.4. Beschreibung der Klassen

4.4.1. Unterteilung der Klassen

Die für die Simulation entworfenen Klassen lassen sich in Gruppen einteilen:

- Klassen für statische Simulationsobjekte wie Warteschlangen oder Bedienanlagen
- Klassen für dynamische Simulationsobjekte wie Forderungen und Nachrichten
- Unterstützende Klassen, die die Basis für die Definition anderer Klassen bilden oder Datenelemente anderer Klassen sind
- Klassen des Simulatorkernes

Letztere sollen dabei nicht gesondert erläutert werden, eine Beschreibung ist im vorhergehenden Abschnitt zu finden. Zum Simulatorkern werden die Klassen TSimulator, TEventList sowie eine Klasse zur Behandlung von Blockierungen gezählt.
Begonnen werden soll mit der Beschreibung der unterstützenden Klassen.

4.4.2. Hilfsklassen

Der objektorientierte Entwurf fördert die Modellierung realer Objekte als Softwareobjekte. Für die Realisierung eines Softwareprojektes sind aber allein diese aus der Anschauung gewonnenen Objekte zumeist nicht ausreichend. Es entsteht die Notwendigkeit des Entwurfs von Klassen, deren Widerspiegelung in der Realität nicht oder nur versteckt vorhanden ist.
Im entwickelten Simulationspaket zählen hierzu eine Basisklasse für die Listenverwaltung, Klassen von Zufallszahlengeneratoren und Statistik-Klassen, die nun näher betrachtet werden sollen.

LISTENVERWALTUNG

Die Basisklasse zur Listenverwaltung, TList, definiert bereits eine einfache Liste mit den entsprechenden Datenelementen und Funktionen. Ihre Deklaration lautet:

```
template <class ListenElement>
class TList
{
protected:
                ListenElement *first;
                ListenElement *last;
                int length;
public:
                TList();
                ~TList();
                ListenElement* get();
                virtual BOOLEAN put(ListenElement*);
                virtual void print(){};
};
```

Die Funktion get liefert das erste Element der Liste zurück und löscht dieses aus der Liste, die virtuelle Funktion put fügt ein Element am Ende der Liste ein. Damit verfügt diese Basisklasse bereits über die Funktionalität einer FIFO-Warteschlange.
Die Deklaration von TList als Schablonenklasse (*template*) erweist sich als nützlich, TList stellt die Basis für die spezialisierten Nachkommen TEventList, TRoutingList und BasisQueue dar.
Der Destruktor TList::~TList übernimmt die Speicherfreigabe für TList und der von TList abgeleiteten Klassen.

ZUFALLSZAHLENGENERATOREN

Zufallszahlengeneratoren sind für die Simulation von existentieller Bedeutung. Wie im Kapitel 2 bereits erwähnt, wurde für das hier vorgestellte Simulationspaket ein multiplikativer linearer Kongruenzgenerator mit den Parametern $a=7^5=16807$ und $m=2^{31}-1=2147483647$ verwendet. Der Basisgenerator ist in nachstehender Weise deklariert:

```
class TRNDGeneratorBasis
{
protected:
        static long seed;                   // Startwert, der bei Initialisierung erzeugt wird
        unsigned long m;                    // Periode
        unsigned long a;                    // Multiplikator
        float Reziprok_m;                   // Zur schnelleren Berechnung: 1/m
public:
        TRNDGeneratorBasis();               // Konstruktor
        virtual ~TRNDGeneratorBasis();      // Destruktor
        float random();                     // Gleichverteilte ZZ in [0,1)
        virtual float RND() = 0;            // liefert Zufallszahl entsprechend der gewünschten Verteilung
};
```

Mit Hilfe der statischen Membervariablen *seed* wird die Initialisierung aller Zufallszahlengeneratoren durchgeführt. Die Initialisierung erfolgt durch Auslesen der Systemzeit und ist nicht portabel. Der Startwert *seed* wird nur einmal, beim erstmaligen Aufruf des Konstruktors TRNDGeneratorBasis::TRNDGeneratorBasis, initialisiert und nicht für alle im System existierenden Zufallszahlengeneratoren separat. Auch beim mehrfachen Aufruf der Simulationskomponente von BNETD wird diese Variable nur einmal initialisiert. Von der Variablen *seed* existiert immer nur eine Kopie, auch wenn während der Simulation mehrere Zufallszahlengeneratoren-Objekte vorhanden sind. Sämtliche Generatoren greifen stets auf die einzige Kopie von

seed zu. Konsequenz ist die "Aufteilung" einer Folge von Pseudozufallszahlen zwischen mehreren Generatoren. Das Abhängigkeitsverhalten der erzeugten Pseudozufallszahlen wird dadurch nicht negativ beeinflußt.
Anmerkung:
Eine vollständige Abhängigkeit zwischen zwei Zufallszahlenfolgen würde der Fall sein, wenn zwei Generatoren jeweils eine separate Initialisierung durch die Systemzeit unmittelbar hintereinander durchführten. Möglicherweise stimmt dann der jeweilige private Initialisierungs-Wert überein und folglich auch die Reihe der erzeugten Zufallszahlen.

Die Funktion random erzeugt nach der multiplikativen Kongruenzenmethode gleichverteilte Pseudozufallszahlen in [0,1).
Die abstrakte virtuelle Funktion RND wird in den abgeleiteten Generatorenklassen definiert. Sie transformiert die durch TRNDGeneratorBasis::random erzeugten Pseudozufallszahlen in verschiedene Verteilungen. Diese Transformation wurden analog der Beschreibung im Kapitel 2 implementiert. Die Basisklasse TRNDGeneratorBasis ist aufgrund der Funktion RND abstrakt, die Bildung von Objekten dieser Klasse ist nicht erlaubt.

Nachkommen der Basisklasse sind Generatoren, welche exponentialverteilte, erlangverteilte, gleichverteilte, deterministische und normalverteilte Zufallszahlen liefern. Der deterministische Generator wurde der Systematik wegen in die Hierarchie der Zufallszahlengeneratoren aufgenommen. Im folgenden sind die Konstruktoren dieser Generatoren angegeben:

```
TRNDGeneratorGV(float a,float b);                 // Gleichverteilung in [a,b)
TRNDGeneratorEXP(float Erwwert);                  // Exponentialverteilung
TRNDGeneratorNormal(float Erwwert,float Streu);   // Normalverteilung
TRNDGeneratorErlang(float Erwwert,int k);         // Erlangverteilung - Erbe der Exponentialverteilung
TRNDGeneratorDet(float ConstValue = 1.0);         // Deterministischer Generator
```

Allen diesen Generatoren ist gemeinsam, daß sie die virtuelle Funktion RND entsprechend der notwendigen Transformation überschreiben.

Die Verwendung der Generatoren erfolgt in der Weise, daß in Objekten, welche stochastisch arbeiten, ein Objekt der Klasse TRNDGeneratorBasis als Datenelement vorgesehen wird. Der entsprechende spezialisierte Generator wird bei der Initialisierung des Simulationsmodells dynamisch eingefügt.
Pseudozufallszahlengeneratoren werden von Objekten der Klassen RandomQueue, TSplit, Quelle und Bedienanlage benötigt (siehe auch Anhang E: Objekthierarchie).

STATISTIK-KLASSEN

Die meisten Klassen der Simulationsbibliothek, darunter alle statischen Simulationselemente, führen eine Statistik, um einerseits Erkenntnisse bezüglich des Modellverhaltens zu ermöglichen sowie ihre Funktion zu dokumentieren.
Für bestimmte Klassen erfüllt bereits eine einfache Zählvariable die gewünschte Statistikfunktion, für andere Klassen ist die zu führende Statistik recht komplex. Letztere Klassen - Bedienanlage und Warteschlange - enthalten deshalb Objekte zur Statistikführung. Die Statistikklasse für Warteschlangen enthält folgende Daten:

```
class WS_Statistik
{
private:
        int AnzKlassen;
        float *LastRefresh;
        long GesCurrentLength;                  // Aktuelle Gesamtlänge
```

```
        float *AverageLength;
        float *AverageWaitingTime;
        long MaxLength;
        long *CurrentLength;                    // Aktuelle Länge pro Klasse
        long *NumberOfCountedEntities;          // diese waren in der WS
        long *TotalNumberOfEntities;            // diese sind durchgelaufen
public:
        WS_Statistik(unsigned char Classes=MAXCLASS);
        ~WS_Statistik();
        void print();
        void refreshAll(float CurTime,float EntryTimeOfEntity,unsigned char AClass);
        void refreshLength(float CurTime,unsigned char AClass,WSEventType);
        void IncreaseTotalNumber(int AClass) { TotalNumberOfEntities[AClass]++; };
};
```

Auf die Berechnung ausgewählter Statistikgrößen wird im Abschnitt 4.6. eingegangen. Eine weitere Klasse, die in diese Gruppe fällt, ist die Schnittstellenklasse *Result*, welche die Ergebnisse der Simulation aufnehmen kann. Diese wird ebenfalls im Abschnitt 4.6. erörtert.

4.4.3. Statische Simulationselemente

Zunächst sollte zur Begriffsklärung angemerkt werden, daß statische Simulationsobjekte streng genommen nicht statisch sind. Sie verfügen über eine dynamische Komponente, ihren Zustand, den sie im Verlauf der Simulation oft ändern. Der Begriff "statisches Simulationsobjekt" drückt dagegen aus, daß derartige Objekte vor Beginn des Simulationslaufes erzeugt werden und erst nach Beendigung der Simulation wieder verschwinden.
Die Klassenhierarchie der statischen Simulationsobjekte wurde bereits im Abschnitt 4.2. wiedergegeben, ebenso die Basisklasse *Simulationsobjekt*. An dieser Stelle soll die Funktionsweise ausgewählter Klassen dieser Hierarchie näher betrachtet werden. Besonderes Gewicht wird auf die virtuellen Funktionen ***handleMessage*** und ***is_Space*** gelegt, jene sind maßgeblich für das Verhalten des Objektes.
Auf die Darstellung der Deklarationen dieser Klassen wird an dieser Stelle verzichtet, sie sind jedoch dem Anhang zu entnehmen. Bevor mit der Betrachtung der Klasse Quelle begonnen wird, sollen die genutzten Übersichten erläutert werden:

- In den Zeilen 1 und 2 der Tabellen steht der Name der Klasse und eine kurze Erläuterung.
- Zeile 3 (Genutzte Klassen) enthält die Klassen, die entweder Basisklasse oder Datenelemente der betrachteten Klasse darstellen.
- Zeile 4 (Nachkommen) enthält bereits abgeleitete Klassen.
- Zeile 5 (Mögliche Vorgänger) enthält diejenigen Objekte, die in einem Simulationsmodell formal als Vorgänger der Klasse möglich sind.
- Zeile 6 (Mögliche Nachfolger) enthält analog dazu mögliche Nachfolge-Objekte, die einen fehlerfreien Lauf der Simulation **garantieren**.
- Zeile 7 (Behandelte Ereignisse) führt die Ereignistypen auf, die von der Ereignisroutine, der Funktion handleMessage, verarbeitet werden.
- Zeile 8 (Erzeugte Ereignisse) führt die von der Ereignisroutine erzeugten Ereignistypen auf.
- Zeile 9 (Neue Datenelemente) enthält wesentliche Erweiterungen gegenüber der Basisklasse Simulationsobjekt.
- Zeile 10 (Platzanfrage an Objekt) erfaßt die möglichen Antworten, die ein Objekt der Klasse auf die Anfrage is_Space liefert.
- Zeile 11 (Platzanfrage von Objekt) beinhaltet schließlich die Antworten, die ein Klassenobjekt erwartet, wenn es die Anfrage is_Space an den Nachfolger stellt. Diese und die vorhergehende Zeile sind das Kriterium für die Bestimmung möglicher Vorgänger und Nachfolger der Modellbausteine. Auf diese Weise wird die Struktur möglicher Modelle eingegrenzt.

Klassenname	Quelle
Kurzbeschreibung	Forderungsquelle
Genutzte Klassen	SimulationsObjekt, TRNDGeneratorBasis
Nachkommen	-
Mögliche Vorgänger	-
Mögliche Nachfolger	TUnify, TSplit,(Warteschlange)
Behandelte Ereignisse	Create_Entity, Ready
Erzeugte Ereignisse	Create_Entity, Entity_Leaves
Neue Datenelemente	Zufallszahlengeneratoren
Platzanfrage an Objekt	-
Platzanfrage von Objekt	YES, WAIT_FOR_READY

Tabelle 4-3: Charakterisierung der Klasse Quelle

Die Klasse Quelle ist während der Simulation offener Systeme für die Erzeugung von Forderungen zuständig, in geschlossenen Systemen ist sie überflüssig. Eine Quelle erzeugt stets Forderungen mit einem Ankunftszeitabstand genau einer Verteilungsfunktion entsprechend. Die Art des Ankunftsstromes wird während der Initialisierung festgelegt, indem ein entsprechender Zufallszahlengenerator integriert wird. Für verschiedene Forderungsklassen können verschiedene Parameter festgelegt werden. Einem abstrakten Modell, in welchem Ankunftsströme zu verschiedenen Knoten definiert sind, entspricht ein Simulationsmodell mit mehreren Quelle-Objekten. Ein Objekt der Klasse Quelle erwartet nach Erzeugung einer Forderung, daß der Nachfolger Platz für diese Forderung bietet, daher kommen nur Objekte der Klassen TUnify, TSplit und Warteschlangen mit unbegrenzter Kapazität als Nachfolger infrage, wenn die Simulation garantiert fehlerfrei laufen soll.
Die Funktion Quelle::handleMessage reagiert auf die Ereignistypen Create_Entity und Ready. Nach Erzeugung einer Forderung wird diese zum Nachfolger geschickt. Gleichzeitig wird eine Eigenstimulanznachricht erzeugt. Diese bewirkt nach einer stochastischen Zeitspanne die Erzeugung einer weiteren Forderung. Ein Quelle-Objekt arbeitet wie ein autonomer Automat, der - einmal angestoßen - unabhängig von seiner Umwelt nach stochastischen Zeitabschnitten Forderungen erzeugt und versendet. Das "Anstoßen" der Quelle ist Aufgabe der Funktion TSimulator::Init.
Wie alle statischen Simulationsobjekte verfügt ein Objekt der Klasse Quelle über die Fähigkeit, sich selbst auszugeben. In einem Quelle-Objekt wird eine Statistik über die Anzahl der erzeugten Forderungen pro Klasse geführt und der mittlere Ankunftszeitabstand festgehalten.

Klassenname	Senke
Kurzbeschreibung	Forderungssenke
Genutzte Klassen	SimulationsObjekt
Nachkommen	-
Mögliche Vorgänger	Quelle, Warteschlange, Bedienanlage, TSplit, TUnify
Mögliche Nachfolger	-
Behandelte Ereignisse	Entity_Leaves
Erzeugte Ereignisse	-
Neue Datenelemente	-
Platzanfrage an Objekt	YES
Platzanfrage von Objekt	-

Tabelle 4-4: Charakterisierung der Klasse Senke

Die Klasse Senke vernichtet während des Simulationslaufes dynamische Simulationselemente - Forderungen. Die Senke repräsentiert in Gemeinschaft mit der Klasse Quelle die Außenwelt des Modells.

Die Funktion Senke::handleMessage kennt nur ein Ereignis, auf welches sie reagiert - das Übersenden einer Forderung. Eine Platzanfrage an die Senke wird stets positiv beantwortet.
Ein Objekt der Klasse Senke führt eine Statistik über die Anzahl der vernichteten Forderungen pro Forderungsklasse und die mittlere Verweilzeit im System (ebenfalls klassenabhängig).

Zur Modellierung von Warteschlangen existiert eine Klassenhierarchie, welche ausgehend von einer Basisklasse *BasisQueue* Warteschlangenklassen definiert, die sich durch ihre Entnahmestrategie (FIFO, LIFO, Random, Priority) unterscheiden. Es folgt die Beschreibung der Basisklasse der Warteschlangenhierarchie:

Klassenname	BasisQueue
Kurzbeschreibung	Basisklasse für Warteschlangenobjekte
Genutzte Klassen	SimulationsObjekt,TList,(TRNDGeneratorBasis)
Nachkommen	FIFOQueue,LIFOQueue,PriorityQueue,RandomQueue
Mögliche Vorgänger	TUnify,TSplit,Warteschlange
Mögliche Nachfolger	Bedienanlage,Warteschlange,Senke
Behandelte Ereignisse	Entity_Leaves, Ready
Erzeugte Ereignisse	Entity_Leaves, Ready
Neue Datenelemente	Statistik für Warteschlangen
Platzanfrage an Objekt	YES, NO
Platzanfrage von Objekt	YES, NO

Tabelle 4-5: Charakterisierung der Klasse BasisQueue

Warteschlangenobjekte bilden die Funktion einer realen Warteschlange nach. Sie sind in der Lage, die Verantwortung für durch das System wandernde Forderungen zu übernehmen und sie - wenn nötig - zu puffern. Warteschlangenklassen stammen von den Basisklassen TList und SimulationsObjekt ab. Zusätzlich wird von der abgeleiteten Klasse RandomQueue ein Zufallszahlengenerator-Objekt genutzt. Die Flexibilität der Listenklassen wurde beibehalten, auch die Warteschlangenklassen wurden als Templates deklariert.
Die Funktion BasisQueue::handleMessage reagiert auf die Ereignisse Entity_Leaves und Ready. Das Ereignis Entity_Leaves übergibt einem Warteschlangenobjekt eine Forderung. Sollte der Nachfolger Übernahmekapazität bereithalten, kann die Forderung sofort weitergegeben werden, indem das Warteschlangenobjekt seinerseits ein Entity_Leaves-Ereignis erzeugt. Ein ankommendes Ready-Signal interpretiert die Funktion BasisQueue::handleMessage als Nachricht, daß der Nachfolger bereit zur Übernahme einer Forderung ist. Die Reaktion besteht im Weiterleiten einer sich in der Warteschlange befindlichen Forderung. Welche Forderung dabei weitergeleitet wird, wird durch die Warteschlangenstrategie bestimmt. Die Realisierung der verschiedenen Warteschlangenstrategien erfolgt durch das Überschreiben der virtuellen Funktion put, die von der Basisklasse TList ererbt wurde, in den von BasisQueue abgeleiteten Klassen.
Weiterhin wird durch das Verlassen der Warteschlange ein Ready-Signal an den Vorgänger ausgelöst, um ihm das Freiwerden eines Platzes in der Warteschlange anzuzeigen. Eine derartige Botschaft ist nur dann notwendig, wenn die Warteschlange zuvor vollständig besetzt war.
Die Funktion BasisQueue::is_Space liefert eine positive oder negative Antwort auf eine Platzanfrage. Eine negative Antwort (NO) wird nur dann zurückgegeben, falls die Warteschlange zum Zeitpunkt der Anfrage vollständig ausgelastet ist.
Die zu erstellenden Statistiken für ein Warteschlangenobjekt besitzen einen Umfang, der die Einführung einer Klasse WS_Statistik sinnvoll erscheinen läßt. Klassenabhängige Statistiken werden für die mittlere Länge und die mittlere Wartezeit geführt.
Im weiteren soll die Klasse Bedienanlage näher beleuchtet werden:

Klassenname	Bedienanlage
Kurzbeschreibung	Entsprechung einer realen Bedienanlage
Genutzte Klassen	SimulationsObjekt,TRNDGeneratorBasis
Nachkommen	-
Mögliche Vorgänger	TUnify,TSplit,Warteschlange
Mögliche Nachfolger	TUnify,TSplit,Senke
Behandelte Ereignisse	Entity_Leaves, Ready, Entity_Bearbeitet
Erzeugte Ereignisse	Entity_Leaves, Ready, Entity_Bearbeitet
Neue Datenelemente	Statistik für Bedienanlagen, Zufallszahlengeneratoren
Platzanfrage an Objekt	YES, NO
Platzanfrage von Objekt	YES, WAIT_FOR_READY

Tabelle 4-6: Charakterisierung der Klasse Bedienanlage

Objekte der Klasse Bedienanlage sollen realen Bedienanlagen in der Funktion entsprechen. Bedienanlagen führen zeitkonsumierende Operationen über Forderungen aus, sie bedienen diese Forderungen.
Solche Objekte sind in der Lage, den Lauf einer Forderung durch das Simulationsmodell entsprechend einer vorgegebenen mathematischen Funktion zu verzögern. Die Art der Verteilung wird - in Analogie zur Quelle - während der Initialisierung festgelegt. Alle Forderungsklassen besitzen innerhalb eines Bedienanlagen-Objektes die gleichen Verteilungsfunktionen, allerdings mit möglicherweise unterschiedlichen Parametern. Die Initialisierung erfolgt bequem mit Hilfe des überladenen Konstruktors der Klasse Bedienanlage. Zur Ermittlung der stochastischen Bedienzeiten werden die Zufallszahlengeneratoren-Klassen genutzt. Bedienanlagen können mehrkanalig sein.
Die Funktion Bedienanlage::handleMessage reagiert auf die Ereignisse Entity_Leaves, Ready und Entity_Bearbeitet:

- Für eine ankommende Forderung (Ereignis Entity_Leaves) wird eine stochastische Bedienzeit ermittelt, anschließend eine Eigenstimulanz-Nachricht gesendet, die das Bedienanlagen-Objekt wieder aktiviert, wenn diese Bedienzeit abgelaufen ist. All dieses geschieht unter der Voraussetzung, daß in der Bedienanlage mindestens ein freier Bedienkanal zur Verfügung steht, sonst wird die Simulation fehlerhaft beendet (Die Prüfung, ob in einer Bedienanlage Bearbeitungskapazität frei ist, wird vom Vorgänger durch Aufruf der Funktion is_Space realisiert).
- Trifft ein Ereignis Entity_Bearbeitet in der Bedienanlage ein, so wird zunächst die Anfrage nach vorhandener Kapazität im Nachfolger gestellt. Kein Platz im Nachfolger erzeugt einen Fehler. Antwortet der Nachfolger mit YES, so kann die Forderung mittels einer Entity_Leaves-Nachricht weitergesendet werden, und der benutzte Bedienkanal wird freigegeben. Die Antwort WAIT_FOR_READY dagegen ist die Antwort der ausgehend von der Klasse TRouter gebildeten Objekte. Sie bewirkt, daß eine Forderung zwar an den Nachfolger weitergegeben werden kann, aber der Bedienkanal noch nicht freigegeben wird. Die Freigabe des Bedienkanals erfolgt erst dann, wenn ein Ready-Signal empfangen wird. Auf diese Weise wird im Zusammenspiel von Bedienanlagen- und TRouter-Objekten eine virtuelle Ausgangswarteschlange in der Bedienanlage geschaffen, die das korrekte Verhalten bei Blockierungen ermöglicht. Nach Freigabe eines Bedienkanals erhält auch der Vorgänger des Bedienanlagen-Objektes eine Ready-Botschaft.

Die Funktion Bedienanlage::is_Space liefert in Abhängigkeit von der Momentanbelegung der Bedienkanäle YES oder NO als Antwort, dagegen erwartet ein Bedienanlagen-Objekt von seinem Nachfolger die Antworten YES oder WAIT_FOR_READY.

Der Umfang der notwendigen Statistik gab den Anlaß zur Definition einer Statistikklasse für Bedienanlagenobjekte. Diese enthält unter anderem (jeweils pro Klasse) die mittlere Anzahl genutzter Kanäle, die mittlere Bedienzeit und die mittlere Dauer von Blockierungen.

Wie im Abschnitt 4.2. beschrieben, ist die Notwendigkeit des Hindurchleitens der Forderungen durch das Simulationsmodell der Anstoß für die Entwicklung von Klassen, welche als Verbindungselement zwischen Warteschlangen, Bedienanlagen, Quellen und Senken dienen. Basis für die Verteilerklassen ist die abstrakte Klasse TRouter, von ihr abgeleitet sind die Klassen TUnify und TSplit. Die Abstammungsverhältnisse sind in der nachstehenden Abbildung skizziert:

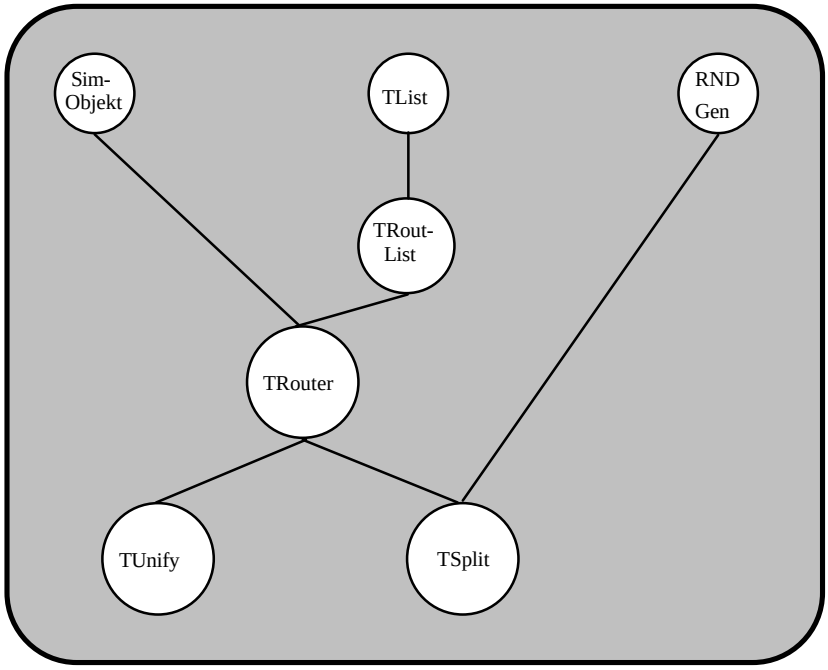

Abbildung 4-7: Abstammung der Klassen TUnify und TSplit

Die von der Klasse TRouter abstammenden Klassen ("Verteilerklassen") verfügen über folgende Besonderheiten:

a. Verwaltung mehrerer Vorgänger oder mehrerer Nachfolger und
b. Fähigkeit zum Puffern von Forderungen.

Die Klasse TRouter definiert einen Vektor von Adressen anderer Simulationsobjekte. Nach entsprechender Initialisierung anhand der im formalen Modell gegebenen Übergangswahrscheinlichkeiten wird somit Eigenschaft a. (siehe oben) ermöglicht.
Für die Behandlung der Blockierungen notwendig ist die Einrichtung von Ausgangswarteschlangen in Bedienanlagen-Objekten. Diese Funktion erfüllen Verteilerobjekte, ihre Fähigkeit zur Pufferung von Forderungen "simuliert" eine Ausgangswarteschlange im Vorgänger-Objekt (Eigenschaft b.). Diese Pufferung geschieht unter Nutzung der Klasse TRoutingList, eines spezialisierten Nachkommens der Basisklasse TList. Die so geschaffene virtuelle Ausgangswarteschlange verwaltet wartende Forderungen physisch, logisch läuft die Simulation allerdings so, als würde sich die Forderung im Vorgängerobjekt befinden. Beispielsweise scheint die Forderung einen Bedienkanal weiterhin zu belegen, obwohl ihre Bearbeitung bereits beendet ist. Detaillierter wird auf die Klasse TRoutingList im Abschnitt 4.5. eingegangen. Eine Statistik wird in der Klasse TRouter nur in eingeschränktem Umfang geführt, sie ist auf die Zählung der gepufferten Forderungen beschränkt.
Mit der Klasse TUnify soll die Beschreibung der Verteilerklassen begonnen werden:

Klassenname	TUnify
Kurzbeschreibung	Verteilerklasse, kann mehrere Vorgänger haben
Genutzte Klassen	TRouter,TRoutingList
Nachkommen	-
Mögliche Vorgänger	TUnify,TSplit,Bedienanlage,Quelle
Mögliche Nachfolger	TUnify,TSplit,Bedienanlage,Warteschlange,Senke
Behandelte Ereignisse	Entity_Leaves, Ready
Erzeugte Ereignisse	Entity_Leaves, Ready, Blockierungstest
Neue Datenelemente	Array von Simulationsobjekten
Platzanfrage an Objekt	WAIT_FOR_READY
Platzanfrage von Objekt	YES, NO, WAIT_FOR_READY

Tabelle 4-7: Charakterisierung der Klasse TUnify

Die Klasse TUnify enthält Modellbausteine, deren Aufgabe es ist, mehrere Vorgänger zu verwalten. Von ihrer Basisklasse TRouter ererbt sie die Fähigkeit zur Pufferung von Forderungen. Entsprechend der im Abschnitt 4.3.5. erläuterten Modellstruktur von "BNETD-Modellen" ist ein TUnify-Objektes unmittelbar vor dem Warteschlangen-Objekt eines Bedienknotens sinnvoll zu plazieren.
Eine weitere von TRouter ererbte Besonderheit ist die Antwort von TUnify auf eine is_Space-Anfrage. Der Rückgabewert der Funktion is_Space ist in jedem Falle die Antwort WAIT_FOR_READY, die -wie erwähnt- dem Vorgänger die Übergabe einer Forderung bereits erlaubt, obwohl sie sich (anschließend) logisch noch im Vorgänger befindet. Auf den genauen Ablauf der Kommunikation wird im Abschnitt 4.5. eingegangen.
Die Funktion TUnify::handleMessage reagiert auf die Ereignisse Entity_Leaves und Ready. Wird einem TUnify-Objekt eine Forderung mittels Entity_Leaves übergeben, so wird zunächst geprüft, ob der Nachfolger Kapazität für diese Forderung bereitstellt. Hierbei ergeben sich drei Möglichkeiten:

1. YES: Es ist Platz vorhanden. Die Forderung wird ohne Zeitverzug weitergeleitet und gleichzeitig der ursprüngliche Sender benachrichtigt (Ready-Signal). Letzterer gibt als Reaktion die logische Verantwortung für diese Forderung an den Nachfolger des TUnify-Objektes ab.
2. NO: Es ist kein Platz vorhanden. Die Forderung wird im TUnify-Objekt gepuffert und der ursprüngliche Sender nicht von seiner logischen Zuständigkeit für diese Forderung befreit. Es ist eine Blockierung aufgetreten. Unter bestimmten Rahmenbedingungen (siehe Abschnitt 4.5.) wird der Simulatorkern aufgefordert, sich auf eine unaufhebbare Blockierung zu prüfen.
3. WAIT_FOR_READY: Nachfolger des TUnify-Objektes ist ein weiteres TRouter-Objekt. Da dieses Objekt gleichfalls Forderungen puffern kann, wird die Forderung weitergeleitet. Als Absender dieser Entity_Leaves-Nachricht wird jedoch der Vorgänger von TUnify angegeben. Damit zieht sich das TUnify-Objekt aus der Kommunikation zwischen TUnify-Vorgänger⇔TUnify⇔TUnify-Nachfolger zurück. Die Forderung wird physisch im Nachfolgerobjekt von TUnify verwaltet, logisch jedoch im Vorgängerobjekt von TUnify.

Erhält ein TUnify-Objekt ein Ready-Signal, so reagiert es in folgender Weise: Befinden sich im Puffer wartende Forderungen, so wird eine Forderung entnommen und weitergeleitet. Die Entnahmestrategie aus dieser virtuellen Ausgangswarteschlange ist FIFO. Gleichzeitig wird durch ein Ready-Signal der logische Eigentümer der Forderung von seiner Verantwortlichkeit für die Forderung befreit. Sollten keine wartenden Forderungen gepuffert sein, so wird die Nachricht als erledigt markiert.

Das letzte zu beschreibende statische Simulationsobjekt sollen Objekte der Klasse TSplit sein.

Klassenname	TSplit
Kurzbeschreibung	Verteilerklasse,Verzweigungsfunktion
Genutzte Klassen	TRouter,TRoutingList,TRNDGeneratorGV
Nachkommen	-
Mögliche Vorgänger	TUnify,TSplit,Bedienanlage,Quelle
Mögliche Nachfolger	TUnify,TSplit,Warteschlangen,Bedienanlage,Senke
Behandelte Ereignisse	Entity_Leaves,Ready
Erzeugte Ereignisse	Entity_Leaves,Ready,Blockierungstest
Neue Datenelemente	Array von Simulationsobjekten, Gleichverteilungsgenerator, Übergangsmatrix
Platzanfrage an Objekt	WAIT_FOR_READY
Platzanfrage von Objekt	YES, NO, WAIT_FOR_READY

Tabelle 4-8: Charakterisierung der Klasse TSplit

Aufgabe der Klasse TSplit ist es, die Verzweigung von Forderungen an mehrere Nachfolger entsprechend einer bestimmten Strategie zu gewährleisten. Die Klasse TSplit implementiert nur eine Strategie der Verzweigung - die Verzweigung nach Wahrscheinlichkeiten, welche für Warteschlangensysteme relevant ist. Um das Anwendungsspektrum zu verbreitern, sind weitere Verzweigungsstrategien denkbar:

- Zyklische Verzweigung an mögliche Nachfolger: Alle Nachfolger werden in gleicher Häufigkeit und gleicher Reihenfolge bedient.
- Ausweichprinzip: Die Forderung wird an den ersten freien Nachfolger vermittelt.

Die implementierte Verzweigungsstrategie nach Wahrscheinlichkeiten macht die Definition einer Übergangsmatrix notwendig. Diese enthält die klassenabhängigen Übergangswahrscheinlichkeiten zu den Nachfolgerobjekten und wird durch die Init-Funktion des Simulatorkerns initialisiert.
Weiterhin verfügt TSplit als Nachkomme von TRouter über die gleichen Fähigkeiten zur Pufferung von Forderungen wie TUnify (siehe oben). Die Realisierung der Funktion handleMessage ähnelt der der Klasse TUnify, deshalb sollen nur einige wichtige Unterschiede herausgestellt werden.
Denkbar ist der Fall, daß eine übergebene Forderung nicht zu einem Nachfolger weitergeleitet werden kann, der zufällig ermittelt wurde. Die Klasse TSplit verfährt nun so, daß sie diese Forderung puffert und dabei einen Vermerk über Absender und gewünschtes Ziel hinzufügt. Erhält ein TSplit-Objekt ein Ready-Signal, so durchsucht es seinen Puffer mit Hilfe der Funktion find_entry, die als Argument die ID-Nummer des gewünschten Empfängers erhält. Das heißt, eine Forderung hält an einem einmal ermittelten Nachfolger fest, sollte dieser blockiert sein, so wartet die Forderung - ungeachtet freier möglicher anderer Nachfolger. Daraus ergibt sich die Idee einer Kombination von Verzweigung nach Wahrscheinlichkeit und dem oben erwähnten Ausweichprinzip, die vorliegende Implementation bleibt jedoch bei der "reinen" Verzweigung nach Wahrscheinlichkeiten.
Die Statistik der TSplit-Objekte zählt die Anzahl der verzweigten Forderungen pro Nachfolger und Klasse und ermöglicht so die Verifikation der Verzweigungsfunktion.
Anzumerken ist, daß innerhalb der Simulationskomponente von BNETD ein TSplit-Objekt nie seinen Puffer benutzen wird, da als Nachfolger von TSplit stets ein TUnify-Objekt (puffert selbst) oder ein Senke-Objekt (kann keine Blockierung verursachen) definiert ist.

4.4.4. Dynamische Simulationselemente

Als dynamische Simulationselemente sollen diejenigen Simulationselemente bezeichnet sein, die während des Simulationslaufes dynamisch erzeugt und vernichtet werden. Dazu zählen Nachrichten und Forderungen.
Die Definition der Nachrichtenklasse TMessage wurde in einem früheren Abschnitt bereits erläutert, daher soll hier vor allem die Klasse der Forderungen betrachtet werden - sie ist wie folgt definiert:

```
class kunde                          // Synonyme: Kunde,Forderung,Entity,Werkstück,Job
{
private:
        long number;
        unsigned char klasse;
        unsigned int priority;
        float EntryTime;
public:
        kunde(float ATime,unsigned char = 0,unsigned int = 0);
        ~kunde();
        virtual void print();
        unsigned int get_priority() { return priority ; } ;
        unsigned char get_klasse()  { return klasse ; } ;
        float getEntryTime()    { return EntryTime; } ;
        long get_number() { return number ; } ;
};
```

Eine Forderung wird beschrieben durch eine fortlaufende Nummer, die Forderungsklasse, eine Angabe der Priorität und den Eintrittszeitpunkt in das System. Forderungen (auch Kunden, Jobs, Werkstücke, Entities) sind die passiven Bestandteile einer solchen ereignisorientierten Simulation, die Aktivitäten gehen während der Simulation von den statischen Simulationsobjekten (siehe vorhergehender Abschnitt) aus.

4.5. Behandlung von Blockierungen

4.5.1. Blockierungen

Ein wesentlicher Vorteil der simulativen Lösung eines Warteschlangenmodells besteht in der Tolerierung und Behandlung von Blockierungen. Es besteht die Möglichkeit, die Reaktion des Simulationsmodells auf auftretende Blockierungen zu prüfen. Zu diesem Zweck können Blokkierungen sogar bewußt herbeigeführt werden.

BEGRIFF DER BLOCKIERUNG

Unter einer Blockierung soll hier der Fall verstanden werden, daß der gewünschte Adressat einer Forderung zum aktuellen Zeitpunkt nicht in der Lage ist, die Forderung zu übernehmen. Dann soll der Absender der Forderung als (momentan) **blockiert** gelten. Ein solcher blockierter Absender (im hier vorgestellten Simulationspaket ausschließlich Objekte der Klasse Bedienanlage) muß somit für die Forderung weiterhin eigene Platzkapazität bereitstellen.
Muß ein solches Absender-Objekt seine gesamte Kapazität zur Speicherung solcher Forderungen, die augenblicklich nicht vermittelt werden können, aufwenden, so soll er als (momentan) **vollständig blockiert** bezeichnet werden. Ein zu einem Zeitpunkt vollständig blockiertes Objekt ist nicht in der Lage, zu diesem Zeitpunkt seinerseits Forderungen von einem Vorgängerobjekt zu übernehmen und verursacht somit gleichfalls eine Blockierung im Vorgänger.
Eine wichtige Unterscheidung betrifft die Dauerhaftigkeit einer Blockierung. Folgende Skizze zeigt ein Beispielmodell:

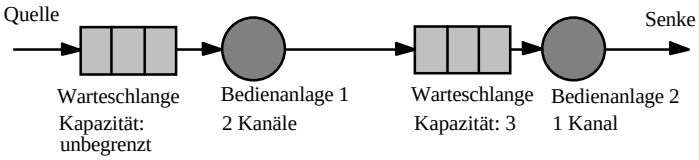

Abbildung 4-8: Dauerhaftigkeit von Blockierungen (Variante a.)

In dem dargestellten Modell kann eine Blockierung zum Zeitpunkt t auftreten, falls die Kapazität der Bedienanlage 2 und der zugehörigen Warteschlange erschöpft ist und die Bedienanlage 1 beabsichtigt, eine Forderung an den zweiten Bedienknoten weiterzuleiten. Die Bedienanlage 1 wird blockiert. Jedoch ist unschwer zu erkennen, daß diese Blockierung wieder aufgehoben wird - und zwar genau dann, wenn Bedienanlage 2 die Bearbeitung einer Forderung beendet hat. Derartige Blockierungen sollen als **aufhebbare Blockierungen** bezeichnet werden. Der simulative Lösungsansatz bietet die Möglichkeit, über aufhebbare Blockierungen eine Statistik zu führen. Diese beinhaltet die mittlere Dauer von Blockierungen und die Wahrscheinlichkeit des Auftretens von Blockierungen jeweils pro Forderungsklasse.
Die nächste Skizze zeigt ein Modell, in dem Blockierungen auftreten können, die nicht wieder aufhebbar sind:

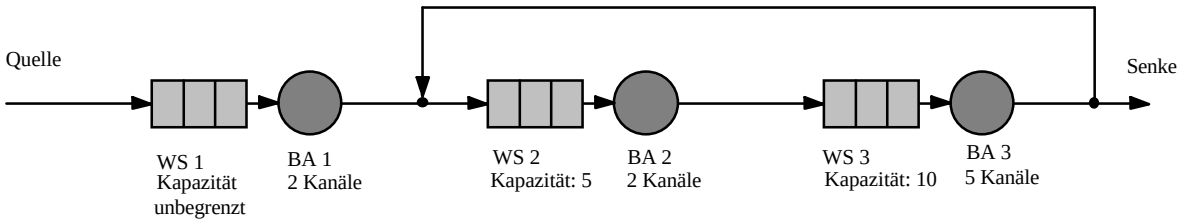

Abbildung 4-9: Dauerhaftigkeit von Blockierungen (Variante b.)

Die Variante b. beinhaltet als augenfälligstes Merkmal eine Rückführung von bearbeiteten Forderungen des dritten Knotens zum zweiten Knoten. Weiterhin existieren innerhalb des so entstandenen Kreises nur Objekte mit beschränkter Platzkapazität.
Vorstellbar ist nun die Situation, daß dieser "Kreis" mit einer Gesamtkapazität von 22 Plätzen vollständig ausgelastet ist. Ein Übergang von Knoten 3 zu Knoten 2 ist damit aktuell nicht möglich, eine Forderung, die diesen Weg wählt, muß im Knoten 3 warten und belegt weiterhin einen Bedienkanal. Ein Übergang einer bearbeiteten Forderung von Knoten 2 zu Knoten 3 ist gleichfalls unmöglich. Der einzige Ausweg aus dieser Situation besteht darin, daß eine im Knoten 3 bearbeitete Forderung diesen Kreis verläßt, indem sie den Übergang zur Forderungssenke wählt. Sollte jedoch der Fall eintreten, daß in der oben beschriebenen Situation alle von Knoten 3 bearbeiteten Forderungen den Übergang zu Knoten 2 wählen, so tritt eine Blockierung auf, die nicht mehr aufhebbar ist. Eine solche **unaufhebbare Blockierung** (**Kreisblockierung**, **Verklemmung**, **Deadlock**) tritt auf, da die Abläufe in einem Simulationsmodell nicht wirklich parallel sind. Im folgenden soll der Begriff **Deadlock** (engl. to reach deadlock: sich festfahren, in eine Sackgasse geraten) für eine solche unaufhebbare Blockierung verwendet werden. Warum die nicht vorhandene echte Parallelität Ursache für das Auftreten von Deadlocks während des Simulationslaufes ist, macht folgendes Minimalbeispiel deutlich:

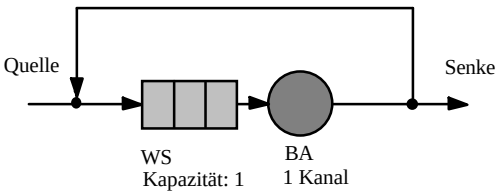

Abbildung 4-10: Minimalbeispiel für Auftreten eines Deadlocks

In der Warteschlange WS und in der Bedieneinrichtung BA befinden sich jeweils eine Forderung. Nach Ende der Bearbeitung der Forderung möchte diese erneut von der Bedienanlage bearbeitet werden. Nachstehender Ablauf ergibt sich:

1. Anfrage, ob in der Warteschlange Platz zur Übernahme vorhanden ist.
2. Antwort auf die Anfrage ist negativ, also geht die Bedieneinrichtung BA in einen (vollständig) blockierten Zustand über.

Die Forderung kann nicht weitergeleitet werden, da sich in der Warteschlange eine andere Forderung befindet. Letztere wiederum kann noch nicht bedient werden, da die Bedieneinrichtung immer noch belegt ist. Wären beide Forderungen in der Lage, sich zeitlich parallel durch das Modell zu bewegen, würde kein Deadlock auftreten.

Damit können **notwendige Bedingungen für das Auftreten eines Deadlocks** angegeben werden:

1. Sequentielle Abarbeitung des Simulationsmodells und
2. Existenz von geschlossenen "Kreisen" von Übergängen und
3. Begrenzte Platzkapazität der Elemente (Warteschlangen, Bedienanlagen) solcher "Kreise".

Durch Prüfung dieser Bedingungen können Abschnitte eines Modells erkannt werden, die Deadlock-gefährdet sind (Deadlock-gefährdete Kreise). Ausgehend von den Ausführungen, die zur Abbildung 4-9 gemacht wurden, können **hinreichende Bedingungen für das Auftreten eines Deadlocks** angegeben werden. Dabei ist zu beachten, daß ein Deadlock ausschließlich in geschlossenen Kreisen von einem oder mehreren Elementen eines Simulationsmodells auftritt:

1. Erfüllung der notwendigen Bedingungen (siehe oben) und
2. Die gesamte Platzkapazität eines Kreises ist ausgeschöpft und
3. Alle Elemente dieses Kreises sind vollständig blockiert und
4. Alle Forderungen, die innerhalb des Kreises auf einen Übergang warten, warten auf einen Übergang zu Elementen dieses Kreises.

Bemerkung: Die Bedingung 3 ist nicht allein hinreichend. Forderungen in einem vollständig blockierten Knoten könnten auf einen Übergang aus dem Deadlock-gefährdeten Kreis heraus warten, die Vermittlung solcher Forderungen würde später gelingen und damit die Blockierung aufgehoben werden.

Während aufhebbare Blockierungen (im folgenden nur kurz mit Blockierungen bezeichnet) für bestimmte Zwecke durchaus erwünscht sein könne, führt das Auftreten eines Deadlocks in der Regel zu Simulationsergebnissen, die nur geringen Erkenntnisgewinn über das modellierte System ermöglichen. Aus diesem Grund ist es notwendig, im Falle möglichen Auftretens von Deadlocks Maßnahmen zur ihrer Vermeidung, Erkennung oder Beseitigung zu treffen. Ein im Rahmen dieser Arbeit entwickelter Algorithmus zur Erkennung von Deadlocks wird später vorgestellt, zunächst wird beschrieben, wie mit (aufhebbaren) Blockierungen umgegangen wird.

BEHANDLUNG VON BLOCKIERUNGEN

Sind in einem Wartenschlangenmodell Bedienknoten vorhanden, bei denen sowohl die Warteschlangenkapazität als auch die Anzahl der Bedienkanäle begrenzt sind, können Blockierungen auftreten.

Für die Behandlung auftretender Blockierungen gibt es unter anderem folgende Möglichkeiten:

a) Verlust der Forderungen, die nicht weitervermittelt können

b) Warten der Forderung im Senderobjekt, bis die Weitervermittlung möglich ist
c) Versuch der Vermittlung an andere freie Nachfolger, bei Nichterfolg Möglichkeit a) oder b)

Möglichkeit a) ist die am einfachsten zu realisierende Variante. Es existieren durchaus Anwendungsmöglichkeiten für die Simulation von Systemen mit Warteschlangenkapazitäten gleich Null (Verlustsysteme). Den Objekten der Simulationsbibliothek kann die Fähigkeit zur Umsetzung dieser Variante leicht vermittelt werden, Erweiterungen würden jedoch auch die Modellbeschreibung innerhalb der Oberfläche und die Schnittstellen betreffen.
Möglichkeit c) ist eine flexible Möglichkeit der Behandlung von Blockierungen, die sicherlich auch bei Erweiterungen Berücksichtigung finden sollte.
Möglichkeit b) wurde in der vorliegenden Version der Simulationsbibliothek implementiert. Die Forderungen verhalten sich weniger flexibel, dafür aber konsequent: Ist ein Zielobjekt für eine Forderung einmal bestimmt, wird daran festgehalten. Diese Strategie ist in der Vorstellung gut nachzuvollziehen. Ein wichtiger Gesichtspunkt bei der Implementation einer Blockierungsbehandlung ist die gute Anwendbarkeit, das heißt, es sollten möglichst viele reale Systeme existieren, auf welche diese Strategie paßt. Diese Entscheidung ist bei der Vielzahl möglicher Systeme mit Unsicherheit behaftet, dessen ungeachtet wurde sich für die Strategie b) entschieden, da sich davon eine gute Anwendbarkeit versprochen wird.

Wie bereits erwähnt, wurde diese Strategie umgesetzt, indem durch die Fähigkeit von TRouter-Objekten zur Pufferung von Forderungen eine virtuelle Ausgangswarteschlange für blokkierte Objekte geschaffen wurde. Logisch werden Forderungen im blockierten Sender-Objekt verwaltet, physisch befinden sie sich in einem TRouter-Objekt, welches sich zwischen Sender-Objekt und Empfänger-Objekt befindet. Die typische Struktur sieht wie folgt aus:

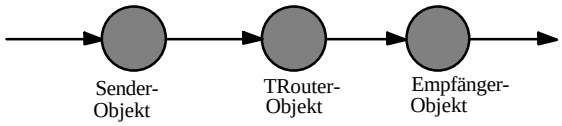

Abbildung 4-11: Pufferung durch TRouter-Objekte

Die eventuell notwendige Zwischenspeicherung erfolgt in TRouter-Objekten unter Zuhilfenahme der von TList abgeleiteten Klasse TRoutingList. Ihre Deklaration lautet:

```
template <class Element>
class TRoutingList : public TList<Element>
{
public:
                        TRoutingList();
                        ~TRoutingList();
                        BOOLEAN is_empty();
                        Element* find_entry(unsigned int ReceiverID);
                        virtual void print();
                        virtual BOOLEAN put(Element*);
protected:
                        long MaxNumberOfElements;
};
```

Gegenüber der Basisklasse TList existieren folgende Erweiterungen: Die Funktion *is_empty* prüft auf das Vorhandensein von gepufferten Forderungen, *find_entry* sucht die erste gepufferte Forderung, die zu dem durch *ReceiverID* beschriebenen Objekt vermittelt werden soll. Die Funktion *put* ist für das Einfügen weiterer zu puffernder Forderungen in die Liste verantwortlich. Durch das Überschreiben dieser Funktion besteht die Möglichkeit, eine andere Strategie für die Reihenfolge der Vermittlung wartender Forderungen zu definieren. In der aktuel-

len Version ist eine FIFO-Strategie realisiert, eine sinnvolle weitere Strategie wäre die zusätzliche Berücksichtigung von Prioritäten.
Die Elemente dieser Liste sind in der folgenden Weise beschrieben:

```
class RouterElementtyp {
public :
        RouterElementtyp(kunde&,unsigned int SenderID,unsigned int ReceiverID,float ATime);
        ~RouterElementtyp();
        RouterElementtyp *next;
        kunde* K;
        unsigned int Sender;
        unsigned int Receiver;
        float ArrivalTime;
};
```

Neben der eigentlichen Forderung beinhalten sie zusätzlich die Angabe von Sender- und Empfänger-ID. Für die Statistikführung über Blockierungen wird die Angabe der Eintrittszeit in die Liste (= Beginn der Blockierung) benötigt.

In den nachstehenden Abbildungsvarianten wird die Kommunikation zwischen sendendem Objekt, TRouter und Empfänger-Objekt dargestellt. Im Fall a) besteht für das TRouter-Objekt keine Notwendigkeit zur Pufferung der Forderung:

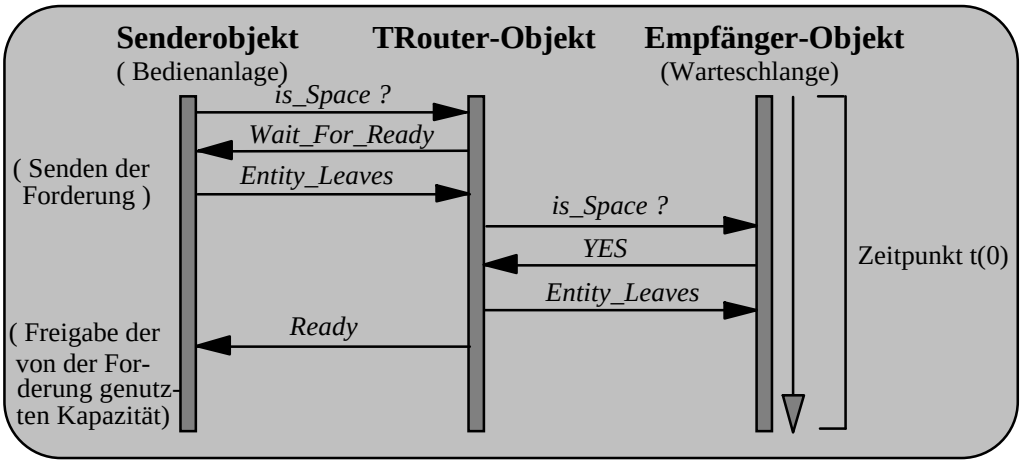

Abbildung 4-12: Kommunikation über ein TRouter-Objekt (Variante a)

In der obigen Abbildung tritt keine Blockierung auf, die Forderung wird vom TRouter-Objekt sofort an ihren Nachfolger vermittelt. Die gesamte Kommunikation findet ohne Modellzeitverbrauch, jedoch in der angegebenen sequentiellen Folge, statt.

Die zweite Variante beschreibt die Kommunikation, wenn eine Blockierung eintreten sollte:

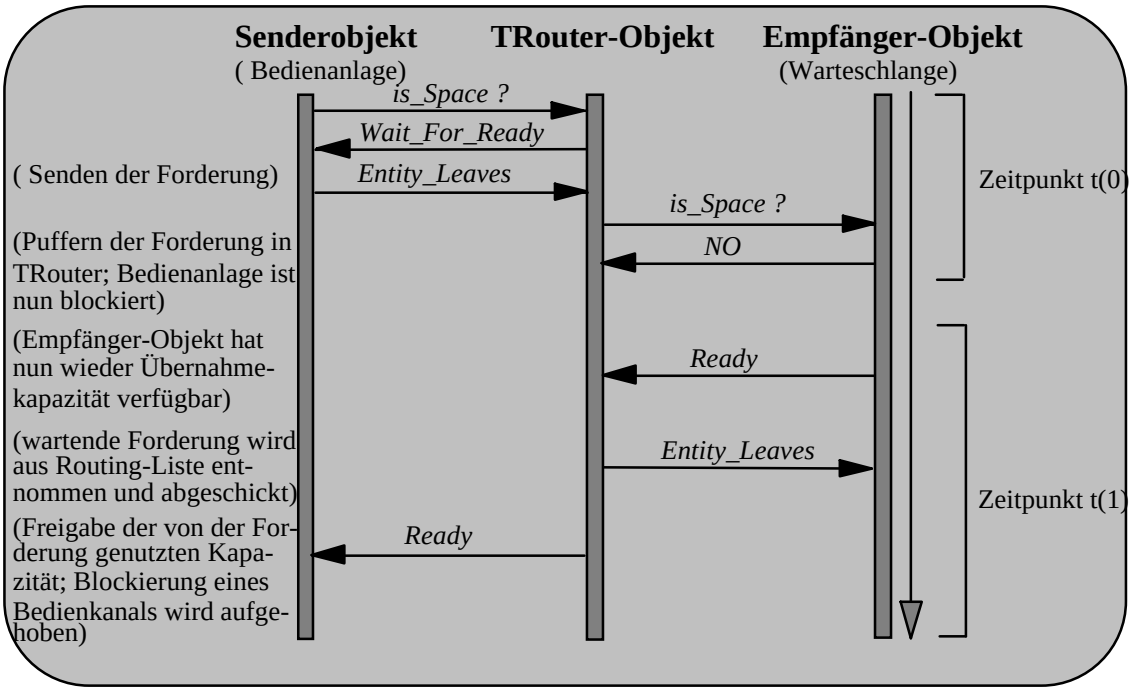

Abbildung 4-13: Kommunikation über ein TRouter-Objekt (Variante b)

In dieser Abbildung kann das TRouter-Objekt die Forderung nicht sofort weitervermitteln, da das Empfänger-Objekt keine freie Kapazität meldet. Somit wird die Forderung im TRouter-Objekt gespeichert. Die Kapazität des Senderobjektes (im obigen Beispiel ein Bedienkanal) wird weiter als belegt geführt, das Senderobjekt ist somit blockiert.
Zu einem späteren Zeitpunkt t_1 meldet das Empfänger-Objekt wieder freie Kapazität (mittels eines Ready-Signales). Das Router-Objekt durchsucht nun seinen Puffer nach einer Forderung, die zum Empfänger-Objekt vermittelt werden soll. Findet es eine Forderung, so wird diese aus der Liste entnommen und abgesendet, anschließend wird das Sender-Objekt von der erfolgreichen Vermittlung in Kenntnis gesetzt. Als Reaktion darauf kann ein Bedienkanal des Senders wieder freigegeben werden.

4.5.2. Ein Algorithmus zur Deadlock-Erkennung

EINFÜHRUNG

Wie bereits erwähnt, ist im Gegensatz zu (aufhebbaren) Blockierungen das Auftreten eines Deadlocks gesondert zu behandeln. Folgende Möglichkeiten sollen betrachtet werden:

A) Keine gesonderte Behandlung
B) Deadlock-Vermeidung
C) Deadlock-Erkennung während des Simulationslaufes
D) Deadlock-Beseitigung während des Simulationslaufes

Möglichkeit A) bedeutet das Ignorieren auftretender Deadlocks. Nachdem ein Deadlock aufgetreten ist, ist der Lauf der Forderungen zumindest durch einen Zweig des Modelles, oft sogar durch das Gesamtmodell, nicht mehr möglich. Daraus ergeben sich zwei Konsequenzen: Die Simulationsergebnisse sind nach dem Auftreten eines Deadlocks schwerlich zum Erkenntnisgewinn über das Realsystem geeignet, und die Anzahl der Forderungen, die auf ein Betreten des Deadlock-Zweiges warten, wächst ständig an und wird somit die vorhandenen Speicherressourcen der Simulationshardware früher oder später erschöpfen. Aus diesen Gründen ist das Ignorieren auftretender Deadlocks ungeeignet.

Zur Realisierung von Möglichkeit B) gibt es mehrere Alternativen. Die einfachste Lösung besteht sicher darin, durch Beachtung der oben angeführten notwendigen und hinreichenden Bedingungen das Auftreten eines Deadlocks auszuschließen. Dieses wäre beispielsweise durch die exklusive Verwendung unlimitierter Warteschlangen möglich. Eine solche Lösung kann natürlich nicht befriedigen.
Eine brauchbare Lösung setzt voraus, daß die Gefahr eines Deadlocks vor Beginn der Simulation erkannt wird. Hernach kann durch entsprechende Erweiterungen an der Struktur diese Gefahr ausgelöscht werden. Beispielsweise kann die Deadlock-Gefahr im Minimalbeispiel aus Abbildung 4-10 wie folgt beseitigt werden:

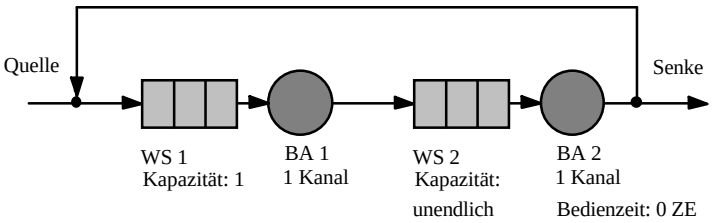

Abbildung 4-14: Beseitigung der Deadlock-Gefahr durch Strukturänderung

Durch das Einfügen eines zusätzlichen Bedienknotens ist die dritte notwendige Bedingung für das Auftreten eines Deadlocks nicht mehr erfüllt (die Gesamtkapazität des "Deadlock-Kreises" ist nun unbeschränkt). Die Wirklichkeitstreue des Modells bleibt dabei unverändert, da die Bedienung in der Bedienanlage 2 keine Zeit verbraucht. Bei der Betrachtung dieser Möglichkeit stellt sich die Frage, ob diese Strukturänderung bereits durch den Nutzer bei der Modellierung (**nutzergesteuerte Deadlock-Vermeidung**) oder durch die Initialisierungsroutine des Simulators geschehen (**automatische Deadlock-Vermeidung**) soll. Die vorliegende Version des Simulationspaketes realisiert die automatische Deadlock-Vermeidung nicht, da in der Umsetzung der Möglichkeit C) eine leistungsfähige Alternative entwickelt werden konnte. Zudem sind Modelle denkbar, die auf das Auftreten eines Deadlocks innerhalb einer bestimmten Simulationsdauer geprüft werden sollen - diese Möglichkeit würde durch die Realisierung der Variante B) verstellt. Natürlich steht es dem Nutzer frei, bei der Modellbildung bereits Deadlocks auszuschließen (nutzergesteuerte Deadlock-Vermeidung).

Möglichkeit C), die Erkennung eines Deadlocks während der Simulation, wurde zur Integration in das Simulationspaket ausgewählt. Die Idee besteht darin, daß der Simulationsprozeß bis zu dem Auftreten eines Deadlocks geführt wird, bei der Erkennung des Auftreten eines Deadlocks wird die Simulation abgebrochen und die Ergebnisse bis zu diesem Zeitpunkt können zur Auswertung benutzt werden. Der hauptsächliche Vorteil einer solchen Methode besteht darin, daß keine Eingriffe in die Modellstruktur notwendig werden.
Ein erster Ansatz für die Umsetzung könnte darin bestehen, in bestimmten Zeitabständen die Bedienanlagen eines Modells zu prüfen, ob eine Zustandsänderung (beispielsweise das Ende eines Bearbeitungsvorganges) stattgefunden hat. Die Zeitabstände könnten dabei durch die mittlere Bedienzeit der Bedienanlage bestimmt sein, denkbar ist das 20fache der normalen Bedienzeit. Hat keine Zustandsänderung stattgefunden, so könnte ein Deadlock diagnostiziert werden. Probleme dieser Methode sind sowohl die schwere Bestimmbarkeit der Zeitabstände und - viel schwerwiegender - die Möglichkeit einer Fehldiagnose, eines falschen Alarms. Es ist keinesfalls zu tolerieren, würde der Simulator das Auftreten eines Deadlocks melden, der möglicherweise nur eine Blockierung darstellt, die durch hohe Auslastung entstanden ist. Ein weiteres Problem stellt die Verzögerung zwischen dem Auftreten und der Erkennung des Deadlocks dar.
Der zweite Ansatz ist eine Verbesserung des ersten. Wenn durch Anwendung der ersten Methode der Verdacht auf einen Deadlock gewonnen wurde, so muß mittels eines deterministischen Algorithmus das Modell irrtumsfrei auf das Auftreten eines Deadlocks geprüft werden

können. Im Rahmen dieser Arbeit konnte ein solcher Algorithmus gefunden und implementiert werden. Auf diese Weise kann die Möglichkeit eines "falschen Alarms" ausgeschlossen werden. Bestehen bleiben jedoch die Probleme der Verzögerungszeit und die schwere Bestimmbarkeit der Prüfabstände (siehe oben).
Der dritte Ansatz verbessert den zweiten hinsichtlich der Bestimmung des Zeitpunktes, wann eine Prüfung auf einen Deadlock stattzufinden hat. Vor Beginn der Simulation wird durch Prüfung der notwendigen Bedingungen für einen Deadlock zunächst ermittelt, ob im gegebenen Modell Deadlocks auftreten können. Besteht die Gefahr eines Deadlocks, so wird die Routine zur Prüfung auf Deadlock ohne Zeitverzug aufgerufen, wenn eine Bedienanlage vollständig blockiert ist. Die Prüfung auf Deadlock wird vom Simulatorkern durchgeführt. Der Algorithmus arbeitet effizient und ermöglicht die Erkennung eines Deadlocks sofort nach dessen Auftreten. Auf Entwurf und Implementation des Algorithmus wird unten detailliert eingegangen.

Auch Möglichkeit D), die Beseitigung des Deadlocks während der Simulation, muß in Erwägung gezogen werden. Sie setzt allerdings die Erkennung eines Deadlocks nach Möglichkeit C) bereits voraus.
Wie oben erwähnt, ist die sequentielle Abarbeitung von parallelen Vorgängen durch den Simulator notwendige Bedingung für das Auftreten eines Deadlocks. Wurde ein Deadlock erkannt, so soll mittels eines geeigneten Mechanismus Quasiparallelität erzeugt werden, so daß der Deadlock aufgelöst werden kann. Die Findung eines solchen Mechanismus ist nicht trivial, insbesondere wenn keine Kompromisse bezüglich der Struktur des Simulationsmodelles (Kriterien der Modularität - siehe Kapitel 3) eingegangen werden sollen. Eine ausführbare Umsetzung der Möglichkeit D) soll an dieser Stelle nicht angegeben werden.

ENTWURF UND IMPLEMENTATION DES ALGORITHMUS

Um die Erkennung von Deadlocks während der Simulation zu ermöglichen, wurde ein Mechanismus zur Deadlockerkennung entworfen, getestet und implementiert, der sich in folgende Bestandteile (Teilalgorithmen) gliedert:

1. Feststellen von potentieller Deadlock-Gefahr vor Simulationsbeginn.
2. Aktualisierung einer Matrix, die die aktuell blockierten Forderungen verwaltet (während der Simulation).
3. Wenn der Verdacht auf einen aufgetretenen Deadlock besteht, wird eine algorithmische Prüfung des aktuellen Zustandes der Simulation durchgeführt.

Die notwendigen Datenstrukturen und Funktionen werden in der Klasse *DLCheck* zusammengefaßt, deren vollständige Deklaration dem Anhang entnommen werden kann. Ein Objekt der Klasse DLCheck ist ein Datum des Simulatorkernes, welcher für die Deadlock-Erkennung zuständig ist.
Voraussetzung für die Anwendung des Mechanismus ist, daß das Simulationsmodell ein BNETD-Modell (siehe Abschnitt 4.3.5.) ist. Im Interesse einer effizienten Abarbeitung wurde sich bei der Umsetzung der Deadlock-Erkennung die feststehende Struktur der BNETD-Modelle zunutze gemacht. Daraus folgt, daß der Mechanismus zur Deadlock-Erkennung als ein Bestandteil der Simulationskomponente von BNETD, nicht als Bestandteil der Simulationsbibliothek gesehen werden sollte. Er ist auf alle mit Hilfe des Programmsystems BNETD erzeugten Modelle anwendbar, also auch auf mehrklassige Modelle. Prinzipiell steht der Übertragung der hier eingeführten Techniken auf erweiterte Modelle nichts im Wege.

Die Erläuterung des Mechanismus soll mit Hilfe eines relativ einfachen Beispiels erfolgen, eines offenen einklassigen Warteschlangennetzes mit 4 Knoten, dessen Struktur aus der folgenden Skizze entnommen werden kann:

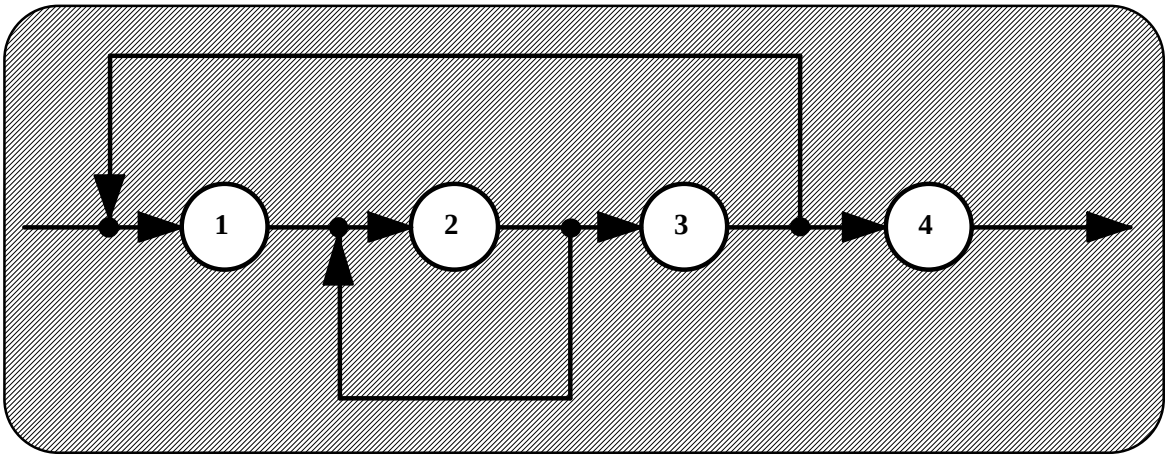

Abbildung 4-15: Erläuterung des Mechanismus zur Erkennung von Deadlocks

Alle Bedienknoten des Modells verfügen über einen Warteraum mit begrenzter Kapazität. Die genaue Kenntnis der Warteraumkapazität ist nicht nötig, es genügt zu wissen, ob die Warteräume begrenzt oder unbegrenzt sind. Dagegen spielt die Kenntnis der Kapazität der Bedienanlagen eine Rolle: Knoten 1 hat drei, Knoten 2 vier, Knoten 3 zwei und Knoten 4 einen Bedienkanal.

Erster Schritt: Algorithmus zur Erkennung einer Deadlock-Gefährdung:
Der erste Schritt besteht in der Prüfung des Modells, ob ein Deadlock theoretisch auftreten kann. Diese Prüfung wird während der Initialisierung der Simulationskomponente durchgeführt. Sie ist pro Simulationslauf einmalig.
Die Idee des ersten Schrittes besteht darin herauszufinden, ob in dem gegebenen Modell Zyklen existieren, die die notwendigen Bedingungen für das Auftreten eines Deadlocks erfüllen. Zu diesem Zweck sind einige Operationen über der Adjazenzmatrix nötig.

Unterschritt A: Aufstellung der Adjazenzmatrix
Die Adjazenzmatrix wird aus den Übergangswahrscheinlichkeiten aufgestellt, die das Modell beschreiben. Im vorliegenden Fall können sie aus der Abbildung 4-15 entnommen werden:

a⇒b	1	2	3	4
1	-	X	-	-
2	-	X	X	-
3	X	-	-	X
4	-	-	-	-

Adjazenzmatrix

Ein mit "X" gekennzeichnetes Feld [a,b] stellt einen direkten Übergang von Knoten a zu Knoten b dar, eine Kennzeichnung mit "-" bedeutet dagegen: Es existiert kein direkter Übergang von a nach b.
Wie erwähnt, arbeitet der Algorithmus auch für mehrklassige Modelle korrekt. Übergänge, die weitere Forderungsklassen betreffen, können wie zusätzliche Übergänge der ursprünglichen Forderungsklasse betrachtet werden. Eine programmtechnische Umsetzung besteht in der Bildung von Adjazenzmatrizen für jede Forderungsklasse und anschließender Oder-Verknüpfung dieser Matrizen zu einer Gesamtmatrix.

Unterschritt B: Streichen aller Übergänge zu Knoten mit unbegrenztem Warteraum
Aus der notwendigen Bedingung für das Auftreten eines Deadlocks folgt, daß ein Knoten mit unbegrenztem Warteraum in einem Zyklus das Auftreten eines Deadlocks in diesem Zyklus verhindert.
Aus diesem Grund werden alle Übergänge zu Knoten mit unbegrenztem Warteraum gestrichen. Praktisch bedeutet dies: Für jeden Knoten mit unbegrenztem Warteraum wird die entsprechende Spalte in der Adjazenzmatrix durchgängig mit "-" markiert.
Im gegebenen Beispiel sind alle Warteräume begrenzt, so daß die Durchführung des Unterschrittes B keine Veränderung ergibt:

a⇒b	1	2	3	4
1	-	X	-	-
2	-	X	X	-
3	X	-	-	X
4	-	-	-	-

Bereinigte Adjazenzmatrix

Unterschritt C: Bildung der transitiven Hülle
Im dritten Unterschritt wird die transitive Hülle der bereinigten Adjazenzmatrix gebildet. Der Grundgedanke bei der Bildung der transitiven Hülle eines gerichteten Graphs lautet: Wenn es einen Weg von Knoten a zu Knoten b gibt, und es gibt einen Weg von Knoten b zu Knoten c, dann gibt es einen Weg von Knoten a zu Knoten c. Ein einfacher, nichtrekursiver Algorithmus zur Bildung der transitiven Hülle wurde zuerst im Jahr 1962 von S. Warshall gefunden [Sedgewick 90]. Die Anwendung des Algorithmus auf die im Unterschritt B ermittelte bereinigte Adjazenzmatrix ergibt:

a⇒b	1	2	3	4
1	X	X	X	X
2	X	X	X	X
3	X	X	X	X
4	-	-	-	-

Transitive Hülle der bereinigten Adjazenzmatrix

Unterschritt D: Bildung einer Deadlock-Matrix
Der Grundgedanke des ersten Teilalgorithmus besteht darin, Deadlock-gefährdete Zyklen zu finden. Aus diesem Grund werden aus der in Unterschritt C gefundenen Matrix die Übergänge gestrichen, die nicht Teil eines Zyklus sind. Programmtechnisch heißt dieses, daß alle Übergänge [a,b] gestrichen werden, für die der Übergang [b,a] nicht existiert. Die Ausführung dieses Schrittes ergibt eine neue Matrix, die mit dem Begriff "Deadlock-Matrix" gekennzeichnet werden soll:

a⇒b	1	2	3	4
1	X	X	X	-
2	X	X	X	-
3	X	X	X	-
4	-	-	-	-

Deadlock-Matrix

Unterschritt E: Bereinigung der Deadlock-Matrix
Die im Unterschritt D gefundene Matrix sagt aus, daß zwischen den Knoten 1, 2 und 3 mindestens ein Zyklus existiert, in dem Deadlocks auftreten können. Durch eine Bereinigung der Deadlock-Matrix können weitergehende Aussagen getroffen werden. Umgesetzt wird die Bereinigung durch eine Und-Verknüpfung von Deadlock-Matrix und Adjazenzmatrix:

a⇒b	1	2	3	4
1	-	X	-	-
2	-	X	X	-
3	X	-	-	-
4	-	-	-	-

Bereinigte Deadlock-Matrix

Aus der gefundenen Matrix lassen sich nachstehende Aussagen ableiten:

- Das Auftreten eines Deadlocks im gegebenen Modell ist möglich.
- Es existieren zwei Zyklen, in denen ein Deadlock auftreten kann. Einerseits ist dies der Zyklus Knoten 1 ⇒ Knoten 2 ⇒ Knoten 3; andererseits der Zyklus, den der Knoten 2 allein bildet.
- **Gefährdete Übergänge** sind [1,2], [2,2], [2,3] und [3,1]. In anderen Worten heißt das: Eine Blockierung des Überganges [3,4] kann nicht Ursache für einen Deadlock sein.

Bemerkung: Im gewählten einfachen Beispiel sind die getroffenen Aussagen aus der Abbildung 4-15 zu erkennen. Die algorithmische Erkennung der Deadlock-Gefährdung ist bei komplizierteren Modellen (höhere Knotenanzahl, mehrere Forderungsklassen) gegenüber der Erkennung durch "Hinschauen" wesentlich zuverlässiger.

Weiterhin wird im ersten Schritt ein Vektor gebildet, in welchem die maximale Kapazität der Bedienanlagen (Anzahl der Bedienkanäle) vermerkt wird. Dieser wird im dritten Schritt benötigt. Im vorliegenden Fall hat der Kapazitätsvektor folgendes Aussehen:

Knoten	1	2	3	4
Kapazität	3	4	2	1

Kapazitätsvektor

Auf die Durchführung der Schritte zwei und drei kann selbstverständlich verzichtet werden, wenn im Schritt eins keine Deadlock-Gefährdung erkannt wurde.

Zweiter Schritt: Führung einer Blockierungsmatrix während der Simulation
Um die Erkennung eines Deadlocks während der Simulation zu ermöglichen, wird die Anzahl momentan blockierter Forderungen mit Angabe von Sender und Ziel benötigt. Diese Informationen werden in einer Matrix verwaltet, die im weiteren Blockierungsmatrix heißen soll. Unter Nutzung der im ersten Schritt gewonnenen Erkenntnisse müssen nur diejenigen Felder der Matrix geführt werden, die gefährdete Übergänge nach der bereinigten Deadlockmatrix darstellen (diese sind in der Matrix dunkler dargestellt). Die Begründung für diese Vereinfachung gibt die vierte hinreichende Bedingung für das Auftreten eines Deadlocks: Alle Forderungen, die innerhalb des Kreises (Zyklus) auf einen Übergang warten, warten auf einen Übergang zu Elementen dieses Kreises (Zyklus).

Die Blockierungsmatrix könnte folgendes Aussehen haben:

a⇒b	1	2	3	4	Σ	Max.Kapazität
1	-	1	-	-	1	3
2	-	1	2	-	3	4
3	0	-	-	-	0	2
4	-	-	-	-	0	1

Blockierungsmatrix A

Im dargestellten Fall sagt die Matrix aus, daß eine Forderung auf den Übergang von Knoten 1 nach Knoten 2, eine Forderung auf den Übergang [2,2] und zwei Forderungen auf den Übergang [2,3] warten.
Wie auch die bereinigte Deadlock-Matrix wird die Blockierungsmatrix von einem Objekt der Klasse DLCheck verwaltet, welches seinerseits zum Simulatorkern (Klasse TSimulator) gehört. Die notwendige Aktualisierung wird von TRouter-Objekten veranlaßt, die jede neu auftretende oder sich aufhebende Blockierung an den Simulatorkern melden.

Dritter Schritt: Algorithmische Erkennung eines Deadlocks
Der Algorithmus zur Deadlock-Erkennung wird von TRouter-Objekten ausgelöst, wenn eine Bedienanlage vollständig blockiert ist. Das TRouter-Objekt sendet eine Nachricht *Blockierungstest* an den Simulatorkern. Diese Nachricht wird an die Funktion TSimulator::handleSystemMessage vermittelt, die dann wiederum die Funktion DLCheck::isDeadLock ausführt.
Zunächst soll anhand des Beispiels gezeigt werden, welche Bedingungen für das Eintreten eines Deadlocks im konkreten Fall hinreichend sind. Durch Anwendung notwendiger und hinreichender Bedingungen sowie der im Schritt eins gewonnenen Erkenntnisse können zwei Möglichkeiten angegeben werden, die zum Auftreten eines Deadlocks führen:

- Im Feld [2,2] der Blockierungsmatrix befinden sich vier Forderungen, damit wartet die gesamte Kapazität des durch den Knoten 2 gegebenen Zyklus auf einen Übergang in diesen Zyklus.
- Ein Deadlock kann ebenso im Bereich des Zyklus Knoten1 ⇒ Knoten 2 ⇒ Knoten 3 auftreten, wenn die Zeilensummen der Zeile i (mit i=1,2,3) jeweils gleich der maximalen Kapazität des Knotens i sind.

Der Grundgedanke der algorithmischen Prüfung ist nachstehender rekursiver Ansatz:

Ein Knoten ist unaufhebbar vollständig blockiert, wenn alle seine Nachfolger, zu denen Forderungen warten, unaufhebbar vollständig blockiert sind oder bereits markiert wurden.

Ein Knoten wird markiert, wenn die zugehörige Zeilensumme in der Blockierungsmatrix gleich der maximalen Kapazität des Knotens ist. Wird **ein** unaufhebbar blockierter Knoten gefunden, so ist ein Deadlock aufgetreten.
Folgende Blockierungsmatrix soll betrachtet werden:

a⇒b	1	2	3	4	Σ	Max.Kapazität
1	-	3	-	-	3	3
2	-	1	3	-	4	4
3	1	-	-	(1)	(2)1	2
4	-	-	-	-	0	1

Blockierungsmatrix B

In der dargestellten Situation erfüllen die Knoten 1 und 2 alle Voraussetzungen, um eine Markierung zu erhalten.
Knoten 3 entspricht dem zu Beginn des Abschnittes 4.5. gegebenen Begriff "vollständig blokkiert", da die gesamte Kapazität des Knotens (zwei Bedienkanäle) auf Vermittlung warten muß. Jedoch ist ein Deadlock nicht möglich, da eine Forderung auf den Übergang zum Knoten 4 wartet. Diese wird früher oder später vermittelt. Es zeigt sich, daß für die Bildung der Zeilensumme nur *gefährdete Übergänge* berücksichtigt werden müssen. Die Begründung ist in der vierten hinreichenden Bedingung für das Auftreten eines Deadlocks gegeben.

Die programmtechnische Umsetzung des rekursiven Ansatzes lautet wie folgt:

```
// Prüfung auf Deadlock

Aktualisiere die Zeilensummen anhand der Blockierungsmatrix;
FOR (alle Knoten des Modells)
        DO
        {
        Setze alle Markierungen zurück;
        Wenn der aktuelle Knoten unaufhebbar vollständig blockiert ist, melde einen
        Deadlock;
        };
```

Die Funktion zur Prüfung auf unaufhebbare vollständige Blockierung lautet:

```
IF (Knoten K bereits markiert) RETURN Blockierung;
IF NOT (Zeilensumme für Knoten K = Maximaler Kapazität des Knotens K) RETURN Keine
Blockierung;
Markiere Knoten K;
FOR (alle Knoten I des Modells)
        {
        IF( (Forderungen auf Übergang von K nach I warten )
        AND NOT (Knoten I unaufhebbar vollständig blockiert))
        RETURN Keine Blockierung;
        }
RETURN Blockierung;
```

Zur besseren Veranschaulichung soll der Algorithmus für obiges Beispiel (siehe Blockierungsmatrix B) durchgeführt werden:

1.Untersuchung von Knoten 1:
Knoten 1 hat noch keine Markierung
Die Zeilensumme - 3 - ist gleich der maximalen Kapazität des Knotens
Knoten 1 wird markiert
Untersuchung der Knoten, zu denen Forderungen aus Knoten 1 vermittelt werden möchten:
Untersuchung von Knoten 2:
Knoten hat noch keine Markierung
Die Zeilensumme - 4 - ist gleich der maximalen Kapazität des Knotens
Knoten 2 wird markiert
Untersuchung der Knoten, zu denen Forderungen aus Knoten 2 vermittelt werden möchten:
Untersuchung von Knoten 2:
Knoten ist bereits markiert - nächsten Übergang suchen

Untersuchung von Knoten 3:
Knoten hat noch keine Markierung
Die Zeilensumme -1 - ist kleiner als die maximale Kapazität des Knotens 3
RETURN Keine Blockierung
RETURN Keine Blockierung
RETURN Keine Blockierung
2. Rücksetzen der Markierungen und Untersuchung von Knoten 2:
- führt analog zur Untersuchung von Knoten 1 zum Rückgabewert Keine Blockierung -
3. Rücksetzen der Markierungen und Untersuchung von Knoten 3:
Führt zum Rückgabewert Keine Blockierung, da Zeilensumme ungleich maximaler Kapazität.
4. Rücksetzen der Markierungen und Untersuchung von Knoten 4:
Führt zum Rückgabewert Keine Blockierung, da Zeilensumme ungleich maximaler Kapazität (Zeilensumme für Knoten 4 ist stets null, da keine gefährdeten Übergänge in dieser Zeile existieren).

Ergebnis: In der durch die Blockierungsmatrix beschriebenen Situation liegt kein Deadlock vor.

4.6. Statistiken

4.6.1. Vorbemerkungen

Durch die Führung von Statistiken werden Erkenntnisse über ein Modell erlangt, welche später auf ein reales System übertragen werden können. Weiterhin dienen Statistiken der Verifizierung des korrekten Laufes der Simulation in der Testphase. Durch die Sammlung von Daten - dieses sind teilweise einfache Zählvorgänge- wird die Grundlage geschaffen, mittels Anwendung mathematischer Methoden Voraussagen über interessierende Ergebnisgrößen (beispielsweise Bedien- und Wartezeiten) zu treffen.
Bei der Führung einer Statistik ist zu beachten, daß die Datensammlung im Modell in der Regel nur von Wert ist, wenn die Simulation einen *stationären Zustand* erreicht hat. Daher liegt die Idee nahe, die Simulation eine gewisse Zeit ohne Führung einer Statistik laufen zu lassen. Nach Beendigung dieser *Einschwingphase* beginnt die Führung einer Statistik. Ein Problem besteht hierbei in der Bestimmung der Dauer der Einschwingphase.
In der hier vorgestellten Simulationskomponente wird dagegen sofort mit der Statistikführung begonnen, eine Einschwingphase wird nicht berücksichtigt. Als wesentlichster Grund läßt sich angeben, daß aufgrund der zur Verfügung stehenden Leistungen der Hardware Einschwingphasen immer relativ kürzer werden. Wenn ein Simulationsmodell sich bereits nach einem Bruchteil einer (realen) Sekunde eingeschwungen hat, lohnt sich der Aufwand für die Berücksichtigung einer Einschwingphase nicht.
Im nächsten Abschnitt soll die Klasse *Result*, die als Ergebnisschnittstelle sowohl für Analyse als auch Simulation bestimmt ist, vorgestellt werden. Weiterhin soll beispielhaft die Berechnung ausgewählter Ergebnisgrößen gezeigt werden.

4.6.2. Ergebnisschnittstelle der Simulation - die Klasse Result

Die Implementation der Simulationskomponente für das Programmsystem BNETD macht die Übertragung und Auswertung der in den einzelnen Objekten geführten Statistiken in eine Schnittstellenklasse notwendig, um innerhalb der Oberfläche auf die gewonnenen Resultate zugreifen zu können. Zu diesem Zweck wurde die Klasse Result definiert. Sie paßt sich dynamisch an die Größe des Modells (Anzahl der Klassen, Anzahl der Knoten) an. Sie wurde als

einheitliche Schnittstelle sowohl für Analyse- als auch Simulationskomponente entwickelt, kann also die Ergebnisse beider Komponenten verwalten. Die gegenwärtig integrierte Analysekomponente nutzt noch eine eigene, weniger flexible Ergebnisdatenstruktur.
Die ermittelten Größen von Analyse- und Simulationskomponente stellt die folgende Tabelle zusammen:

Größe (pro Knoten)	**Analyse**	**Klassenabhängig**	**Simulation**	**Klassenabhängig**
Bedienzeit	ja	nein	ja	ja
Durchsatz	ja	ja	ja	ja
Auslastung	ja	ja	ja	ja
WarteWkt	ja	nein	ja	ja
LeerWkt	ja	nein	nein	nein
Verweilzeit	ja	ja	ja	ja
Wartezeit	ja	ja	ja	ja
Füllung	ja	ja	ja	ja
Mittl. Anzahl genutzter Kanäle	nein	nein	ja	ja
Max. Anzahl genutzter Kanäle	nein	nein	ja	nein
Mittl. Länge der Warteschlange	ja	ja	ja	ja
Max. Länge der Warteschlange	nein	nein	ja	nein
Blockierdauer	nein	nein	ja	ja
Max. Zahl blokkierter Kanäle	nein	nein	ja	nein
Blockierungs-Wkt	nein	nein	ja	ja
Größe (Gesamtsystem)	**Analyse**	**Klassenabhängig**	**Simulation**	**Klassenabhängig**
Durchsatz	ja	ja	ja	ja
Füllung	nein	nein	ja	ja
Verweilzeit	nein	nein	ja	ja
Systemauslastung	nein	nein	ja	nein
Schwachstelle	nein	nein	ja	nein

Tabelle 4-9: Ergebnisgrößen von Analyse- bzw. Simulationskomponente von BNETD

Die Klasse Result verwaltet diese Daten. Weiterhin definiert die Klasse Result eine Funktion, die das Schreiben der Ergebnisse in ein binäres Ergebnisfile ermöglicht. Dieses File steht zur Weiterverarbeitung durch externe Statistikprogramme zur Verfügung, bildet also eine Schnittstelle BNETD $\Rightarrow$ Außenwelt. Für die Übertragung der Daten aus den Simulationsobjekten in die Ergebnisschnittstelle ist die Funktion TSimulator::Done zuständig. Innerhalb der BNETD-Oberfläche wird die Klasse Result für die Auswertung und Ergebnisrepräsentation genutzt.
Die Deklaration der Klasse Result ist im Anhang zu finden.

4.6.3. Darstellung der Berechnung ausgewählter Statistikgrößen

Im vorliegenden Abschnitt soll anhand ausgewählter Beispiele gezeigt werden, wie die Führung der Statistik während der Simulation vonstatten geht. Zunächst soll der Zusammenhang zwischen Bedienzeiten, Wartezeiten, Verweilzeiten einerseits und Anzahl Forderungen in der Bedienanlage, Warteschlangenlänge und Füllung andererseits dargestellt werden (siehe auch die in Abschnitt 1.3. beschriebenen Grundlagen der Warteschlangentheorie):
So ergibt sich die mittlere Verweilzeit t_v in einem Knoten aus der mittleren Bedienzeit t_b und der mittleren Wartezeit t_w (ohne Berücksichtigung von Blockierungen):

$$\mathbf{t_v = t_b + t_w.}$$

Die Füllung n_v des Knotens (oder mittlere Anzahl von Forderungen) kann aus der mittleren Anzahl Forderungen in der Bedienanlage n_b (oder mittlere Anzahl belegter Bedienkanäle) und der mittleren Länge der Warteschlange n_w berechnet werden:

$$\mathbf{n_v = n_b + n_w.}$$

Durch die in der Simulationskomponente ermöglichte Führung von Statistiken über Blockierungen müssen diese Werte bei der Berechnung von mittlerer Verweilzeit und Füllung berücksichtigt werden. Im einzelnen sind dies die mittlere Dauer von Blockierungen t_{bl} und die mittlere Anzahl blockierter Bedienkanäle n_{bl}. Die Berechnungsformel für die mittlere Verweilzeit in einem Knoten unter Berücksichtigung von Blockierungen muß nun lauten:

$$\mathbf{t_v = t_b + t_{bl} + t_w.}$$

Für die mittlere Anzahl blockierter Kanäle n_{bl} existiert keine separate Statistik, die Größe n_{bl} ist vielmehr bereits in der mittleren Anzahl belegter Kanäle n_b enthalten. Somit wird reflektiert, daß ein Bedienkanal während der Simulation aus zwei Gründen belegt sein kann:

- Bearbeitung einer Forderung und
- Blockierung.

Als Beispiel für die Führung einer Statistik soll die laufende Berechnung der Größen t_w und n_w angeführt werden:

$$t_w = \frac{t_w * (c-1) + t_{aktuell}}{c}$$

mit einer aktuellen Wartezeit $t_{aktuell}$ und der Anzahl der gezählten Forderungen c. Diese Formel kann unter Nutzung des C-Ausdrucks += vereinfacht werden zu:

$$t_w += \frac{t_{aktuell} - t_w}{c}$$

Die mittlere Warteschlangenlänge n_w berechnet sich wie folgt:

$$n_w = \frac{n_w * t_{last\_refresh} + n_{aktuell} * \left(t_{current} - t_{last\_refresh}\right)}{t_{current}}$$

Dabei ist $n_{aktuell}$ die aktuelle Warteschlangenlänge, $t_{current}$ die aktuelle Simulatorzeit und $t_{last\text{-}refresh}$ der Zeitpunkt der letzten Aktivierung der Warteschlange.

4.7. Beschreibung der Programmierschnittstelle zur Nutzung der Simulationsbibliothek

4.7.1. Schnittstellen-Funktionen von TSimulator

Ziel der vorliegenden Arbeit ist es, ein Instrumentarium zur Simulation von Warteschlangennetzen zur Verfügung zu stellen. Dabei sollen in erster Linie zwei Anforderungen beachtet werden:

- Unterstützung der Nachnutzbarkeit, das heißt Beachtung von Kriterien wie Universalität, Erweiterbarkeit und Wartbarkeit
- Schaffung einer spezialisierten Simulationskomponente für BNETD

Diese Ansprüche konkurrieren miteinander, trotzdem wurde versucht, beim Entwurf des Simulationssystems die obigen Anforderungen im Auge zu behalten. Ein Beispiel für diesen "Konflikt" ist einerseits die Schaffung leistungsfähiger Methoden zur Blockierungsbehandlung, welche andererseits dazu verleiten, einen Teil der Universalität aufzugeben.
Der gewählte Ansatz beinhaltet den Entwurf einer Reihe von Klassen in einer Simulations-Klassenbibliothek, mit deren Hilfe eine gewünschte spezielle Realisierung geschaffen werden kann. Diese Klassen (Zufallszahlengeneratoren, Listen, Warteschlangen, Bedienanlage, Vereiniger, Verzweiger, Quelle und Senke) wurden bereits beschrieben. Sie stehen für Modifikationen und Erweiterungen zur Verfügung.
Zur Nachnutzung der bereits vorhandenen Klassen muß ein Programmierer folgende Schritte unternehmen:

A) Bildung eines Simulationsmodells aus der formalen Modellbeschreibung. Dies ist der Verantwortungsbereich der Funktion **TSimulator::Init**;
B) Anstoßen der Simulation - die Funktion **TSimulator::Run**;
C) Extrahieren der in den Objekten gesammelten statistischen Daten. Erreicht wird jenes durch Überschreiben der Funktion **TSimulator::Done**;
D) Verifizierung des formalen Modells oder des Simulationsmodells. Dazu existiert die Funktion **Verify**.

Diese Vorgehensweisen sollen anhand des Beispiels der Realisierung der speziellen Simulationskomponente von BNETD erläutert werden.

Zu Punkt A): Aufgabe der Funktion TSimulator::Init ist es zunächst, aus der formalen Modellbeschreibung eines Warteschlangennetzes ein Simulationsmodell mit Hilfe der Klassen aus dem Simulationspaket zu erstellen:

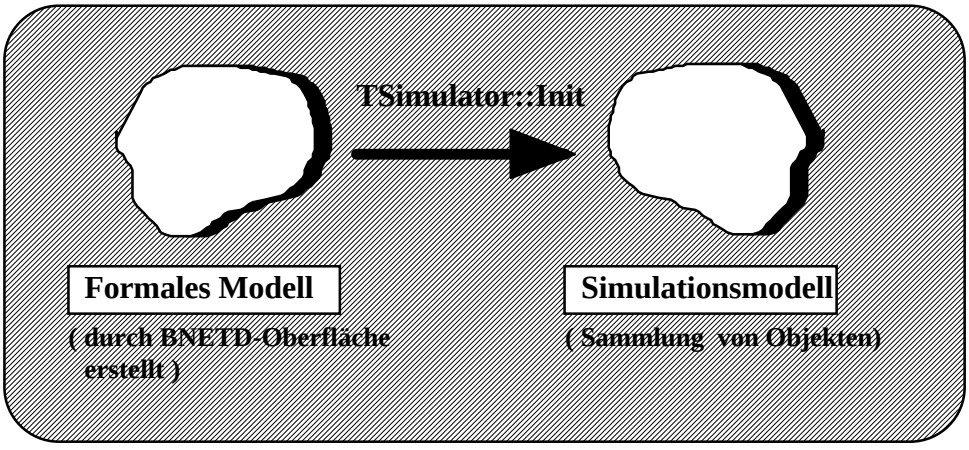

Abbildung 4-16: Die Funktion TSimulator::Init

Die Funktion TSimulator::Init erledigt folgende Aufgaben:

- Aufruf der Funktion EventList->InitFinalEvent zum Einfügen des Ereignisses "Simulationsende".
- Prüfung der Integrität des formalen Modells durch Aufruf der Funktion Verify.
- Sukzessiver Aufbau der Modellstruktur aus der formalen Modellbeschreibung und Parametrisierung der entstehenden Objekte.
- Initialisierung der Deadlock-Verwaltung.

Zu Punkt B): Die Funktion TSimulator erzeugt die notwendigen Startereignisse für die Simulation durch Aufruf der Funktion TSimulator::InitDynObjects. Solche Startereignisse können sein: Erzeugung von Create_Entity-Nachrichten für Quelle-Objekte oder die Bereitstellung einer begrenzten Anzahl von Forderungen (in einem geschlossenen Modell). Anschließend wird durch Aufruf der Funktion TSimulator::handleMessage dem Simulatorkern die Steuerung der Simulation übergeben.

Zu Punkt C): Nach Beendigung der Simulation ist die Funktion TSimulator::Done für das Abschließen der Statistiken verantwortlich. In der weiteren Folge ermittelt die Funktion anhand der gesammelten statistischen Daten in den einzelnen Modellelementen Leistungsgrößen des Modells und überträgt diese in die Klasse Result, die als Ergebnisschnittstelle zwischen Simulationskomponente und Oberfläche fungiert:

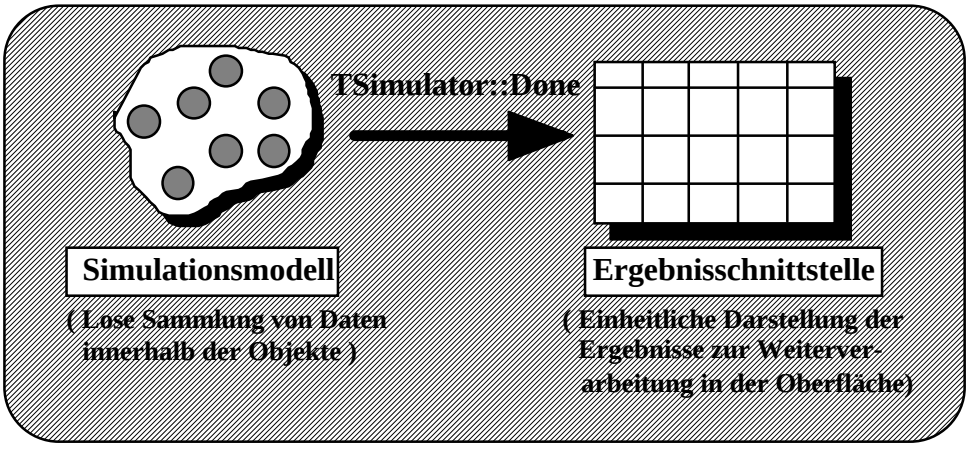

Abbildung 4-17: Die Funktion TSimulator::Done

Zu Punkt D): Zur Umsetzung der Verifizierungsfunktion gibt es zwei Möglichkeiten. Erstens ist es möglich, ein gegebenes formales Modell zu überprüfen. Diese Möglichkeit wurde bei der Entwicklung der BNETD-Simulationskomponente angewandt. Sie hat den Vorteil, daß vor dem Aufbau des Simulationmodells eventuelle Fehler entdeckt werden können. Die zweite Möglichkeit besteht in der Verifizierung des aufgebauten Simulationsmodells. Für diese Herangehensweise spricht, daß eine solche Funktion Verify recht universell ist und weiterverwendet werden kann.
Verifiziert werden unter anderem Typ und Parameter von Ankunfts- und Bedienstrom und die Definition von Lastabhängigkeiten und Preemptivität. Andere Funktionen der Verifizierungsschicht, wie Strukturprüfung und Prüfung der Übergangswahrscheinlichkeiten, werden durch die Oberfläche zur Verfügung gestellt.

4.7.2. Konventionen und Potential der Simulations-Klassen

Der Entwurf der Klassen der Simulationsbibliothek erfolgte so, daß diese ein Potential besitzen, welches die notwendigen Fähigkeiten zur Simulation von "klassischen" Warteschlangennetzen übertreffen soll. Äußern soll sich dieser Anspruch in einer hohen Kombinierkeit ver-

schiedener Objekte. Aufgrund der Orientierung auf eine leistungsfähige spezielle Simulationskomponente für das Programmsystem BNETD wurden Kompromisse in der Universalität eingegangen, die der aktuellen Version der Klassen der Simulationsbibliothek einige Konventionen auferlegen. In der nachstehenden Matrix wird die Kombinierkeit der Objekte wesentlicher Klassen der Bibliothek zusammengestellt:

VOR / NACH	QUELLE	TROUTER	QUEUE	BEDIEN-ANLAGE	SENKE
QUELLE	-	X	-	-	X
TROUTER	-	X	X	X	X
QUEUE	-	-	X	X	X
BEDIEN-ANLAGE	-	X	-	-	X
SENKE	-	-	-	-	-

Tabelle 4-10: Kombinierbarkeit der Objekte der Simulationsbibliothek

Als sinnvolle Erweiterung kann abgeleitet werden, daß die Hierarchie der Warteschlangen um einen Nachfolger erweitert werden sollte, der in der Lage ist, WAIT_FOR_READY-Nachrichten zu verarbeiten und somit als Vorgänger von TRouter-Objekten infrage kommt.
Eine zweite Erweiterung der Klassenhierarchie betrifft Bedienanlagen-Klassen. Für die Anwendung als Werkzeug zur Leistungsbewertung von Rechensystemen sollte eine Bedienanlagen-Klasse geschaffen werden, die nach den Strategien Round Robin oder Processor Sharing arbeiten kann.

4.7.3. Fehlerbehandlung

Die Simulationskomponente von BNETD kennt mehrere Kategorien von Fehlern:

- Warnungen
- Fehlermeldungen der Verifikationsschicht
- Fehlermeldungen des Simulatorkerns
- Fehlermeldungen der statischen Simulationsobjekte

Warnungen führen nicht zum Abbruch der Simulation, sie werden jedoch nach Beendigung derselben als Statuswert an den Aufrufer (hier: die BNETD-Oberfläche) zurückgegeben. Beispiele für Warnungen sind:

- Warnung, daß im Modell Deadlocks möglich sind,
- Warnung, daß während der Simulation negative Zwischenankunftszeiten oder Bedienzeiten auftraten. Möglich ist dieser Fall bei Verwendung einer Normalverteilung. Derartige Zeiten werden auf einen Wert geringfügig größer als null korrigiert.

Warnungen werden durch später auftretende Fehler überschrieben. Fehler führen zum Abbruch der Simulation. In der Regel sind die bis zu diesem Zeitpunkt vorliegenden Ergebnisse verwendbar.

4.7.4. Die Konstante BMODELL

Wenn ein Simulationsmodell des Typs "BNETD-Modell" vorliegt, ist eine bestimmte Effizienzsteigerung und ein erweiterter Leistungsumfang der Simulation möglich. Durch die Kenntnis der festgelegten Struktur eines Bedienknotens *TUnify* $\Rightarrow$ *Warteschlange* $\Rightarrow$ *Bedienanlage* $\Rightarrow$ *TSplit* sind folgende Verbesserung in Funktion und Leistung der Simulation ermöglicht worden:

- Vermeidung überflüssiger Ready-Signale an Quelle-Objekte

- Effizientere Kommunikation zwischen Sender und Empfänger einer Forderung
- Führung einer Statistik über Blockierungen
- Erkennung von Deadlocks zur Laufzeit der Simulation

Die Realisierung erfolgte mit Hilfe der Präprozessor-Konstanten BMODELL. Ist diese zur Übersetzungszeit definiert, so erzeugt der Compiler Objekte, die von der Struktur eines BNETD-Modells ausgehen. Durch diese Konstante ist es möglich, ohne großen Änderungsaufwand eine allgemeine oder eine auf BNETD spezialisierte Version der Simulationsbibliothek zu erstellen.

4.7.5. Anmerkungen zur Portabilität

Die Erfüllung des Kriteriums der Portabilität war ein Teilziel während des Entwurfs des Simulationspaketes. Aufgrund der hohen Verbreitung der Sprache C++ bestehen bei der Nutzung dieser Sprache gute Voraussetzungen für ein portierbares Softwareprodukt. Für die Sprache C++ existiert ein Quasi-Standard, der durch die AT&T-cfront-Compilerversionen gegeben wird. Der AT&T-Standard 2.1 definiert alle wichtigen Erweiterungen von C++ gegenüber C, zu nennen sind Klassen, einfache und mehrfache Vererbung, Polymorphismus, überladbare Operatoren und abstrakte Klassen. An der Erweiterung und Veröffentlichung eines aktuellen Standards arbeitet ein ANSI-C++-Komitee. Der für die Implementation von BNETD verwendete Compiler ist AT&T-cfront 3.0 kompatibel (Unterstützung von Templates). Die momentan neueste Version, AT&T-cfront 3.1, definiert als wichtigste Erweiterung die Ausnahmebehandlung. Compiler, die dem Quasi-Standard 3.0 entsprechen, stehen unter anderem für UNIX (GNU C++) und OS/2 (Borland C++ 1.5) zur Verfügung. Unter DOS/WINDOWS liegt eine Reihe von Compilern vor, die diesen Standard erfüllen (Borland, Microsoft, Watcom und andere).
Für die Realisierung der Simulationsbibliothek wurden keine nicht portablen Bibliotheken verwendet.
Eine Problemstelle bei einer Portierung wird die Initialisierung der Zufallszahlengeneratoren mittels der Systemzeit darstellen, diese müßte systemabhängig neu implementiert werden.
Ein Ansatz zur Erreichung leichter Portierbarkeit wird in [Frank 93] vorgestellt. In einem globalen Headerfile werden bestimmte Standard-Datentypen umdefiniert, um sie einer eventuell anderen Wortlänge des Prozessors anzupassen. Dadurch wird der Änderungsaufwand bei Portierungen herabgesetzt. Ein solches Vorgehen erweist sich jedoch nur als notwendig, wenn auf Datenstrukturen direkt durch Adreßmanipulationen zugegriffen wird.

4.8. Externe Werkzeuge

4.8.1. Kommandozeilenversion des Simulators

Die Kommandozeilenversion des Simulators entstand während der Entwicklung der Simulationskomponente für BNETD, um Korrekturen und Änderungen schneller durchführen zu können. Die Kommandozeilenversion bietet für den fortgeschrittenen Nutzer gegenüber der in BNETD eingebundenen Version bei gleicher Funktionalität einige Vorteile:

- Geringer Speicherbedarf des Programms (etwa 120 kByte)
- Geringfügig höhere Simulationsgeschwindigkeit als die integrierte Version
- Pseudografische Darstellung von Deadlock-Gefahren (bereinigte Deadlock-Matrix, siehe dazu Abschnitt 4.5.)
- Wiedergabe der Blockierungsmatrix (siehe Abschnitt 4.5.), falls ein Deadlock auftrat
- Mögliche Umleitung der Ergebnisdaten in ein File oder zum Drucker

- Mögliche Erzeugung eines binären Ergebnisfiles unterstützt die Weiterverarbeitung der Resultate
- Verifizierbarkeit der Arbeitsweise der Simulation durch die print-Funktionen der Simulationsobjekte

Demgegenüber bietet die integrierte Version die Möglichkeit zum Stoppen der Simulation vor Ablauf der vorgegebenen Realzeit oder Modellzeit und bessere Möglichkeiten zur Ergebnisrepräsentation.
Die Syntax des Aufrufes der Kommandozeilenversion lautet wie folgt:

SIM Modellfilename[.BND] [Maximale reale Simulationsdauer [binäres Ergebnisfile]]

Zu empfehlen ist die Umleitung der Ausgabe in ein File. Die Nutzung der Kommandozeilenversion könnte wie folgt geschehen:

SIM c:\Modelle\ws_netz 60 c:\res\binres >c:\output\a.txt

Letzteres Beispiel löst das durch das File ws_netz.bnd beschriebene Modell simulativ. Die reale Simulationsdauer beträgt maximal 60 Sekunden. Die Ergebnisdatenstruktur (Klasse Result) wird binär im File binres.bnr abgelegt, die Auswertung der Simulation erfolgt in das File a.txt. Beispiele für die von der Kommandozeilenversion erzeugten Ausgabefiles können dem Anhang F entnommen werden.

4.8.2. CHISQUAR und ZZTEST

Zur Testung der Funktion der Zufallszahlengeneratoren wurden zwei kleinere Testprogramme implementiert. Das Programm Chisquar führt einen an den χ^2-Test angelehnten Test durch, das Programm ZZTest testet Erwartungswert und Streuung der in der Simulationsbibliothek implementierten Zufallszahlengeneratoren für spezielle Verteilungen.

CHISQUAR

In [Sedgewick 90] wird nachstehende Methode zur Prüfung der Qualität eines Zufallszahlengenerators angegeben, die sich leicht implementieren läßt. Zunächst wird eine Testgröße analog dem χ^2-Test ermittelt:

$$T = \frac{\sum_{i=1}^{r}\left(f_i - \frac{N}{r} \right)^2}{\frac{N}{r}}$$

Es werden N gleichverteilte Zufallszahlen erzeugt, die in r Klassen gleicher Breite eingeteilt werden. Die erwartete Häufigkeit innerhalb der Klassen beträgt N/r, die empirisch ermittelte Häufigkeit f_i. Die Nullhypothese H_0 sagt aus, daß die so ermittelte empirische Verteilungsfunktion einer Gleichverteilung entspricht. Der χ^2-Test erlaubt eine Entscheidung über die Ablehnung der Nullhypothese durch Vergleich des Testwertes T mit einem aus Tabellen entnommenen Wert der χ^2-Verteilung mit r-1 Freiheitsgraden und einer Irrtumswahrscheinlichkeit α.
In [Sedgewick 90] ist eine Näherungsmethode angegeben, die ohne das Nachschlagen des Wertes der χ^2-Verteilung auskommt. Voraussetzung ist, daß N mindestens 10mal größer als r sei (Vermeidung "leerer" Klassen). Die Irrtumswahrscheinlichkeit α für das ungerechtfertigte Ablehnen der Nullhypothese beträgt 10%.

Die Nullhypothese wird abgelehnt, wenn der Testwert T größer als $r+2\sqrt{r}$ ist. [Sedgewick 90] und sinngemäß [Langendörfer 92] verschärfen die Testbedingungen - es wird verlangt, daß die Testgröße T innerhalb eines Intervalls $\left[r-2\sqrt{r}, r+2\sqrt{r}\right]$ liegen soll. Die Begründung lautet "*...der Kern des Tests besteht jedoch darin: Die Häufigkeit des Auftretens der Werte sollte nicht **exakt gleich** sein - dieses wäre kein Zufall.*"[Sedgewick 90].
Das Programm Chisquare ist eine direkte Umsetzung dieser Berechnungsvorschrift. Die Syntax des Programmaufrufs lautet:

CHISQUAR Anzahl Zufallszahlen Anzahl der Klassen

Im Kapitel 5 ist sowohl ein Test des verwendeten Generators nach dem Näherungsverfahren von Sedgewick als auch ein "klassischer" χ^2-Test enthalten.

ZZTEST

Das Programm ZZTest ermöglicht den Test der vom Basisgenerator abgeleiteten speziellen Zufallszahlengeneratoren hinsichtlich Erwartungswert und Streuung. Die Syntax des Programmaufrufs lautet:

ZZTEST Anzahl Zufallszahlen Generator Parameter1 [Parameter2]

mit "Generator" entsprechend den aufgeführten Schaltern:

- e: Exponentialverteilung; Parameter 1 ist der Erwartungswert
- n: Normalverteilung; Parameter 1 ist der Erwartungswert, Parameter 2 die Standardabweichung
- g: Gleichverteilung; Parameter 1 ist die untere, Parameter 2 die obere Grenze des Intervalls
- l: Erlangverteilung; Parameter 1 ist der Erwartungswert, Parameter 2 die Ordnung der Erlangverteilung

Das Programm ermittelt für die gegebene Anzahl Zufallszahlen empirische Werte für den Erwartungswert und die Streuung. Die gleichfalls angegebenen theoretischen Größen für Erwartungswert und Streuung ermöglichen einen unmittelbaren Vergleich.

5. Test des Simulators

5.1. Test der Zufallszahlengeneratoren

5.1.1. Vorbemerkungen

Im vorliegenden Abschnitt sollen Testmethoden und -ergebnisse, die die implementierten Zufallszahlengeneratoren betreffen, dargestellt werden.
Für die Prüfung von Zufallszahlengeneratoren existieren zwei Hauptgruppen von Tests:

A. Anpassungstests. Diese prüfen die Anpassung einer empirischen Folge von Zufallszahlen an eine vorgegebene theoretische Verteilung. Die wichtigsten Vertreter sind der χ^2-Anpassungstest und der Kolmogorov-Smirnov-Anpassungstest (KSA).

B. Unabhängigkeitstests. Solche Testverfahren prüfen die Unabhängigkeit der erzeugten Zufallsgrößen. Vertreter der Unabhängigkeitstests sind der Serientest, der Lückentest, der Karten-Sammler-Test, der Pokertest, der Maximum-von-t-Test, der Kollisionstest und der Run-Test.

Bei der Anwendung der Tests sind folgende Punkte zu beachten:

- Es ist kein endgültiger Beweis für einen "guten" Generator möglich, trotzdem sollten möglichst viele verschiedene Tests mit mehreren Testläufen durchgeführt werden.
- Das negative Ergebnis eines Tests muß nicht zum Verwerfen eines Generators führen.

Die oben angeführten Testverfahren sind in [Langendörfer 92] näher erläutert.
In diesem Abschnitt soll exemplarisch der χ^2-Anpassungstest durchgeführt werden. Anschließend werden Testergebnisse, die mit Hilfe der Programme Chisquar und ZZTest erhalten wurden, dargestellt und interpretiert.

5.1.2. χ^2-Anpassungstest

Der χ2-Anpassungstest wurde bereits um das Jahr 1900 von Karl Pearson entwickelt. Zur Prüfung des Basisgenerators auf Anpassung an eine Gleichverteilung geht man in folgender Weise vor:
1.Schritt:
Vom Zufallszahlengenerator werden N diskrete Werte erzeugt. Diese Zufallszahlen werden in r Klassen gleicher Breite eingeteilt. Die Klassenbreite ist so zu wählen, daß keine Klasse leer ist.
2.Schritt:
Aufstellung der Nullhypothese H_0: "Die durch den Zufallszahlengenerator erzeugte Folge unterliegt einer Gleichverteilung".
3.Schritt:
Prüfung der theoretischen Häufigkeiten: Die theoretischen Häufigkeiten betragen aufgrund der einheitlichen Klassenbreite N/r. Die Klassenbreite ist zusätzlich so zu wählen, daß die theoretische Häufigkeit für jede Klasse mindestens fünf beträgt.
4.Schritt:
Ermittlung einer Testgröße T:
Aus den empirisch ermittelten Häufigkeiten f_i und den theoretischen Häufigkeiten wird eine Testgröße T ermittelt (analog zu der bereits im Abschnitt 4.8.2. angeführten Berechnungsmethode):

$$T = \frac{\sum_{i=1}^{r}\left(f_i - \frac{N}{r} \right)^2}{\frac{N}{r}}$$

5.Schritt:
Auswahl einer Irrtumswahrscheinlichkeit α. α gibt eine Schranke für die Wahrscheinlichkeit einer ungerechtfertigten Ablehnung der Nullhypothese an. Die Ablehnung einer richtigen Hypothese stellt einen Fehler 1.Art dar, die Nichtablehnung einer falschen Hypothese einen Fehler 2.Art. Die Wahrscheinlichkeit für einen Fehler 2.Art kann durch Erhöhung des Stichprobenumfangs N verringert werden.
6.Schritt:
Testentscheidung. Vergleich des Testwertes T mit dem Quantil $\chi^2_{r-1,\alpha}$, der aus Tabellen [Göhler 89] entnommen werden kann.

Beispiel:
Mittels des Basiszufallszahlengenerators werden 1000 Zufallszahlen erzeugt und in 10 Klassen eingeteilt. Die jeweiligen Häufigkeiten sind der nachstehenden Tabelle zu entnehmen:

Klasse i	**Häufigkeit f_i**
1	101
2	95
3	101
4	76
5	100
6	124
7	101
8	105
9	101
10	96
Σ	1000

Tabelle 5-1: χ 2-Anpassungstest

Als Nullhypothese wird formuliert: "Die erzeugten Zufallszahlen sind gleichverteilt." Die theoretischen Häufigkeiten betragen jeweils 100. Die Berechnung der Testgröße T ergibt:

$$T = \frac{\sum_{i=1}^{10}\left(f_i - 100 \right)^2}{100} = 12{,}22$$

Relevante Quantile der χ2-Verteilung mit 9 Freiheitsgraden lauten:

Irrtumswahrscheinlichkeit α	0.01	0.05	0.10	0.20	0.90
Quantil der χ2-Verteilung	21.7	16.9	14.68	12.24	4.17

Tabelle 5-2: Quantile der χ 2-Verteilung

Wegen 12.22 < 12.24 kann die Nullhypothese H_0 bis zu einer Irrtumswahrscheinlichkeit von α =20% nicht abgelehnt werden.

Der nächste Abschnitt zeigt eine Variante des χ2-Anpassungstests, für welche die Nutzung von Tabellen nicht erforderlich ist und sich daher besonders für eine Implementierung als Rechnerprogramm eignet.

5.1.3. Test des Basisgenerators

In diesem Abschnitt sollen Resultate des Tests des Basisgenerators mit Hilfe des Programmes Chisquare (siehe vorhergehendes Kapitel) angeführt werden. Der Test wurde durchgeführt, indem 1000 Zufallszahlen erzeugt und diese in 20 Klassen eingeteilt wurden. Der Test wurde 10fach wiederholt. Nach der Näherungsregel [Sedgewick 90] wird die Zufallsgröße T im Intervall [11.05,28.94] erwartet.
Anmerkung: An dieser Stelle ergibt eine Prüfung des angegebenen Intervalls:
Der Wert 11.05 entspricht etwa dem Quantil der χ^2-Verteilung mit 19 Freiheitsgraden und einer Irrtumswahrscheinlichkeit α=90%, der Wert 28.94 dem Quantil χ^2_{19} mit α=10%.
Sämtliche ermittelten Testwerte T lagen innerhalb des gegebenen Intervalls. Die folgende Tabelle stellt die Testergebnisse mit entsprechenden Irrtumswahrscheinlichkeiten (aus Tabellen der χ^2-Verteilung entnommen) zusammen:

Nummer des Testlaufes	Testwert T	Irrtumswahrscheinlichkeit
1	22.2	25%
2	17.5	55%
3	24.1	20%
4	19.0	45%
5	17.4	55%
6	16.3	60%
7	17.8	55%
8	19.9	40%
9	22.3	25%
10	16.4	60%
Mittel:	19.3	45%

Tabelle 5-3: Test des Basiszufallszahlengenerators mittels des Programmes Chisquar

Das Ergebnis des Tests sagt aus, daß die Hypothese "Der Basiszufallszahlengenerator erzeugt Pseudozufallszahlen, die einer Gleichverteilung unterliegen." auch bei 45% Irrtumswahrscheinlichkeit nicht abgelehnt werden kann.

5.1.4. Test der transformierten Generatoren

Das Programm ZZTest wurde entwickelt, um die korrekte Transformation der im Intervall [0,1) erzeugten gleichverteilten (Pseudo-)Zufallszahlen zu prüfen. Das Programm arbeitet nach der Abfolge:

1. Erzeugung von N Zufallszahlen einer Verteilung.
2. Vergleich des theoretischen Erwartungswertes und der theoretischen Streuung mit den empirisch ermittelten Werten.

Für den durchgeführten Test wurden jeweils 10000 Zufallszahlen einer Verteilung erzeugt. In der Kopfzeile sind die theoretischen Werte für Erwartungswert und Streuung der entsprechenden Verteilung aufgeführt. Folgende Resultate wurden erzielt:

1.Untersuchung einer Exponentialverteilung mit λ=0.2

Mittelwert ($1/\lambda$ = 5)	**Streuung ($1/\lambda^2$=25)**
4.90	24.04
5.03	25.45
4.98	24.83

Tabelle 5-4: Untersuchung des Exponentialverteilungsgenerators

2.Untersuchung einer Erlangverteilung mit λ=0.4 und der Ordnung k=2

Mittelwert (k/λ = 5)	**Streuung (k/λ^2=12.5)**
4.97	12.28
4.98	12.52
4.96	12.53

Tabelle 5-5: Untersuchung des Erlangverteilungsgenerators

3.Untersuchung einer Normalverteilung mit μ = 10 und σ = 3

Mittelwert (μ = 10)	**Streuung (σ^2 = 9)**
10.04	8.96
10.02	8.98
9.99	9.07

Tabelle 5-6: Untersuchung des Normalverteilungsgenerators

4.Untersuchung einer Gleichverteilung in [1,6] mit a=1 und b=6

Mittelwert ($\frac{a+b}{2}$ = 3.5)	**Streuung ($\frac{(b-a)^2}{12}$ = 2.08)**
3.50	2.08
3.49	2.07
3.52	2.10

Tabelle 5-7: Untersuchung des Gleichverteilungsgenerators

Im Ergebnis der Untersuchungen wurden keine wesentlichen Mängel an den implementierten Zufallszahlengeneratoren festgestellt.

5.2. Vergleich mit analytischen Ergebnissen

Die Überprüfung des ordnungsgemäßen Funktionierens der Simulationskomponente gestaltete sich komfortabel, da die bereits bestehende Analysekomponente des Programmsystems BNETD genutzt werden konnte. Der Vorteil besteht insbesondere in der raschen Vergleichsmöglichkeit mit exakten Resultaten, nachteilig dagegen ist die eingegrenzte Funktionalität der Analysekomponente, welche lediglich Größen für einen engen Kreis von Modellen ermittelt. Als Nebeneffekt konnten durch diese Vorgehensweise Fehler der Analysekomponente entdeckt werden. Grundlage für die durchgeführten Versuche waren vorrangig Beispielmodelle, die in [Bolch 82] angegeben sind.
Für die Bewertung der Resultate sind nachstehende Kriterien interessant:

- Die simulativen Ergebnisse konvergieren gegen die analytischen Ergebnisse. (grundlegendes Kriterium der Korrektheit)
- Nach welcher realen Zeit wird eine "befriedigende" Konvergenz erreicht? (Kriterium der Effizienz)

Anhand des folgenden Beispiels soll ein Vergleich von simulativen Resultaten und exakten analytischen Ergebnissen erfolgen:

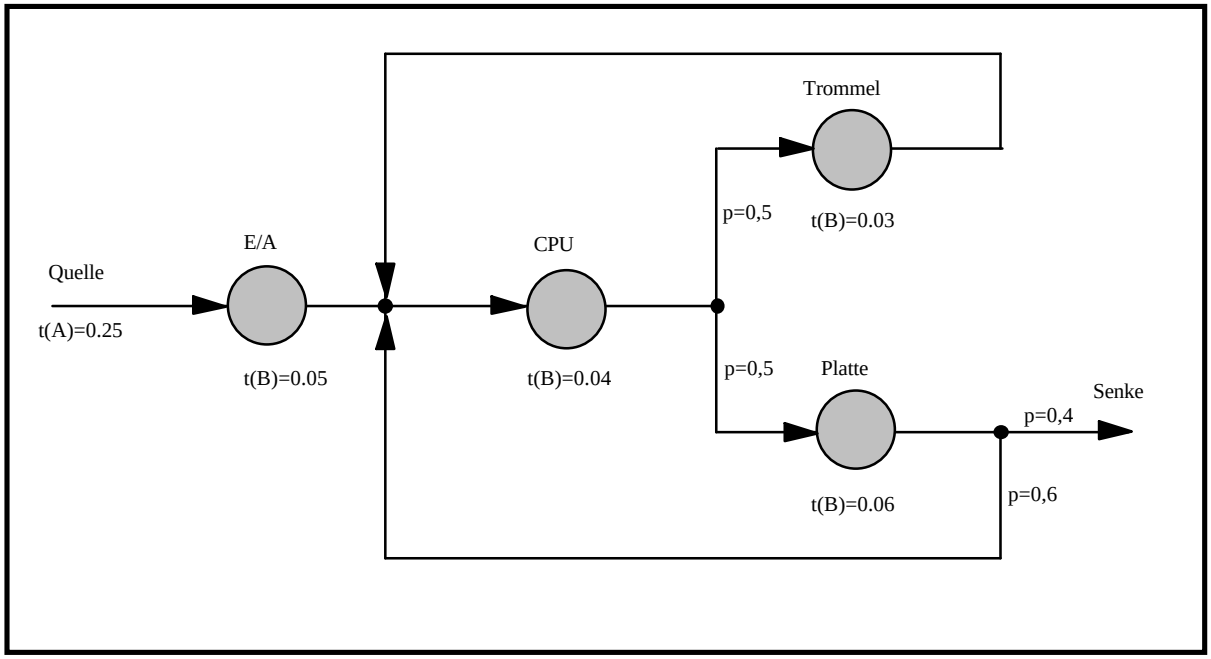

Abbildung 5-1: Beispiel eines Rechensystems nach [Bolch 82]

Das gegebene offene Warteschlangennetz soll untersucht werden. Sämtliche Bedienknoten verfügen über einen Warteraum mit unbeschränkter Kapazität, als Warteschlangenstrategie wurde jeweils FIFO festgelegt. Die Bedienzeiten der Forderungen sind exponentiell verteilt. Es existiert eine Klasse von Forderungen.
Unter Nutzung der BNETD-Analysekomponente wurden folgende exakte Ergebnisse ermittelt:

	CPU	Trommel	Platte	E/A
Durchsatz	20	10	10	4
Auslastung	0.8	0.3	0.6	0.2
Bedienzeit	0.04	0.03	0.06	0.05
Wartezeit	0.16	0.0129	0.09	0.0125
WS-Länge	3.2	0.129	0.9	0.05
WarteWkt	0.8	0.3	0.6	0.2
Verweilzeit	0.2	0.0429	0.15	0.0625
Füllung	4	0.429	1.5	0.25

Tabelle 5-8: Analyseergebnis für Rechensystem nach [Bolch 82]

Die gleichfalls durchgeführten Simulationsläufe wurden mehrfach mit verschiedener Simulationsdauer (30 Sekunden, 120 Sekunden, 1800 Sekunden und 10800 Sekunden) durchgeführt. Die Hypothese, daß mit zunehmender Simulationsdauer die Ergebnisse stärker gegen die analytisch ermittelten Ergebnisse konvergieren, konnte bestätigt werden. Als Testhardware wurde ein PC 486SX mit 20 MHz Taktfrequenz genutzt. Durch Nutzung leistungsfähigerer Hardware werden die Ergebnisse weiter verbessert.
Die folgenden Tabelle zeigt die ermittelten Resultate bei einer Simulationsdauer von 30 Sekunden. Neben den festgestellten Ergebnissen (diese können gleichfalls dem Anhang entnommen werden), ist die prozentuale Abweichung vom analytisch ermittelten Wert eingetragen. Die Fußzeile enthält die mittlere Abweichung für den jeweiligen Bedienknoten.

(Dauer: 30sec)	CPU	Abw. in %	Trom-mel	Abw. in %	Platte	Abw. in %	E/A	Abw. in %
Durchsatz	20.9	4.5	10.6	6	10.2	2	4.15	3.7
Auslastung	0.82	2.5	0.36	20	0.65	8.3	0.22	10
Bedienzeit	0.039	2.5	0.032	6.7	0.063	5	0.054	8
Wartezeit	0.20	25	0.015	16	0.11	22	0.021	68
WS-Länge	4.25	33	0.16	24	1.20	33	0.087	74
WarteWkt	0.84	5	0.33	10	0.69	15	0.27	35
Verweilzeit	0.24	20	0.047	9.5	0.18	20	0.075	20
Füllung	5.06	26	0.49	14	1.85	23	0.31	24
mittlere Abweichung in %		14.8		13.3		16.0		30.3

Tabelle 5-9: Simulationsergebnisse bei einer Dauer von 30 Sekunden

Die Resultate zeigen, daß eine qualitative Aussage zu Leistungsgrößen getroffen werden kann, es sind jedoch teilweise erhebliche Abweichungen (bis zu 74%) vom exakten Wert zu registrieren. Es ist zu vermuten, daß die Nichtberücksichtigung der Einschwingphase eine Rolle spielt.
Die Erhöhung der Simulationsdauer führt zu einer Annäherung an die exakten Werte. Die folgende Tabelle stellt die ermittelten Resultate für eine Simulationsdauer von 3 Stunden dar.

(Dauer: 3 h)	CPU	Abw. in %	Trom-mel	Abw. in %	Platte	Abw. in %	E/A	Abw. in %
Durchsatz	20.22	1.1	10.15	1.5	10.07	0.7	4.04	1
Auslastung	0.814	1.7	0.301	0.3	0.613	2.2	0.204	2
Bedienzeit	0.040	0	0.0296	1.3	0.061	1.7	0.050	0
Wartezeit	0.154	3.7	0.0123	4.6	0.086	4.4	0.013	4
WS-Länge	3.12	2.5	0.125	3.1	0.86	4.4	0.053	6
WarteWkt	0.809	1.1	0.297	1	0.605	0.8	0.207	3.5
Verweilzeit	0.194	3	0.0419	2.4	0.147	2	0.063	0.8
Füllung	3.93	1.8	0.426	0.7	1.473	1.8	0.257	2.8
mittlere Abweichung in %		1.86		1.86		2.25		2.51

Tabelle 5-10: Simulationsergebnisse bei einer Dauer von 3 Stunden

Wie zu erwarten war, stimmen die ermittelten Werte weitgehend mit den exakten Werten überein, die Abweichung beträgt im Mittel etwa 2 Prozent. Auffällig ist erneut, daß die Statistik für Warteschlangen mit einer größeren Ungenauigkeit behaftet ist als die Statistik der Bedienanlagen. Diese Tatsache war allen durchgeführten Simulationsläufen und Modellen gemeinsam.
Die Abweichungen der Simulationsergebnisse bei einer Dauer von 120 und 1800 Sekunden lagen erwartungsgemäß zwischen den Abweichungen der hier ausgewerteten Resultate. Auf ihre Darstellung soll daher verzichtet werden.
Eine Empfehlung für die Festlegung der Simulationsdauer kann nur unter Kenntnis der zur Verfügung stehenden Hardware erfolgen. Vor der Überbewertung von Resultaten, die bei einer kurzen Simulationsdauer (weniger als 30 Sekunden) erzielt wurden, ist zu warnen. Auf jeden Fall bietet sich die Möglichkeit an, die Simulationsdauer recht groß zu wählen und bei Bedarf zu stoppen (diese Möglichkeit existiert vorerst nur für die in BNETD integrierte Simulationskomponente, nicht für die Kommandozeilenversion).

Aus dem eingangs des Abschnittes Gesagten geht hervor, daß insbesondere für die "fortgeschrittenen" Möglichkeiten der Simulationskomponente (begrenzte Warteschlangenkapazitäten, Blockierungen, weitere implementierte Verteilungen und Warteschlangenstrategien) andere Testmethoden gefunden werden müssen.

5.3. Vergleich mit anderen Simulationssystemen

Für die Prüfung der Resultate der BNETD-Simulationskomponente wurde das GPSS-kompatible Simulationssystem PS SIMDIS-2 verwendet. Für ausführliche Hinweise zu SIMDIS soll Herrn Dr.R.Koch herzlich gedankt werden.
Die besondere Eigenschaft der simulativen Lösung macht ein breites Anwendungsspektrum aus. Für den durchzuführenden Test wurde folgendes offene mehrklassige Warteschlangennetz benutzt:

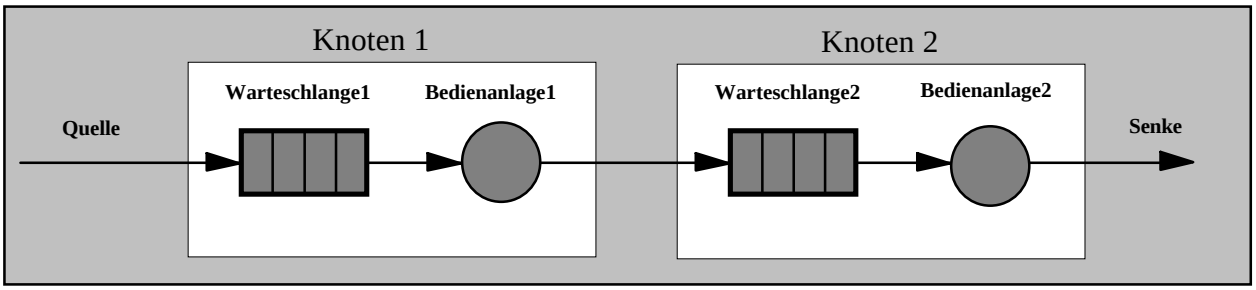

Abbildung 5-2: Beispiel eines offenen Warteschlangennetzes

Im gegebenen Netz existieren zwei Forderungsklassen. Beide Forderungsklassen nehmen den oben abgebildeten Lauf durch das Modell. Die Zwischenankunftszeiten sind exponentiell verteilt mit einem Erwartungswert von 65 (Klasse 1) bzw. 80 (Klasse 2). Die Bedienzeit im Knoten 1 ist für die Klasse 1 normalverteilt mit μ=20 und σ=3, für die Klasse 2 normalverteilt mit μ=25 und σ=6. Die Warteschlange 1 kann unendlich lang sein, die Bedienanlage 1 verfügt über einen Bedienkanal. Die Bedienzeit im Knoten 2 genügt einer Gleichverteilung im Intervall [50,70] für die Klasse 1 bzw. [60,80] für die Klasse 2. Der Warteraum des 2.Knotens hat eine Kapazität von 3 Plätzen, die Bedienanlage 2 verfügt über 2 Bedienkanäle.
Aufgrund der beschränkten Warteraumkapazität können in diesem Modell Blockierungen auftreten. Die simulative Lösung erfolgte sowohl mit SIMDIS als auch mit BNETD. Die Simulationsdauer wurde auf 50000 Zeiteinheiten beschränkt. Damit konnte gleichfalls ein Vergleich der benötigten Rechenzeit für die Lösung des Modells erfolgen. Im Ergebnis des Versuchs benötigte SIMDIS etwas weniger als eine Minute, BNETD dagegen etwas mehr als eine Minute.
Eine Auswahl wesentlicher Ergebnisse zeigt die nachstehende Übersicht.

Leistungsgröße	SIMDIS	BNETD
Auslastung Bedienanlage 1	0.70	0.721
Auslastung Bedienanlage 2	0.89	0.884
mittl. Dauer von Blockierungen in BA1	-	3.796
BlockierungsWkt. in BA1	-	0.247
mittl. Bedienzeit (BA 1)	25.53-t(Block)	22.41
mittl. Bedienzeit (BA 2)	64.4	64.54
mittl. Wartezeit in WS 1	45.12	82.53
mittl. Wartezeit in WS 2	51.9	53.84
mittl. WS-Länge 1	1.24	2.26
mittl. WS-Länge 2	1.4	1.47

Tabelle 5-11: Vergleich von SIMDIS- und BNETD-Resultaten

Zunächst soll die Zeile "mittlere Bedienzeit (Bedienanlage 1)" erläutert werden. Das genutzte SIMDIS-Modell ermittelte im Vergleich zu BNETD relativ wenig Leistungsgrößen. So setzt sich der Wert 25.53 aus der mittleren Bedienzeit in der Bedienanlage 1 und der mittleren Dauer von Blockierungen zusammen. Die entsprechende Summierung für die von BNETD ermittelten Leistungsgrößen ergibt 22.41+3.796 = 26.206.
Auffällig bei der Betrachtung der Resultate ist die deutliche Abweichung der Leistungsgrößen für die Warteschlange 1. Ein vermuteter Modellierungsfehler konnte nicht entdeckt werden, zumal die Leistungsgrößen für den Bedienknoten 2 gut übereinstimmen. Eine grobe Abschätzung durch die Anwendung von Formeln, die für ein M/M/1-Modell gültig sind, ergeben bei einer angenommenen Auslastung von 0.71 und einer mittleren Bedienzeit von 25 ZE folgende Werte: Mittlere Wartezeit = 25 * 0.71 / (1-0.71) = 61.2 und mittlere Warteschlangenlänge = $(0.71)^2$ / (1-0.71) = 1.74.
Mögliche Ursachen für die aufgetretenen Abweichungen können weiterhin sein:

- Zu kurze Simulationsdauer
- SIMDIS erzeugt Ereignisse nur zu ganzzahligen Zeitpunkten
- Statistik von SIMDIS stimmt mit der in BNETD geführten Statistik nicht überein. Eine Anwendung des Gesetzes von Little (siehe Abschnitt 1.3.) konnte die von BNETD ermittelten Leistungsgrößen bestätigen, jedoch nicht die mit SIMDIS ermittelten Resultate.

Die vollständigen Resultate und das SIMDIS-Listing können dem Anhang entnommen werden.

Das abschließende Kapitel beschreibt die Integration der Simulationskomponente in das Programmsystem BNETD.

6. Einbindung in das BNETD-Projekt

6.1. Struktur und Schnittstellen von BNETD

Die Entwicklung der vorliegenden zweiten Programmversion des BNETD-Systems wurde durch mehrere Autoren im Rahmen von Beleg- und Diplomarbeiten realisiert. Gegenüber anderen Systemen zeichnet sich BNETD durch die simultane Verfügbarkeit sowohl von Analyse- als auch Simulationskomponente zur Behandlung von Warteschlangensystemen/Bediensystemen aus. Eine pseudografische CUA-konforme Oberfläche ermöglicht die intuitive Nutzung des Programmes. Ebenso wurden Komponenten zur grafischen Ein- und Ausgabe sowie zur Ergebnisrepräsentation integriert.
Die Struktur des Programmsystems BNETD skizziert die Abbildung 6-1:

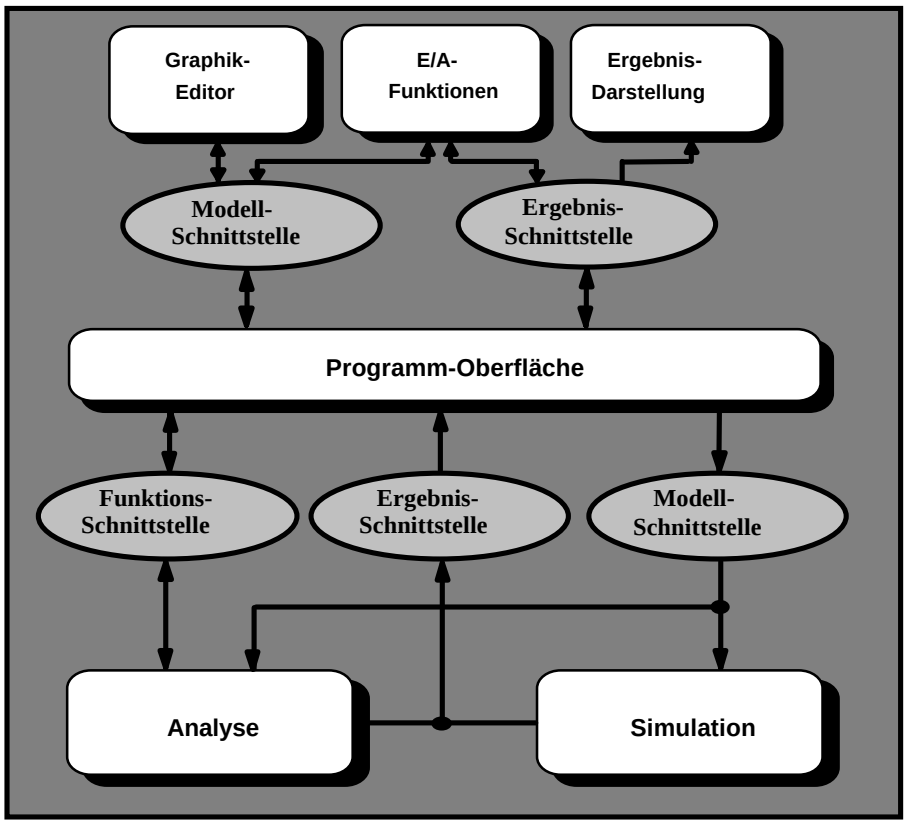

Abbildung 6-1: Struktur und Schnittstellen des Programmsystems BNETD

In der Abbildung wurde die Verifikationsschicht nicht dargestellt. Sie wird von einer Menge von Funktionen gebildet, die in den Hauptkomponenten realisiert sind. Die Hauptkomponenten von BNETD sind die Programm-Oberfläche, die Grafikkomponente, die Analyse- und die Simulationskomponente. Diese sollen nun kurz beschrieben werden:

1. **Programmoberfläche**: Die Nutzerschnittstelle des Programmsystem BNETD ist SAA-CUA konform. Der somit eingehaltene Standard fördert die schnelle Einarbeitung in die Nutzung des Programmes. Der Anwender bestimmt durch Interaktionen den Programmablauf. Die Oberfläche unterstützt mehrere Fenster. Alle Steuerungsaktionen lassen sich sowohl mittels einer Maus als auch durch die Tastatur durchführen. Das Ziel eines Programmsystems, welches eine schriftliche Dokumentation unnötig macht, wird mit einer ausführlichen Hilfefunktion angestrebt, die kontextsensitiv arbeitet. Es ist ein ständiger Überblick über das aktuelle Modell sichtbar, sowohl in einem Statusfenster mit mehreren Unterfenstern als auch als Grafik. Die in der obigen Abbildung nicht dargestellte Verifika-

tionsschicht wird innerhalb der Programmoberfläche unter anderem von einer leistungsfähigen Funktion zur Strukturprüfung für Warteschlangennetze gebildet. Zur detaillierten Beschreibung der Oberfläche des Programmsystems und der Grafikkomponente wird auf [Borriss 93] verwiesen.

2. **Grafikkomponente**: In der aktuellen Version umfaßt die Grafikkomponente Möglichkeiten zur grafischen Eingabe eines Modells, zur Anzeige der Struktur des aktuellen Modells und zur grafischen Darstellung von Leistungsgrößen:
    - Grafische Eingabe: Zum Umfang des Programmsystems gehört ein grafischer Editor. Zur Nutzung des Grafikeditors ist eine Maus, eine VGA-kompatible Grafikkarte und ein Farbmonitor erforderlich. Der Grafikeditor ermöglicht eine schnelle, einfache und anschauliche Modellierung. Ein mit dem Grafikeditor erstelltes Modell wird innerhalb der Oberfläche parametrisiert.
    - Grafische Darstellung: Während der Arbeit mit einem Modell kann der Anwender sich jederzeit das aktuelle Modell grafisch darstellen lassen. Diese Komponente erstellt eine Grafik aus der Modellbeschreibung. Sie arbeitet "intelligent", da sie die Darstellung des Modells auf Wunsch verbessert.
    - Grafische Darstellung von Leistungsgrößen: Diese Funktion wurde neu implementiert und wird im Abschnitt 6.3. beschrieben.
3. **Analysekomponente**: Die Analysekomponente bietet die exakte Lösung eines Modelles an. Es sind Algorithmen zur Lösung einfacher Bediensysteme und von Bedienungs-(Warteschlangen-)netzen realisiert. In der gegenwärtigen Version sind zwei Berechnungsvorschriften für die Behandlung von Netzen enthalten, der Jackson-Algorithmus für offene Netze und die Mittelwertanalyse für geschlossene Netze.
4. **Simulationskomponente**: Die Simulationskomponente ist Gegenstand dieser Arbeit.

Die modulare Aufteilung des Programmsystems macht die klare Definition von Schnittstellen notwendig, über welche die Kommunikation der einzelnen Programmbestandteile erfolgt. In Kapitel 3 wurde ausgesagt, daß, über einen längeren Zeitraum betrachtet, die Daten den stabilen Aspekt eines Softwareprodukts bilden [Meyer 90]. Folglich wurde beim Entwurf der Schnittstellen von BNETD auf Daten, nicht auf Funktionen, orientiert.
Eine wichtige Schnittstelle ist die Modellbeschreibungsstruktur. Alle von BNETD zu behandelnden Modelle können durch eine derartige Struktur erfaßt werden. Wie aus der obigen Abbildung ersichtlich ist, bildet diese Modellbeschreibung für die Komponenten des Programmsystems die Arbeitsgrundlage. Die Integrität der Modellbeschreibungsdateien (*.BND) wird durch Prüfzeichen gewahrt.
Die ermittelten Leistungsgrößen geben Anlaß für die Definition einer weiteren Schnittstelle. Die Ergebnisschnittstelle (die Klasse "Result" - siehe auch Deklaration im Anhang) wurde im Kapitel 4 beschrieben. Ihre wichtigsten Merkmale sind die variable Größe und die Anwendbarkeit sowohl für Analyse- als auch Simulationskomponente. Die Unterbringung der Ergebnisse von Analyse und Simulation in einem gemeinsamen Datenobjekt ist im Hinblick auf die Weiterverarbeitung der Resultate von Interesse. Die momentane Umsetzung erfolgt allerdings nicht konsequent nach diesem Konzept. Die (noch) existierende Ergebnisschnittstelle der Analysekomponente wird in ein Objekt der Klasse Result umgewandelt. Verantwortlich für die Umwandlung ist der Teil der Oberfläche, der für die Ergebnisrepräsentation zuständig ist.
Weiterhin ist ein Funktionsinterface für die Übermittlung der Analyseresultate definiert worden. Die spezielle Problematik der Ermittlung von Zustands- und Randwahrscheinlichkeiten macht dieses erforderlich, da die Unterbringung der möglicherweise unbegrenzten Anzahl von Werten in einer Datenstruktur scheitert.

6.2. Leistungsumfang von BNETD

Im vorliegenden Abschnitt sollen Umfang und Einschränkungen der Modellbildung und Lösung durch das Programmsystem BNETD dargestellt werden. Die Oberfläche (einschließlich des grafischen Editors) erlaubt die Modellierung von einfachen Warteschlangensystemen sowie offenen und geschlossenen Warteschlangennetzen. Alle Systeme können mehrklassig sein. BNETD beschränkt die Modellierung auf Systeme, deren Knoten aus genau einem Warteraum und einer (mehrkanaligen) Bedienanlage bestehen. Die Bedienknoten eines Modells werden durch die Definition eines Ankunftsstromes, eines Bedienstromes, Regeln für den Übergang bearbeiteter Forderungen (Übergangswahrscheinlichkeiten) und Spezifizierung der Warteschlange beschrieben.

Ankunfts- und Bedienstrom
Mögliche Verteilungen der Zwischenankunftszeit und der Bedienzeit in einem Knoten sind Exponentialverteilung, Erlangverteilung, Cox-Verteilung, Hyperexponentialverteilung, Hypoexponentialverteilung, Normalverteilung, Gleichverteilung und deterministische Verteilung. Entsprechend der gewählten Verteilung wird der Ankunfts- bzw. Bedienstrom eines Knotens durch ein oder zwei klassenabhängige Parameter festgelegt. Die Angabe eines Ankunftsstromes ist fakultativ, die Angabe eines Bedienstromes obligatorisch.

Regeln für den Forderungslauf durch das Modell
Der Lauf der Forderungen durch das Modell wird durch die Definition von Übergangswahrscheinlichkeiten realisiert. Diese sind ebenfalls klassenabhängig.

Spezifizierung der Warteschlange
Der Warteraum eines Knotens wird durch seine maximale Kapazität und durch die Entnahmestrategie von Forderungen aus der Warteschlange beschrieben. Zur Verfügung stehende Strategien sind FIFO, LIFO, Entnahme nach Prioritäten und zufällige Entnahme. Die Strategien LIFO und Entnahme nach Prioritäten können preemptiv sein.

Sämtliche Merkmale eines Knotens können dabei kombiniert werden.

Durch die Oberfläche von BNETD gebildete Modelle unterliegen folgenden Einschränkungen, die gleichzeitig Anregungen für zukünftige Erweiterungen geben können:

- Ein Netz beinhaltet maximal 10 Knoten und 5 Forderungsklassen.
- Klassenübergänge sind nicht vorgesehen.
- Erweiterte Warteschlangennetze können nicht modelliert werden. Darunter sind gemeinsame Warteschlangen für einen Knoten, Vereinigen und Aufteilen von Forderungen oder die Modellierbarkeit von Abhängigkeiten (Zuliefervorgänge) zu verstehen.
- Lastabhängigkeit von Ankunfts- und Bedienstrom wird nicht weiter definiert.
- Für den gegebenen Anwendungsbereich (Leistungsanalyse von Rechensystemen) wäre die Implementation der Strategien Processor Sharing (PS) und Round Robin (RR) von Belang.

Ein solches Modell (BNETD-Modell) kann nun analytisch oder simulativ gelöst werden. Eine analytische Lösung setzt voraus:

1. *Einfaches Bediensystem:*
   - Ein einfaches Bediensystem muß "offen" sein
   - Es kommt genau eine Forderungsklasse im Modell vor
   - Die Warteschlangenkapzität muß unbegrenzt sein
   - Warteschlangenstrategie FIFO oder preemptives LIFO

- Poissonsche Ankunftszeitverteilung
- Exponentialverteilte Bedienungszeiten

2. *Offenes Netz:*
   - Zwischenankunftszeit und Bedienzeit unterliegen einer Exponentialverteilung
   - Es existiert genau eine Auftragsklasse
   - Warteraumkapazitäten sind jeweils unbegrenzt
   - Entnahmestrategie ist FIFO

3. *Geschlossenes Netz:*
   - Zwischenankunftszeit und Bedienzeit unterliegen einer Exponentialverteilung
   - Warteraumkapazitäten sind jeweils unbegrenzt

Die Anwendung simulativer Methoden erfordert weniger Einschränkungen. Zu nennen sind die Nichtunterstützung der Warteschlangenstrategien LIFO preemptiv und Priority preemptiv. Von den modellierbaren Verteilungen (siehe oben) unterstützt die Simulationskomponente die Exponentialverteilung, Erlangverteilung, Normalverteilung, Gleichverteilung und deterministische Verteilung.

6.3. Erweiterungen der Oberfläche

Im Zuge der Entwicklung der Simulationskomponente wurden zeitgleich einige Erweiterungen und Anpassungen innerhalb der BNETD-Oberfläche vorgenommen. Als neue Verteilungen wurden Gleichverteilung und Normalverteilung in den Leistungsumfang integriert. Die Hilfefunktion wurde der neuen Funktionalität (Einbindung von Analyse- und Simulationskomponente) angepaßt.
Die Komplexität des Gesamtsystems BNETD führte zu einer Überarbeitung der Speicherbelegung. Aufgrund der verwendeten Overlays kann dem hohen Speicherbedarf der Analysekomponente besser genügt werden. Für die Auslagerung nicht benötigter Module wird vorhandener Erweiterungsspeicher benutzt.
Neben einer Verfeinerung oder Neuentwicklung von Dialogen besteht eine wichtige Erweiterung in der Fähigkeit der Statuszeile des Programmes, kontextsensitive Informationen anzuzeigen. Der Anwender erhält dadurch stets eine Kurzinformation beispielsweise über die Wirkung beabsichtigter Aktionen.
Die Einbindung von Simulations- und Analysekomponente erfolgte in der Weise, daß diese eine Interaktion des Anwenders in den Lösungsprozeß ermöglichen. Während des Lösungsprozesses wird die Arbeitsspeicheranzeige aktualisiert. Der Nutzer hat die Möglichkeit, die simulative oder analytische Lösung abzubrechen. Bei Anwendung der Simulationsmethode gehen durch diesen Schritt keine Daten verloren, die Auswertung erfolgt mit den bis zum Unterbrechungszeitpunkt ermittelten Daten. Eine Wiederaufnahme der Simulation nach einer Unterbrechung ist prinzipiell möglich, wurde aber in der vorliegenden Version nicht realisiert.
Eine signifikante Erweiterung der Funktionalität der Programmoberfläche besteht in der Realisierung der geplanten Komponente zur Ergebnisrepräsentation.

ERGEBNISREPRÄSENTATION

Die Ergebnisrepräsentation kann in BNETD sowohl tabellarisch als auch grafisch erfolgen. Grundlage für die Funktionen zur Ergebnisrepräsentation ist die bereits erwähnte Ergebnisschnittstelle (die Klasse "Result").
Die Ausgabe in Tabellenform innerhalb der BNETD-Oberfläche ist wie folgt realisiert: Ergebnisse, welche für das gesamte Modell gelten ("Globaldaten") werden in einem Dialogfenster zusammengefaßt. Derartige Resultate sind etwa Durchsatz, mittlere Verweilzeit der Forderungen im System und mittlere Anzahl Forderungen im System. Weitere Dialogfenster dienen der

Anzeige von Daten, die für einen Knoten gelten. Betreffende Leistungsgrößen sind beispielsweise Durchsatz, Auslastung, mittlere Verweil-, Warte-, Blockier- und Bedienzeiten, Warte- und Blockierwahrscheinlichkeiten und die mittlere Anzahl von Forderungen in der Warteschlange. Fast alle Leistungsgrößen (bis auf einige Werte wie maximale Warteschlangenlänge, die von untergeordnetem Interesse sind) existieren gleichfalls in einer klassenabhängigen Darstellung. Die klassenabhängigen Leistungsgrößen eines Knotens werden in einem Unterfenster des Knotendatenfensters wiedergegeben. Einen Nachteil stellt das Fehlen einer komfortablen Druckfunktion dar.
Wie eingangs erwähnt, stellt BNETD ebenfalls eine Komponente zur grafischen Wiedergabe der Resultate zur Verfügung (siehe auch Anhang C). Die Komponente stellt eine ausgewählte Leistungsgröße der Knoten eines Modells in einem Säulendiagramm dar. Darstellbare Leistungsgrößen sind Durchsatz, Auslastung, mittlere Anzahl Forderungen in der Bedienanlage, mittlere Anzahl Forderungen in der Warteschlange, mittlere Anzahl Forderungen im Bedienknoten, mittlere Verweil-, Warte-, Blockier- und Bedienzeiten sowie Warte- und Blockierwahrscheinlichkeiten. Ein Säulendiagramm für die Auslastung der Knoten des Beispiels 4.1. aus [Bolch 82] könnte wie folgt aussehen:

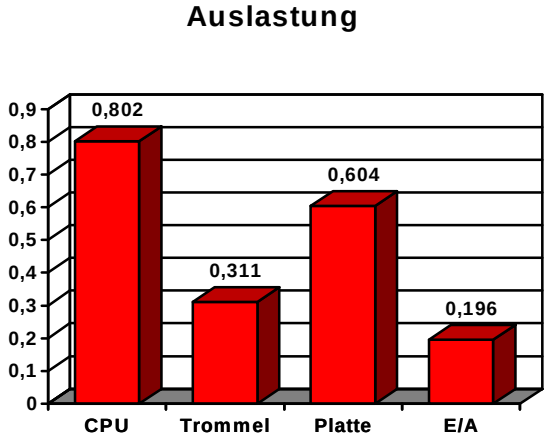

Abbildung 6-2: Darstellung der Knotenauslastung

Die grafische Darstellung ermöglicht einen raschen Überblick. Beispielsweise kann in obigen Diagramm sofort der Knoten "CPU" als Schwachstelle des Modells identifiziert werden.

6.4. Zusammenfassung

Mit der Fertigstellung der Simulationskomponente hat das Programmsystem BNETD seinen angestrebten Funktionsumfang erreicht. Ansätze für Erweiterungen und Verbesserungen wurden im Text verschiedentlich erwähnt. Nichtsdestoweniger hat das System in der gegenwärtigen Version einen Stand erreicht, der BNETD zu einem praktikablen Werkzeug macht, welches in seiner Gesamtfunktionalität auf keinen Fall hinter seinen Vorgängern zurücksteht. Ein Einsatz von BNETD als Praktikumssystem ist empfehlenswert, auch im Hinblick auf die Verifizierung des angestrebten kommerziellen Niveaus des Softwareproduktes.
Als wesentliche Pluspunkte von BNETD sind zu sehen:

- standardisierte pseudografische Nutzeroberfläche, die eine intuitive und komfortable Bedienung des Programms ermöglichen sollte
- leistungsfähige Grafikkomponente zur einfachen Struktureingabe, Modelldarstellung und Ergebnisrepräsentation
- breites Einsatzspektrum durch analytische und simulative Lösungsmethoden
- hohes Niveau von Korrektheit und Robustheit, welches durch kontinuierliche Entwicklung und Testung des Systems in einem kleinen Programmierteam über einen Zeitraum von fast 1,5 Jahren erreicht wurde

Demgegenüber ist als Problem lediglich die Wahrung der Kontinuität bei der Weiterentwicklung des Systems zu sehen, da eine Personalwiederverwendbarkeit (siehe Kapitel 3) meist nicht gegeben ist und eine Arbeit an dem Programmsystem durch den gleichen Personenkreis über einen längeren Zeitraum, wie es im Rahmen dieses Projektes der Fall war, schwer möglich sein wird.