Universität Stuttgart

Institut für Nachrichtenvermittlung und Datenverarbeitung
Prof. Dr.-Ing. habil. Dr. h. c. mult. P. J. Kühn

69. Bericht über verkehrstheoretische Arbeiten

Entwicklung eines objektorientierten Werkzeugs für verschiedene Verfahren der parallelen ereignisgesteuerten Simulation

von

Thomas Necker

1998

D 93

Druck: E. Kurz & Co., Druckerei + Reprografie GmbH., Stuttgart
ISBN 3-922403-79-4

University of Stuttgart

Institute of Communication Networks and Computer Engineering
Prof. Dr.-Ing. habil. Dr. h. c. mult. P. J. Kühn

69th Report on Studies in Congestion Theory

Development of an Object Oriented Tool for Various Algorithms for Parallel Discrete Event Simulation

by
Thomas Necker
1998

Development of an Object Oriented Tool for Various Algorithms for Parallel Discrete Event Simulation

Summary

The performance evaluation of technical systems is a necessary part of their development, improvement and the estimation of the resulting consequences. Due to several reasons experiments or measurements are sometimes not possible; building a prototype or using an existing system for conducting experiments is usually time consuming and expensive. Moreover, in many cases, the behaviour of the system in critical situations is of interest which often presents an unacceptable risk for the system itself or its environment.

Therefore, the performance evaluation of a technical system should be carried out without the use of the real implementation. To achieve this, the real or planned system, or subsets thereof, are represented by a model which contains all the relevant aspects for the given problem. Finding out the core of a system in respect to a given problem is called abstraction. Abstraction makes the handling of the model feasible by reducing its complexity. Idealizing certain attributes of the original system, either to further reduce complexity or because of a lack of knowledge in relation to them, is another step in the model building process.

Analytical methods are one way of analysing models. However, this approach often delivers only approximations and requires the use of limiting assumptions. They are additionally not suitable for increased complexity.

Another possibility is computer simulation. The model is transferred into a suitable "simulation model" which is implemented and executed on a computer, possibly using a simulation tool. Simulation has proven to be a flexible method in the evaluation of technical systems.

In the field of telecommunications, simulation is used, for example, in the analysis of transmission and switching equipment (which is often modeled as queueing systems). As systems become more complex, problems arise in handling and implementing the models. For fast and simple simulation of complex systems powerful tools are necessary to support the user. Growing complexity in conjunction with higher transmission rates and new protocols leads to increasing computation times of the event driven simulators commonly used in this area. One example of this is ATM (Asynchronous Transfer Mode) with high transmission rates and short cells where one whishes to study very small cell loss probabilities. Another example is the signaling system Nr. 7, the real implementations of which potentially consist of hundreds of signaling points each node of which contains considerable complexity. In spite of computers becoming more powerful, the gap between the available and necessary computation power is increasing. Therefore, other ways of accelerating the simulations have to be found.

As a result of the extensibility possibilities inherent within parallel and distributed computing systems, they offer almost unlimited computation and memory resources. Therefore, they make the simulation of very complex models feasible and can accelerate the simulation of a given problem. Parallel discrete event simulation (PDES) can be realized in several ways. In this thesis, the distribution and parallel execution of the simulation model is used. Using PDES in this way several problems arise concerning the synchronization of the parallel processes. To solve these problems various algorithms have been developed each having its specific strengths and weaknesses.

In spite of the fact that parallel discrete event simulation has been around for almost two decades, it is still predominantly applied by a minority of experts and not the majority of simulation users. What are the reasons for this? On the one hand, there is no simulation algorithm currently available which achieves good performance in all applications. Rather the performance is very much dependent on the characteristics of the simulation model. This leads to the situation where the user needs to be familiar not only with the particular simulation model but also with the various algorithms. On the other hand, building a parallel simulation is much more difficult than building a sequential one. In contrast to the large number of simulation tools for sequential simulation hardly any exist for parallel simulation. The few existing parallel simulation tools are normally either dedicated to a limited field of application or only support one class of PDES algorithms.

Consequently, the application of parallel discrete event simulation is much more difficult than that of sequential simulation. Moreover, it is very badly supported by tools. Besides the research for commonly usable PDES algorithms, the development of efficient and easy to use simulation tools is therefore necessary. This thesis aims to address this problem.

An all-purpose tool for parallel discrete event simulation of queueing systems is developed especially addressing the problems of simple and flexible usage. One major objective is the separation of the simulation model and the PDES algorithms. The PDES algorithms are encapsulated and exchangeable in a simple way. Hence, it is possible to implement a simulation model without knowing about the algorithm to be used; rather the best suitable one can be chosen later. In particular, it is possible to use different methods from a toolkit and to evaluate their performance for a given simulation model. Furthermore, a sequential tool has been used as the basis of the parallel tool thus enabling an easy transition from sequential to parallel simulation. Therefore, the user of sequential simulation can more easily familiarize himself with parallel simulation and existing sequential simulation models can be easily parallelized. Additionally, the simulation models can be implemented sequentially and may be parallelized on demand (an important point for the acceptance of parallel simulation).

Besides the user friendly aspects, the internal architecture of the tool is designed for easy extensibility. A framework of basic functionality is provided that is independent and transparent in respect to the different PDES algorithms. The algorithms themselves are imple-

mented in such a way so that they fit into this framework. Since the number of algorithms is large this flexible approach is very advantageous when implementing new methods or their variants. Easy portability of the tool is another feature that is necessary due to the dynamic development in the field of parallel and distributed computing systems.

This thesis is divided into eight chapters. After a short introduction in chapter 1, chapter 2 describes the problems of parallel discrete event simulation when distributing the simulation model onto so called "Logical Processes" together with the corresponding solutions. The problem of synchronization of the Logical Processes is addressed and the basic categories of conservative and optimistic methods are described and discussed. Specific mechanisms for optimistic simulation, for example state saving or calculation of the global virtual time, are also addressed.

The first part of chapter 3 is dedicated to parallel computation. The architectures and programming models of MIMD computers (Multiple Instruction Stream – Multiple Data Stream) are described; especially in regards to message passing systems which are important for this thesis. The second part of the chapter deals with object oriented software design and the concept of design patterns. A selection of design patterns important for this work is presented. The chapter closes with sections covering the programming language C++ and object oriented approaches for building parallel applications.

The overall architecture of the developed tool is shown in chapter 4. The hierarchy of parallel executable units is presented which allows a flexible adaptation to the potential parallelism of the simulation model. The structure and the functionality of these units are described as well as the realization of a transparent and encapsulated message exchange system. A flexible concept for optimistic parallel simulation is addressed which allows the easy use and exchange of different mechanisms for state saving. In this context new methods for state saving are presented.

Chapter 5 describes the structure and the key concepts of the tool in detail, especially in regards to the use of object oriented techniques for managing of the complexity, the encapsulation of PDES mechanisms and the building of a framework into which these mechanisms fit. It is shown how the separation of the simulation model from these mechanisms and their easy exchangeability and extensibility is achieved.

The realization of the simulation tool as an object oriented library is shown in chapter 6. This chapter also contains guidelines for the use of the tool and the transition from sequential to parallel simulation.

Chapter 7 contains exemplary applications and performance investigations to show the successful applicability of the developed concepts. Firstly, different topologies of artificial models built from generic node and link elements are tested. These generic elements were designed

for examining the most important performance parameters of parallel discrete event simulation. Secondly, the parallelization of a simulation model for the Signaling System Nr. 7 is discussed. It is shown that such a parallelization can easily be achieved thanks to the support of the developed tool and that this can be advantageous for the simulation of large models.

Chapter 8 concludes this thesis with a summary and an outlook on further work.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungen

ACM	Association for Computing Machinery (im SS7-Kontext auch: Address Complete Message)
AEÜ	Archiv für Elektronik und Übertragungstechnik
ALU	Arithmetic Logical Unit
ANM	Answer Message
ANSI	American National Standards Institute
ASS	Annual Simulation Symposium
ATM	Asynchronous Transfer Mode
CC	Call Control
CCITT	Comité Consultatif International Téléphonique et Télégraphique
CCO	Component Coordinator
COMA	Cache-Only Memory Access
COOP	Concurrent Object-Oriented Programming
CORBA	Common Object Request Broker Architecture
CPCI	Call Processing Control Incoming
CPCO	Call Processing Control Outgoing
CPU	Central Processing Unit
DCE	Distributed Computing Environment
DHA	Dialog Handling
DII	Dynamic Invocation Interface
E/A	Ein-/Ausgabe
FCFS	First Come First Serve
FIFO	First In First Out
GVT	Global Virtual Time
HMDC	Message Discrimination
HMDT	Message Distribution
HMRT	Message Routing
IAM	Initial Address Message
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronics Engineers
IFIP	International Federation for Information Processing
IND	Institut für Nachrichtenvermittlung und Datenverarbeitung
I/O	Input/Output
IR	Interface Repository
ISDN	Integrated Services Digital Network
ISM	Invocation State Machine
ISO	International Organization for Standardization

ISUP	ISDN User Part
ITC	International Teletraffic Congress
ITU	International Telecommunication Union
ITU-T	Telecommunication Sector of ITU
LAN	Local Area Network
LN	Logical Node
LP	Logical Process
LVT	Local Virtual Time
MASCOTS	International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems
MB	2^{20} Byte
MDSC	Message Distribution Control
MIMD	Multiple Instruction Stream – Multiple Data Stream
MISD	Multiple Instruction Stream – Single Data Stream
MIT	Massachusetts Institute of Technology
MPI	Message Passing Interface
MPMD	Multiple Program – Multiple Data Stream
MPSS	Message-Passing-Subsystem
MSDC	Message Sending Control
MTP	Message Transfer Part
NORMA	No-Remote Memory Access
NUMA	Nonuniform Memory Access
OA	Object Adapter
OMA	Object Management Architecture
OMG	Object Management Group
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
OOPSLA	Annual Conference on Object-Oriented Programming Systems, Languages and Applications
ORB	Object Request Broker
ORSA	Operations Research Society of America
OSF	Open Software Foundation
OSI	Open Systems Interconnection
PADS	Workshop on Parallel and Distributed Simulation
PDES	Parallel Discrete Event Simulation
PE	Prozessorelement
PVM	Parallel Virtual Machine
REL	Release Message
RLC	Release Complete Message

RTTI	Run-Time Type Information
SCCP	Signalling Connection Control Part
SCLR	SCCP Connectionless Control Reception
SCLT	SCCP Connectionless Control Transmission
SCOR	SCCP Connection-Oriented Control Reception
SCOT	SCCP Connection-Oriented Control Transmission
SCRR	SCCP Routing Control Reception
SCRT	SCCP Routing Control Transmission
SCS	Society for Computer Simulation
SIGPLAN	Special Interest Group on Programming Languages
SIMD	Single Instruction Stream – Multiple Data Stream
SISD	Single Instruction Stream – Single Data Stream
SP	Signalling Point
SPMD	Single Program – Multiple Data Stream
SS7	Signalisiersystem Nr. 7
STP	Signalling Transfer Point
SZE	Simulationszeiteinheit
TCAP	Transaction Capabilities Application Part
TSL	Transaction Sublayer
TUP	Telephone User Part
TWOS	Time Warp Operating System
UMA	Uniform Memory Access

Formelzeichen

A	Alter eines Ereignisses
C	Maß für Größe und Komplexität einer Anwendung
E	Ereignis
G	Generator
h_I	mittlere Bediendauer des „Infinite Server“ (normiert auf 1 SZE)
h_S	mittlere Bediendauer der Bedieneinheit S (normiert auf 1 SZE)
id'	eindeutiger Bezeichner eines Ereignisses
λ_G	mittlere Erzeugungsrate von Nachrichten im Generator G (normiert auf $\frac{1}{\text{SZE}}$)
L	Lookahead
$n_{E_{SI}}$	Maximale Anzahl bearbeiteter Ereignisse pro Zeitschlitz
$n_{E_{SS}}$	Anzahl Ereignisse zwischen zwei Zustandssicherungen
n_{In}	Anzahl Eingänge eines generischen Knotenelements
n_{Nd}	Anzahl generischer Knotenelemente
n_{M_S}	mittlere Anzahl von Durchläufen einer Nachricht durch die Bedieneinheit eines generischen Knotenelements
n_{Out}	Anzahl Ausgänge eines generischen Knotenelements
$n_{S_{GVT}}$	Anzahl berabeiteter Ereignisse zwischen zwei GVT-Anforderungen
p	Anzahl Prozessoren
p_r	Rückkoppelwahrscheinlichkeit im generischen Knotenelement
S	Bedieneinheit
Sc	Scaleup
Sp	Speedup
t	Echtzeit
t_{elp}	Ausführungszeit
T	Simulationszeit (normiert auf 1 SZE)
T_E	Ereigniszeitpunkt (normiert auf 1 SZE)
T_{GVT}	globale virtuelle Zeit (normiert auf 1 SZE)
T_{In}	Eingangskanalzeit (normiert auf 1 SZE)
T_{LVT}	lokale virtuelle Zeit (normiert auf 1 SZE)
T_M	Nachrichtenzeitpunkt (normiert auf 1 SZE)
T_{Out}	Ausgangskanalzeit (normiert auf 1 SZE)
T_{PTL}	Teiltestlänge (normiert auf 1 SZE)
T_S	Nachzüglerzeitpunkt (normiert auf 1 SZE)
T_{SS}	Zeitpunkt einer Zustandssicherung (normiert auf 1 SZE)
$T_{\Delta S}$	Zeitpunkt einer Zustandsänderung (normiert auf 1 SZE)
τ	erweiterter Zeitstempel

Kapitel 1

Einleitung

1.1 Parallele Simulation komplexer Systeme

Die Evaluierung technischer Systeme ist ein unverzichtbarer Bestandteil ihrer Entwicklung, Verbesserung und Folgenabschätzung. Die Durchführung von Experimenten und Messungen ist dabei aus verschiedenen Gründen nicht immer möglich: Zum einen ist der Aufbau eines Prototypen oder die Entbindung eines existierenden Systems von seinen eigentlichen Aufgaben zur Durchführung von Experimenten meist zeitaufwendig und teuer, zum anderen soll häufig das Verhalten in kritischen Situationen ermittelt werden – ein für das System oder seine Umwelt oftmals inakzeptables oder gar gefährliches Unterfangen.

Es besteht also Bedarf an der Untersuchung eines technischen Systems, ohne einer realen Implementierung habhaft zu sein. Um dies zu ermöglichen, wird das reale (oder gedachte) System oder interessierende Teile davon auf ein Modell abgebildet, welches die für die gegebene Fragestellung relevanten Aspekte berücksichtigt. Das Herausarbeiten des wesentlichen Kerns eines Systems bezüglich einer bestimmten Problemstellung wird als Abstraktion bezeichnet und macht die Handhabung des Modells durch Reduzierung seiner Komplexität in der Regel überhaupt erst möglich. Ein weiterer Vorgang bei der Modellbildung ist die Idealisierung bestimmter Eigenschaften des Originals, entweder um die Komplexität weiter zu reduzieren oder weil genaue Kenntnisse dieser Eigenschaften nicht vorliegen.

Das entwickelte Modell kann dann zum einen mit Hilfe analytischer Methoden untersucht werden. Diese bieten aber oft nur approximative Lösungen unter Zuhilfenahme einschränkender Annahmen oder versagen ab einer gewissen Komplexität vollständig.

Eine andere Möglichkeit ist die Simulation auf einem Rechner. Das Modell wird dazu in eine für diesen geeignete Form gebracht und dieses Simulationsmodell dann u. U. unter Zuhilfenahme eines Simulationswerkzeugs auf dem Rechner implementiert und ausgeführt. Die

Simulation hat sich in den verschiedensten Bereichen als ein äußerst flexibles Mittel bei der Untersuchung technischer Abläufe und Systeme etabliert.

Im Bereich der Telekommunikationstechnik wird Simulation z. B. zur Untersuchung von Übertragungs- und Vermittlungseinrichtungen verwandt, wobei die Systeme häufig als Warteschlangennetze modelliert werden. Es ist dabei ein zunehmender Trend zu immer höherer Komplexität festzustellen, was zum einen die Problematik der Handhabbarkeit dieser Modelle und der Schwierigkeit ihrer Implementierung aufwirft. Leistungsfähige Simulationswerkzeuge sind nötig, um dem Anwender eine schnelle, sichere und möglichst einfache Simulation auch komplexer Systeme zu ermöglichen. Zum anderen führt die zunehmende Komplexität zusammen mit höheren Übertragungsraten und neuen Übertragungsverfahren bei der in diesem Bereich meist verwandten zeitdiskreten ereignisgesteuerten Simulation zu immer längeren Rechenzeiten. Ein Beispiel hierfür ist *ATM* (*Asynchronous Transfer Mode*, siehe z. B. [Händel et al.,1994]), mit hohen Übertragungsraten, kurzen Zellen und sehr kleinen zu untersuchenden Zellverlustwahrscheinlichkeiten. Ein weiteres Beispiel ist das Signalisiersystem Nr. 7 [ITU-T,1993], dessen reale Implementierungen je nach Abgrenzung aus hunderten Signalisierpunkten mit jeweils erheblicher Komplexität bestehen können. Zwar werden auch die eingesetzten Rechner durch technologische Fortschritte immer schneller, doch es sind einerseits deren Leistungsgrenzen absehbar und andererseits bereits heute eine sich zunehmend öffnende Schere zwischen benötigter und verfügbarer Rechenleistung erkennbar. Aus diesen Gründen muß auch nach anderen Wegen zur Beschleunigung von Simulationen gesucht werden.

Durch die Verwendung paralleler und verteilter Rechnersysteme stehen im Prinzip nahezu beliebig große Rechen- und Speicherressourcen zur Verfügung. Dadurch eröffnet sich die Möglichkeit einer beschleunigten Simulation eines gegebenen Problems bzw. überhaupt erst die Möglichkeit der Simulation sehr komplexer Modelle. Für die Realisierung einer parallelen ereignisgesteuerten Simulation (*Parallel Discrete Event Simulation, PDES*) gibt es verschiedene Möglichkeiten, von denen in dieser Arbeit die Aufteilung und parallele Bearbeitung des Simulationsmodells betrachtet wird. Dabei ergeben sich einige Probleme im Zusammenhang mit der Synchronisation der parallelen Simulation. Im Lauf der Jahre wurden für deren Lösung verschiedene Algorithmen mit spezifischen Stärken und Schwächen entwickelt (gute Übersichten bieten z. B. [Fujimoto,1990] oder [Ayani,1993]).

Obwohl seit nunmehr fast zwei Jahrzehnten weltweit an dem Problem der parallelen ereignisgesteuerten Simulation gearbeitet wird und auch durchaus Erfolge vorzuweisen sind, führt sie nach wie vor ein Nischendasein und wird nur von wenigen Experten angewandt, nicht aber von der großen Masse der Simulationsanwender [Fujimoto,1993]. Woran liegt dies? Zum einen gibt es bisher keinen PDES-Algorithmus, der in allen Anwendungsfällen gute Leistungen erbringt. Diese sind vielmehr stark abhängig von den Eigenschaften des Simulationsmodells, so daß der Anwender nicht nur mit diesen vertraut sein muß, sondern

auch mit denen der verschiedenen Algorithmen. Zum anderen ist die Implementierung einer parallelen Simulation deutlich schwieriger als die einer sequentiellen. Im Gegensatz zu der inzwischen reichhaltigen Auswahl an Simulationswerkzeugen für sequentielle Simulation, existieren solche für parallele Simulation kaum. Bei den wenigen existierenden handelt es sich dann oft um spezielle Werkzeuge für ein sehr eng eingegrenztes Anwendungsfeld, oder es wird nur eine kleine Klasse von PDES-Algorithmen zur Verfügung gestellt.

1.2 Ziele der Arbeit

Die Anwendung paralleler ereignisgesteuerter Simulation ist also auf der einen Seite deutlich schwieriger als die der sequentiellen, wird auf der anderen Seite aber auch noch wesentlich schlechter durch Werkzeuge unterstützt. Neben der weiteren Forschung nach allgemein anwendbaren PDES-Algorithmen muß deshalb vor allem die Entwicklung leistungsfähiger und einfach anwendbarer Werkzeuge im Vordergrund stehen.

Genau diese Problematik wird in der vorliegenden Arbeit aufgegriffen. Es soll ein universelles Werkzeug für die parallele Simulation von Warteschlangennetzen entwickelt werden, wobei das Hauptaugenmerk auf eine einfache und flexible Anwendung gerichtet ist. Als erstes wichtiges Merkmal soll eine weitgehende Trennung von Simulationsmodell und eingesetzten PDES-Algorithmen realisiert werden. Zum zweiten sollen letztere gekapselt und einfach gegeneinander austauschbar sein. Dadurch ist es möglich, ein Modell ohne Rücksicht auf den einzusetzenden Algorithmus zu implementieren und den am besten geeigneten erst später auszuwählen. Vor allem können nach dem Baukastenprinzip verschiedene Verfahren eingesetzt und ihre Leistungsfähigkeit für das gegebene Simulationsmodell getestet werden. Drittens soll das parallele Werkzeug an ein existierendes sequentielles angelehnt werden, wodurch der Übergang von sequentieller zu paralleler Simulation erleichtert wird. Ein daraus resultierender Vorteil ist die erheblich erleichterte Einarbeitung für den Anwender sequentieller Simulation, ein anderer die einfache Parallelisierung bereits existierender sequentieller Simulationsanwendungen. Simulationen sollen also weiterhin konventionell sequentiell entwickelt werden und dann bei Bedarf parallelisiert werden können – ein wichtiger Punkt für die breitere Akzeptanz der parallelen Simulation.

Neben dem gerade erläuterten Aspekt der einfachen Anwendbarkeit der parallelen Simulation soll außerdem bei der internen Architektur des Werkzeugs auf eine einfache Erweiterbarkeit geachtet werden. Es ist deswegen ein Grundgerüst an Basisfunktionalität bereitzustellen, das bezüglich spezieller PDES-Algorithmen unabhängig und transparent gehalten ist. Die eigentlichen PDES-Verfahren werden auf diesem Gerüst aufbauend bzw. es ausfüllend realisiert. Aufgrund deren großer Anzahl, ist die dadurch gewonnene Flexibilität bei der Implementierung neuer Verfahren oder Varianten davon äußerst vorteilhaft. Da auf dem Gebiet der

parallelen und verteilten Rechenplattformen eine große Dynamik herrscht, muß das Werkzeug außerdem leicht portierbar sein.

1.3 Übersicht über die Arbeit

Nach einer kurzen Einordnung der Arbeit in das allgemeine Gebiet der Simulation gibt Kapitel 2 eine Übersicht über die Problemstellungen der parallelen ereignisgesteuerten Simulation bei Aufteilung des Simulationsmodells in Logische Prozesse und die zu ihrer Lösung verwandten Algorithmen. Es wird das Problem der Synchronisation der Logischen Prozesse besprochen und die grundlegenden Kategorien der konservativen und optimistischen Lösungsansätze erläutert und diskutiert. Es wird dabei auch auf spezielle Verfahren der Zustandssicherung und GVT-Berechnung (*Global Virtual Time*) für optimistische Simulation eingegangen.

Kapitel 3 widmet sich im ersten Teil der Parallelverarbeitung. Dazu gehört nach einer kurzen allgemeinen Einführung die Beschreibung von Architekturen und Programmiermodellen von MIMD-Rechnern (*Multiple Instruction Stream – Multiple Data Stream*) unter besonderer Berücksichtigung der für diese Arbeit wichtigen Message-Passing-Systeme. Im zweiten Teil des Kapitels wird auf die Grundlagen des objektorientierten Softwareentwurfs und das Konzept der Softwareentwurfsmuster eingegangen. Eine Auswahl solcher Entwurfsmuster wird vorgestellt, soweit sie für die weiteren Betrachtungen von Bedeutung sind. Nach einem Einschub über die Programmiersprache *C++* werden schließlich noch einige objektorientierte Ansätze für die parallele und verteilte Softwareentwicklung aufgeführt.

Die Gesamtarchitektur des entwickelten Werkzeugs wird in Kapitel 4 ausführlich erläutert. Dabei wird die Hierarchie der parallel ausführbaren Einheiten vorgestellt, die eine flexible Anpassung an die dem Simulationsmodell innewohnende Parallelität erlaubt. Der Aufbau und die Funktion dieser Einheiten wird ebenso beschrieben wie die Realisierung eines transparenten und gekapselten Nachrichtenaustauschs. Es wird auf das implementierte flexible Konzept für die Zustandssicherung bei optimistischer paralleler Simulation eingegangen, das den Einsatz und einfachen Austausch verschiedener hierfür bekannter Verfahren ermöglicht. In diesem Zusammenhang werden auch neu entwickelte Verfahren für die Zustandssicherung vorgestellt.

Kapitel 5 beschreibt den Aufbau und die Schlüsselkonzepte des Werkzeugs im Detail. Insbesondere wird auf die Anwendung von Techniken des objektorientierten Softwareentwurfs zur Beherrschung der Komplexität, zur Kapselung der PDES-Mechanismen und für die Erstellung eines Gerüsts an Basisfunktionalität für diese eingegangen. Es wird gezeigt, daß sich dadurch die Forderung hinsichtlich Trennung des Simulationsmodells von diesen Mechanismen sowie deren einfacher Austausch- und Erweiterbarkeit elegant erreichen lassen.

Die Realisierung des Simulationswerkzeugs als objektorientierte Bibliothek wird in Kapitel 6 behandelt. Vor allem werden dort auch pragmatische Richtlinien für die Anwendung des Werkzeugs und den Übergang von sequentieller zu paralleler Simulation gegeben.

Kapitel 7 beinhaltet dann beispielhafte Anwendungen und Leistungsuntersuchungen der Bibliothek, um die erfolgreiche Anwendbarkeit der entwickelten Konzepte zu zeigen. Dazu werden zuerst verschiedene Topologien künstlicher Modelle mit eigens entworfenen generischen Knoten- und Verbindungselementen getestet, mit Hilfe derer sich die wichtigsten PDES-Leistungsparameter untersuchen lassen. Danach werden Erfahrungen bei der Parallelisierung einer Simulation des Signalisiersystems Nr. 7 berichtet. Es wird gezeigt, daß eine solche Parallelisierung dank der Unterstützung durch die implementierte Bibliothek einfach durchführbar ist und sich bei der Simulation großer Modelle auszahlen kann.

Die Arbeit wird in Kapitel 8 mit einer Zusammenfassung und einem Ausblick auf mögliche weitere Entwicklungen abgeschlossen.

Kapitel 2

Parallele ereignisgesteuerte Simulation

2.1 Verfahren der Simulation

Laut [Duden,1993] versteht man in der Informatik unter Simulation „die Nachbildung von Vorgängen auf einer Rechenanlage auf der Basis von Modellen... Sie wird meist zur Untersuchung von Abläufen eingesetzt, die man in der Wirklichkeit aus Zeit-, Kosten-, Gefahren- oder anderen Gründen nicht durchführen kann".

Zum Zwecke der Simulation wird zuerst das zu untersuchende System auf ein abstrahierendes Simulationsmodell abgebildet, das die für die gewünschten Untersuchungen wichtigen Aspekte enthält. Die Wahl geeigneter Abstraktionen bei der Modellbildung ist von entscheidender Bedeutung, da sie eine Abwägung zwischen der zur Erlangung aussagekräftiger Ergebnisse erforderlichen Detailtreue und der handhabbaren Komplexität erfordert. Das entwickelte Simulationsmodell wird anschließend auf einem Rechner implementiert und simuliert. In Abhängigkeit von der Art des Voranschreitens der Zeit bei der Simulation läßt sich die in Bild 2.1 dargestellte Klassifizierung verschiedener Verfahren angeben.

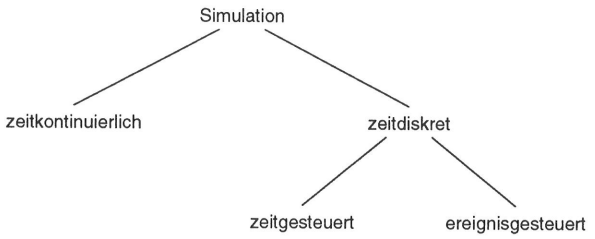

Bild 2.1: Klassifizierung verschiedener Simulationsarten nach zeitlichem Ablauf

Bei der *zeitkontinuierlichen* Simulation lassen sich die Zustandsänderungen des Modells durch kontinuierliche Funktionen beschreiben. Auf heute üblichen Digitalrechnern lassen sich solche Simulationen aufgrund der prinzipbedingten diskreten Natur solcher Rechner nur angenähert kontinuierlich, *quasikontinuierlich*, bearbeiten. Diese Art der Simulation wird beispielsweise für die Untersuchung von Strömungsvorgängen bei der Flugzeugkonstruktion oder Klimaforschung angewandt.

Die *zeitdiskrete* Simulation, bei der nur Zustandsänderungen an diskreten Punkten der Zeitachse betrachtet werden, läßt sich weiter unterteilen in *zeitgesteuerte* und *ereignisgesteuerte* Simulation. Bei ersterer schreitet die Zeit im Simulationsmodell immer in festen Schritten ΔT voran. Alle Vorgänge innerhalb dieser Zeitspanne werden erst an deren Ende simuliert. Diese Art der Simulation eignet sich z. B. im Zusammenhang mit Kommunikationsnetzen für Medienzugriffsprotokolle mit fester Zeitschlitz-Struktur. Bei ereignisgesteuerter Simulation schließlich werden alle Zeitpunkte betrachtet, an denen sich der Zustand des Simulationsmodells ändert. Im folgenden wird ausschließlich diese Art der Simulation behandelt.

2.2 Zeitdiskrete ereignisgesteuerte Simulation

Technische Systeme, z. B. im Bereich der Telekommunikation, der Rechnersysteme oder der Fabrikation, können häufig als Warteschlangennetze modelliert werden, die aus einzelnen Grundelementen (Strukturkomponenten) aufgebaut sind und von Aufträgen (Nachrichten) durchlaufen werden. Solche Komponenten sind z. B. Verkehrsquellen (Generatoren), Bedieneinheiten, Warteschlangen oder Koppelnetzwerke (siehe z. B. [Kühn,1995]).

Für die Simulation solcher Warteschlangennetze wird die ereignisgesteuerte Simulation benutzt, zu deren Erläuterung das in Bild 2.2 dargestellte einfache Warteschlangennetz als Beispiel diene. Es besteht aus einem Generator, der über eine Verteilungsfunktion gesteuert

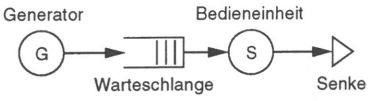

Bild 2.2: Einfaches Warteschlangennetz

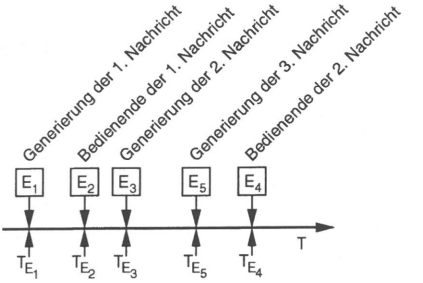

Bild 2.3: Ereigniskalender

in bestimmten Zeitabständen Nachrichten erzeugt, einer nachfolgenden Warteschlange, in der diese Nachrichten bis zu ihrer Bearbeitung eingereiht werden, sowie einer Bedieneinheit, die nach einer ebenfalls durch eine Verteilungsfunktion bestimmten Zeit die Nachrichten an eine Senke weiterleitet, in der sie vernichtet werden.

Bei der sequentiellen Simulation erfolgt die Steuerung des Simulationsablaufs über einen zentralen *Ereigniskalender*, in den jede Modellkomponente zum Simulationszeitpunkt, zu dem sich ihr Zustand ändern wird, ein *Ereignis* einträgt. Die Simulationssteuerung entnimmt jeweils das Ereignis mit der kleinsten Zeit und ruft eine zugehörige Bearbeitungsroutine auf. Während der Bearbeitung eines Ereignisses können Folgeereignisse erzeugt werden, welche wiederum in den Kalender eingetragen werden. Die Simulationszeit schreitet also von Ereignis zu Ereignis voran, die dazwischenliegenden Spannen werden übersprungen. Rechenzeit, d. h. Echtzeit (Realzeit), wird dagegen fast ausschließlich während der Bearbeitung eines Ereignisses verbraucht, d. h. während die Simulationszeit stillsteht.

Bild 2.3 zeigt den Ereigniskalender für das Beispiel aus Bild 2.2. Zu Beginn der Simulation (z. B. innerhalb einer Initialisierungsroutine) hat der Generator den Zeitpunkt T_{E_1} eingetragen, zu dem er seine erste Nachricht erzeugen muß. Die Simulationssteuerung entnimmt dieses Ereignis dem Kalender und ruft eine entsprechende Bearbeitungsroutine auf. In dieser wird vom Generator eine Nachricht erzeugt und durch die (leere) Warteschlange an die (freie) Bedieneinheit weitergereicht. Diese bestimmt die Bediendauer für die Nachricht und trägt ein Ereignis zum Zeitpunkt T_{E_2} für das Bedienende in den Kalender ein. Vor Rückgabe der Kontrolle an die Simulationssteuerung wird außerdem noch T_{E_3} als nächster Zeitpunkt für die Erzeugung einer Nachricht durch den Generator in den Kalender eingetragen.

Da Ereignisse Folgeereignisse niemals mit kleineren Zeiten (d. h. niemals in der simulierten Vergangenheit) erzeugen können, werden sie während eines Simulationslaufs immer in monoton steigender Zeitenfolge bearbeitet. Damit garantiert der zentrale Ereigniskalender die Einhaltung der Kausalität innerhalb des Simulationsmodells.

2.3 Parallelisierung ereignisgesteuerter Simulationen

2.3.1 Ziele

Die Problematik sehr langer Rechenzeiten bei komplexen Systemen mit detaillierter Modellierung wurde bereits in Kapitel 1 angesprochen. Müssen sehr viele Ereignisse bearbeitet werden, wie das z. B. bei der Simulation von *ATM*-Systemen auf Zellebene der Fall ist, kann der Rechenzeitbedarf enorm ansteigen, da bei ereignisgesteuerter Simulation gerade die Bearbeitung von Ereignissen Rechenzeit erfordert (siehe Unterkapitel 2.2).

Die benötigte Rechenkapazität kann durch verteilte Rechnersysteme im Prinzip zur Verfügung gestellt werden. Für deren Einsatz zur Lösung eines Problems gibt es i. allg. zwei Motivationen[1]: Ein gegebenes Problem soll in kürzerer Zeit oder in gegebener Zeit soll ein größeres Problem gelöst werden. Entsprechend lassen sich Kenngrößen definieren. Sei p die Anzahl eingesetzter Prozessoren, $t_{elp}(1)$ die Ausführungszeit der Anwendung auf einem Prozessor, $t_{elp}(p)$ die Ausführungszeit auf p Prozessoren und C ein Maß für die Größe und Komplexität einer Anwendung, dann ist der *Speedup*, $Sp(p)$, definiert als

$$Sp(p) = \frac{t_{elp}(1)}{t_{elp}(p)} \qquad \text{für} \quad C(1) = C(p)$$

und der *Scaleup*, $Sc(p)$, als

$$Sc(p) = \frac{C(p)}{C(1)} \qquad \text{für} \quad t_{elp}(1) = t_{elp}(p) \ .$$

Auch für die ereignisgesteuerte Simulation treffen diese beiden Punkte zu. Es besteht der Wunsch, längere Zeiträume oder größere Systeme mit einem hohen Detaillierungsgrad zu simulieren. In den folgenden Abschnitten sollen die für eine Parallelisierung bestehenden Möglichkeiten und Probleme beschrieben werden.

2.3.2 Möglichkeiten

Um eine ereignisgesteuerte Simulation auf mehrere Rechenknoten zu verteilen und parallel abzuarbeiten, gibt es verschiedene Möglichkeiten (siehe z. B. [Lehnert,1993]).

Eine erste Möglichkeit besteht in der Verteilung und parallelen Bearbeitung von unabhängigen Simulationsprogrammen (*Replicated Simulation*, siehe z. B. [Lin,1994]). Bei stochastischer Simulation ist es z. B. zur Bestimmung der statistischen Aussagesicherheit im allgemeinen notwendig, einen Simulationslauf in mehrere unabhängige Teiltests aufzuspalten. Ein Nachteil bei dieser Art der Parallelisierung ist, daß der erzielbare Speedup durch die Anzahl durchzuführender Teiltests begrenzt ist. Ebenfalls möglich ist die gleichzeitige unabhängige Simulation verschiedener Parametersätze. Auch hier wird der Speedup durch die Anzahl der Parametersätze begrenzt. Zudem kommt erschwerend hinzu, daß die Wahl der Parameter eines Simulationslaufs oftmals vom Ergebnis eines vorangegangenen Simulationslaufs abhängt.

Die Auslagerung einzelner Funktionsblöcke eines Simulationsprogramms auf eigene Rechenknoten stellt eine zweite Möglichkeit der Parallelisierung dar. So können z. B. die statistische Auswertung, die Zufallszahlenerzeugung oder die grafische Aufbereitung der Ergebnisse auf

[1] Sicherheitskritische Problemstellungen, bei denen mehrere Rechner die Ausfallsicherheit erhöhen oder sich gegenseitig überwachen, werden hier nicht betrachtet.

eigenen Prozessoren durchgeführt werden und somit den Hauptprozessor entlasten. Da es aber nur eine begrenzte Anzahl auslagerbarer Funktionen gibt, ist auch hier der maximal erzielbare Speedup begrenzt.

Die dritte und im Sinne des maximal erzielbaren Speedup vielversprechendste Möglichkeit zur Parallelisierung besteht in der Verteilung des Simulationsmodells selbst. Sie ist allerdings gleichzeitig die am schwierigsten beherrschbare. Im folgenden soll ausschließlich diese Art der Parallelisierung betrachtet werden.

2.4 Parallelisierung des Simulationsmodells

Bei der Parallelisierung des Simulationsmodells werden die einzelnen Simulationsmodellkomponenten auf sog. *Logische Prozesse* (*Logical Process*, *LP*) aufgeteilt. Bei dem in Bild 2.4 gezeigten einfachen Warteschlangennetz handelt es sich um ein „Central-Server"-Modell (siehe z. B. [Fujimoto,1988]), bei dem Nachrichten, die die zentrale Bedieneinheit verlassen, durch den nachfolgenden Verzweigungsknoten (Demultiplexer) entweder in den oberen oder den unteren Bearbeitungszweig weitergeleitet werden. Bei der Verteilung dieses Modells werden jeweils eine Warteschlange und Bedieneinheit, sowie der Multiplexer und der Verzweigungsknoten auf einen Logischen Prozeß aufgeteilt (grau unterlegt in Bild 2.4).

Um die Änderung der Simulationszeit in einem verteilten Simulationssystem zu beschreiben, wurde in [Jefferson,1985] der Begriff der *Virtuellen Zeit* eingeführt. Jeder Logische Prozeß i besitzt eine eigene *Lokale Virtuelle Zeit* (*Local Virtual Time*, *LVT*) T^i_{LVT}, einen eigenen lokalen Ereigniskalender und kommuniziert mit anderen LPs nur über mit Zeitstempeln

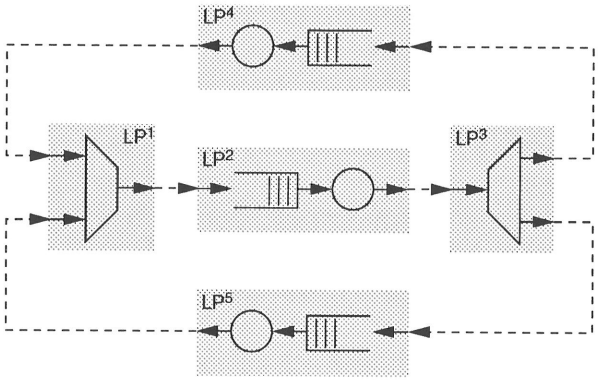

Bild 2.4: Verteiltes „Central-Server"-Modell

behaftete Nachrichten. Auf einem Rechenknoten kann jeweils entweder ein einzelner LP oder mehrere LPs können quasiparallel ausgeführt werden.

Definition 2.1 (Lokale Virtuelle Zeit)
Die *Lokale Virtuelle Zeit* (*Local Virtual Time, LVT*) des Logischen Prozesses i, T^i_{LVT}, ist gleich dem Zeitstempel des gerade bearbeiteten Ereignisses bzw. des zuletzt bearbeiteten, falls momentan kein Ereignis bearbeitet wird.

Definition 2.2 (Globale Virtuelle Zeit)
Die *Globale Virtuelle Zeit* (*Global Virtual Time, GVT*), T_{GVT}, ist gleich dem Minimum aller lokalen virtuellen Zeiten und der Zeitstempel aller im Transit befindlichen, d. h. gesandten und noch nicht empfangenen, Nachrichten im Simulationssystem.

T^i_{LVT} ist also ein Maß für den Fortschritt der Simulation im Logischen Prozeß i und T_{GVT} ein Maß für den Fortschritt der gesamten verteilten Simulation.

Durch die Verteilung auf die einzelnen Logischen Prozesse entfällt die Synchronisation durch den zentralen Ereigniskalender. Die Kausalität der Simulation bleibt gewahrt, falls alle LPs Ereignisse jeweils nur in monoton steigender Zeitstempelfolge bearbeiten (*Lokale Kausalitätsbedingung*, siehe [Fujimoto,1990]). Bild 2.5 enthält ein Beispiel, welches zeigt, daß dies nicht ohne weitere Maßnahmen gewährleistet ist. Dort sind die Ereigniskalender zweier nicht synchronisierter Logischer Prozesse dargestellt. Die jeweils grau unterlegten Ereignisse wurden schon bearbeitet und die lokalen Uhren stehen auf den Zeiten T^1_{LVT} bzw. T^2_{LVT}. Das Ereignis E^1_2 zum Zeitpunkt T^1_2 in LP^1 hat ein Folgeereignis E^2_S zum Zeitpunkt T^2_S in LP^2 generiert. Als dieses bei LP^2 eintrifft, hat dieser allerdings schon das Ereignis E^2_2 zum Zeitpunkt $T^2_2 > T^2_S$ abgearbeitet. Da durch eine Ereignisbearbeitung i. allg. der Systemzustand verändert wird, kann eine nachträgliche Bearbeitung des Ereignisses E^2_S eine Kausalitätsverletzung bedeu-

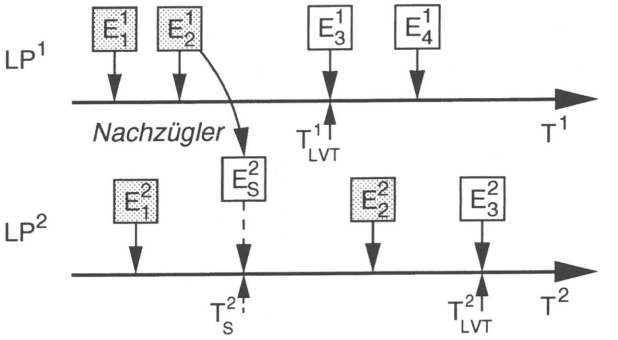

Bild 2.5: Synchronisationsproblem bei verteilten Kalendern

ten und muß deshalb ausgeschlossen werden. Ein Ereignis, welches solchermaßen „zu spät“ eintrifft wird *Nachzügler* oder *Straggler* genannt.

Die Synchronisation der verteilten Logischen Prozesse ist eines der Hauptprobleme der parallelen ereignisgesteuerten Simulation. Es gibt hierfür viele unterschiedliche Verfahren, die sich im wesentlichen in die beiden Kategorien *konservativ* und *optimistisch* einordnen lassen. Gute Übersichten sind z. B. in [Fujimoto,1990] und [Ayani,1993] zu finden. Im nächsten Abschnitt werden einige Verfahren näher vorgestellt und diskutiert.

2.5 Synchronisationsverfahren

2.5.1 Konservative Verfahren

2.5.1.1 Grundprinzip

Konservative Verfahren (auch *Pessimistische Verfahren* genannt) verhindern das Auftreten einer in Unterkapitel 2.4 beschriebenen Nachzügler-Nachricht grundsätzlich. Prinzipiell dürfen die einzelnen Logischen Prozesse dazu Ereignisse aus ihrem Ereigniskalender nur bearbeiten, wenn sie sich darauf verlassen können, keine Nachricht mit niedrigerem Zeitstempel mehr zu erhalten – solche Ereignisse sind „sicher“.

Bild 2.6 zeigt einen Logischen Prozeß mit n *Eingangskanälen* und m *Ausgangskanälen*. Jeder LP schickt über seine Ausgänge garantiert nur Nachrichten in monoton steigender Zeitstempelfolge, wodurch gewährleistet ist, daß jeder LP über seine Eingänge Nachrichten auch nur in monoton steigender Zeitstempelfolge empfängt.

Definition 2.3 (Kanalzeit)
Die *Kanalzeit* $T^j_{In,i}$ bzw. $T^j_{Out,i}$ ist gleich dem Zeitstempel der zuletzt über einen Kanal i im Logischen Prozeß j empfangenen bzw. gesandten Nachricht.

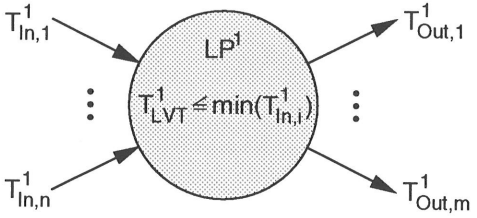

Bild 2.6: Blockieren eines Logischen Prozesses bei konservativer Synchronisation

Da über die Kanäle nur Nachrichten mit monoton steigenden Zeitstempeln übertragen werden, gibt die Kanalzeit gleichzeitig den kleinsten Zeitstempel an, den die nächste übertragene Nachricht haben kann. Jeder LP bearbeitet Ereignisse in seinem Ereigniskalender nur, falls deren Zeitstempel kleiner oder gleich dem Minimum aller Eingangskanalzeiten ist. Existiert kein Ereignis, das dieser Bedingung genügt, ist der LP blockiert bis durch eintreffende Nachrichten dieses Minimum erhöht wird. Dadurch wird sichergestellt, daß niemals Nachzügler-Ereignisse auftreten können und die lokale Kausalitätsbedingung erfüllt ist.

2.5.1.2 Behandlung von Verklemmungen

Bei konservativen Verfahren kann es vorkommen, daß sich die einzelnen Logischen Prozesse gegenseitig blockieren und es zu Verklemmungen kommt. Bild 2.7 zeigt einen solchen Fall anhand des verteilten „Central-Server"-Modells. Ohne Beschränkung der Allgemeinheit sei angenommen, es befinde sich nur eine Nachricht im System. Diese Nachricht ist zum Zeitpunkt[2] 7 am oberen Eingang von LP^1 angekommen. Da LP^1 über seinen unteren Eingang noch keine Nachricht empfangen hat, ist die entsprechende Eingangskanalzeit 0, und er kann deswegen die an seinem oberen Eingang anliegende Nachricht nicht simulieren. Die Eingangskanalzeit an seinem unteren Eingang wird sich aber auch nicht erhöhen, da sich keine weitere Nachricht im System befindet. Es ist somit eine Verklemmung entstanden.

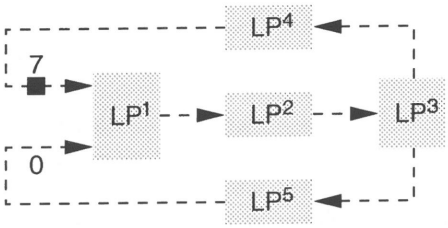

Bild 2.7: Auftreten einer Verklemmung bei konservativer Synchronisation

In [Chandy&Misra,1979], [Peacock et al.,1979] oder [Misra,1986] werden solche Verklemmungen durch das Versenden von *Garantienachrichten*, auch *NULL*-Nachrichten genannt, verhindert. Dabei teilt ein LP seinen Nachfolgern mit, welche Zeit die nächste von ihm versandte Nachricht mindestens haben wird. Wann ein LP solche NULL-Nachrichten sendet, kann bei verschiedenen Varianten des Verfahrens unterschiedlich sein: Er kann dies z. B. bei jeder Änderung seiner lokalen Zeit tun oder auch erst auf Verlangen eines nachfolgenden LPs [Misra,1986]. Im obigen Beispiel kann LP^3 eine NULL-Nachricht an LP^5 versenden, sobald er die Nachricht an LP^4 weitergereicht hat. Möge die lokale Zeit von LP^3 beim Weiterreichen z. B. 3 sein, dann garantiert er LP^5 durch die NULL-Nachricht, ihm keine Nachricht

[2] Alle Simulationszeiten sind normiert auf 1 *SZE* (Simulationszeiteinheit) und deswegen dimensionslos.

mit kleinerer Zeit als 3 zu senden. Daraufhin erhöht sich dessen Eingangskanalzeit und er kann seinerseits eine NULL-Nachricht senden. Die Zeit einer NULL-Nachricht berechnet sich dabei aus dem Minimum der Eingangskanalzeiten und dem *Lookahead* des LPs.

Definition 2.4 (Lookahead)
Kennt ein Logischer Prozeß alle seine Ereignisse bis zur Zeit T und kann er daraus sämtliche Folgeereignisse, die er mit Zeiten kleiner oder gleich $T + L$ generieren wird, vorhersagen, dann wird L sein *Lookahead* genannt [Fujimoto,1988].

Die Größe des Lookaheads hängt i. allg. von der Zeit und von der Art der Ereignisse ab. Hätte die Bedieneinheit von LP^5 aus Bild 2.4 bzw. 2.7 z. B. eine konstante Bedienzeit von 1, so wäre der Lookahead von LP^5 bei leerer Warteschlange ebenso 1, da eine zur Zeit T ankommende Nachricht zur Zeit $T + 1$ weitergesandt würde.

Wenn man zur Bedingung macht, daß die Summe der Lookahead-Werte in jeder Schleife des aus verbundenen LPs bestehenden Netzes größer als 0 ist, läßt sich beweisen, daß das gesamte System bei Verwendung von NULL-Nachrichten frei von Verklemmungen ist [Peacock et al.,1979]. So wird auch im Beispiel aus Bild 2.7, u. U. nach mehreren Umläufen, am unteren Eingang von LP^1 letztlich eine NULL-Nachricht ankommen, die einen Zeitstempel größer oder gleich 7 hat und somit die Blockierung von LP^1 auflöst.

Verschiedene Untersuchungen (z. B. [Fujimoto,1988], [Fujimoto,1989], [Reed et al.,1988]) haben gezeigt, daß die Leistungsfähigkeit dieses Ansatzes entscheidend von der Größe des Lookaheads abhängt. Bei kleinen Lookahead-Werten ist das Verhältnis der Anzahl von NULL-Nachrichten zur Anzahl realer Nachrichten sehr ungünstig. Da andererseits die Größe des Lookaheads eines Logischen Prozesses stark vom Simulationsmodell abhängt, reagiert das Verfahren sehr empfindlich auf Änderungen desselben. So kann z. B. die aus Anwendersicht kleine Änderung des Wechsels von konstanten auf negativ-exponentiell verteilte Bedienzeiten der Bedieneinheiten die Leistungsfähigkeit der parallelen Simulation aufgrund des verringerten Lookaheads stark mindern oder sogar Verklemmungen verursachen (Die Summe der Lookahead-Werte in Schleifen kann nun 0 ergeben).

Andere Verfahren verhindern Verklemmungen nicht, sondern erkennen und lösen diese durch zusätzliche Maßnahmen auf ([Chandy&Misra,1981], [Misra,1986]). Nachteilig bei diesen Verfahren ist der zusätzliche Aufwand für die Erkennung der Verklemmungen. Zudem kommt es vor, daß Systeme über längere Zeit größtenteils blockiert sind, bevor eine Verklemmung des gesamten Systems auftritt und aufgelöst werden kann.

2.5.1.3 Verbesserungen und weitere Verfahren

Wie schon erwähnt, ist die Größe des verfügbaren Lookaheads entscheidend für die Leistungsfähigkeit konservativer Verfahren. Sind die Bediendauern einer Bedieneinheit aber

z. B. negativ-exponentiell verteilt, so ist deren Lookahead bei Eintreffen einer neuen Nachricht in einer vorgeschalteten FIFO-Warteschlange (First In First Out) unabhängig von der aktuellen Warteschlangenlänge 0. In [Nicol,1988] wird für FIFO-Warteschlangen der Lookahead erhöht, indem die Bedienzeit einer Nachricht schon bei ihrem Eintreffen in der Warteschlange vorab ausgewürfelt wird. Dadurch liegen die Bedienzeiten aller wartenden Nachrichten fest, und es kann zu jedem Zeitpunkt eine Mindesttransferzeit einer eventuell neu ankommenden Nachricht bestimmt werden. Diese Mindesttransferzeit ist der (zeitlich veränderliche) Lookahead des Systems Warteschlange/Bedieneinheit. In [Lin&Lazowska,1990] und [Nicol,1992] wird ein ähnliches Prinzip für Warteschlangen mit Prioritäten vorgestellt.

[De Vries,1990] und [Nevison,1990] entwickeln spezielle Algorithmen für bestimmte Topologien des simulierten Netzes (einfach vorwärtsgerichtet, einfach rückgekoppelt, vorwärtsgerichtet mit Verzweigungen usw.), um die Anzahl benötigter NULL-Nachrichten zur Aufhebung von Blockierungen zu reduzieren.

Weitere konservative Ansätze zielen auf eine synchrone Abarbeitung der Simulation ab, d. h. die Simulation wird in Phasen zerlegt, zwischen denen die einzelnen Logischen Prozesse immer wieder synchronisiert werden. In [Lubachevsky,1988], [Lubachevsky,1989a] und [Lubachevsky,1989b] wird z. B. für jeden Logischen Prozeß ein Zeitfenster berechnet, in dem sich nur „sichere“ Ereignisse befinden. Anschließend werden diese Ereignisse bearbeitet, bevor wieder eine Synchronisation erfolgt. Ein ähnlicher Ansatz wird in [Ayani,1989] verfolgt. In [Chandy&Sherman,1989] wird die Kenntnis von Abhängigkeiten zwischen Ereignissen dazu benutzt, „sichere“ Ereignisse zu bestimmen und zu simulieren.

2.5.2 Optimistische Verfahren

2.5.2.1 Grundprinzip

Eine andere „Philosophie“ als bei konservativen wird bei *optimistischen Verfahren* verfolgt, zuerst vorgestellt unter dem Namen *Time Warp* in [Jefferson,1985]. Hier wird das kleinste im lokalen Kalender jedes Logischen Prozesses vorhandene Ereignis immer simuliert, ohne Rücksicht darauf, ob noch eine Nachricht mit kleinerem Zeitstempel eintreffen kann oder nicht. Es wird optimistisch davon ausgegangen, daß dies nicht der Fall sein wird. War diese Annahme falsch, und tritt nun, wie in Unterkapitel 2.4 beschrieben, ein Nachzügler-Ereignis auf, so muß der Logische Prozeß „zurücksetzen“, er muß einen sog. *Rollback* durchführen. Hinter optimistischen Verfahren steckt also die Idee, lieber etwas möglicherweise Falsches zu machen und bei Bedarf zu korrigieren, als überhaupt nichts zu tun.

In Bild 2.8 sind die lokalen Ereigniskalender dreier Logischer Prozesse dargestellt. Die lokale Zeit von LP^2 ist 95, als ein Nachzügler-Ereignis von LP^1 mit Zeit $T_S = 40$ eintrifft. LP^2 muß

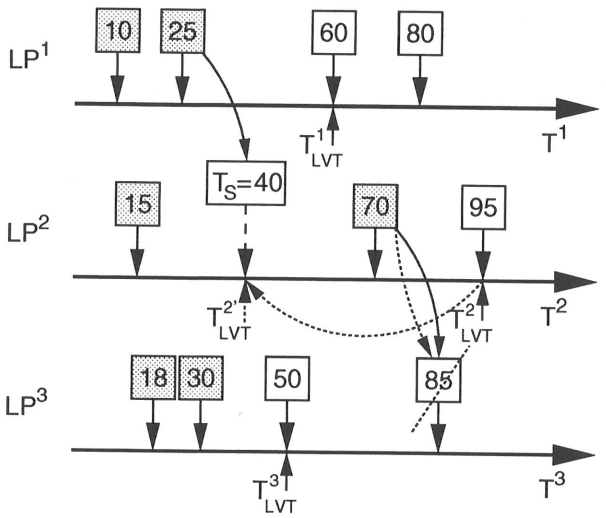

Bild 2.8: Zurücksetzen eines Logischen Prozesses

nun alle Effekte der schon bearbeiteten Ereignisse mit Zeit $T^2_{E_i} \geq T_S$ rückgängig machen, da diese unter falschen Voraussetzungen entstanden sind. Er stellt hierzu seine lokale Zeit T^2_{LVT} auf die Zeit $T^{2'}_{LVT} = T_S$ zurück und stellt seinen internen Zustand zu diesem Zeitpunkt wieder her. In der Zwischenzeit versandten Nachrichten schickt er sog. *Anti-Nachrichten* hinterher, um diese „auszulöschen". Der Empfänger einer Anti-Nachricht muß die Effekte der Bearbeitung der zugehörigen Nachricht seinerseits rückgängig machen. In Bild 2.8 hatte das Ereignis mit $T^2_E = 70$ in LP^2 eine Nachricht erzeugt, die mit Zeitstempel 85 an LP^3 gesandt wurde. Letzterer hatte die Ankunft der Nachricht als Ereignis in seinen Kalender eingetragen. Durch das Zurücksetzen von LP^2 muß dieser eine Anti-Nachricht mit Zeitstempel 85 an LP^3 verschicken, worauf LP^3 das Ankunftsereignis der Nachricht aus seinem Kalender streicht. Dies geht natürlich nur, da LP^3 dieses Ereignis und damit die zugehörige Nachricht noch nicht bearbeitet hat. Falls dies doch der Fall wäre, müßte LP^3 seinerseits zurücksetzen.

2.5.2.2 Aggressive und Lazy Cancellation

Im Beispiel aus Bild 2.8 werden Anti-Nachrichten sofort beim Zurücksetzen versandt. Diese Vorgehensweise wird als *Aggressive Cancellation* bezeichnet. Es besteht allerdings eine gewisse Wahrscheinlichkeit, daß der auslösende Nachzügler die entsprechende Nachricht überhaupt nicht beeinflußt, so daß die gleiche Nachricht bei der nochmaligen Simulation der zurückgesetzten Zeitspanne erneut erzeugt wird. In diesem Falle wäre unnötigerweise

eine Anti-Nachricht versandt worden und hätte u. U. Folge-Rollbacks ausgelöst. *Lazy Cancellation* [Gafni,1988] versendet aus diesem Grund beim Zurücksetzen vorerst keine Anti-Nachrichten. Erst wenn bei der erneuten Vorwärtssimulation festgestellt wird, daß eine Nachricht nicht wieder oder nur in veränderter Form erzeugt wird, wird die Anti-Nachricht versandt. Neben dem Vorteil, daß korrekte Nachrichten nicht unnötigerweise zurückgeholt werden, hat das Verfahren aber den Nachteil, daß inkorrekte Nachrichten sich durch das verspätete Versenden von Anti-Nachrichten weiter im System ausbreiten können. Vergleiche beider Verfahren finden sich z. B. in [Gafni,1988] und [Reiher et al.,1990].

2.5.2.3 Zustandssicherung

Um zurücksetzen und einen alten Zustand wiederherstellen zu können, muß der Logische Prozeß diesen Zustand zuvor gesichert haben. Die Art und Weise der Zustandssicherung ist ein kritischer Punkt bei optimistischen Simulationsverfahren, da sie sowohl Zeit als auch Speicher verbraucht. Die beiden zentralen Ansätze sollen im folgenden kurz erläutert werden.

2.5.2.3.1 Zustandskopien

Eine Möglichkeit der Zustandssicherung stellt die periodische Erstellung von Zustandskopien (*Checkpointing* oder *Copy State Saving*) dar, die den gesamten Zustand des Simulationsmodells enthalten. Bild 2.9 zeigt den Vorgang des Zurücksetzens eines Logischen Prozesses bei dieser Art der Zustandssicherung. Ein LP mit der lokalen Zeit T_{LVT} empfängt einen Nachzügler mit dem Zeitstempel T_S. Da normalerweise aufgrund des Speicherplatz- und Zeitbedarfs nicht nach jedem bearbeiteten Ereignis sondern nur zu bestimmten Zeitpunkten $T_{SS,i}$ eine Zustandskopie erstellt wird, gliedert sich die Wiederherstellung des Zustands zum Zeitpunkt T_S in zwei Phasen. Zuerst wird die Zustandskopie mit der größten Zeit kleiner

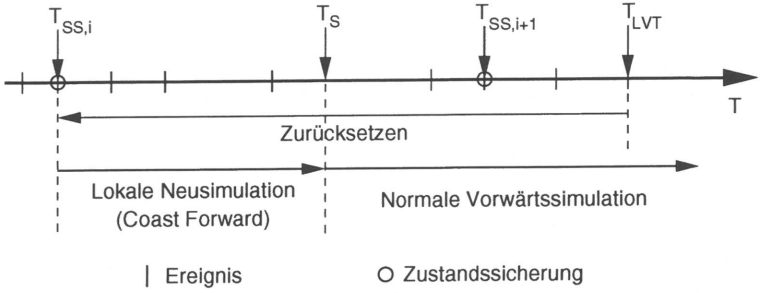

Bild 2.9: Wiederherstellung eines Zustands bei Sicherung mittels Zustandskopien

als der Zeit des Nachzüglers wiederhergestellt. Seien $T_{SS,i}$, $0 \leq i \leq n$ die Zeiten aller gespeicherten Zustandskopien, dann ergibt sich zuerst die neue lokale Zeit des Logischen Prozesses aus

$$T'_{LVT} = \max_{0 \leq i \leq n} \{T_{SS,i} \mid T_{SS,i} < T_S\} \ .$$

Anschließend werden die Ereignisse mit Zeitstempeln $T'_{LVT} \leq T_E < T_S$ während der sog. *Coast-Forward-Phase* erneut bearbeitet und dadurch der Zustand zum Zeitpunkt T_S wiederhergestellt. Die Auswirkungen des Zurücksetzens auf einen Zeitpunkt $T'_{LVT} < T_S$ müssen rein lokal gehalten werden, da ansonsten ein Voranschreiten der Simulation aufgrund der Möglichkeit des fortgesetzten wechselseitigen Versendens von Anti-Nachrichten nicht garantiert werden kann. Deshalb dürfen für versandte Nachrichten mit Zeitstempel $T_M < T_S$ keine Anti-Nachrichten und entsprechend keine während der Coast-Forward-Phase nochmals erzeugten Nachrichten versandt werden.

Die Wahl des zeitlichen Abstands zwischen zwei Zustandskopien, des sog. *Checkpoint-Intervalls*, ist kritisch für die Leistung der optimistischen Simulation. [Preiss et al.,1992] und [Bellenot,1992] untersuchen den Zusammenhang zwischen der Größe des Checkpoint-Intervalls und dem Zurücksetzverhalten eines Logischen Prozesses, dem Speicherplatz- und dem Zeitbedarf. In [Lin et al.,1993] werden auf der Basis eines analytischen Modells obere und untere Schranken für den durch die Zustandssicherung und das Zurücksetzen verursachten Zeitaufwand hergeleitet. Daraus wiederum werden Schranken für das optimale Checkpoint-Intervall, das diesen Zeitaufwand minimiert, bestimmt. Diese Schranken begrenzen das sog. *Lin-Lazowska-Intervall*, in dem sich das optimale Checkpoint-Intervall befindet. Außerdem wird ein iterativer Algorithmus angegeben, dieses zu bestimmen.

Weitere Ansätze zur dynamischen und adaptiven Bestimmung des Checkpoint-Intervalls sind z. B. in [Rönngren&Ayani,1994] und [Fleischmann&Wilsey,1995] beschrieben.

2.5.2.3.2 Inkrementelle Zustandssicherung

Bei Simulationsmodellen mit sehr großen Zustandsräumen mit nur wenig Zustandsänderungen zwischen den einzelnen Sicherungen, wie sie z. B. bei der Simulation großer digitaler Schaltungen auftreten, sind Zustandskopien hinsichtlich Speicherplatz- und Zeitbedarf ineffizient. Hier kann sich die *Inkrementelle Zustandssicherung* (*Incremental State Saving*, siehe z. B. [Bauer&Sporrer,1993]), bei der nur die Zustandsänderungen gespeichert werden, als vorteilhaft erweisen. Wie in Bild 2.10 gezeigt, müssen in diesem Fall beim Zurücksetzen eines Logischen Prozesses alle Zustandsänderungen mit $T_S \leq T_{\Delta S,i} < T_{LVT}$ in umgekehrter Reihenfolge ihrer Erzeugung rückgängig gemacht werden, wobei $T_{\Delta S,i}$ der Zeitpunkt der i-ten Zustandsänderung ist.

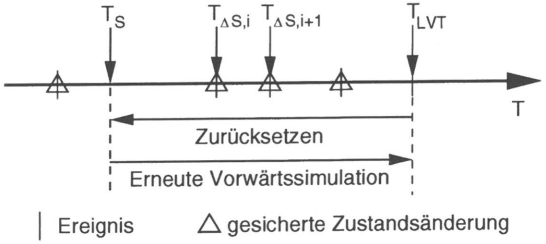

Bild 2.10: Wiederherstellung eines Zustands bei inkrementeller Zustandssicherung

Welche Methode der Zustandssicherung vorzuziehen ist, hängt von verschiedenen Parametern wie der Häufigkeit des Zurücksetzens, der mittleren Anzahl zurückgesetzter Ereignisse, der mittleren Dauer der Bearbeitung eines Ereignisses und dem Aufwand zur Herstellung einer Zustandskopie ab [Palaniswamy&Wilsey,1993].

2.5.2.4 Approximation der Globalen Virtuellen Zeit

Während die Lokalen Virtuellen Zeiten der einzelnen Logischen Prozesse durch Zurücksetzen auch kleiner werden können, wächst die Globale Virtuelle Zeit (GVT, siehe Definition 2.2) monoton [Jefferson,1985]. Außerdem ist sichergestellt, daß kein Nachzügler mit einer kleineren Zeit als der GVT auftritt, d. h. kein Logischer Prozeß auf eine kleinere Zeit als die GVT zurücksetzen muß. In der Praxis kann es aufgrund unvollständig durchgeführter Zustandssicherungen zwar vorkommen, daß der Zeitpunkt der wiederherzustellenden Zustandskopie kleiner ist als die GVT. Dies ist aber ohne Bedeutung, da beim Zurücksetzen die externen Effekte von Ereignissen mit $T_E < T_S$ nicht rückgängig gemacht werden und das Zurücksetzen somit virtuell auf den Zeitpunkt T_S erfolgt (siehe Unterabschnitt 2.5.2.3).

Die GVT ist also eine Garantie dafür, daß Ereignisse mit $T_E \leq T_{GVT}$ gültig sind und nicht wieder zurückgesetzt werden. Die GVT kann damit durch das Löschen nicht mehr benötigter Zustandssicherungen zum Aufräumen des Speichers (*Fossil Collection*) benutzt werden. Ausgabeoperationen, die nicht wieder zurückgenommen werden können, müssen zurückgehalten werden, bis durch die GVT sichergestellt werden kann, daß kein Zurücksetzen auf einen kleineren Zeitpunkt als den der Operation selbst mehr erfolgen kann. Schließlich kann die GVT auch noch zum Erkennen des Simulationsendes eingesetzt werden.

Die Berechnung der GVT soll möglichst genau erfolgen, den Fortgang der Simulation selbst möglichst wenig beeinflussen und mit geringem Berechnungs- und Kommunikationsaufwand verbunden sein.

Um die GVT exakt berechnen zu können, müßten alle Logischen Prozesse während der Berechnung angehalten und der Empfang aller im Transit befindlichen Nachrichten abgewartet

werden [Beraldi&Nigro,1995]. Da dies die Simulation natürlich stark verlangsamt, wird in der Regel, während die Simulation weiterläuft, eine untere Schranke für den tatsächlichen Wert der GVT bestimmt.

Die aus der Literatur bekannten Algorithmen zur Bestimmung der GVT unterscheiden sich im wesentlichen in der Genauigkeit der Approximation, dem Aufwand für die Berechnung und Kommunikation sowie den Annahmen über die Eigenschaften der Kommunikation zwischen den Logischen Prozessen.

[Bellenot,1990] und [D'Souza et al.,1994] gehen davon aus, daß ein LP für eine empfangene Nachricht eine Bestätigungsmeldung zurück an den sendenden LP schickt. Dadurch können durch die jeweiligen sendenden LPs die sich im Transit befindlichen Nachrichten festgestellt und ihr minimaler Zeitstempel bestimmt werden. Außerdem muß der Übertragungskanal zwischen sendendem und empfangendem LP nicht unbedingt fehlerfrei und nicht reihenfolgegesichert sein. Durch das Versenden von Bestätigungsmeldungen verdoppelt sich der Kommunikationsaufwand.

[Tomlinson&Garg,1993] kommen ohne Bestätigungsmeldungen aus, erwarten dafür aber einen fehlerfreien Kanal. [Chandy&Lamport,1985] benötigen zusätzlich noch einen FIFO-Kanal, d. h. einen reihenfolgegesicherten Kanal. Durch zusätzlich über die einzelnen Kanäle versandte Steuernachrichten kennzeichnet ein LP, daß er die folgenden Nachrichten für die GVT-Berechnung berücksichtigt hat. Auf diese Weise kann der Empfänger bis zum Erhalt dieser Steuernachricht auf die im Transit befindlichen Nachrichten warten. Die im Transit befindlichen Nachrichten werden aus dem Kanal unter Ausnutzung dessen FIFO-Eigenschaft sozusagen „ausgespült".

[Bauer,1994] geht von den gleichen Voraussetzungen aus. Hier wird aber keine verteilte GVT-Berechnung angestoßen, sondern die GVT wird aus Statusinformationen gewonnen, die von den einzelnen Logischen Prozessen von Zeit zu Zeit automatisch an einen zentralen Prozeß gesandt wird.

[Mattern,1993] stellt einen Algorithmus vor, der weder Bestätigungsmeldungen noch FIFO-Kanäle benötigt, wohl aber eine gesicherte Übertragung. Jedem Logischen Prozeß ist eine „Farbe" zugeordnet, die er nach Eintreffen einer entsprechenden Aufforderung und Erstellen einer lokalen „Momentaufnahme" (*Snapshot*) der LVT, ändert. Versandte Nachrichten werden mit der jeweiligen Farbe des sendenden LPs „eingefärbt", anhand derer sich bei Empfang der Nachricht durch einen anderen LP feststellen läßt, ob ihr Zeitstempel schon in der Momentaufnahme des Senders berücksichtigt wurde. Ein verteilter Terminierungsalgorithmus wird benutzt, um festzustellen, wann die Gesamt-Momentaufnahme und damit die Approximation der GVT abgeschlossen ist.

2.5.3 Hybride Synchronisationsansätze

Ein großes Problem bei optimistischen Verfahren ist das kaskadierte Zurücksetzen nachfolgender Logischer Prozesse. Ein zurücksetzender LP versendet Anti-Nachrichten, die die nachfolgenden Prozesse ihrerseits zum Zurücksetzen veranlassen, wodurch weitere Anti-Nachrichten generiert werden. Dies kann zu einem Schneeballeffekt und einer lawinenartig steigenden Zahl von zurücksetzenden Logischen Prozessen führen, besonders, wenn das Simulationsmodell Rückkopplungen enthält. Auf der anderen Seite blockieren konservative Verfahren die Logischen Prozesse oftmals unnötigerweise und sind ganz besonders von hohen Lookahead-Werten abhängig. Deshalb gibt es verschiedene Ansätze, die die Vorteile beider Welten kombinieren und ihre Nachteile eliminieren wollen. Einige dieser hybriden Ansätze sollen im folgenden kurz vorgestellt werden.

2.5.3.1 Risikofreie optimistische Simulation

Die Idee der risikofreien optimistischen Simulationsverfahren besteht darin, brachliegende Rechenkapazität von ansonsten blockierten Logischen Prozessen durch optimistische Simulation zu nutzen, die Effekte der nicht sicheren Ereignisse, d. h. der Ereignisse, die u. U. zurückgesetzt werden können, aber lokal zu halten. Generierte Nachrichten an andere Logische Prozesse werden zurückgehalten, bis sie sicher sind, d. h. die optimistische Vorausberechnung sich als richtig erwiesen hat. Erweist sich diese jedoch als falsch, so setzt der Logische Prozeß lokal zurück. Andere Logische Prozesse sind von diesem Zurücksetzen aber nicht betroffen, da sich alle unsicheren Nachrichten noch im Ausgangspuffer befinden und dort lediglich entfernt werden müssen. Auf Anti-Nachrichten kann bei diesen Verfahren also verzichtet werden, und es kann damit auch kein lawinenartiges Zurücksetzen mehrerer Logischer Prozesse erfolgen.

Die in [Mehl,1991a] vorgestellte *Spekulative Simulation* geht von konservativer Simulation aus. Wenn ein Logischer Prozeß blockiert ist, simuliert er auf einer Zustandskopie des Simulationsmodells optimistisch weiter. Erwies sich die spekulative Ausführung als falsch, muß lediglich die Zustandskopie verworfen werden.

In [Steinman,1992b] wird ein Algorithmus namens *Breathing Time Buckets* vorgestellt. Die Simulation wird dabei in zeitliche Abschnitte aufgeteilt, an deren Ende sich alle Logischen Prozesse synchronisieren. Die Dauer eines Abschnitts wird durch den sog. *Ereignishorizont* bestimmt, dem Zeitpunkt des frühesten durch ein Ereignis des aktuellen Abschnitts erzeugten neuen Ereignisses. Es wird zwischen „lokalem“ und „globalem“ Ereignishorizont unterschieden. Ersterer berücksichtigt nur die lokal in einem Logischen Prozeß vorhandenen Ereignisse als mögliche Quellen neuer Ereignisse, letzterer ist das Minimum aller lokalen Ereignishorizonte. Die Bearbeitung aller Ereignisse innerhalb des globalen Ereignishorizonts

ist sicher. Da der globale Ereignishorizont erst am Ende eines Abschnitts berechnet wird, die einzelnen Logischen Prozesse sich aber erst am Ende ihres lokalen Ereignishorizonts synchronisieren, kann es vorkommen, daß ein Logischer Prozeß zurücksetzen muß. Nachrichten an andere Prozesse hält ein Logischer Prozeß aber so lange zurück, bis sie durch die Berechnung des globalen Ereignishorizonts sicher geworden sind. Aus diesem Grund ist auch dieses Verfahren risikofrei.

In [Bellenot,1993] wird eine Erweiterung des *Time Warp Operating System* (*TWOS*) aus [Jefferson et al.,1987] vorgestellt, welches eine der ersten Implementierungen optimistischer Verfahren ist. Dieses *Riskfree TWOS* genannte System arbeitet nach einem ähnlichen Prinzip wie Breathing Time Buckets, verzichtet aber auf die Synchronisation der Logischen Prozesse.

2.5.3.2 Zeitfenster

Die Wahrscheinlichkeit für das Zurücksetzen eines Logischen Prozesses ist umso größer, je weiter die einzelnen lokalen virtuellen Zeiten auseinanderliegen. Außerdem wächst die mittlere Zeitspanne, um die zurückgesetzt werden muß, was die Leistung der parallelen Simulation ebenfalls negativ beeinflußt. Aus diesen Gründen wurden Ansätze entwickelt, die ein solches Auseinanderlaufen durch die Einführung von Zeitfenstern begrenzen. Am Ende eines Zeitfensters angelangt, wird ein Logischer Prozeß so lange blockiert, bis dieses vorangeschoben wird (bzw. er aufgrund eines Nachzüglers zurücksetzen muß).

In [Reiher et al.,1989] wird eine naheliegende Zeitfenstermethode untersucht. Beim *Window-Based Throttling* dürfen Ereignisse nur dann bearbeitet werden, wenn die Zeitdifferenz zur kleinsten LVT im System – gleichbedeutend mit der GVT – geringer als ein bestimmter fester Wert ist. Das Zeitfenster erstreckt sich also von der GVT über einen einzustellenden Zeitraum – die Größe des Zeitfensters – und wird vorangeschoben, sobald sich die GVT erhöht. Hauptprobleme dieses Ansatzes sind die Abhängigkeit der geeigneten Zeitfenstergröße vom Simulationsmodell (dessen Zeitskala und der Variation seiner Ereigniszeitabstände), die Verhinderung korrekter optimistischer Simulation (und damit Einbußen der ausgenutzten Parallelität) und schließlich die berichteten geringen Leistungssteigerungen gegenüber klassischem *Time Warp*.

Beim *Bounded Time Warp* aus [Turner&Xu,1992] wird die Simulationsdauer in Zeitintervalle aufgeteilt. Jeder Logische Prozeß simuliert Ereignisse innerhalb eines Zeitfensters, das nach oben hin durch das Ende des aktuellen Zeitintervalls begrenzt wird. Mittels eines verteilten Terminierungsalgorithmus wird überprüft, ob alle Ereignisse innerhalb der Zeitfenster der einzelnen Logischen Prozesse abgearbeitet sind. Ist dies der Fall, wird die Simulation für das nächste Intervall gestartet.

[Ferscha,1995] versucht aus der Vorgeschichte der Nachrichtenankünfte eines Logischen Prozesses mit Hilfe statistischer Methoden den Ankunftszeitpunkt der nächsten Nachricht vor-

auszuberechnen, um damit die Zeitfenstergröße optimal einstellen zu können. Nachteilig ist hierbei allerdings der meist hohe Berechnungsaufwand für die Vorhersage und die bei stochastischen Modellen oftmals geringe Autokorrelation des Nachrichten-Ankunftsprozesses, die eine Vorhersage unmöglich macht [ten Brink,1995].

Das *Penalty-Based Throttling* aus [Reiher et al.,1989] ist nicht direkt ein Zeitfenstermechanismus, soll aber der Vollständigkeit halber aufgeführt werden. Hier werden LPs, die fehlerhaft optimistisch simuliert haben und deswegen zurücksetzen und eine Anti-Nachricht versenden müssen, „bestraft", indem sie bei der Rechenzeitvergabe auf einem Rechenknoten eine Zeitlang zugunsten anderer LPs übergangen und dadurch gebremst werden. Auch für dieses Verfahren werden nur sehr geringe Leistungssteigerungen berichtet.

2.5.3.3 Filter-Ansätze

Bei den klassischen Ansätzen zur optimistischen Simulation werden Anti-Nachrichten über dieselben Wege wie die entsprechenden positiven Nachrichten versandt, sie werden den Simulationsnachrichten „hinterhergeschickt". Dadurch kann es bei kaskadiertem Zurücksetzen mehrerer Logischer Prozesse vorkommen, daß Anti-Nachrichten und Folge-Anti-Nachrichten die jeweiligen Simulationsnachrichten durch einen großen Teil des Simulationssystems hindurch „jagen", bevor sie diese einholen. Es kann also sehr lange dauern, bis ein sich ausbreitender Fehler gestoppt werden kann. Es gibt deshalb Ansätze, bei denen ein zurücksetzender Logischer Prozeß an alle, bzw. an alle möglicherweise betroffenen, Logischen Prozesse sofort Informationsnachrichten über sein Zurücksetzen sendet. Deren Verbreitung erfolgt auf anderen, schnelleren Wegen als für die Simulationsnachrichten zur Verfügung stehen. Deswegen ist ein empfangender Logischer Prozeß günstigenfalls schon vor Eintreffen einer Simulationsnachricht über deren mögliche Fehlerhaftigkeit informiert, und er kann diese „herausfiltern".

Beim *WOLF*-Verfahren aus [Madisetti et al.,1988] ist für jeden Logischen Prozeß eine Einflußsphäre definiert als die Menge Logischer Prozesse, die eine von ihm versandte Nachricht innerhalb einer bestimmten Echtzeit erreichen kann. Die Einflußsphäre hängt also nicht nur von den Verbindungen der Logischen Prozesse untereinander, sondern auch von den realen Nachrichtenübertragungszeiten der zugrundeliegenden Rechenplattform ab. Setzt ein Logischer Prozeß zurück und erweist sich eine Simulationsnachricht dadurch als fehlerhaft, so wird eine entsprechende Information an alle Logischen Prozesse der für diese Nachricht geltenden Einflußsphäre gesandt. Letztere ergibt sich u. a. aus der Zeitspanne zwischen Senden der Nachricht und dem Feststellen ihrer Fehlerhaftigkeit.

Bei dem in [Prakash&Subramanian,1991] vorgestellten *Filter*-Algorithmus wird jeder versandten Simulationsnachricht eine Liste von Annahmen mitgegeben, unter denen sie gültig ist. Setzt ein Logischer Prozeß zurück, so informiert er hierüber alle anderen Logischen Prozesse. Widersprechen die Annahmen einer empfangenen Nachricht den gespeicherten

Informationen über zurückgesetzte Logische Prozesse, so wird die Nachricht „gefiltert“ und nicht bearbeitet, da sie sowieso durch eine nachfolgende Anti-Nachricht ausgelöscht würde.

2.6 Wiederholbarkeit der Simulation

Eine wichtige Forderung an ein ereignisgesteuertes Simulationssystem ist die Reproduzierbarkeit der Ergebnisse. Zwei mit den gleichen Parametern durchgeführte Simulationsläufe sollen exakt gleiche Ergebnisse liefern. Im Normalfall ist dies gewährleistet, da die Abarbeitung von Ereignissen immer in aufsteigender Folge ihrer Zeitstempel erfolgt und deshalb die Bearbeitungsreihenfolge in jedem Lauf exakt vorgegeben ist. Probleme ergeben sich, wenn mehrere Ereignisse den gleichen Zeitstempel haben. In diesem Fall ist durch das Simulationsmodell keine Reihenfolge vorgegeben, die Priorisierung erfolgt durch das Simulationssystem z. B. nach der Reihenfolge des Eintrags in den Ereigniskalender. Bei der sequentiellen Simulation ist diese Reihenfolge in jedem Lauf einfach reproduzierbar, anders bei paralleler Simulation. Dort hängt die Reihenfolge des Eintrags u. a. von Laufzeiteffekten zwischen den verteilten Logischen Prozessen ab, die sich nicht reproduzieren lassen. Hier sind aufwendigere Mechanismen zur Konfliktauflösung „gleichzeitiger“ Ereignisse, sog. *Tie-Breaking-Verfahren*, erforderlich, wie sie z. B. in [Mehl,1991b] und [Mehl,1992] beschrieben werden.

Das dort beschriebene Verfahren erweitert den Zeitbegriff in drei Schritten. Im ersten wird einem Zeitstempel ein zusätzliches *Alter* zugeordnet. Plant ein Ereignis E_1 ein Ereignis E_2 ein, so ergibt sich das Alter A_2 von E_2 zu

$$A_2 = \begin{cases} A_1 + 1 & \text{für } T_{E_1} = T_{E_2} \\ 1 & \text{für } T_{E_1} \neq T_{E_2} \end{cases} .$$

Durch das Alter allein lassen sich nicht alle Konflikte lösen. So bekommen mehrere durch E_1 zum gleichen Zeitpunkt eingeplante Ereignisse auch das gleiche Alter. In einem zweiten Schritt wird deswegen jedem Ereignis zusätzlich ein global eindeutiger Bezeichner id' zugeordnet. Dieser kann sich z. B. aus einem global eindeutigen Bezeichner des einplanenden Logischen Prozesses und einem Zähler aller durch diesen eingeplanten und nicht zurückgesetzten Ereignisse zusammensetzen. Der erweiterte Zeitstempel läßt sich also schließlich als 3-Tupel darstellen:

$$\tau_E = (T_E, A, id') .$$

Bei Lazy Cancellation (siehe Abschnitt 2.5.2.2) kann es vorkommen, daß ein zurückgesetztes Ereignis ein korrektes Ereignis erzeugt hat, welches selbst nicht zurückgesetzt wird. Dieses Folgeereignis kann deshalb mit einem falschen Tupel versehen sein. In einem dritten Schritt werden aus diesem Grund für solche Fälle Korrekturnachrichten vorgesehen.

Kapitel 3

Parallelverarbeitung und objektorientierter Softwareentwurf

3.1 Parallelverarbeitung

3.1.1 Motivation

Die Fortschritte in der Leistungsfähigkeit moderner Rechnersysteme sind nicht zuletzt einem hohen Maß an Parallelverarbeitung zu verdanken, deren sich der Anwender meistens nicht bewußt ist. Breitere Bussysteme, Fließbandverarbeitung und in Zukunft zunehmend Multiprozessorsysteme erschließen dem Workstationbereich immer mehr Gebiete, die seither Großrechnern vorbehalten waren. Bei letzteren zeichnet sich ein Trend zu Multiprozessorsystemen mit einer großen Anzahl Prozessoren ab, um weitere Leistungssteigerungen zu ermöglichen. Dieses Unterkapitel soll einen Überblick über verschiedene Aspekte der Parallelverarbeitung geben und die für die vorliegende Arbeit wichtigen Punkte herausheben.

3.1.2 Klassifikation

3.1.2.1 Rechnertypen

Obwohl Parallelverarbeitung auch in klassischen „Einprozessorsystemen“ zu finden ist (s. u.), sollen hier vor allem Systeme mit mehreren Prozessoren betrachtet werden. Es ergibt sich die Frage der Kategorisierung solcher Systeme aufgrund verschiedener Gesichtspunkte. Eine erste grundlegende Definition ist die folgende:

Definition 3.1 (Multiprozessorsystem)
Ein *Multiprozessorsystem* ist ein Rechnersystem, das als wesentliches Merkmal über mehr als einen Prozessor verfügt. Diese Prozessoren – von durchaus unterschiedlicher Komplexität – können miteinander kommunizieren und teilweise kooperieren, um Aufgaben verschiedenster Ausprägung gemeinsam zu bearbeiten. Das schließt eine asynchrone Parallelverarbeitung mit ein. Die Komponenten des Gesamtsystems (Prozessoren, Speicher, Ein-/Ausgabe) sind über unterschiedlich ausgeprägte Verbindungsnetze miteinander gekoppelt und werden von einem gemeinsamen Betriebssystem kontrolliert. [Gonauser&Mrva,1989]

Die Begriffe *Multiprozessorsystem* und *Parallelrechner* werden in dieser Arbeit synonym gebraucht. Im Gegensatz dazu bestehen sog. *Multicomputersysteme* aus weitgehend autonomen Rechnereinheiten mit je einem eigenen Betriebssystem und eigenen Ein-/Ausgabemöglichkeiten [Gonauser&Mrva,1989]. Diese Systeme heißen auch *verteilte Systeme* und stellen den Systemverbund meist über ein lokales Kommunikationsnetz (*Local Area Network*, *LAN*) her.

Mit einer Klassifikation aus Hardware-Sicht lassen sich nach [Flynn,1966] alle Rechner in die Kategorien aus Tabelle 3.1 einteilen. Da diese Einteilung recht grob ist und die Klasse *MISD* zudem keine praktische Bedeutung besitzt, wird die Klasse *MIMD* auch noch in die beiden in Tabelle 3.2 dargestellten Unterkategorien aufgespalten (siehe [Geiger,1991], [Lewis,1991]).

SISD	Single Instruction Stream Single Data Stream	klassische „von-Neumann-Rechner“ mit Einzelprozessor
MISD	Multiple Instruction Stream Single Data Stream	ohne praktische Bedeutung
SIMD	Single Instruction Stream Multiple Data Stream	Rechner, die mit einer Instruktion mehrere Daten verarbeiten können (Vektor-, Arrayrechner)
MIMD	Multiple Instruction Stream Multiple Data Stream	Rechner mit mehreren Prozessoren, die unterschiedliche Instruktionen auf unterschiedlichen Daten ausführen können

Tabelle 3.1: Klassifikation von Rechnern nach [Flynn,1966]

SPMD	Single Program Multiple Data Stream	MIMD-Rechner, die auf allen Prozessoren das gleiche Programm abarbeiten, das sich aber in unterschiedlichen Zweigen befinden kann
MPMD	Multiple Program Multiple Data Stream	MIMD-Rechner, bei denen auf allen Prozessoren unterschiedliche Programme abgearbeitet werden können

Tabelle 3.2: Unterklassifikation von MIMD-Rechnern

3.1.2.2 Parallelitätsebenen

Eine weitere Art der Klassifikation läßt sich nach dem Abstraktionsgrad der Parallelität finden, wofür aber leider keine vollständig einheitliche Definition existiert. Die in Tabelle 3.3 aufgeführten Ebenen sind aus [Geiger,1991], [Bräunl,1993] und [Waldschmidt,1995] abgeleitet. In der Tabelle steigt die *Granularität* der Parallelität von unten nach oben an. Grobgranulare Parallelität ist vor allem auf MIMD-Systemen zu finden, während feine Granularität vor allem auf SIMD-Rechnern und im Design moderner Mikroprozessoren zu finden ist. Besonders auf Bitebene durch breitere Datenbusse und Register und teilweise auf Instruktionsebene ist die Parallelverarbeitung auch in kleineren Rechnersystemen verbreitet. Die Parallelität auf Ebene der Datenstrukturen heißt auch *Datenparallelität* und ist häufig auf *SIMD*-Rechnern vorzufinden. Hier existiert nur ein einziger Kontrollfluß, der gleichzeitig auf viele verschiedene Datenelemente angewandt wird.

Programmebene	Ausnutzung der Parallelität zwischen verschiedenen, gleichzeitig laufenden Programmen, die vom Betriebssystem verwaltet werden (Multi-Tasking, Multi-User-Mode); diese Ebene steigert im wesentlichen den Durchsatz an Programmen und nicht deren Bearbeitungsgeschwindigkeit
Ebene kooperierender Prozesse	Ausnutzung der Parallelität von Abschnitten innerhalb eines Programms durch Zerlegung in weitgehend unabhängige Teilprobleme und Abspaltung, z. B. einzelner Unterprogramme, in separate „Prozesse“ oder „Tasks“, Parallelisierung erfolgt durch den Programmierer auf algorithmischer Ebene
Ebene der Datenstrukturen	Parallele Ausführung unabhängiger Teiloperationen auf Datenstrukturen, z. B. bei der Addition oder Multiplikation von Vektoren oder Matrizen; Parallelisierung erfolgt durch den Programmierer auf Sprachebene
Anweisungsebene	Parallele Verarbeitung einzelner Anweisungen oder Anweisungsblöcke, wie z. B. von Schleifen; Parallelisierung erfolgt durch den Programmierer oder den Compiler
Instruktionsebene	Komponentenweise parallele Verarbeitung von Basisoperationen eines arithmetischen Ausdrucks oder von Maschinenbefehlen; Parallelisierung erfolgt durch den Compiler oder direkt durch die Hardware
Bitebene	Parallele Ausführung von Bit-Operationen, wie z. B. einer 8-Bit-Addition in der Arithmetisch-Logischen Einheit (*Arithmetic Logical Unit, ALU*) eines Prozessors; Parallelisierung erfolgt auf Hardware-Ebene

Tabelle 3.3: Parallelitätsebenen

In der vorliegenden Arbeit wird die Parallelität auf der Ebene kooperierender Prozesse ausgenutzt, d. h. die parallele Simulation wird, wie schon in Kapitel 2 beschrieben, in einzelne weitgehend unabhängige und asynchron laufende Prozesse aufgeteilt, die auf verschiedenen Rechenknoten bearbeitet werden[1]. Es sollen im folgenden nun die dafür geeigneten MIMD-Rechner näher vorgestellt werden.

3.1.3 MIMD-Rechner

3.1.3.1 Kriterien für eine Klassifikation

Nach [Waldschmidt,1995] lassen sich MIMD-Rechner nach den in Bild 3.1 aufgeführten Merkmalen kategorisieren. Obwohl alle diese Merkmale orthogonal zueinander sind, sind nicht alle möglichen Kombinationen von Interesse. So ist z. B. auf Systemen mit gemeinsamem Speicher fast ausschließlich der globale Adreßraum als Programmiermodell vorzufinden und nicht der explizite Nachrichtenaustausch. Das häufigste Kriterium für die Unterscheidung von MIMD-Rechnern ist die physikalische Anordnung des Speichers (nicht zu verwechseln mit der logischen Sicht des Adreßraums, die davon abweichen kann). Die wichtigsten diesbezüglichen Kategorien werden im nächsten Unterabschnitt kurz vorgestellt, gefolgt von einem Unterabschnitt über Programmiermodelle von MIMD-Rechnern. Einzelheiten zu den übrigen Merkmalen finden sich z. B. in [Bräunl,1993], [Hwang,1993] oder [Waldschmidt,1995].

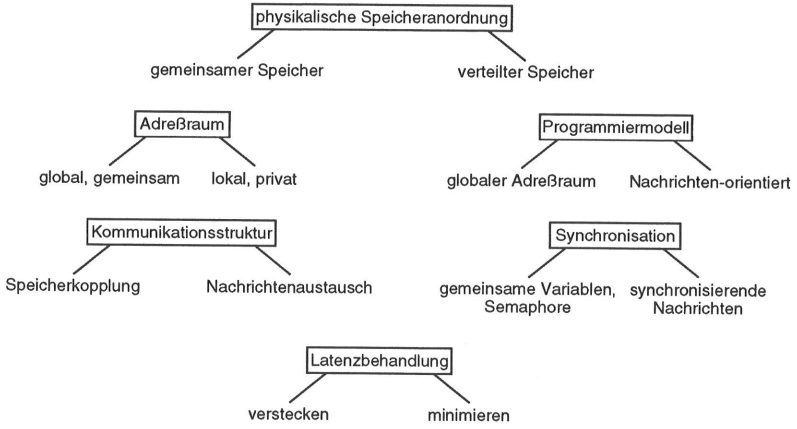

Bild 3.1: Merkmale für eine Einteilung von MIMD-Rechnern (nach [Waldschmidt,1995])

[1] Selbstverständlich wird auf jedem Rechenknoten auch Parallelität auf Bit- und teilweise auf Instruktionsebene genutzt. Dies geschieht aber automatisch durch die verwandten Mikroprozessoren und Compiler und ist deswegen hier ohne Bedeutung.

3.1.3.2 Anordnung des physikalischen Speichers

Die beiden Hauptkategorien bei der Anordnung des physikalischen Speichers von MIMD-Rechnern bilden Systeme mit *gemeinsamem Speicher* (*Shared-Memory*), auch *eng gekoppelte* Systeme genannt, und mit *verteiltem Speicher* (*Distributed-Memory*), auch *lose gekoppelte* Systeme genannt. Zwischen diesen beiden Extremen gibt es verschiedene Mischformen, so daß die Grenzen durchaus fließend sind.

Bei Systemen mit gemeinsamem Speicher ist der gesamte, meist in einzelne Module aufgeteilte Speicher von jedem Prozessorelement (*PE*) aus über ein Verbindungsnetz zugänglich. Nach der Art des Speicherzugriffs kann eine weitere Unterteilung erfolgen.

Bild 3.2 zeigt eine Architektur mit *gleichförmigem Speicherzugriff* (*Uniform Memory Access*, *UMA*), bei der die Zugriffsweise aller PEs auf die Speichermodule identisch ist. Die Anforderungen an die Bandbreite des Verbindungsnetzes sind hier sehr hoch und können nur bei einer begrenzten Anzahl von Prozessorelementen erfüllt werden. Aus diesem Grund ist diese Architektur nur eingeschränkt skalierbar.

Um die Begrenzung durch das Verbindungsnetz zu umgehen, bieten sich Architekturen mit *ungleichförmigem Speicherzugriff* (*Nonuniform Memory Access*, *NUMA*) an. Dazu kann zum einen jedes PE seinen eigenen lokalen *Cache*-Speicher zugeordnet bekommen, in dem häufig benötigte Daten zwischengespeichert werden, so daß nicht jedesmal über das Verbindungsnetz auf die eigentliche Speicherstelle zugegriffen werden muß. Dabei ergibt sich allerdings das Problem der Konsistenz der Daten im Cache und im Hauptspeicher, für deren Einhaltung z. T. aufwendige Protokolle erforderlich sind.

Eine andere Möglichkeit, das Verbindungsnetz zu entlasten, besteht in der Zuordnung der Speichermodule zu einzelnen Prozessorelementen (siehe Bild 3.3). Dadurch können lokale Zugriffe eines PEs auf das ihm zugeordnete Modul ohne Belastung des Verbindungsnetzes

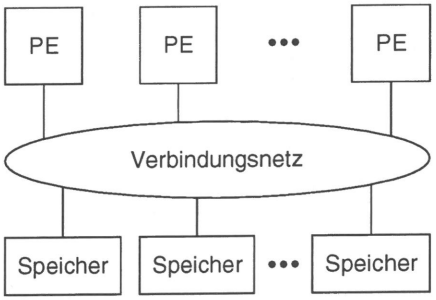

Bild 3.2: MIMD-Rechner mit *UMA*-Architektur

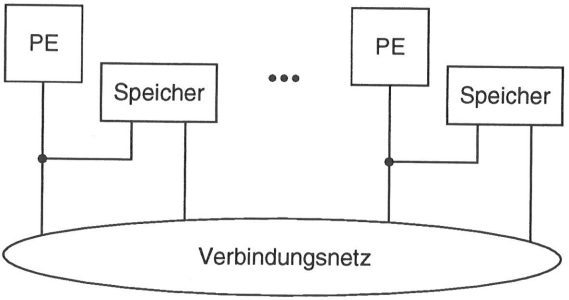

Bild 3.3: MIMD-Rechner mit *NUMA*-Architektur (nach [Waldschmidt,1995])

erfolgen. Lediglich globale Zugriffe auf Module, die anderen PEs zugeordnet sind, müssen weiterhin über dieses abgewickelt werden.

Der Vollständigkeit halber sei noch die *COMA*-Architektur (*Cache-Only Memory Access*) aufgeführt, bei der die Speichermodule ausschließlich als Cache-Speicher benutzt werden. Greift ein Prozessorelement auf Daten zu, die sich nicht in seinem lokalen Cache-Speichermodul befinden, werden diese automatisch in einem anderen Speichermodul lokalisiert und von dort geholt.

Bild 3.4 zeigt einen MIMD-Rechner mit verteiltem Speicher, bei dem jedes Speichermodul einem bestimmten Prozessorelement physikalisch zugeordnet ist, wobei jedes PE nur auf seinen lokalen Speicher Zugriff hat (*No-Remote Memory Access*, *NORMA*). Hier dient das Verbindungsnetz dazu, Nachrichten zwischen den einzelnen Knoten auszutauschen. Ein Knoten besteht dabei jeweils aus einem PE, seinem lokalen Speicher und einer Schnittstelle zur Anbindung an das Verbindungsnetz.

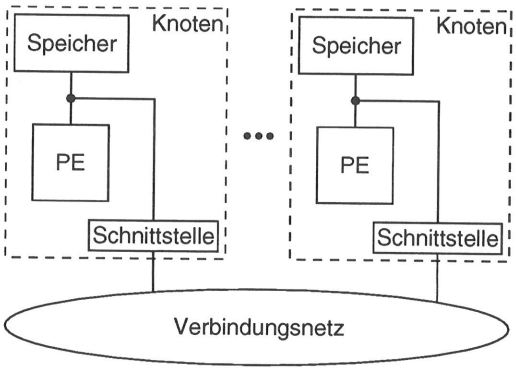

Bild 3.4: MIMD-Rechner mit *NORMA*-Architektur (nach [Waldschmidt,1995])

3.1.3.3 Programmiermodelle

3.1.3.3.1 Globaler Adreßraum

Steht allen Prozessorelementen ein globaler Adreßraum zur Verfügung, erleichtert dies die Parallelisierung bestehender sequentieller Programme erheblich. Zur Vermeidung von Inkonsistenzen muß allerdings der gleichzeitige Zugriff mehrerer PEs auf denselben Speicherbereich verhindert werden, wozu sich z. B. Semaphore [Dijkstra,1965] oder Monitore [Hoare,1974] einsetzen lassen.

Handelt es sich um einen Rechner mit physikalisch gemeinsamem Speicher, dann ist es einfach und folgerichtig, einen globalen Adreßraum bereitzustellen. Um dem Programmierer aber auch auf Rechnern mit verteiltem Speicher die Vorteile eines gemeinsamen Speicherbereichs bieten zu können, stellt das Konzept des *Virtual Shared Memory* einen globalen Adreßraum auf einer solchen Maschine zur Verfügung. Problematisch bei diesem Konzept ist, daß es zu einem ähnlichen Programmierstil wie bei Rechnern mit gemeinsamem Speicher verleiten und z. B. in der häufigen Benutzung globaler Variablen münden kann. Da der gemeinsame Speicher hier aber nur vorgetäuscht ist, kann dies z. T. erhebliche Leistungseinbußen zur Folge haben.

3.1.3.3.2 Message-Passing

Ein anderes Programmiermodell ist das *Message-Passing*-Modell, d. h. der explizite Austausch von Nachrichten. Dies ist das auf MIMD-Rechnern mit verteiltem Speicher am häufigsten anzutreffende Modell. Hier existieren nur asynchron laufende Prozesse auf den einzelnen PEs, die bei Bedarf explizit, d. h. vom Programmierer initiiert, mittels Nachrichten miteinander in Kontakt treten. Dieses Konzept ist auch deswegen weit verbreitet, da es prinzipiell auf jeder MIMD-Rechnerarchitektur einfach zur Verfügung zu stellen ist und parallele Programme, die es benutzen, theoretisch portabel gehalten werden können. Es läßt sich nicht zuletzt auch in heterogenen verteilten Systemen einsetzen.

In einem Message-Passing-System müssen im Prinzip nur die beiden Operationen „Senden“ und „Empfangen“ von Nachrichten zu bzw. von bestimmten Prozessen implementiert sein. Vorhanden sind aber meist auch Möglichkeiten für das Rundsenden (*Broadcast*) und Mehrfachsenden (*Multicast*) sowie u. U. weitere Operationen wie z. B. für das Starten neuer Prozesse oder zur globalen Synchronisation. Daneben gibt es oftmals eine Vielzahl von Varianten der Sende- und Empfangsoperationen. Diese können z. B. blockierend sein oder nicht, die Übertragung kann zuverlässig sein oder nicht oder eine Reihenfolgesicherung ist vorhanden oder nicht (siehe z. B. [Rothermel,1993]).

Message-Passing wird heutzutage hauptsächlich durch Bibliotheken realisiert, vereinzelt auch durch Erweiterungen von Programmiersprachen. Das Vorhandensein dieser Bibliotheken bzw. entsprechender Compiler auf der gewünschten Zielplattform ist Voraussetzung für die Portabilität der auf dem Message-Passing-Konzept aufbauenden Programme. Im folgenden werden kurz einige Systeme, die das Message-Passing-Modell auf Parallelrechnern und verteilten Systemen realisieren, samt ihren wichtigsten Eigenschaften vorgestellt.

PVM Das wohl am weitesten verbreitete und damit einen Quasi-Standard darstellende System ist *Parallel Virtual Machine* oder kurz *PVM*[Geist et al.,1994]. Es ist auf einer großen Anzahl verschiedener Plattformen verfügbar – sowohl auf allen gängigen Workstations als auch vielen Parallelrechnern mit gemeinsamem oder verteiltem Speicher. Mit *PVM* lassen sich unterschiedliche Einzelrechner zu einem Rechner-Pool und damit zu einem „virtuellen Parallelrechner" zusammenschließen, wobei dessen Konfiguration dynamisch geändert werden kann. Die innerhalb dieses virtuellen Parallelrechners laufenden Anwendungsprozesse können beliebig miteinander Nachrichten austauschen. Hierzu läuft z. B. auf verteilten Workstations mit *UNIX*-Betriebssystem auf jedem Rechner ein Hintergrundprozeß (*Daemon*), der für die Zustellung der Nachrichten lokaler Anwendungsprozesse verantwortlich ist. Auf Parallelrechnern werden die PVM-Aufrufe dagegen meist direkt in die Aufrufe des jeweiligen lokalen Message-Passing-Systems umgesetzt. *PVM* erlaubt das Versenden von aus einfachen Datentypen bestehenden Nachrichten in heterogenen Umgebungen, wozu eine einheitliche Codierung benutzt werden kann. Neben Mechanismen für den Nachrichtenaustausch stellt *PVM* außerdem Funktionen für das dynamische Starten neuer Prozesse bereit.

P4 *P4* [Butler&Lusk,1993] ist *PVM* sehr ähnlich. Auch *P4* steht auf einer großen Anzahl verschiedener Workstations und Parallelrechner zur Verfügung und erlaubt die Ausführung einer parallelen Anwendung in einer heterogenen Umgebung. Hauptunterschiede sind, daß *P4* keine Hintergrundprozesse für die Kommunikation benutzt und auf einem Master-Slave-Modell basiert. Das Starten der einzelnen Prozesse geschieht in *P4* statisch über eine Konfigurationsdatei beim Start einer neuen Anwendung.

MPI Bei *MPI* (*Message Passing Interface*) [MPI-Draft,1995] handelt es sich um eine internationale Initiative zur Standardisierung von Funktionen einer Message-Passing-Bibliothek, ausgehend von Entwicklern und Anwendern paralleler Soft- und Hardware, die sich im *MPI Forum* zusammengefunden haben. Hauptaugenmerk liegt auf der einfachen Portabilität der mit *MPI* geschriebenen Anwendungen auf eine möglichst große Anzahl von Rechnerplattformen. *MPI* definiert nur die Kommunikationsmechansimen und z. B. keine Funktionen zum Starten neuer Prozesse. Es wird derzeit allerdings über eine erweiterte Version *MPI 2* mit entsprechenden Ergänzungen diskutiert.

NX Das *NX*-System [Intel,1995] wurde von der Firma *Intel* speziell für ihre Parallelrechnerserie entwickelt. Es stellt neben synchronen, asynchronen und Interrupt-gesteuerten Sende- und Empfangsbefehlen u. a. auch Mechanismen zum Starten und Steuern einer parallelen Anwendung und zur globalen Synchronisation bereit. Anwendungen haben über eine Bibliothek Zugriff auf diese Funktionen. Obwohl es Implementierungen auf vernetzten Workstations gibt, ist das *NX*-System für *Intel*-Rechner optimiert und daher wenig portabel. Andererseits stellt es dort das leistungsfähigste Message-Passing-System dar, da z. B. die *PVM*-Implementierung für die *Paragon* auf dem *NX*-System aufsetzt.

3.1.4 Intel Paragon

Mit der *Paragon XP/S-5* [Intel,1995] der Firma *Intel* soll nun ein nach dem Message-Passing-Prinzip arbeitender Rechner mit *NORMA*-Architektur herausgegriffen und kurz vorgestellt werden, da eine Maschine dieses Typs für die in Kapitel 7 folgenden Messungen verwandt wurde. Ihre Rechenknoten sind in einem zweidimensionalen Gitter angeordnet und über jeweils vier bidirektionale Kommunikationskanäle miteinander verbunden. Als Betriebssystem dient das *UNIX*-Derivat *OSF/1-AD*.

Die für diese Arbeit verwandte Maschine war mit 73 Compute-, 5 Service- und 7 E/A-Knoten ausgestattet. Die Service-Knoten bearbeiten die interaktiven Anwendungen der Benutzer, während die E/A-Knoten für die Massenspeicher- und die Netzanbindung zuständig sind. Auf den Compute-Knoten werden die parallelen Anwendungen ausgeführt, wobei jede Anwendung auf einer eigenen Partition läuft, in der sie die benötigten Compute-Knoten exklusiv zugeteilt bekommt. Jeder Knoten des benutzten Systems beherbergte zwei Prozessoren des Typs *Intel i860XP* – einen für die Bereitstellung der Rechenleistung (Applikationsprozessor), den anderen zur Abwicklung der Kommunikation (Kommunikationsprozessor). Die Compute-Knoten waren mit jeweils 32 MB Halbleiterspeicher ausgestattet.

3.1.5 Vernetzte Workstation-Rechner

Vernetzte Workstations gehören in die Kategorie Multicomputersysteme. Mehrere solcher verbundenen Rechner können bei Anwendung des Message-Passing-Programmiermodells im Prinzip wie ein Parallelrechner agieren, wobei es allerdings einige Besonderheiten gibt:

- In der Regel handelt es sich um eine Ansammlung von Einzelrechnern, die einer parallelen Anwendung keine einheitliche Umgebung zur Verfügung stellen. Eine solche läßt sich durch zusätzliche Programmpakete, wie z. B. *DCE* (*Distributed Computing Environment*, [OSF™DCE,1992]), *PVM* (*Parallel Virtual Machine*, [Geist et al.,1994]) oder verschiedene *CORBA*-Implementierungen (*Common Object Request Broker Architecture*, [OMG,1995]), realisieren.

- Oftmals sind heterogene Umgebungen von Rechnern unterschiedlicher Leistung und mit unterschiedlichen Betriebssystemen vorhanden. Eine parallele Anwendung muß deswegen auf alle Zielplattformen portiert werden und bei Bedarf die Anpassung an unterschiedliche Datendarstellungen entweder selbst vornehmen oder auf die entsprechende Funktionalität oben erwähnter Programmpakete zurückgreifen.
- Die Kommunikationsnetze, mit deren Hilfe die Workstations verbunden sind, und die Protokolle, mit deren Hilfe die Kommunikation abgewickelt wird, haben normalerweise eine weitaus geringere Bandbreite und höhere Latenzzeit (Übertragungsdauer für „leere“ Nachrichten) als bei einem Parallelrechner.
- Ein weiteres Problem stellt die Ausfallsicherheit der oftmals räumlich verteilten Workstations dar. Bei einer großen Anzahl steigt auch die Wahrscheinlichkeit der Nichtverfügbarkeit einer Maschine oder der Störung der Kommunikationsverbindung zwischen den Maschinen.

Entgegen diesen eher negativen Punkten hat eine Workstation allerdings meistens mehr Speicher zur Verfügung als ein einzelner Knoten eines Parallelrechners. Zudem sind verteilte Workstations deutlich billiger als ein spezieller Parallelrechner und auch häufiger anzutreffen.

3.2 Objektorientierter Entwurf

3.2.1 Einführung

Zusammen mit der Leistungsfähigkeit der Rechner nahm auch die Komplexität von Software immer mehr zu. Waren zu Beginn dieser Entwicklung noch hauptsächlich mathematische Probleme zu lösen, wurden die Aufgaben später immer vielfältiger. Entsprechend entwickelten sich auch die Programmiersprachen. Frühe Programmiersprachen, wie FORTRAN I oder ALGOL 58, boten im wesentlichen Möglichkeiten zur Formulierung von mathematischen Ausdrücken. In späteren Entwicklungen kamen Konzepte für die Modularisierung der Programme und die Abstraktion von Datentypen hinzu. Mit jeder Sprachgeneration hatte es der Programmierer einfacher, sein Problem in einer mehr dem Problem als der Maschine angepaßten Form zu beschreiben.

Um die immer komplexer werdenden Softwaresysteme beherrschen zu können, wurde der objektorientierte Ansatz für den Softwareentwurf entwickelt. Dieser gliedert sich in die drei Phasen *objektorientierte Analyse* (*Object-Oriented Analysis, OOA*), *objektorientierter Entwurf* (*Object-Oriented Design, OOD*) und *objektorientierte Programmierung* (*Object-Oriented Programming, OOP*) [Booch,1994]. Bei der objektorientierten Analyse werden die Anforderungen an das Softwaresystem mit Hilfe der Begriffswelt des zu analysierenden Problems identifiziert und daraus ein Modell entwickelt. In der anschließenden Phase des

objektorientierten Entwurfs wird mittels einer objektorientierten Dekomposition und unter Zuhilfenahme einer Notation für die statischen und dynamischen sowie logischen und physikalischen Aspekte eine Lösung erarbeitet. Diese wird schließlich während der Programmierphase in einer objektorientierten Programmiersprache implementiert und getestet.

Wann gilt aber eine Programmiersprache als objektorientiert? In [Cardelli&Wegner,1985] werden hierfür die folgenden drei Anforderungen definiert:

- Sie unterstützt Objekte, die Datenabstraktionen mit einer Schnittstelle aus mit Namen versehenen Operationen und einem verborgenen lokalen Zustand sind.
- Objekte haben einen zugeordneten Objekttyp [Klasse].
- Typen [Klassen] können Eigenschaften von Obertypen [-klassen] erben.

Fehlt die Möglichkeit der Vererbung, wird von einer *objektbasierten* Sprache gesprochen (siehe [Wegner,1990] und [Wegner,1992]). Die obige Definition einer objektorientierten Programmiersprache kann auch auf den objektorientierten Entwurf im allgemeinen ausgedehnt werden. Im Laufe der Zeit wurden verschiedene Ansätze für den objektorientierten Entwurf entwickelt, [Fichman&Kemerer,1992] und [Stein,1993] bieten dazu gute Übersichten. In dieser Arbeit wird die Methode nach [Booch,1994] verwandt, die ein *Objektmodell* als Rahmen für den objektorientierten Entwurf samt einer dazugehörigen Notation zur Verfügung stellt. Ihre Grundzüge werden im folgenden erläutert.

3.2.2 Elemente des Objektmodells

Das Objektmodell nach [Booch,1994] besteht aus den vier Hauptelementen:

- Abstraktion,
- Kapselung,
- Modularität,
- Hierarchie.

Danach ist *Abstraktion* definiert als „die grundlegenden Eigenschaften eines Objekts, die es von allen anderen Objekten unterscheiden und dadurch eine klare konzeptionelle Abgrenzung aus Sicht des Betrachters bieten". Abstraktion ermöglicht die Handhabung von Komplexität, indem sie die aus Sicht des Betrachters wichtigen Dinge heraushebt und unwichtige beiseite läßt. Daraus folgt, daß es unterschiedliche Abstraktionen desselben Objekts in Abhängigkeit dieser Sichtweise geben kann. [Booch,1994] fügt hier noch das *Prinzip des geringsten Erstaunens* hinzu, durch das „eine Abstraktion das gesamte Verhalten eines Objekts umfaßt, nicht mehr und nicht weniger, und keine Überraschungen oder Seiteneffekte bietet, die über den Umfang der Abstraktion hinausgehen". Objekte können dabei das Ergebnis von vier Arten

von Abstraktion sein: Sie können ein sinnvolles Modell eines Elements aus dem Problem- oder Lösungsbereich repräsentieren (*Entity Abstraction*), einen allgemeinen Satz gleichartiger Funktionen zur Verfügung stellen (*Action Abstraction*), Funktionen zusammenfassen, die alle von einer übergeordneten Instanz benötigt werden oder die alle einen untergeordneten Satz von Funktionen benutzen (*Virtual Machine Abstraction*) oder einen Satz von Funktionen ohne Beziehung zueinander „verpacken" (*Coincidental Abstraction*). Das Verhalten eines Objekts läßt sich einerseits charakterisieren durch die Dienste, die es nach außen hin zur Verfügung stellt, andererseits durch die Dienste, die es von anderen Objekten in Anspruch nimmt. Um festzulegen, wie ein Objekt einen Dienst unter welchen Bedingungen zu erbringen hat, führt [Meyer,1992] den *Entwurf durch Vertrag* (*Design by Contract*) ein. Dazu werden für jede Operation eines Objekts Vor- und Nachbedingungen festgelegt, die der Klient (der die jeweilige Operation benutzt) erfüllen muß bzw. auf die er sich verlassen kann.

Wie ein Objekt eine bestimmte Operation tatsächlich implementiert, bleibt nach außen hin unsichtbar – die Implementierung ist *gekapselt*. *Kapselung* bedeutet, daß die gesamte innere Struktur eines Objekts nur diesem selbst zugänglich ist (*Information Hiding*). Dies führt dazu, daß ein Objekt aus zwei Teilen besteht: einer *Schnittstelle*, die ein klar definiertes Verhalten für die Außenwelt festlegt und einer verborgenen *Implementierung*.

Ein weiteres Mittel, um die Komplexität eines Systems zu reduzieren, besteht in seiner Aufspaltung in einzelne, möglichst lose gekoppelte *Module*, die jeweils für einzelne Teilaufgaben zuständig sind. Was Kapselung auf Objektebene ist, stellt *Modularität* auf Systemebene dar: Auch Module besitzen eine klar definierte Schnittstelle nach außen hin und ihre Implementierung bleibt verborgen.

Da es in einem komplexen System sehr viele Abstraktionen gibt, ist für ein einfaches Verständnis deren hierarchische Anordnung notwendig. Die beiden zentralen *Hierarchien* in einem objektorientierten Entwurf sind dabei die Klassen- und die Objekthierarchie (s. u.).

3.2.3 Objekte

3.2.3.1 Definition

In den letzten Abschnitten wurde häufig der Ausdruck „Objekt" gebraucht. Was ist darunter aber genau zu verstehen? [Booch,1994] gibt die folgende Definition:

Definition 3.2 (Objekt)
Ein *Objekt* hat einen Zustand, ein Verhalten und eine Identität; die Struktur und das Verhalten gleichartiger Objekte sind in ihrer gemeinsamen *Klasse* definiert; die Ausdrücke *Instanz* und *Objekt* sind gegeneinander austauschbar.

Der *Zustand* umfaßt dabei alle statischen Eigenschaften eines Objekts zusammen mit deren aktuellen dynamischen Werten. Konkret wird der Zustand durch die internen Datenstrukturen (*Felder*) eines Objekts und deren meistens dynamische Werte festgelegt. Er ist letztlich das Resultat des gesamten zurückliegenden *Verhaltens* des Objekts, welches wiederum eine Funktion des Zustands und der ausgeführten Operationen ist. Diese Operationen sind dabei Aktionen, die Objekte auf anderen Objekten ausführen, um sie zu einer Reaktion zu veranlassen. Man spricht auch davon, daß Meldungen oder Nachrichten zwischen den Objekten ausgetauscht bzw. daß Methoden eines Objekts aufgerufen werden. Die Identität eines Objekts schließlich ist die Eigenschaft, die es von allen anderen Objekten unterscheidet.

3.2.3.2 Beziehungen zwischen Objekten

Erst das Zusammenspiel vieler verschiedener Objekte, die sich gegenseitig mit Diensten beauftragen und diese anderen gegenüber erbringen, ergibt in einem objektorientierten Entwurf ein funktionsfähiges Gesamtsystem. Nach [Booch,1994] können zwei Objekte über eine Verbindung (*Link*) miteinander in Beziehung stehen. Sie können in dieser Beziehung entweder ausschließlich selbst handeln (*Actor*), ausschließlich Dienste erbringen (*Server*) oder beide Rollen übernehmen (*Agent*). In einer Verfeinerung wird weiter unterschieden, auf welche Weise der Diensterbringer für den Klienten sichtbar ist (als globales Objekt, als Parameter des Aufrufs einer Methode des Klienten, als Teil des Klienten oder als lokales Objekt einer Methode des Klienten) und wie der Zugriff auf den Diensterbringer synchronisiert wird (die Funktionalität des Diensterbringers ist garantiert, sofern nur ein Kontrollfluß exisitert (*sequential*), sofern mehrere Kontrollflüsse existieren, wobei die Klienten kooperieren und sich an bestimmte Zugriffsprotokolle halten müssen (*guarded*) oder sofern mehrere Kontrollflüsse existieren, wobei der Diensterbringer den gegenseitigen Ausschluß garantiert (*synchronous*)).

Daneben können Objekte auch eine Eltern-Kind-Beziehung eingehen (*Aggregation* oder *Containment*), d. h. ein Objekt kann ein anderes enthalten, wobei „enthalten“ hier nicht unbedingt physikalisch sein muß, sondern auch einen logischen Zusammenhang darstellen kann. Diese Beziehung bildet eine der beiden in Abschnitt 3.2.2 erwähnten Hierarchien beim objektorientierten Entwurf, die *Objekthierarchie*. Eine solche Beziehung läßt sich auch charakterisieren mit den Ausdrücken „besteht aus“ oder „ist ein Teil von“ ([Booch,1994] nennt dies eine *„part of“ hierarchy*).

3.2.4 Klassen

3.2.4.1 Definition

Schon in der Definition des *Objekts* kam der Begriff der *Klasse* vor, der nach [Booch,1994] wie folgt definiert ist:

Definition 3.3 (Klasse)
Eine *Klasse* repräsentiert einen Satz von Objekten, die eine gemeinsame Struktur und ein gemeinsames Verhalten aufweisen.

Eine Klasse legt also die gemeinsamen Eigenschaften gleichartiger Objekte fest. Man sagt auch, daß ein Objekt eine *Instanz* einer Klasse ist. Als Beispiel möge die Verwaltung von Daten an einer Schule dienen. Dort gibt es die Klasse *TSchüler*[2], die verschiedene Datenfelder für Name, Vorname und Noten in einzelnen Fächern enthält, sowie die Methode *Notendurchschnitt()*, die eben diesen berechnet. Für jeden Schüler der Schule wird nun eine Instanz dieser Klasse erzeugt, wobei die Datenfelder jedes dieser Objekte unterschiedlichen Inhalt haben und die Methode *Notendurchschnitt()* in Abhängigkeit vom Objekt, auf das sie angewandt wird, unterschiedliche Ergebnisse liefert.

3.2.4.2 Schnittstelle und Implementierung

Wie schon erwähnt, läßt sich eine Klasse in die beiden Teile Schnittstelle und Implementierung aufspalten. Erstere kann nach dem Grad der Zugänglichkeit weiter unterteilt werden in die drei Kategorien *öffentlich* (*public*), d. h. allgemein zugänglich, *geschützt* (*protected*), d. h. nur der Klasse selbst und davon abgeleiteten Klassen zugänglich, sowie *privat* (*private*), d. h. nur der Klasse selbst zugänglich.

3.2.4.3 Beziehungen zwischen Klassen

3.2.4.3.1 Assoziation

Die einfachste Beziehung zwischen Klassen ist die *Assoziation*. Sie sagt im wesentlichen nur aus, daß zwischen den entsprechenden Klassen eine semantische Abhängigkeit besteht. Über die Art dieser Abhängigkeit wird nichts ausgesagt (außer durch eventuell angefügte Kommentare), lediglich u. U. über die Anzahl der an der Beziehung beteiligten Instanzen (*Kardinalität*).

Bild 3.5: Assoziation

[2]Im weiteren Verlauf der Arbeit beginnen Klassennamen mit dem Buchstaben *T* (für „Typ"). Diese Konvention erleichtert die Lesbarkeit der Diagramme und Programme und damit deren Verständnis.

Bild 3.5 zeigt mit Hilfe der in [Booch,1994] eingeführten Notation, daß jede Instanz der Klasse *TFirma* Beziehungen zu n Instanzen der Klasse *TMitarbeiter* unterhält, eine Assoziation, durch die entsprechende Beschäftigungsverhältnisse modelliert werden. Assoziationen werden häufig zu Beginn des Entwurfsprozesses eingesetzt und in späteren Phasen durch verfeinerte Beziehungen ersetzt.

3.2.4.3.2 Vererbung

Einfache und mehrfache Vererbung Die wichtigste Beziehung beim objektorientierten Entwurf ist die *Vererbung*. Es gibt oft Klassen, die ähnliche Eigenschaften aufweisen und sich nur in Teilaspekten unterscheiden. Um nicht alle Datenfelder und Methoden mehrfach definieren zu müssen, ist es sinnvoll, diese Klassen hierarchisch anzuordnen. Gemeinsame Eigenschaften werden dabei in *Basis-* oder *Oberklassen* definiert, von denen spezialisierte Klassen *abgeleitet* werden. Datenfelder und Methoden der Basisklasse werden an die abgeleitete Klasse *vererbt*, welche zusätzliche hinzufügen oder vererbte Methoden verändern kann. Besitzt eine abgeleitete Klasse genau eine direkte Oberklasse, spricht man von *einfacher Vererbung* (*Single Inheritance*), ist sie von mehreren Oberklassen abgeleitet von *mehrfacher Vererbung* (*Multiple Inheritance*). Die Bilder 3.6 und 3.7 zeigen entsprechende Beispiele. Über Vererbungsbeziehungen entsteht die zweite wichtige Hierarchie in einem objektorientierten Entwurf, die sog. *Klassenhierarchie* (s. a. Abschnitt 3.2.2). Bei der Beziehung zwischen Ober- und Unterklasse handelt es sich um eine „ist ein"-Beziehung, d. h. für die Bilder 3.6 und 3.7: Ein Auto „ist ein" Verkehrsmittel und ein Rennwagen „ist ein" Auto und ein Sportgerät ([Booch,1994] nennt dies eine *„is a" hierarchy*).

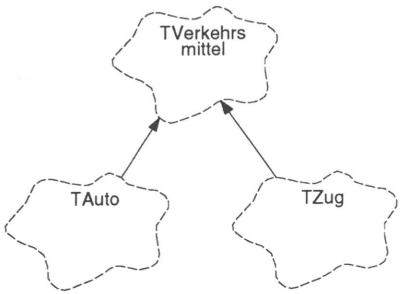

Bild 3.6: Einfache Vererbung

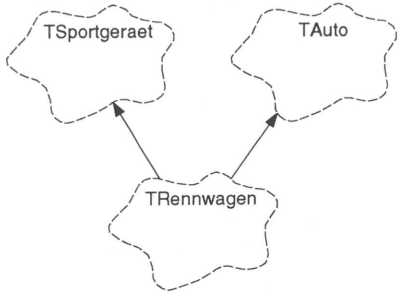

Bild 3.7: Mehrfache Vererbung

Polymorphie Es wurde schon erwähnt, daß abgeleitete Klassen Methoden, die sie von ihren Basisklassen erben, verändern können. Welche Methode welcher Klasse der Ableitungshierarchie bei einem Aufruf ausgeführt wird, kann nun dynamisch zur Laufzeit bestimmt

werden (*Dynamisches Binden*). Um beim obigen Beispiel zu bleiben, wäre eine Liste von Verkehrsmitteln denkbar, die aus Objekten des Typs *TAuto*, *TZug* und evtl. weiterer von *TVerkehrsmittel* abgeleiteter Klassen besteht. Es gäbe eine Methode *Fahrpreis*, die diesen aufgrund einer vorzugebenden Fahrstrecke ermittelt. Obwohl diese Methode für jedes Objekt der Liste für die Basisklasse *TVerkehrsmittel* aufgerufen wird, wird die entsprechende Methode der abgeleiteten Klasse ausgeführt. Ein solches Verändern einer Methode durch eine abgeleitete Kalsse wird *Überschreiben* (*Overriding*) einer Methode genannt[3], in *C++* heißen diese Methoden *virtuell* (*virtual*)[4]. Hinter einem Objekt, von dem nur bekannt ist, daß es vom Typ der Basisklasse ist, können sich also Instanzen verschiedener abgeleiteter Klassen verbergen. Diese Eigenschaft heißt *Polymorphie* (Vielgestaltigkeit).

Abstrakte Basisklassen Basisklassen fassen gemeinsame Eigenschaften ihrer Unterklassen zusammen. Oftmals sind sie aber noch unvollständig und müssen erst durch abgeleitete Klassen komplettiert und präzisiert werden. Es macht deshalb auch wenig Sinn, Objekte vom Typ dieser Basisklassen zu generieren. Solche Basisklassen, von denen keine Instanzen erzeugt werden dürfen, werden *abstrakte Basisklassen* genannt. Methoden, die in diesen Klassen zwar deklariert aber nicht implementiert werden, heißen *abstrakte Methoden* oder im *C++*-Sprachgebrauch *reine virtuelle* Methoden (*pure virtual*)[5].

Virtuelle Basisklassen In der Vererbungshierarchie einer Unterklasse kann eine bestimmte Basisklasse auch über verschiedene Ableitungspfade erreichbar sein. Eine solche Basisklasse wird *virtuelle Basisklasse* genannt. In Bild 3.8 ist ein Beispiel hierfür gezeigt: Ein Taxi ist sowohl ein Auto als auch ein öffentliches Verkehrsmittel. Diese sind aber wiederum beide Verkehrsmittel. Wäre *TVerkehrsmittel* nicht virtuell, so käme diese Basisklasse zweimal in der Vererbungshierarchie von *TTaxi* vor, und entsprechend hätte jede Instanz von *TTaxi* alle Datenfelder von *TVerkehrsmittel* doppelt. Im Normalfall ist dies unerwünscht, da es zu Mehrdeutigkeiten und unnötigem Speicherplatzverbrauch führt.

Mixin-Klassen Eine *Mixin-Klasse* (*Mixin Class*, [Booch,1994] und [Gamma et al.,1994]) fügt einer bestehenden Klasse weitere Eigenschaften hinzu. In Bild 3.7 kann *TSportgeraet* z. B. als Mixin-Klasse aufgefaßt werden, da sie einem *TAuto* die Eigenschaften eines Sportgeräts hinzufügt und daraus einen Rennwagen macht. Da es keine Instanzen einer

[3]Nicht zu verwechseln mit dem *Überladen* (*Overloading*) einer Methode, bei dem mehrere Methoden gleichen Namens aufgrund der Anzahl und des Typs ihrer Argumente (in manchen Sprachen auch des Typs des Rückgabewerts) unterschieden werden.

[4]Virtuelle Methoden werden im folgenden durch Voranstellen des *C++*-Schlüsselworts *virtual* gekennzeichnet, also z. B. *virtual AnyMethod()*.

[5]Abstrakte Methoden werden im folgenden, wie in *C++*, als zu Null initialisiert gekennzeichnet, also z. B. *virtual AnyMethod() = 0*.

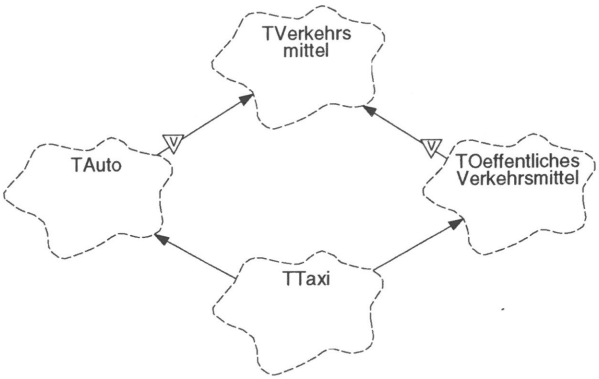

Bild 3.8: Virtuelle Basisklasse

Mixin-Klasse selbst geben sollte, handelt es sich dabei in der Regel um abstrakte Klassen. Mixin-Klassen erfordern mehrfache Vererbung.

3.2.4.3.3 Enthaltensein

Die *Enthaltensein*-Beziehung bei Klassen (*Aggregation* oder *Containment*) entspricht ganz der *Enthaltensein*-Beziehung bei Objekten. Sie sagt letztlich nichts anderes aus, als daß die Instanzen der entsprechenden Klassen die gleichen Beziehungen unterhalten. Bild 3.9 bedeutet z. B., daß jede Instanz der Klasse *TAuto* eine Instanz der Klasse *TGetriebe* enthält oder konkret, daß *TAuto* ein Feld namens *fGetriebe*[6] des Typs *TGetriebe* besitzt. Letzteres kann entweder physikalisch enthalten sein (*has by value*), d. h. es kann nicht ohne sein Elternobjekt existieren, oder logisch durch eine Referenz (*has by reference*), d. h. die Lebensdauern beider Objekte können unterschiedlich sein. Insbesondere kann ein Objekt in mehreren anderen Objekten durch Referenz enthalten sein, aber nur in maximal einem physikalisch.

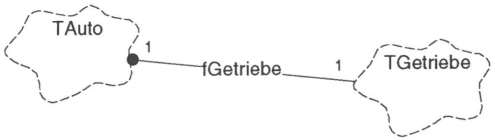

Bild 3.9: Enthaltensein

[6] Alle Bezeichnungen von Datenfeldern einer Klasse beginnen im folgenden nach Konvention mit einem *f*.

3.2.4.3.4 Benutzen

Die *Benutzen*-Beziehung (*Using*) zwischen zwei Klassen (Bild 3.10) spiegelt ein Verhältnis eines Klienten zu einem Diensterbringer wider. Auch diese Beziehung findet ihre Entsprechung in einer Verbindung der Instanzen dieser Klassen (Unterabschnitt 3.2.3.2).

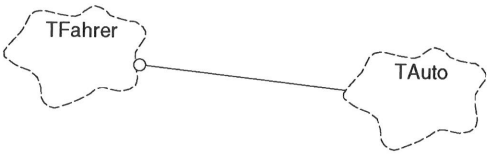

Bild 3.10: Benutzen

3.2.4.3.5 Instanziieren

Parametrisierte Klassen (auch *Generische Klassen* genannt) bieten die Möglichkeit, als *Schablonen* (*Templates*) für andere Klassen zu dienen. In Bild 3.11 ist das Beispiel einer generischen Liste gezeigt. Das Verhalten einer solchen Liste ist prinzipiell unabhängig vom Typ der darin enthaltenen Elemente. Andererseits soll sichergestellt werden, daß in einer Liste jeweils nur Elemente eines Typs vorkommen. Eine Lösung besteht darin, die Funktionalität einer Liste durch *TListe* bereitzustellen, vorerst aber offenzulassen, welchen Typ ein Listenelement *Element* hat. Diese Schablone kann dann dazu benutzt werden, Klassen für verschiedene Listen zu *instanziieren*. Im Bild ist dies für eine Liste von Zügen (z. B. für einen Fahrplan) gezeigt. Aus diesen instanziierten Klassen können dann wieder Objekte erzeugt

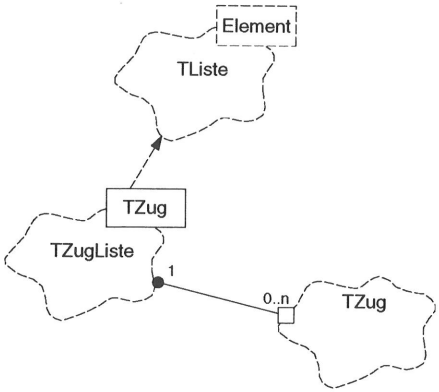

Bild 3.11: Instanziieren

werden. Mittels parametrisierter Klassen lassen sich sehr kompakte und flexible Bibliotheken häufig benötigter Datenstrukturen erstellen (siehe z. B. [Booch&Vilot,1993]).

3.2.5 Entwurfsmuster

3.2.5.1 Motivation

Die Hauptaufgabe beim objektorientierten Entwurf besteht darin, geeignete Abstraktionen zu finden und ein Netz von Klassen und Objekten zu entwerfen, die im Zusammenspiel die gestellte Aufgabe bewältigen. Obwohl dies für jedes System individuell erfolgen muß, treten dabei häufig ähnliche Problemstellungen auf. *Softwareentwurfsmuster* (*Design Patterns*) [Gamma et al.,1994] stellen für deren Lösung bestimmte Grundmuster zur Verfügung, denen die konkrete Implementierung in eigenen Entwürfen folgen kann. Sie wurden beeinflußt durch die Entwicklung von Entwurfsmustern auf dem Gebiet der Architektur [Alexander et al.,1977], die ganz entsprechend Lösungen für architektonische Grundprobleme der Gebäude- und Städteplanung bereitstellen. Softwareentwurfsmuster sind damit auf einer höheren Abstraktionsebene als die der Klassen ein Ansatz zur Wiederverwendbarkeit von objektorientierten Entwürfen. Es wurden in der Zwischenzeit eine ganze Reihe solcher Entwurfsmuster allgemeiner Natur oder spezieller Anwendungsbereiche extrahiert (z. B. [Schmidt,1995]), von denen einige grundlegende im folgenden kurz vorgestellt werden.

3.2.5.2 Handle

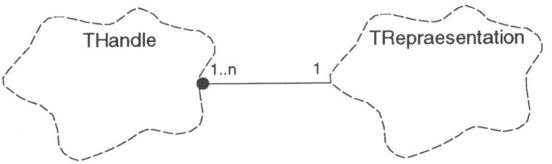

Bild 3.12: Handle

Über eine *Handle*-Klasse [Stroustrup,1991], auch *Bridge Pattern* [Gamma et al.,1994] genannt, läßt sich der Zugriff auf ein Objekt von dessen eigentlicher Repräsentation trennen (Bild 3.12). Verschiedene Zugriffsobjekte (*Handles*) können sich dasselbe Repräsentationsobjekt teilen und verhindern so einerseits unnötiges Replizieren redundanter Daten, andererseits erlauben sie verschiedene Sichtweisen auf ein und dasselbe Objekt. Insbesondere wird dadurch eine Speicherverwaltung durch Zählung der Zugriffsreferenzen möglich [Coplien,1992]. Dafür erhält die Repräsentation einen Zähler für die Anzahl der sie referenzierenden Zugriffsobjekte und wird automatisch gelöscht, sobald das letzte Zugriffsobjekt

gelöscht wird. Weitere Anwendungsgebiete von Handle-Klassen sind das dynamische Binden einer Abstraktion (Handle) an eine Implementation (Repräsentation) zur Laufzeit oder die Möglichkeit, eine Implementierung ändern zu können, ohne den Code eines den Handle benutzenden Klienten neu zu übersetzen.

3.2.5.3 Adapter

Adapter dienen dazu, die Schnittstelle einer vorhandenen Klasse an die Bedürfnisse einer Umgebung anzupassen. *Adapter* (auch *Wrapper* [Gamma et al.,1994] oder, speziell für *C++*, *Schnittstellen-Klasse* [Stroustrup,1991] genannt) lassen sich dabei entweder mittels mehrfacher Vererbung (Bild 3.13) oder Enthaltensein (Bild 3.14) realisieren. In beiden Beispielen erwartet der Klient ein Objekt vom Typ *TAngepasst*, vorhanden ist aber ein Objekt vom Typ *TOriginal* mit der gewünschten Funktionalität aber unterschiedlicher Schnittstelle. In beiden Fällen wird der Aufruf der Methode *Aufruf()* durch den Adapter in einen Aufruf von *OriginalAufruf()* der Klasse *TOriginal* umgesetzt. Dabei übernimmt der Adapter die Anpassung der Parameter und Rückgabewerte der umgesetzten Methode.

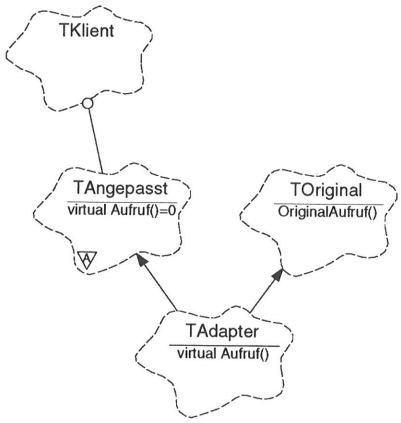

Bild 3.13: Adapter mittels Vererbung (nach [Gamma et al.,1994])

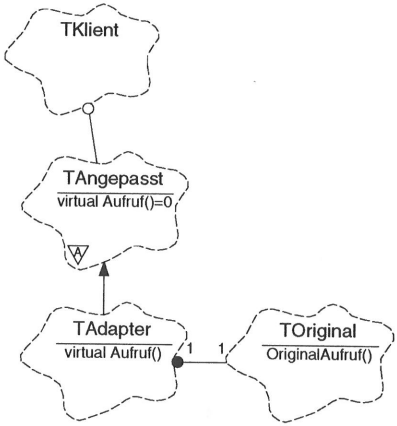

Bild 3.14: Adapter mittels Enthaltensein (nach [Gamma et al.,1994])

3.2.5.4 Prototypen

Prototypen [Gamma et al.,1994] ermöglichen es, den Typ eines zu erzeugenden Objekts zur Laufzeit dynamisch zu bestimmen. Bild 3.15 zeigt das Grundprinzip: *TKlient* erzeugt Objekte einer von *TPrototyp* abgeleiteten Klasse. Um welche Klasse es sich tatsächlich handelt,

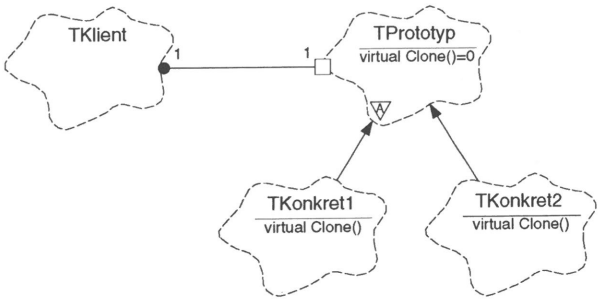

Bild 3.15: Prototyp (nach [Gamma et al.,1994])

wird zur Laufzeit festgelegt. Dazu hält *TKlient* eine Referenz auf ein „Musterobjekt", das von dem Typ ist, der tatsächlich erzeugt werden soll. *TPrototyp* enthält eine abstrakte Methode *Clone()*, die von den abgeleiteten Klassen implementiert wird und die ein neues Objekt des jeweiligen Typs erzeugt. Soll ein neues Objekt erzeugt werden, so ruft *TKlient* die *Clone()*-Methode des Musterobjekts auf.

3.2.5.5 Kette der Verantwortlichkeit

Sollen mehrere Objekte in der Lage sein, eine Anfrage zu bearbeiten, und soll die Menge dieser Objekte u. U. auch zur Laufzeit veränderbar sein, empfiehlt sich das Muster der *Kette der Verantwortlichkeit* (*Chain of Responsibility*) [Gamma et al.,1994]. In Bild 3.16 sind die von der abstrakten Basisklasse *THandler* abgeleiteten Handler („Bearbeiter") in einer verketteten Liste angeordnet. *THandler* enthält die abstrakte Methode *Bearbeite()*, die in den abgeleiteten Klassen implementiert wird. Nach dem Aufruf von *Bearbeite()* des ersten Handlers durch den Klienten bearbeitet dieser die Anfrage falls erforderlich und reicht sie gegebenenfalls an einen in der Liste vorhandenen Nachfolger weiter. Dadurch muß der Klient das Objekt (oder die Objekte), das seine Anfrage letztlich bearbeitet, überhaupt nicht kennen.

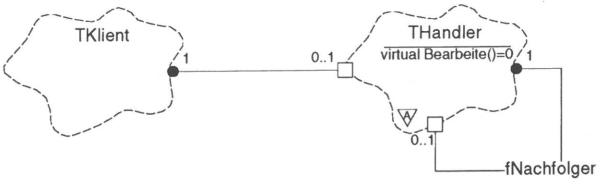

Bild 3.16: Kette der Verantwortlichkeit (nach [Gamma et al.,1994])

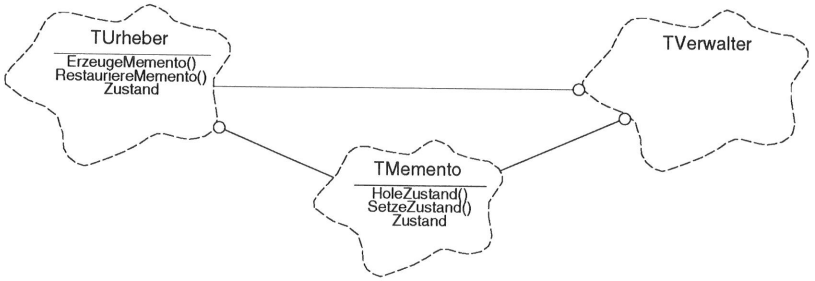

Bild 3.17: Memento (nach [Gamma et al.,1994])

3.2.5.6 Memento

Zur Sicherung des Zustands eines Objekts (oder eines Teils davon), ohne daß dabei dessen Kapselung verletzt wird, dient das Muster des *Memento* (Erinnerung) [Gamma et al.,1994]. Die in Bild 3.17 gezeigte Klasse *TUrheber* muß durch *Zustand* angedeutete interne Daten sichern (z. B. verschiedene Felder einfacher Datentypen). *TVerwalter* fordert *TUrheber* durch Aufruf von *ErzeugeMemento()* zur Sicherung auf, worauf letzterer zuerst ein neues Memento-Objekt erzeugt. Anschließend sichert *TUrheber* seinen Zustand durch Aufruf der *SetzeZustand()*-Methode von *TMemento* und reicht dieses Objekt, das nunmehr seinen aktuellen Zustand enthält, an den Verwalter zurück. Stellt der Verwalter später fest, daß der Zustand wiederhergestellt werden muß, übergibt er das entsprechende Memento-Objekt an den Urheber durch Aufruf von *RestauriereMemento()* und fordert diesen dadurch auf, den Inhalt des Memento-Objekts wiederherzustellen. *TUrheber* erhält dann seinen alten Zustand durch Aufruf der *HoleZustand()*-Methode von *TMemento* zurück. In [Gamma et al.,1994] sind Memento-Objekte rein passiv, d. h. das Sichern und Wiederherstellen von Zuständen wird vom Urheber selbst übernommen. *TMemento* dient lediglich als Datenspeicher mit den beiden Methoden *SetzeZustand()* und *HoleZustand()* für den Datenzugriff.

3.2.5.7 Singleton

In einem Softwareentwurf gibt es oftmals zentrale Funktionen, die von einem globalen Objekt wahrgenommen werden (z. B. die Verwaltung zentraler Ressourcen). Dabei ist es wichtig, daß genau ein solches Objekt existiert und eine wohldefinierte Möglichkeit besteht, darauf zuzugreifen. Das *Singleton*-Entwurfsmuster [Gamma et al.,1994] aus Bild 3.18 stellt genau dies zur Verfügung. *TSingleton* enthält die statische[7] Variable *fInstanz*, die eine Referenz auf

[7]Der Begriff der *statischen* Methoden und Felder ist der Programmiersprache *C++* entnommen und bezeichnet Methoden und Felder, die im Kontext einer Klasse und nicht eines Objekts definiert sind. Sie existieren für alle Instanzen einer Klasse nur einmal. In der Sprache *Smalltalk* z. B. entsprechen ihnen die Klassenmethoden und -variablen.

die einzige globale Instanz der Klasse enthält. Ein Aufruf der statischen Methode *LiefereInstanz()* liefert die Referenz auf diese Instanz als Resultat. Durch eine automatisch bei der Erzeugung durchgeführte Instanzen-Zählung kann *TSingleton* sicherstellen, nur ein einziges Mal instanziiert zu werden.

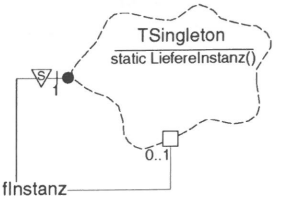

Bild 3.18: Singleton

3.2.5.8 Schablonen-Methode

Ein letztes wichtiges Entwurfsmuster, das hier kurz vorgestellt werden soll, ergibt sich direkt aus der Möglichkeit der Polymorphie (Paragraph 3.2.4.3.2). In [Gamma et al.,1994] wird es *Schablonen-Methode* (*Template Method*) genannt. Es ist vor allem dann nützlich, wenn ein Algorithmus mit verschiedenen Ausprägungen implementiert werden soll. Dazu wird, wie in Bild 3.19 durch die Methode *SchablonenMethode()* der Klasse *TAbstrakteKlasse* angedeutet, das Grundgerüst des Algorithmus in einer abstrakten Basisklasse implementiert. Einzelne veränderliche Teile des Algorithmus werden in virtuelle Methoden (*Operation1()* und *Operation2()* im Beispiel) ausgelagert und erst in abgeleiteten Klassen tatsächlich implementiert. Somit läßt sich die Ausprägung des Algorithmus durch Wahl einer entsprechenden abgeleiteten Klasse sehr einfach wählen, ohne daß der gesamte Algorithmus repliziert werden muß. Natürlich lassen sich durch Ableitung weiterer Klassen auch neue Variationen implementieren und einfach in bestehenden Entwürfen verwenden.

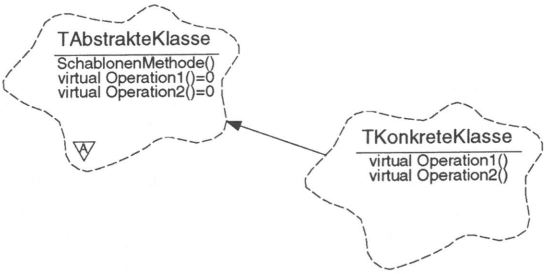

Bild 3.19: Schablonen-Methode (nach [Gamma et al.,1994])

3.2.6 Toolkits

Toolkits stellen nach [Gamma et al.,1994] eine Sammlung nützlicher und miteinander in Beziehung stehender Klassen bereit, die eine allgemein anwendbare Funktionalität erbringen. Beispiele hierfür sind die *Booch-Komponenten* [Booch,1993], die häufig benötigte Strukturen wie Listen, Warteschlangen etc. in Form parametrisierter Klassen beinhalten, oder die *C++ I/O Stream*-Bibliothek [Teale,1993]. Mit der Hilfe von Toolkits muß oft gebrauchte Funktionalität nicht immer wieder neu implementiert werden.

3.2.7 Frameworks

Auf einem höheren Abstraktionsniveau als Entwurfsmuster sind *Frameworks* angesiedelt [Johnson&Foote,1988]. Als Framework bezeichnet man eine Sammlung von Klassen, die einen wiederverwendbaren Entwurf für ein bestimmtes Anwendungsfeld darstellen. Sie stellen ein Gerippe zur Verfügung, das vom Programmierer für seine spezielle Applikation ausgefüllt werden muß. Die gesamte Architektur und Struktur der Anwendung und ihrer Datenstrukturen wird durch das Framework festgelegt. Dieses wird entweder durch Ableitung von abstrakten Basisklassen oder die Konfiguration von internen Komponenten an die Bedürfnisse der Applikation angepaßt. Während Entwurfsmuster Lösungskonzepte für allgemeine Problemstellungen darstellen, die erst noch in angepaßter Form implementiert werden müssen, sind Frameworks schon in einer Programmiersprache codiert und viel stärker spezialisiert. Frameworks übernehmen oftmals auch die Rolle des Hauptprogramms und rufen Methoden abstrakter Basisklassen auf, die vom Programmierer in abgeleiteten Klassen entsprechend der applikationsabhängigen Bedürfnisse implementiert werden. Dies ist ein Hauptunterschied zwischen Toolkits und Frameworks: Bei ersteren schreibt der Programmierer das Hauptprogramm und benutzt dabei Code, den der Toolkit zur Verfügung stellt, bei letzteren stellt der Programmierer den Code zur Verfügung, der vom Framework benutzt wird.

3.3 Die Programmiersprache „C++“

Ein objektorientierter Entwurf ist zunächst unabhängig von der Implementierungssprache. Allerdings unterstützen verschiedene objektorientierte Programmiersprachen einige Konzepte besser als andere, was bei der Wahl berücksichtigt werden muß. Diese Wahl einer geeigneten Sprache hängt zudem von vielen externen Einflüssen ab. Hierzu zählen z. B. die Verfügbarkeit und Leistungsfähigkeit von Compilern und Entwicklungswerkzeugen auf der

Zielplattform, die Verbreitung der Sprache, vorhandene Erfahrung der Entwickler bzw. einfache Erlernbarkeit der Sprache, Integrierbarkeit bestehender Software, Notwendigkeit der Anbindung von externen Geräten (z. B. Datenbanken) und vieles mehr.

C++ ist die am weitesten verbreitete objektorientierte Programmiersprache und auf vielen Parallelrechnern oftmals die einzig verfügbare. Sie wurde Anfang der 80er Jahre von Bjarne Stroustrup an den AT&T Bell Laboratories entworfen und seither kontinuierlich weiterentwickelt. Eine Einführung und umfassende Übersicht findet sich z. B. in [Stroustrup,1991], der aktuelle Stand der derzeitig laufenden Standardisierung in [C++-Draft,1995]. Eine ausführliche Darstellung der Ursprünge und der Entwicklungsgeschichte sowie der zuletzt hinzugekommenen Spracheigenschaften enthält [Stroustrup,1994].

Der Sprachumfang von *C++* ist größtenteils eine Obermenge der Programmiersprache *C* [Kernighan&Ritchie,1988] und fügt letzterer wesentliche objektorientierte Konzepte hinzu. Hierzu gehören u. a. Klassen, (mehrfache) Vererbung, Polymorphie und parametrisierte Klassen (*Templates*). *C++* ist eine hybride Sprache: Sie läßt sich sowohl rein prozedural als „besseres *C*“ verwenden als auch voll objektorientiert. Dadurch bietet sie Anwendern einen einfachen Übergang von der prozeduralen in die objektorientierte Welt und ermöglicht eine einfache Nutzung oftmals in *C* implementierter Systemfunktionen – ein wesentlicher Grund für ihre weite Verbreitung. Da *C++* objektorientierte Programmierung zwar unterstützt aber nicht erzwingt, handelt es sich diesbezüglich aber auch um eine problematische Sprache. Sie erfordert vom Programmierer Selbstdisziplin, damit Konzepte wie z. B. Kapselung oder strenge Typüberprüfung nicht unterlaufen werden.

3.4 Objektorientierte Ansätze in der Parallelverarbeitung

3.4.1 Einführung

Da der objektorientierte Entwurf von Objekten ausgeht, deren Daten gekapselt sind und die untereinander durch den Aufruf von Methoden bzw. den Austausch von Nachrichten miteinander kommunizieren, ist seine Anwendung in der Parallelverarbeitung nur folgerichtig. Ein Konzept für ein Gerüst dieses *Concurrent Object-Oriented Programming* (*COOP*) ist das *Aktoren-Modell* (*Actor-Model*) [Agha,1990], wobei Aktoren in sich geschlossene, unabhängige und gegenseitig aufeinander einwirkende Komponenten eines Rechensystems sind, die mittels asynchronen Nachrichtenaustauschs miteinander kommunizieren.

3.4.2 Programmiersprachen

Ein Weg für die Realisierung einer parallelen objektorientierten Programmierung besteht in der Entwicklung spezieller Programmiersprachen bzw. in der Erweiterung bestehender Sprachen um entsprechende Konstrukte. Eine Auswahl solcher Ansätze soll im folgenden kurz vorgestellt werden.

Charm++ [Kale&Krishnan,1993] ist eine auf *C++* basierende parallele Programmiersprache, die sowohl regelmäßige wie unregelmäßige Berechnungen unterstützt. Parallele Prozesse, sog. *Chares*, kommunizieren über den Austausch von Nachrichtenobjekten oder den Zugriff auf spezielle gemeinsame Datenbereiche miteinander. Gegenüber *C++* gibt es einige zusätzliche Datenstrukturen, wie Chare-Klassen und Messages, deren Struktur sich stark an *C++*-Klassen anlehnt. *Charm++* besteht aus einem Übersetzer, der diese speziellen Strukturen in *C++*-Strukturen übersetzt, und einem Laufzeitsystem.

Ein ähnliches Konzept verfolgt *Mentat* [Grimshaw,1993], wobei hier spezielle *Mentat*-Klassen definiert werden können, deren Methoden ausreichend rechenintensiv sind, damit der Gewinn einer Parallelverarbeitung den Kommunikationsaufwand aufwiegen kann. Jedes aus einer *Mentat*-Klasse erzeugte *Mentat*-Objekt hat seinen eigenen Kontrollfluß, und ein Methodenaufruf blockiert das aufrufende Objekt erst dann, wenn dessen Ergebnis benötigt wird.

pSather [Stoutamire,1995] basiert auf der objektorientierten Sprache *Sather*, die wiederum ursprünglich auf der Sprache *Eiffel* [Meyer,1988] beruhte. *pSather* beinhaltet zusätzlich Threads und Synchronisationsmechanismen und basiert auf einem Shared-Memory-Programmiermodell. Unterstützung für die Ausnutzung von Datenparallelität auf SIMD-Rechnern und die Programmierung von SPMD-Rechnern schließlich bietet *pC++* [Bodin et al.,1993].

3.4.3 Bibliotheken

Eine weitere Möglichkeit zur Unterstützung paralleler Programmierung besteht im Einsatz von Bibliotheken. Einige (z. B. *PARA++* [Coulaud&Dillon,1995] oder *ACE* [Schmidt,1994]) kapseln konventionelle Message-Passing-Systeme (siehe Paragraph 3.1.3.3.2) und Betriebssystemmechanismen durch spezielle objektorientierte Konstrukte. Sie benutzen dazu Adapter (siehe Unterabschnitt 3.2.5.3) für typsichere, objektorientierte Schnittstellen und erreichen dadurch eine bessere Integration dieser Systeme in objektorientierte Entwürfe. Neben der daraus resultierenden einfacheren und sichereren Benutzung kommt die Kapselung des zugrundeliegenden Message-Passing-Mechanismus der Portabilität zugute. *ACE* stellt darüber hinaus noch Konstrukte auf einem höheren Abstraktionsniveau in Form eines Framework bereit (siehe Abschnitt 3.2.7).

Andere Bibliotheken unterstützen umfangreichere objektorientierte Konzepte. *PRESTO* [Bershad,1991] z. B. stellt auf einem Multiprozessorsystem mit gemeinsamem Speicher einen

einheitlichen Adreßraum über mehrere UNIX-Prozesse hinweg zur Verfügung. Es stehen Methoden zur Implementierung mehrerer Kontrollflüsse sowie verschiedene Synchronisationsmöglichkeiten bereit.

3.4.4 CORBA

Die *Object Management Group* (*OMG*), ein Konsortium verschiedener Hard- und Softwarehersteller sowie Softwarentwickler und Endanwender, hat sich zum Ziel gesetzt, eine Architektur für verteilte objektorientierte Applikationen zu definieren. Diese *Object Management Architecture* (*OMA*) legt auf hohem Abstraktionsniveau ein Referenzmodell fest, das die beteiligten Komponenten, Schnittstellen und Protokolle charakterisiert, nicht jedoch genau definiert. Im wesentlichen besteht die Architektur aus den *Object Services*, den *Application Objects* und den *Common Facilities*, die über den *Object Request Broker*, als eine Art Bus, miteinander kommunizieren.

Kernstück der *OMA* ist, wie schon angedeutet, der *Object Request Broker* (*ORB*), dessen Eigenschaften im aktuellen Standard für die *Common Object Request Broker Architecture* (*CORBA*) 2.0 spezifiziert sind [OMG,1995]. Der *ORB* stellt Verbindungen zwischen Objekten im System her, wobei er insbesondere für das Auffinden der Objekte und das Ver-

ORB-Kern	übermittelt Aufrufe an Objekte und liefert eventuelle Rückgabewerte an den Klienten zurück
Interface Definition Language (*IDL*)	Spezifikationssprache für Schnittstellen der Objekte; IDL-Konstrukte werden durch einen Compiler in Konstrukte einer Standard-Programmiersprache (z. B. in *C++*) übersetzt
Interface Repository (*IR*)	persistenter Speicher aller *IDL*-Schnittstellen-Deklarationen
Dynamic Invocation Interface (*DII*)	Möglichkeit zum Ansprechen von Objekten, über deren Schnittstellen erst zur Laufzeit Informationen gewonnen werden können
Object Adapters (*OA*)	Objekte nutzen die Dienste des *ORB* über *Object Adapters*, wodurch verschiedene Objekt-Implementierungen unterstützt werden können (z. B. eigenständige Programme, leichtgewichtige Objekte oder Datenbankanbindungen); es sind vor allem auch verschiedene Schnittstellen zum selben ORB-Kern möglich, ohne diesen ändern zu müssen

Tabelle 3.4: Hauptkomponenten von CORBA

bergen systemspezifischer Aspekte des Objekts, wie z. B. dessen Programmiersprache, Betriebssystem o. ä., verantwortlich ist. Auf diese Weise können Objekte in einem heterogenen verteilten System transparent zusammenarbeiten. *CORBA* besteht dabei aus den fünf in Tabelle 3.4 aufgelisteten Hauptkomponenten (s. a. [Vinoski,1993]).

Weitere Komponenten der *OMA* neben dem *ORB* sind die *Object Services* für das Management von Objekten, die *Domain Interfaces*, die Schnittstellen für bestimmte Anwendungsgebiete anbieten, die *Common Facilities* für das Konfigurationsmanagement und schließlich die *Application Objects*, die über den *ORB* zur Verfügung gestellten Applikationen.

Kapitel 4

Konzept eines universellen Werkzeugs für parallele ereignisgesteuerte Simulation

4.1 Situation

Die derzeitige Situation auf dem Gebiet der Simulationswerkzeuge für parallele Simulation bei Verteilung des Simulationsmodells ist unbefriedigend. Sie ist hauptsächlich gekennzeichnet durch sehr spezielle Simulatoren, die oftmals für genau eine Anwendung oder genau ein Synchronisationsverfahren entwickelt und speziell dafür optimiert wurden. Beispiele hierfür finden sich in [Habermann,1992] mit der Simulation von ATM-Koppelnetzstrukturen auf Transputersystemen oder in [Phillips&Cuthbert,1991] mit einer Bibliothek speziell für die Simulation von Telekommunikationsnetzen mittels konservativer Algorithmen.

Das *Time Warp Operating System* [Jefferson et al.,1987] war eines der ersten parallelen Simulationswerkzeuge, in dem zumindest verschiedene optimistische Strategien implementiert waren. [Preiss,1989] stellt einen Ansatz für ein Simulationswerkzeug vor, das auch konservative Verfahren miteinbezieht.

Weitere Entwicklungen gab es im Bereich spezieller Sprachen für parallele Simulation. Hier sind beispielsweise *Modsim* [West&Mullarney,1988], *Sim++* [Baezner et al.,1990] sowie *Maisie* [Bagrodia&Liao,1994] zu erwähnen. In allen diesen Sprachen kann der Anwender sein Simulationsmodell in Logische Prozesse aufspalten. Ziel einer Sprache für (sequentielle) Simulation ist es normalerweise, dem Benutzer eine weitestgehend an seinem Problem orientierte Möglichkeit zu geben, sein Simulationsmodell zu entwickeln. Zugrundeliegende Implementierungsdetails werden dabei idealerweise vor ihm verborgen. Bei Sprachen für die

parallele Simulation kann dies aber zu einer drastisch verminderten Leistung führen. Sprachen, die zuviel von der zugrundeliegenden Rechnerarchitektur und dem benutzten Synchronisationsverfahren verbergen, täuschen den Anwender und verleiten ihn dazu, bequeme aber aufwendige Sprachkonstrukte häufig zu benutzen [Fujimoto,1993]. Aus diesem Grund geben die derzeit verfügbaren Sprachen dem Benutzer nur ein Hilfsmittel an die Hand, sein Simulationsmodell auf die zugrundeliegende Maschine und die Algorithmen anzupassen.

In [Tinker&Agre,1989] findet sich ein erster Ansatz für die Anwendung objektorientierter Methoden. Dort wird die Sprache *Objective-C* für die Implementierung eines Time-Warp-Systems verwandt. Das von [Steinman,1992b] in *C++* entwickelte *Speedes*-Werkzeug ist ebenfalls ein objektorientierter Ansatz. *Speedes* erlaubt neben Time-Warp auch den Einsatz von hybriden Algorithmen wie Breathing-Time-Buckets (siehe Abschnitt 2.5.3).

4.2 Anforderungen

Wie in Kapitel 2 beschrieben, gibt es bei einer Verteilung des Simulationsmodells neben der Synchronisation der Logischen Prozesse auch andere Bereiche, für die unterschiedliche Verfahren existieren (z. B. Zustandssicherung, Approximation der GVT). Diese Verfahren sind eingeschlossen, wenn im folgenden allgemein von Synchronisationsverfahren oder -algorithmen gesprochen wird. Jedes dieser Verfahren hat seine spezifischen Vor- und Nachteile und ist damit für bestimmte Simulationsmodelle gut geeignet, für andere dagegen überhaupt nicht. Die Entwicklung von Algorithmen, die unempfindlich gegenüber Änderungen des Simulationsmodells sind oder die sich auf diese Änderungen adaptiv einstellen können, ist derzeit Gegenstand der Forschung. Solange solche Verfahren nicht zur Verfügung stehen (und es ist diesbezüglich kein schneller Durchbruch zu erwarten), kommt man nicht umhin, ein zu simulierendes Modell genau zu analysieren und speziell dafür geeignete Synchronisationsalgorithmen einzusetzen. Diese Wahl erfordert nicht nur eine genaue Kenntnis der Eigenschaften der einzelnen Verfahren, sondern auch der des Simulationsmodells und kann deswegen nur mit entsprechenden Spezialkenntnissen vorgenommen werden. Selbst dann sind im voraus nicht immer eindeutige Vorhersagen möglich.

Eines der drängendsten Probleme der parallelen ereignisgesteuerten Simulation ist es, Methoden für ihre einfachere Anwendbarkeit zu entwickeln [Fujimoto,1993]. Bisher ist sie nur einem kleinen Kreis von Experten zugänglich und bleibt der überwiegenden Mehrheit der Simulationsanwender verschlossen. Um dies zu ändern, ist es erstrebenswert, das Simulationsmodell ohne genaue Kenntnis der später einzusetzenden Synchronisationsverfahren implementieren zu können. Diese Verfahren sollen weitgehend vor dem Endanwender gekapselt und durch diesen einfach austauschbar sein. Dies ermöglicht es, mit verschiedenen Verfahren nach dem Baukastenprinzip ohne großen Aufwand zu experimentieren, um die am besten geeigneten zu ermitteln.

Einen weiteren Aspekt stellt der einfache Übergang von sequentieller zu paralleler Simulation dar. Es wäre wünschenswert, daß ein mit einem sequentiellen Simulationswerkzeug vertrauter Anwender auf diesem Werkzeug aufbauend eine Parallelisierung vornehmen kann. Oder wie es in [Nicol&Heidelberger,1995] ausgedrückt wird: „Da der Berg nicht zum Propheten kommt, muß der Prophet zum Berg gehen." Dort wird auch gleich der Ansatz einer Erweiterung eines existierenden sequentiellen Simulators für parallele Simulation vorgestellt, allerdings nur für die Anwendung konservativer Algorithmen.

Um dem Ziel einer einfachen Anwendbarkeit der parallelen Simulation ein Stück näher zu kommen, wurde im Rahmen dieser Arbeit ein Simulationswerkzeug entwickelt, implementiert und getestet, das folgenden Anforderungen genügt:

- Simulationsmodell und Synchronisationsverfahren sind weitestgehend voneinander entkoppelt.
- Die Synchronisationsverfahren sind einfach austauschbar.
- Durch Anlehnung an ein existierendes sequentielles Simulationswerkzeug [Kocher,1994] entsteht ein einfacher Übergang von sequentieller zu paralleler Simulation.
- Es ist einfach erweiter- und portierbar.

Die Kombination all dieser Eigenschaften findet sich in keinem der im vorigen Unterkapitel beschriebenen bisherigen Ansätze. Als Implementierungssprache wurde *C++* gewählt, da diese eine weite Verbreitung besitzt und vor allem auf Parallelrechnern oftmals die einzig verfügbare objektorientierte Programmiersprache ist.

In den folgenden Unterkapiteln wird die Konzeption des Systems vorgestellt, während Kapitel 5 eingehend die vorteilhafte Anwendung objektorientierter Methoden bei der Bewältigung dieser komplexen Aufgabe im allgemeinen und bei der Kapselung der Synchronisationsalgorithmen im besonderen beschreibt.

4.3 Hierarchie der parallel ausführbaren Einheiten

Eine zentrale Rolle bei der Aufteilung des Simulationsmodells spielen die Logischen Prozesse. Sie besitzen eine eigene lokale Uhr, einen eigenen Ereigniskalender und simulieren intern rein sequentiell (siehe Kapitel 2). Um eine Gruppierung von Logischen Prozessen zu ermöglichen, werden einer oder mehrere von ihnen in *Logischen Knoten* (*Logical Node*, *LN*) zusammengefaßt. Innerhalb eines Logischen Knotens werden Logische Prozesse quasi-parallel abgearbeitet. Es gibt im verteilten System genau einen ausgezeichneten Logischen Knoten, den sog. *Manager-Knoten* (*Manager Node*, *MN*), der für zentrale Verwaltungsaufgaben zuständig ist. Ein Logischer Knoten stellt aus Rechnersicht einen Betriebssystemprozeß dar,

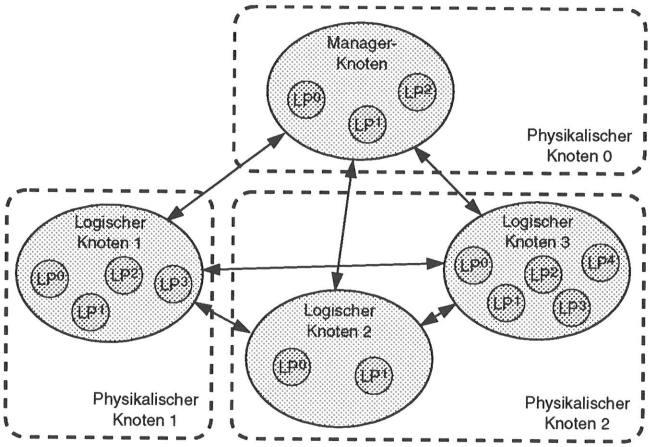

Bild 4.1: Hierarchie der parallel ausführbaren Einheiten

wobei ein *Physikalischer Knoten* (*Physical Node*, *PN*) einen oder mehrere Logische Knoten beherbergen kann. Bei einem Physikalischen Knoten handelt es sich ganz allgemein um einen Rechenknoten, hinter dem sich entweder ein Prozessorknoten eines Parallelrechners oder eine komplette Workstation in einem Netz verbergen kann.

Durch die unterschiedlich starke Kopplung der Einheiten dieses in Bild 4.1 nochmals dargestellten Konzepts ist eine optimale Anpassung an die im Simulationsmodell vorhandene Parallelität möglich.

4.4 Logische Knoten

Bild 4.2 zeigt den Aufbau eines Logischen Knotens. Die enthaltenen Logischen Prozesse erhalten durch einen *Scheduler* Rechenzeit und durch eine Verwaltungseinheit unkorrelierte Zufallszahlengeneratoren zugewiesen. Eine weitere Verwaltungseinheit steuert die Warmlaufphase und die einzelnen Teiltests zur Ermittlung der statistischen Aussagesicherheit.

Für die Zuteilung der Rechenzeit sind verschiedene Strategien möglich (z. B. zyklisch oder prioritätengesteuert). Ein eingeplanter Logischer Prozeß gibt die Kontrolle nach einer gewissen Zeit selbständig wieder an den Logischen Knoten zurück, was in der Regel geschieht, wenn der LP blockiert ist oder eine maximal erlaubte Anzahl Ereignisse bearbeitet wurde. Da die anderen LPs diesbezüglich auf die Zusammenarbeit des gerade aktiven LPs angewiesen sind, entspricht dieses Verfahren einem *kooperativen Multi-Tasking* im Gegensatz zu *preemptivem Multi-Tasking*, bei dem der aktive LP auf jeden Fall nach einer gewissen Zeit

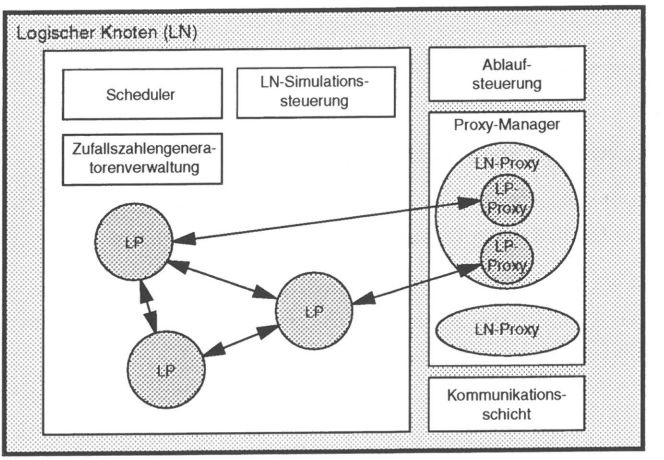

Bild 4.2: Aufbau eines Logischen Knotens

unterbrochen würde. Zwischen den Zeitschlitzen zweier aktiver LPs empfängt der Logische Knoten anstehende externe Nachrichten.

Die Ablaufsteuerung steuert und synchronisiert die Ausführung der parallelen Simulation, die sich in die folgenden Phasen gliedert (siehe auch Bild 4.3):

- Aufbau des Simulationsmodells,
- Herstellen der Verbindungen der Logischen Prozesse untereinander,
- eigentlicher Simulationslauf,
- Anhalten des Logischen Knotens,
- Beenden der Ausführung des Logischen Knotens.

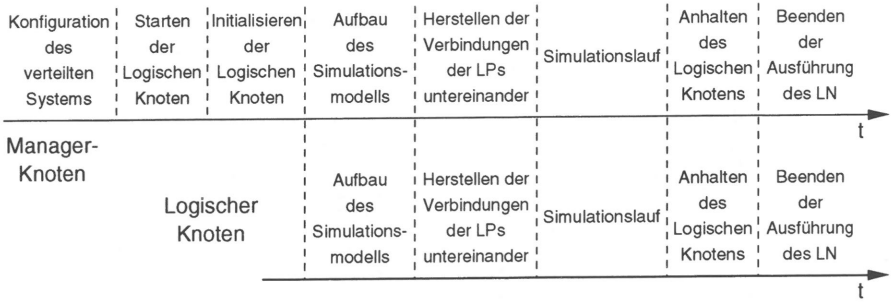

Bild 4.3: Ausführungsphasen im Manager-Knoten und den übrigen Logischen Knoten

Im Manager-Knoten kommen zuvor noch Phasen zur Etablierung des verteilten Systems:

- Konfiguration des verteilten Systems,
- Starten der übrigen Logischen Knoten,
- Initialisieren der übrigen Logischen Knoten.

Das Starten der einzelnen Ausführungsphasen übernimmt der Manager-Knoten, wobei am Ende einer jeden Phase alle Logischen Knoten synchronisiert werden.

Die in Bild 4.2 eingezeichneten Proxy-Objekte sind wichtige Bestandteile des Kommunikationssystems und werden zusammen mit der Kommunikationsschicht in Unterkapitel 4.6 ausführlich erläutert.

4.5 Logische Prozesse

Der prinzipielle Aufbau eines Logischen Prozesses ist in Bild 4.4 zu sehen. Neben einem Ereigniskalender und einer lokalen Uhr enthält er noch einen eigenen Zufallszahlengenerator und eine Verwaltungseinheit zur Steuerung der Teiltests. Bei einer optimistischen Simulation kommt noch ein Manager zur Steuerung der Zustandssicherung hinzu. Simulationsmodellkomponenten innerhalb eines Logischen Prozesses werden direkt verbunden und kommunizieren durch den Austausch von Nachrichten. Sie benutzen dabei das in [Kocher,1994] beschriebene Handshake-Protokoll. Sobald eine Komponente über die Grenzen eines Logischen Prozesses hinweg kommuniziert, erfolgt dies über *Eingangs-* und *Ausgangskanäle*. Diese sind ein wichtiges Element bei der Synchronisation der Logischen Prozesse untereinander. Bei konservativen Verfahren können sie z. B. für die Blockierung des Logischen Prozesses und

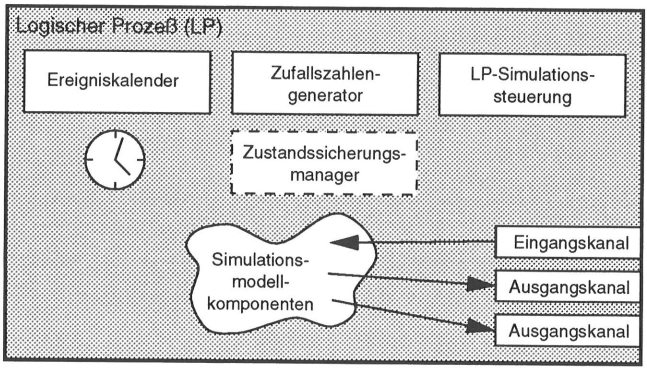

Bild 4.4: Aufbau eines Logischen Prozesses

das Versenden von NULL-Nachrichten verantwortlich sein, bei optimistischen für die Erkennung von Nachzüglern und das Versenden von Anti-Nachrichten. Die Kanäle bestimmen zusammen mit dem Kalender und der Verwaltungsstruktur für den Logischen Prozeß an sich im wesentlichen das verwandte Synchronisationsverfahren.

4.6 Kommunikationsmechanismen

4.6.1 Anforderungen

Ein wichtiger Aspekt jeder verteilten Anwendung ist die Kommunikation der verteilten Einheiten untereinander. Im vorliegenden Fall muß das Kommunikationssystem die folgenden Leistungsmerkmale besitzen:

- Übertragung von Nachrichtenobjekten, die
  - Objekte beliebiger Datentypen enthalten dürfen,
  - hierarchisch aufgebaut sein dürfen,
- Transparenz von lokaler und nicht-lokaler Kommunikation,
- Trennung der Kommunikationsmechanismen vom Simulationsmodell,
- Unterstützung verschiedener Synchronisationsverfahren,
- Effizienz hinsichtlich Laufzeit und Speicherverbrauch,
- Portabilität zwischen verschiedensten Plattformen (Workstation-Verbund und Parallelrechner).

In Kapitel 3 wurden einige Kommunikationssysteme vorgestellt, von denen aber keines alle o. a. Anforderungen erfüllt. Entweder die Möglichkeiten zur Übertragung von Daten beschränken sich auf einfache Datentypen (*PVM*, *MPI*, *NX* etc.), die Verfügbarkeit auf verschiedenen Rechnerplattformen, insbesondere Parallelrechnern, ist eingeschränkt oder das System ist für diese spezielle Anwendung überdimensioniert und verbraucht unnötig viele Ressourcen (*CORBA*).

Aus diesen Gründen wurde ein großer Teil des Systems für die Interprozeßkommunikation neu entwickelt. Vorausgesetzt wird lediglich, daß die Rechenplattform ein Subsystem für den Nachrichtenaustausch (*Message-Passing-Subsystem*, *MPSS*) mit der Möglichkeit des Versendens unstrukturierter Byte-Ströme sowie einfache Mechanismen für das Starten und Stoppen von Prozessen zur Verfügung stellt. Dadurch ist es möglich, das Kommunikationssystem speziell auf die Erfordernisse der parallelen Simulation einzustellen und damit zugleich sehr leistungsfähig und effizient zu gestalten.

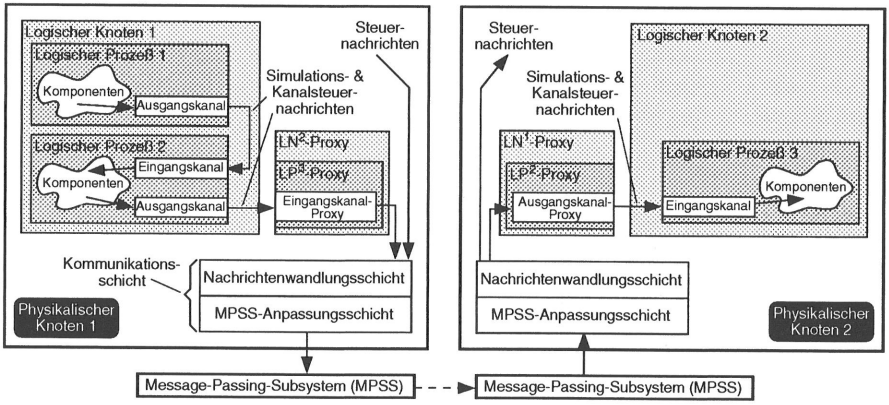

Bild 4.5: Prinzip des Nachrichtenaustauschs

Bild 4.5 zeigt anhand eines Beispiels für zwei Physikalische Knoten eine Übersicht des entwickelten Kommunikationssystems, das in den nächsten Abschnitten näher erläutert wird.[1]

4.6.2 Proxy-Konzept

In Unterkapitel 4.5 wurde erläutert, daß sämtliche Kommunikation zwischen Simulationsmodellkomponenten, die sich in verschiedenen Logischen Prozessen befinden, über Kanäle abgewickelt wird. Dabei wird jeweils ein Ausgangskanal mit einem Eingangskanal verbunden und so eine unidirektionale Verbindung etabliert. Diese Verbindung kann sehr einfach hergestellt werden, wenn beide Logischen Prozesse sich im selben Logischen Knoten befinden (siehe Verbindung zwischen LP^1 und LP^2 in Bild 4.5).

Ist dies nicht der Fall, dann muß die Nachricht über die Grenze eines Betriebssystemprozesses hinweg gesandt werden (siehe Verbindung zwischen LP^2 und LP^3 in Bild 4.5). Zu diesem Zweck werden für den jeweiligen Kommunikationspartner lokale „Stellvertreter", sog. *Proxies*, erzeugt, die eine Schnittstelle anbieten, als ob der entfernte Partner lokal vorhanden wäre. Statt mit dem tatsächlichen (entfernten) Kanal wird der lokale Kanal mit dessen Proxy verbunden. Es bleibt ihm verborgen, daß es sich um eine nicht-lokale Kommunikation handelt, da Wandlung, Senden und Empfang von Nachrichten automatisch zwischen den jeweiligen Kanal-Proxies durchgeführt werden. Jeder Proxy kennt hierzu den Ort seines Originals und ist in der Lage, zu versendende Nachrichten entsprechend zu adressieren. In

[1] Dabei ist die Numerierung der Logischen Knoten, Logischen Prozesse und Kanäle nach didaktischen Gesichtspunkten gewählt. In Wirklichkeit beginnt die Zählung der LPs in jedem Logischen Knoten und der Kanäle in jedem LP jeweils bei 0, so daß das Numerierungsschema insgesamt hierarchisch aufgebaut ist.

jedem Logischen Knoten befindet sich ein Proxy für jeden anderen Logischen Knoten des Systems, während Proxies für Logische Prozesse und Kanäle nur bei Bedarf erzeugt werden, d. h. wenn eine entsprechende Kommunikationsverbindung aufgebaut wird. Damit ist die Skalierbarkeit des Ansatzes auch für große Systeme mit vielen LPs sichergestellt.

Sofern ein benötigter Proxy noch nicht vorhanden ist, wird er automatisch erzeugt und holt sich alle erforderlichen Informationen von seinem entfernten Original. Bild 4.6 zeigt die

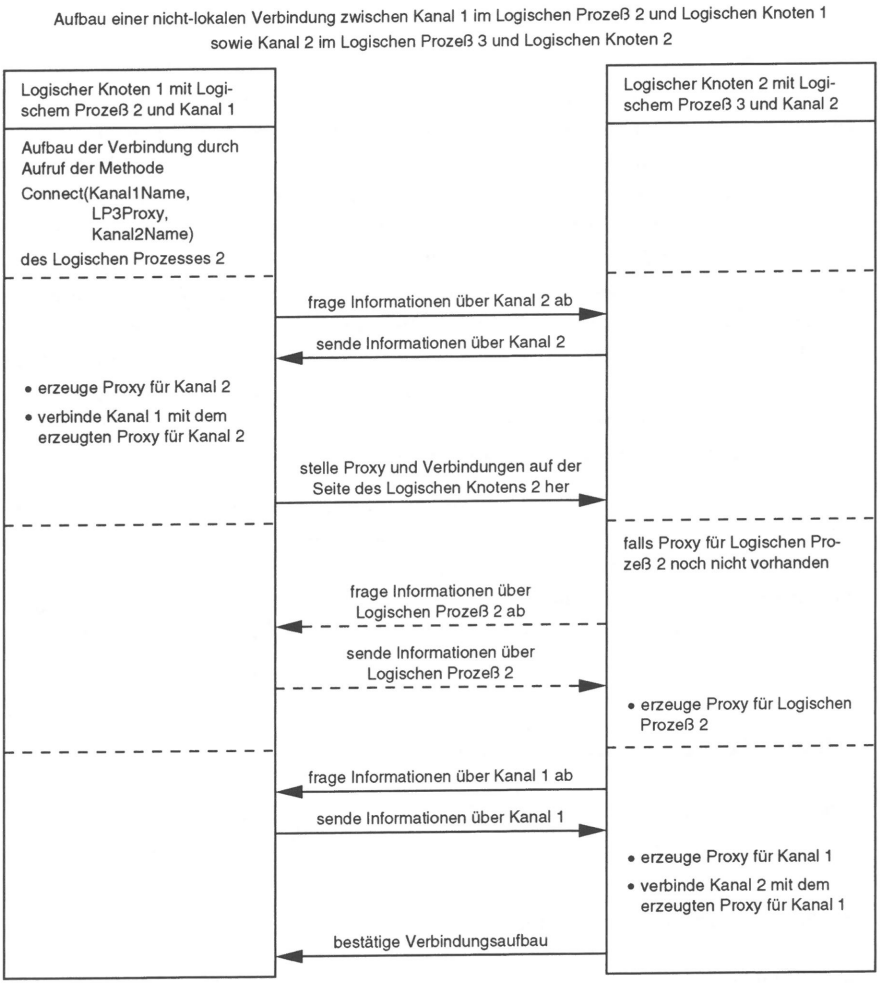

Bild 4.6: Verbinden von Kanälen in unterschiedlichen Logischen Knoten

Abläufe für die in Bild 4.5 dargestellte Verbindung zwischen dem Ausgangskanal (Kanal 1) in LP^2 und dem Eingangskanal (Kanal 2) in LP^3. Die beiden Kanäle seien bereits erzeugt und sollen nun miteinander verbunden werden. Die Herstellung der Verbindung wird im Logischen Knoten 1 durch Aufruf der *Connect()*-Methode des Logischen Prozesses 2 initiiert, als deren Parameter der Name des lokalen Kanals 1, eine Referenz auf den Proxy des nicht-lokalen Logischen Prozesses 3 und der Name des nicht-lokalen Kanals 2 anzugeben sind. Falls der Proxy des Logischen Prozesses 3 nicht schon aufgrund einer anderen Kommunikationsverbindung vorhanden ist, kann er vom Proxy des nicht-lokalen Logischen Knotens 2 zuvor unter Angabe des Namens des Logischen Prozesses erzeugt werden.

In einem ersten Schritt nach Aufruf der *Connect()*-Methode wird eine Steuernachricht (siehe Unterabschnitt 4.6.4.1) mit einer Anfrage über Kanal 2 versandt und eine Steuernachricht mit entsprechenden Informationen zurückgeschickt. Diese werden zur Erzeugung eines Proxies für Kanal 2 verwandt und dieser mit Kanal 1 verbunden. Damit sind auf der Seite des Logischen Knotens 1 alle notwendigen Verbindungen hergestellt, und er veranlaßt durch Senden einer Steuernachricht das gleiche auf seiten des Logischen Knotens 2. Falls dort noch kein Proxy für den Logischen Prozeß 2 vorhanden ist, wird zuerst dieser erzeugt. Die notwendigen Informationen hierzu werden wiederum über Steuernachrichten ausgetauscht (in Bild 4.6 durch gestrichelte Pfeile angedeutet). Die weiteren Abläufe zur Erzeugung des Kanal-Proxies und zu seiner Verbindung mit Kanal 2 verlaufen dann völlig spiegelbildlich zu den Vorgängen bei der Erzeugung und Verbindung des Kanal-Proxies im Logischen Knoten 1. Ist dies geschehen, wird eine Bestätigungsmeldung an den Logischen Knoten 1 gesandt und der gesamte Vorgang dadurch abgeschlossen.

Der beschriebene Ablauf bleibt dem Anwender verborgen, so daß die Verbindung zwischen zwei Kanälen, die sich nicht im gleichen Logischen Knoten befinden, genauso einfach herzustellen ist wie zwischen Kanälen, die sich im gleichen LN befinden. Der nächste Schritt, auch die Erzeugung der Kanäle und die Verbindungen der Simulationsmodellkomponenten mit diesen Kanälen für den Anwender transparent zu gestalten, kann durch Mechanismen höherer Ebenen erfolgen – darauf wird in Unterkapitel 6.6 näher eingegangen.

4.6.3 Kommunikationsschicht

Eine zu versendende Nachricht wird zusammen mit ihrer Zieladresse an die *Kommunikationsschicht* übergeben, welche ihrerseits in zwei Unterschichten unterteilt ist.

4.6.3.1 Nachrichtenwandlungsschicht

In der *Nachrichtenwandlungsschicht* wird ein komplexes Nachrichtenobjekt in einen mit einer eindeutigen Kennung versehenen unstrukturierten Byte-Strom umgewandelt, bzw. um-

gekehrt aus einem empfangenen Byte-Strom das entsprechende Nachrichtenobjekt rekonstruiert. Die Einführung dieser Schicht bietet den großen Vorteil, daß das zugrundeliegende Message-Passing-Subsystem (MPSS) nur primitive Grundfunktionalität für den Nachrichtenaustausch zur Verfügung stellen muß. Ein MPSS mit solchen Eigenschaften ist praktisch auf jeder Rechenplattform zu finden.

4.6.3.2 MPSS-Anpassungsschicht

Die *MPSS-Anpassungsschicht* entkoppelt die höheren Schichten des Kommunikationssystems vom eingesetzten Message-Passing-Subsystem. Sie stellt zur Nachrichtenwandlungsschicht hin eine einheitliche Schnittstelle für den Nachrichtenaustausch bereit und setzt die Schnittstellenaufrufe in entsprechende Aufrufe des MPSS um. Wird das Kommunikationssystem portiert und ein neues MPSS zugrundegelegt, so muß nur diese Schicht neu implementiert werden. Dadurch läßt sich auf jeder Rechenplattform das jeweils optimal geeignete MPSS nutzen. Derzeit realisiert sind Anpassungsschichten für *PVM* und *NX* (Paragraph 3.1.3.3.2).

4.6.4 Austausch von Steuerinformation

Bisher wurde nur der Austausch von Simulationsnachrichten betrachtet, mittels derer die einzelnen Simulationsmodellkomponenten miteinander kommunizieren. Darüber hinaus müssen aber zahlreiche weitere Informationen zur Steuerung des verteilten Systems ausgetauscht werden. Für diesen Informationsaustausch stehen je nach Bedarf folgende Möglichkeiten zur Verfügung: der explizite Austausch von *Steuernachrichten* und *Kanalsteuernachrichten* oder das Anhängen von Steuerinformation an Simulationsnachrichten mittels sog. *Nachrichtenmarkierungen*.

4.6.4.1 Steuernachrichten und Kanalsteuernachrichten

Bei der in Abschnitt 4.6.2 beschriebenen Verbindung zweier Kanäle in unterschiedlichen Logischen Knoten war schon vom Austausch von Steuernachrichten die Rede. Auch weitere, beliebige Objekte in verschiedenen Logischen Knoten können miteinander über Steuernachrichten kommunizieren, z. B. zum Zweck der Ablauf- oder Simulationssteuerung oder auch zur Berechnung der GVT. Sie haben dazu direkten Zugriff auf die Kommunikationsschicht (siehe Bild 4.5).

Daneben gibt es auch Kanalsteuernachrichten, die zwischen miteinander verbundenen Kanälen ausgetauscht werden. Hierzu zählen z. B. NULL- oder Anti-Nachrichten. Während Steuernachrichten immer zwischen Logischen Knoten ausgetauscht werden, dienen Kanalsteuernachrichten auch der Kommunikation von Kanälen innerhalb eines LNs.

4.6.4.2 Kanalfilter und Nachrichtenmarkierungen

Eine andere Möglichkeit zur Übermittlung von Steuerinformation bietet das Anhängen von Zusatzinformation an normale Simulationsnachrichten. Der GVT-Berechnungsalgorithmus aus [Mattern,1993] erfordert z. B. eine „Einfärbung" der Simulationsnachrichten, um unterscheiden zu können, ob sie vor oder nach einem bestimmten Zeitpunkt gesandt wurden. Zu diesem Zweck ist es möglich, an einem Kanal einen oder mehrere *Kanalfilter* zu installieren, die jede über diesen Kanal versandte oder empfangene Nachricht untersuchen können. Zudem kann ein Filter an einem Ausgangskanal mittels *Nachrichtenmarkierungen* jeder Nachricht beliebige Zusatzinformation mitgeben, die von einem entsprechenden Filter am empfangenden Eingangskanal ausgewertet werden kann. Dadurch ist es beliebigen Mechanismen einfach möglich, die vom Simulationsmodell ausgehende Kommunikation unter Kontrolle zu behalten, ohne in diese eingreifen oder gar Simulationsnachrichten oder Kanäle verändern zu müssen.

4.7 Zustandssicherung

Verschiedene Mechanismen für die Zustandssicherung bei optimistischen Verfahren wurden bereits in Unterabschnitt 2.5.2.3 vorgestellt. Jeder dieser Mechanismen hat seine spezifischen Stärken und Schwächen und eignet sich für bestimmte Simulationsmodellkomponenten in bestimmten Fällen besser oder schlechter. Aus diesem Grund bietet das entwickelte Simulationswerkzeug die Möglichkeit, auch die Zustandssicherungsmechanismen flexibel und transparent auszutauschen. Im einzelnen hat das realisierte Konzept folgende Eigenschaften:

- Es werden verschiedene Zustandssicherungsmechanismen unterstützt.
- Diese lassen sich einfach austauschen.
- Für jede Komponente im Simulationsmodell kann individuell der am besten geeignete Mechanismus verwandt werden. Unterschiedliche Mechanismen können im selben Modell koexistieren.
- Es sind nur wenige Eingriffe in die Modellkomponente selbst erforderlich.

Der Grundgedanke des Konzepts besteht darin, daß in jedem Logischen Prozeß ein zentraler Zustandssicherungs-Manager sog. *Sicherungsaufpunkte* festlegt, das sind die Zeitpunkte, zu denen der Zustand der lokalen Simulationsmodellkomponenten wiederherstellbar sein muß. Wie dies gewährleistet wird, liegt in der Verantwortung der einzelnen Komponenten selbst. Auf diese Weise können für die jeweiligen Komponenten unterschiedliche Mechanismen verwandt werden.

Implementiert sind bisher drei verschiedene Zustandssicherungsmechanismen (weitere lassen sich leicht realisieren): eine optimierte Variante der Erstellung von Zustandskopien, inkrementelle Zustandssicherung und eine Mischung von beidem.

Das Grundprinzip der Erstellung von Zustandskopien wurde schon in Paragraph 2.5.2.3.1 erläutert. Beim klassischen Verfahren würde an jedem Sicherungsaufpunkt der gesamte Zustand einer Simulationsmodellkomponente gesichert. Da sich dieser zwischen zwei Aufpunkten aber nicht notwendigerweise ändert, wurde das Verfahren optimiert. Mit jeder Zustandskopie wird zusätzlich eine Zeitspanne gespeichert, während der sie gültig ist. Hat sich der Zustand einer Komponente seit der letzten Erstellung einer Zustandskopie nicht verändert, wird am neuen Sicherungsaufpunkt lediglich die „Gültigkeitsdauer" dieser Kopie verlängert. Bei seltenen Zustandsänderungen läßt sich dadurch erheblich Zeit und Speicherplatz einsparen, womit die Erstellung von Zustandskopien auch effizient auf Komponenten angewandt werden kann, deren Zustand sich nicht häufig ändert. Für diese Optimierung müssen die Simulationsmodellkomponenten nun Zustandsänderungen anzeigen, was aber durch das in Unterkapitel 5.4 näher erläuterte Protokoll für die universelle Zustandssicherung gewährleistet ist.

Die inkrementelle Zustandssicherung ist in ihrer klassischen Form, wie in Paragraph 2.5.2.3.2 beschrieben, implementiert.

Schließlich wurde noch eine Mischform beider Verfahren entwickelt und realisiert. Dieses am besten mit *Copy-Restore Incremental State Saving* umschriebene Verfahren arbeitet bei der Sicherung nach dem Prinzip der inkrementellen Zustandssicherung, speichert aber nicht die einzelnen Zustandsänderungen ab, sondern benutzt diese, um eine separat gehaltene Zustandskopie zu aktualisieren. Diese wird entsprechend dem ausschließlich mit Zustandskopien arbeitenden Verfahren an jedem Sicherungsaufpunkt kopiert und in einer Liste gespeichert (sofern sie sich geändert hat, ansonsten wird wiederum nur die „Gültigkeitsdauer" der zuletzt gespeicherten Kopie verlängert). Da dadurch gesicherte Zustandskopien vorliegen, können diese beim Zurücksetzen auch auf einmal wiederhergestellt werden, so daß nicht jede einzelne Zustandsänderung rückgängig gemacht werden muß.

Kapitel 5

Kapselung der Mechanismen für parallele Simulation mit Hilfe objektorientierter Methoden

5.1 Motivation

In Kapitel 2 wurde gezeigt, daß parallele Simulation an sich eine komplexe Aufgabe darstellt. Sollen verschiedene Synchronisations- und Steuerverfahren bereitgestellt werden, so multipliziert sich der erforderliche Aufwand. Sollen diese Verfahren auch noch gekapselt und austauschbar sein, so ist ein höchst komplexes Softwaredesign zu entwerfen, zu implementieren und bei Bedarf zu erweitern. In den folgenden Unterkapiteln wird gezeigt, wie diese Ziele bei der Umsetzung der im letzten Kapitel vorgestellten Konzeption durch konsequente Anwendung objektorientierter Entwurfstechniken erreicht werden. Dabei wird auf die in [Booch,1994] eingeführte Notation zurückgegriffen (s. a. Anhang A). Alle Klassen- und Objektdiagramme zeigen, soweit nicht anders angegeben, die Verhältnisse in einem Logischen Knoten der aus mehreren LNs bestehenden parallelen Applikation.

5.2 Kapselung der Synchronisationsmechanismen

5.2.1 Basisklassen

Wie schon in Unterkapitel 4.5 erwähnt, wird das verwandte Synchronisationsverfahren im wesentlichen durch die Kanäle, den Ereigniskalender und die Verwaltungsstruktur für den Logischen Prozeß bestimmt. Die grundlegenden Beziehungen zwischen den entsprechenden

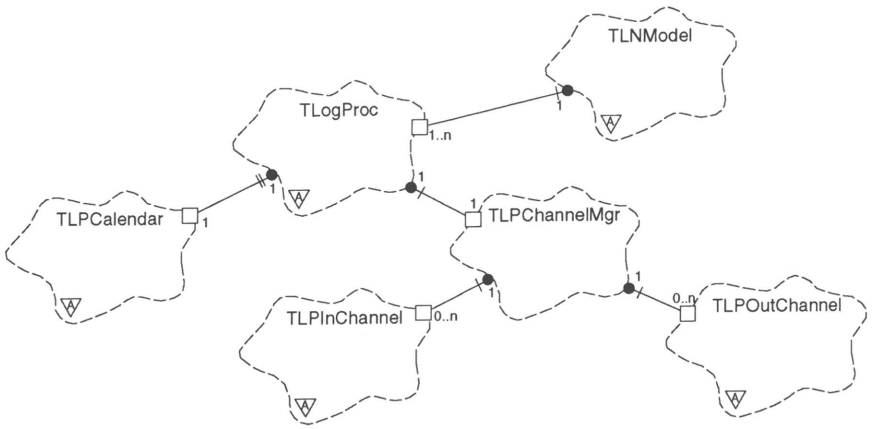

Bild 5.1: Basisklassen für Synchronisation

Klassen *TLPInChannel*, *TLPOutChannel*, *TLPCalendar* und *TLogProc* sind in Bild 5.1 dargestellt. *TLNModel* dient der zentralen Verwaltung aller lokalen Logischen Prozesse, und genau eine Instanz einer davon abgeleiteten Klasse ist in jedem Logischen Knoten enthalten.

Zur Kapselung der Synchronisationsmechanismen wird in hohem Maße auf das Entwurfsmuster der Schablonen-Methode zurückgegriffen (siehe Unterabschnitt 3.2.5.8) und damit auf die Konzepte der Polymorphie und der abstrakten Basisklasse. Die oben erwähnten Klassen stellen ein Grundgerüst dar, in dem nur grundlegende Verwaltungsfunktionalität implementiert ist. Erst durch von ihnen abgeleitete Klassen wird das genaue Verhalten festgelegt, und nur von diesen abgeleiteten Klassen können Objekte instanziiert werden. Der Simulationsanwender findet aber alle für ihn wichtigen Schnittstellen in den Basisklassen wieder, die zu einem Großteil durch (reine) virtuelle Funktionen gebildet werden. Welche Implementierung sich tatsächlich hinter der Schnittstelle verbirgt, ist für ihn nicht sichtbar.

TLogProc verwaltet seine Ein- und Ausgangskanäle mittels eines separaten „Kanal-Managers“, *TLPChannelMgr*. Dadurch kann die Schnittstelle von *TLogProc* kleiner gehalten werden, und es lassen sich verschieden angepaßte Implementierungen realisieren. In den folgenden Abschnitten werden die einzelnen Klassen und ihr Zusammenwirken näher erläutert.

5.2.2 Ereigniskalender

5.2.2.1 Zeitbegriff

In Unterkapitel 2.6 wurde die Notwendigkeit eines erweiterten Zeitbegriffs erläutert. Dieser wird mit den in Bild 5.2 dargestellten Klassen realisiert. *TCounterTime* hat zusätzlich

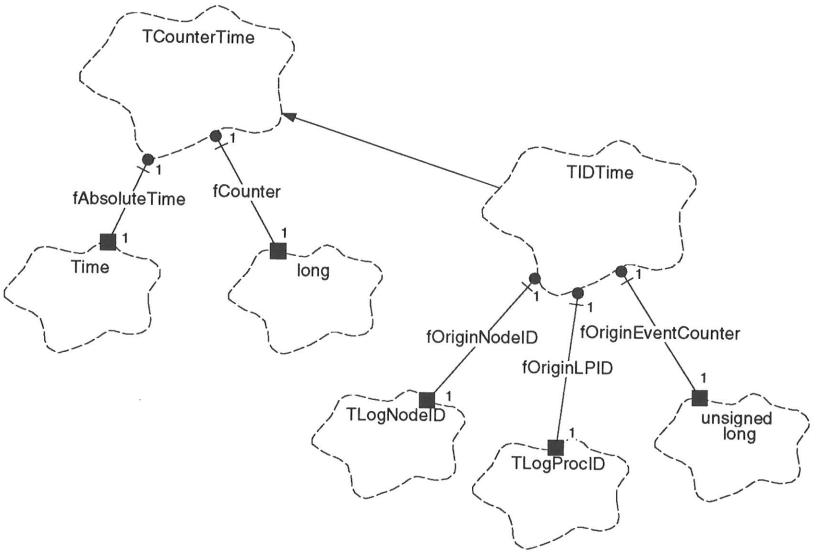

Bild 5.2: Realisierung des erweiterten Zeitbegriffs

zur einfachen Zeit, hier absolute Zeit genannt, einen Zähler. Mittels *TCounterTime* läßt sich eine lokale Reihenfolge bei gleicher absoluter Zeit festlegen. Dies wird z. B. bei der inkrementellen Zustandssicherung benötigt. Davon abgeleitet ist die Klasse *TIDTime*, die zusätzlich einen global eindeutigen Bezeichner realisiert. Dieser besteht aus den Bezeichnern des Logischen Knotens (*TLogNodeID*) und des Logischen Prozesses (*TLogProcID*), in denen er erzeugt wurde, sowie einem Zähler für die schon bearbeiteten und nicht wieder zurückgesetzten Ereignisse. *TIDTime* ist eine Umsetzung des in Unterkapitel 2.6 erläuterten Konzepts aus [Mehl,1991b] und [Mehl,1992]. Durch das Überladen verschiedener Zuweisungs- und Vergleichsoperatoren sind für beide Klassen die gewünschten Ordnungsrelationen und Umwandlungen festgelegt, so daß Ihre Anwendung ähnlich komfortabel wie beim einfachen Datentyp *Time* erfolgen kann.

5.2.2.2 Klassenhierarchie

Die abstrakte Basisklasse *TLPCalendar* aus Bild 5.1 ist die Wurzel der in Bild 5.3 dargestellten Klassenhierachie für den Ereigniskalender. Es gibt jeweils eine weitere, davon abgeleitete abstrakte Basisklasse für sequentielle (*TIsolatedLPCal*)[1], konservative (*TConsLPCal*) sowie

[1] Die Bezeichnung „Isolated" rührt daher, daß LPs, die diesen Kalender benutzen, keine Außenbeziehungen zu anderen LPs über Kanäle unterhalten dürfen, da keine Synchronisationsmechanismen vorhanden sind (in einer sequentiellen Simulation gibt es nur einen LP).

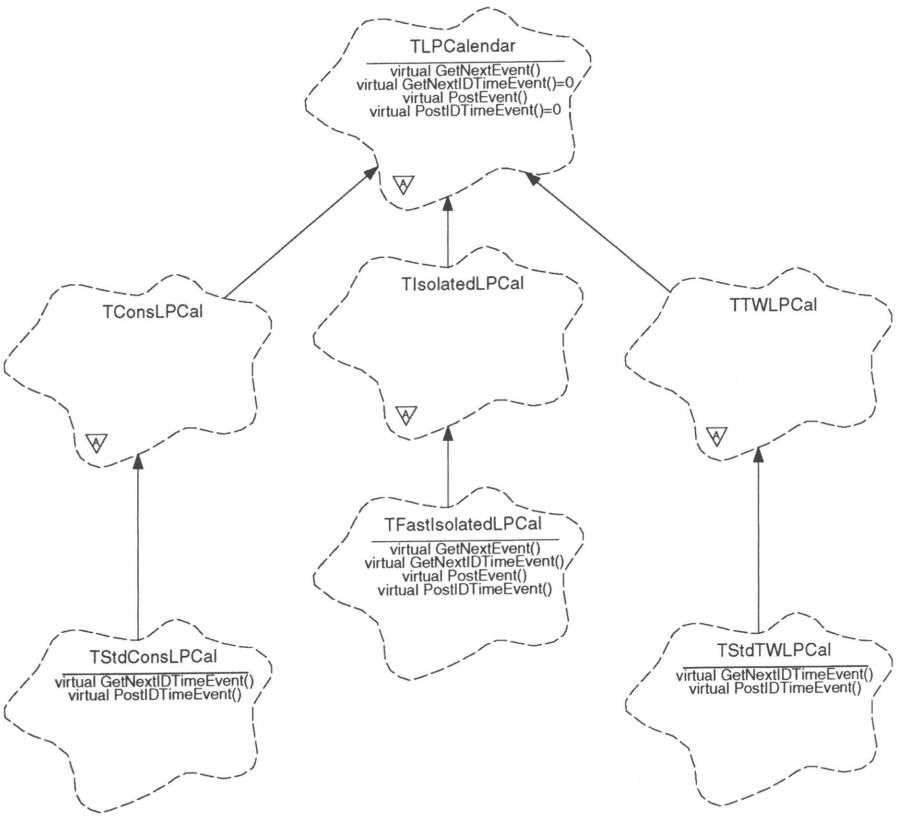

Bild 5.3: Klassenhierarchie für Ereigniskalender

optimistische Simulation (*TTWLPCal*). Diese legen die Kategorie des benutzten Synchronisationsverfahrens fest. Davon wiederum abgeleitet sind Klassen, die jeweils eine Unterart des Synchronisationsverfahrens implementieren. Im Bild sind dies *TFastIsolatedLPCal* für einen speziell für sequentielle Simulation optimierten Kalender entsprechend [Brown,1988], *TStdConsLPCal* für einen konservativen Kalender, der die Blockierung der Ereignisbearbeitung zu bestimmten Zeitpunkten erlaubt und schließlich noch *TStdTWLPCal* für einen optimistischen Kalender mit klassischem Time Warp und Aggressive Cancellation.

5.2.2.3 Allgemeine Schnittstelle

Die wichtigsten öffentlich zugänglichen Operationen der Klassen sind ebenfalls in Bild 5.3 dargestellt. Die öffentliche Schnittstelle von *TLPCalendar* bietet z. B. Methoden zum Ein-

tragen neuer Ereignisse in den Ereigniskalender. Dazu wird i. allg. die virtuelle Methode *PostEvent()* verwandt, bei der als Ereigniszeitpunkt lediglich eine Absolutzeit-Angabe erfolgt, während die erweiterte Zeit automatisch bestimmt wird. Der tatsächliche Eintrag erfolgt dann unter Verwendung der reinen virtuellen Methode *PostIDTimeEvent()*, der die erweiterte Zeit übergeben wird. Für externe Ereignisse oder spezielle Anwendungen kann diese Methode auch direkt aufgerufen werden.

Das nächste zu bearbeitende Ereignis erhält man durch Aufruf der virtuellen Methoden *GetNextEvent()* und *GetNextIDTimeEvent()*, die sich wiederum nur darin unterscheiden, ob sie als Ereigniszeitpunkt eine erweiterte oder eine absolute Zeit zurückliefern.

Die Implementierung der o. a. Funktionen für das Ein- und Austragen von Ereignissen erfolgt erst in den abgeleiteten Klassen. So prüft *TStdConsLPCal::GetNextIDTimeEvent()*[2] vor dem Austragen des nächsten verfügbaren Ereignisses, ob dessen Zeitstempel größer als der Zeitstempel einer eventuell vorhandenen Blockierung ist. Auf der anderen Seite prüft *TStdTWLPCal::PostIDTimeEvent()* vor Eintrag eines Ereignisses, ob es sich um einen Nachzügler handelt und löst gegebenenfalls ein Zurücksetzen des LPs aus.

5.2.2.4 Schnittstelle für Simulationsmodellkomponenten

Simulationsmodellkomponenten werden von den Einzelheiten dieser Implementierung nicht berührt. Sie tragen Ereignisse durch Aufruf von *TLPCalendar::PostEvent()* in den Kalender ein. Die Generierung einer erweiterten Zeit und insbesondere die Einzelheiten des Synchronisationsverfahrens bleiben vollständig vor der Komponente verborgen. Der Kalender stellt damit für die Modellkomponenten die gleiche Schnittstelle wie im sequentiellen Werkzeug aus [Kocher,1994] bereit, was die Portierung der dafür entwickelten Komponenten erleichtert.

5.2.3 Kanäle

Die Wurzel der Klassenhierarchie der Kanäle stellt die Klasse *TLPChannel* dar (Bild 5.4). Von ihr werden sowohl die eigentlichen Kanäle (*TLocalLPCh*) als auch deren Proxies (*TLPChannelProxy*) abgeleitet. Danach erfolgt durch weitere Ableitung die Aufspaltung in Eingangs- (*TLPInChannel*) und Ausgangskanäle (*TLPOutChannel*) bzw. Eingangs- (*TLPInputChProxy*) und Ausgangskanal-Proxies (*TLPOutputChProxy*). Die gemeinsame Basisklasse von Kanälen und Kanal-Proxies ermöglicht eine einfache Verbindung von Instanzen beider Klassen und ist somit ein wichtiger Teil des in Unterkapitel 4.6 vorgestellten Konzepts für die Transparenz lokaler und nicht-lokaler Kommunikation.

[2] *X::Y* ist die in *C++* verwandte Schreibweise zur Bezeichnung der Methode *Y* der Klasse *X*.

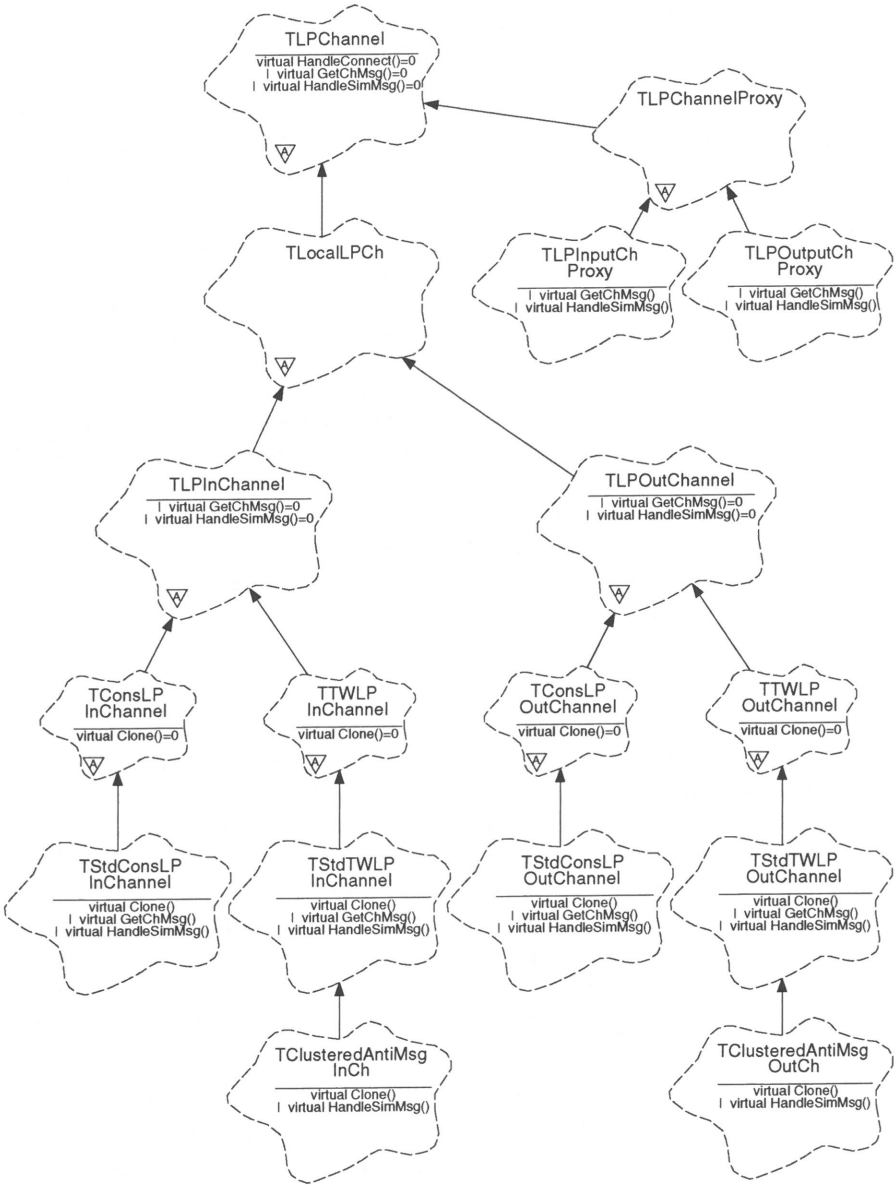

Bild 5.4: Klassenhierarchie für Kanäle

TLPInChannel und *TLPOutChannel* sind die zentralen abstrakten Basisklassen, über die der Aufbau der Kommunikationsbeziehungen erfolgt. Auch hier wird wieder Polymorphie angewandt, um Implementierungsaspekte und damit die Art der Synchronisation zu verbergen. Beim Versenden/Empfangen einer Nachricht über einen Kanal (siehe Abschnitt 5.3.3 für Details) werden die reinen virtuellen Funktionen *HandleSimMsg()* beim Eintritt der Nachricht in den Kanal und *GetChMsg()* beim Austritt aufgerufen. Die Implementierung dieser Methoden geschieht in den abgeleiteten Klassen. Es gibt wiederum abstrakte Basisklassen für konservative und optimistische Kanäle. Diese bieten für die Realisierung eines Prototypen-Entwurfsmusters (siehe Unterabschnitt 3.2.5.4) eine reine virtuelle Methode *Clone()* an, mit deren Hilfe aus einem vorgegebenen Kanal ein Objekt gleichen Typs generiert werden kann, ohne daß der genaue Typ dieses Objekts bekannt sein muß. Dies wird dazu benutzt, den Typ des Kanals, der vom Logischen Prozeß im Normalfall erzeugt wird (siehe Unterkapitel 5.2.4), „voreinzustellen".

In Bild 5.4 sind Klassen mit verschiedenen Implementierungen gezeigt: konservative Kanäle mit NULL-Nachrichten (*TStdConsLPXXXChannel*)[3], optimistische Kanäle mit klassischem Versenden von Anti-Nachrichten für jede falsche Nachricht (*TStdTWLPXXXChannel*) sowie eine optimierte optimistische Version, bei der unter Ausnutzung der FIFO-Eigenschaft der Verbindungen zwischen den Kanälen pro Zurücksetzen nur eine Anti-Nachricht für die falsche Nachricht mit dem kleinsten Zeitstempel gesandt wird (*TClusteredAntiMsgXXXCh*).

5.2.4 Logische Prozesse

Bild 5.5 zeigt die Klassenhierarchie für Logische Prozesse. Auch hier werden ausgehend von der abstrakten Basisklasse *TLogProc* weitere Klassen mit den tatsächlichen Implementierungen für die einzelnen Synchronisationsverfahren abgeleitet. Mittels der reinen virtuellen Methoden *CreateInChannel()* und *CreateOutChannel()* lassen sich jeweils geeignete Ein- und Ausgangskanäle erzeugen. Dies geschieht durch Anwendung der *Clone()*-Methode der in den Klassen *TConsLogProc* bzw. *TTWLogProc* vorhandenen Kanal-Prototypen (siehe vorheriger Abschnitt). Diese können entweder bei der Konstruktion des Logischen Prozesses oder beispielsweise durch eine für die Konfiguration verantwortliche Instanz erzeugt und zugewiesen werden. In jedem Fall bleibt dadurch der tatsächliche Typ eines Kanals beim Aufbau von Kommunikationsbeziehungen verborgen – dies ist wichtig für die Kapselung der Synchronisationsmechanismen.

TLogProc enthält zusätzlich noch Referenzen auf den lokalen Zufallszahlengenerator *TParRNG*, den Kanal-Manager *TLPChannelMgr* und *TLPSimCtrlMgr*, der die Steuerung der Teiltests übernimmt. *TParRNG* ist ebenfalls eine abstrakte Basisklasse, wodurch sich einfach verschiedene Methoden der Zufallszahlenerzeugung realisieren lassen.

[3] „XXX" steht hierbei für „In" bzw. „Out"

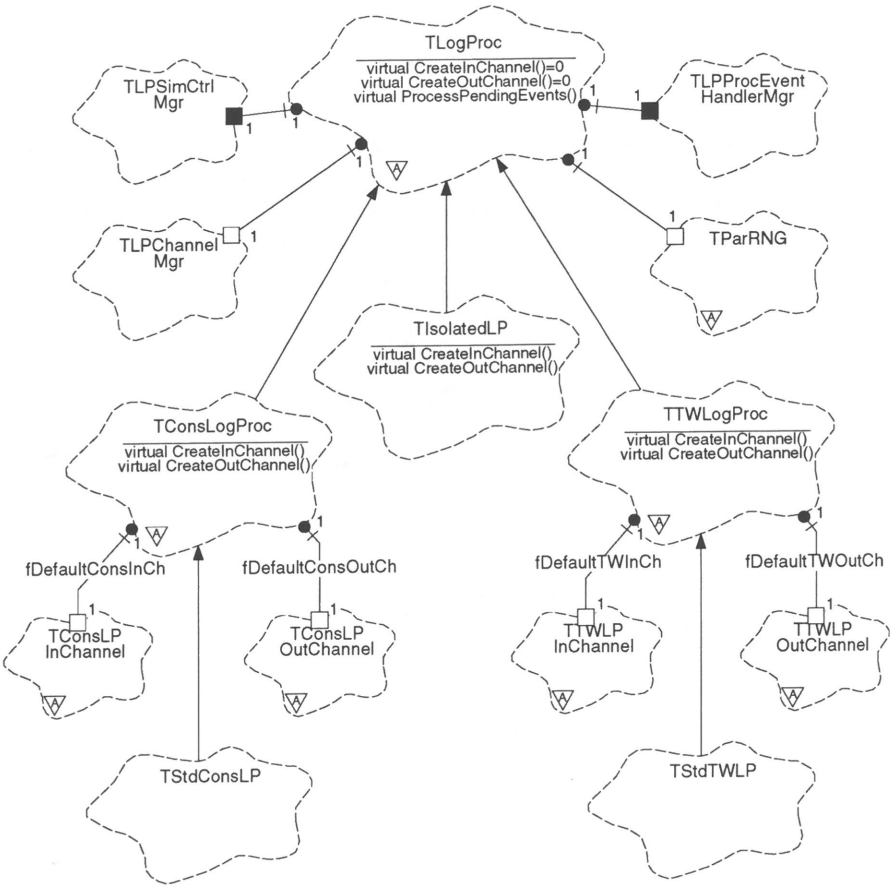

Bild 5.5: Klassenhierarchie für Logische Prozesse

TLPProcEventHandlerMgr, über den sich dynamisch Handler registrieren lassen, ist eine Implementierung des Entwurfsmusters der „Kette der Verantwortlichkeit“ (siehe Unterabschnitt 3.2.5.5). Bei Aufruf und Verlassen der zentralen Routine für die Ereignisbearbeitung, *TLogProc::ProcessPendingEvents()*, sowie nach jedem abgearbeiteten Ereignis werden für alle registrierten Handler entsprechende virtuelle Bearbeitungsroutinen aufgerufen, so daß diese flexibel reagieren können. Verschiedene Synchronisationsverfahren können damit durch die Ereignisbearbeitung beeinflußt werden, ohne daß *TLogProc::ProcessPendingEvents()* geändert werden muß – bei der Implementierung neuer Synchronisationsverfahren muß also nicht jedesmal in diese Basisfunktionalität eingegriffen werden. Das Versenden von NULL-Nachrichten ist z. B. mittels eines solchen Handlers realisiert.

5.2.5 Logische Knoten

5.2.5.1 Ablaufsteuerung

In jedem Logischen Knoten gibt es genau eine Instanz einer von *TExecManager* abgeleiteten Klasse, die für die Steuerung des in Unterkapitel 4.4 beschriebenen Ablaufs zuständig ist (Bild 5.6). Es handelt sich dabei um ein innerhalb des Logischen Knotens globales Objekt, das mit Hilfe des *Singleton*-Entwurfsmusters aus Unterabschnitt 3.2.5.7 realisiert und über

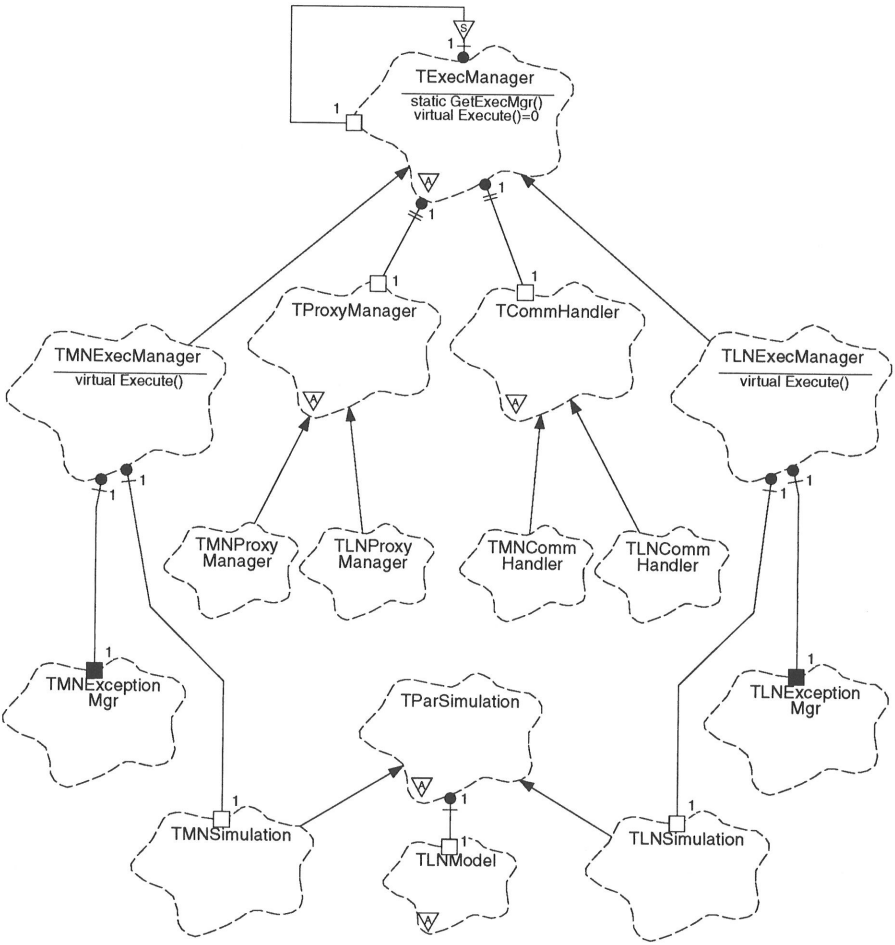

Bild 5.6: Klassendiagramm für Ablaufsteuerung

die statische Funktion *TExecManager::GetExecMgr()* zugänglich ist. Über diese Einsprungstelle lassen sich auch die ebenfalls innerhalb eines Logischen Knotens globalen Manager für Kommunikation, *TCommHandler*, und für Proxy-Objekte, *TProxyManager*, erreichen, deren Aufgaben in den Unterabschnitten 5.3.2.3 und 5.3.3.3 genauer erklärt werden. Die beiden Klassen *TMNExceptionMgr* und *TLNExceptionMgr* sind Teil des in Unterkapitel 6.5 vorgestellten Konzepts zur Ausnahmebehandlung.

Der Ablauf im Manager-Knoten unterscheidet sich grundlegend vom Ablauf in den übrigen Logischen Knoten. Ersterer ist der Master, der die übrigen Slaves steuert. Diesem fundamentalen Unterschied wird durch Ableitung der beiden Klassen *TMNExecManager* und *TLNExecManager* Rechnung getragen. Der in Unterkapitel 4.4 beschriebene Ablauf wird dann durch Aufruf der virtuellen Methode *Execute()* gestartet. *TMNExecManager* und *TLNExecManager* haben Referenzen auf jeweils ein Objekt der Klassen *TMNSimulation* und *TLNSimulation*, das für die Abwicklung des eigentlichen Simulationslaufs verantwortlich ist. Über deren gemeinsame Basisklasse *TParSimulation* ist schließlich eine Instanz einer von *TLNModel* abgeleiteten Klasse erreichbar, die alle LPs des Logischen Knotens verwaltet.

5.2.5.2 Scheduling

Die Verwaltung der Logischen Prozesse in einem Logischen Knoten ist in Bild 5.7 dargestellt. Es existiert genau eine Instanz einer von *TLNModel* abgeleiteten Klasse, die eine oder meh-

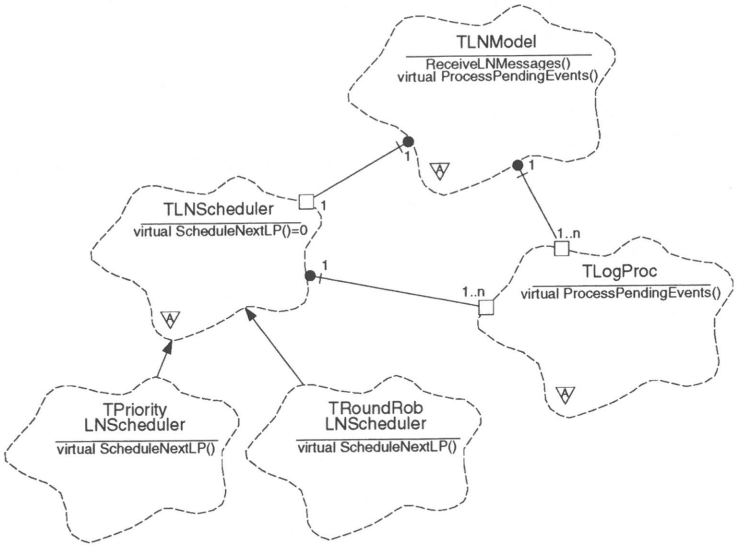

Bild 5.7: Scheduling der Logischen Prozesse in einem Logischen Knoten: Klassen

rere Instanzen von Unterklassen von *TLogProc* (eine pro Logischem Prozeß) durch Referenz enthält. Zusätzlich dargestellt ist noch die Klasse *TLNScheduler*, die für die Zuteilung von Rechenzeit an die einzelnen Logischen Prozesse verantwortlich ist (s. a. Unterkapitel 4.4). Auch diese ist eine abstrakte Basisklasse und die Strategie der Rechenzeitverteilung wird erst durch abgeleitete Klassen festgelegt. Möglich sind z. B. ein zyklisches oder ein prioritätengesteuertes Bearbeiten der Logischen Prozesse (*TRoundRobLNScheduler* bzw. *TPriorityLNScheduler*). Die Bearbeitungsstrategie bleibt dem Logischen Knoten verborgen, da er das Bearbeiten eines LPs lediglich durch Aufruf der reinen virtuellen Funktion *ScheduleNextLP()* initiiert, die erst in den abgeleiteten Klassen implementiert ist.

Die Abläufe in der zentralen Bearbeitungsmethode *TLNModel::ProcessPendingEvents()*, die zu Beginn einer Simulationsphase (Warmlaufphase oder Teiltest) aufgerufen und erst an deren Ende wieder verlassen wird, sind in Bild 5.8 dargestellt. Nachdem innerhalb von *ScheduleNextLP()* ein LP ausgewählt wurde, wird dessen *ProcessPendingEvents()*-Methode aufgerufen und dadurch die Ereignisbearbeitung dieses LPs angestoßen. Nach einer gewissen Zeit stoppt der LP die Bearbeitung und gibt die Kontrolle zurück (siehe Unterkapitel 4.4). Bevor der nächste LP eingeplant wird, werden durch Aufruf von *ReceiveLNMessages()* noch anstehende externe Nachrichten empfangen und an ihre Ziele weitergeleitet. Diese Nachrichten waren vom Message-Passing-Subsystem zwischengespeichert worden und werden nun durch evtl. mehrmaligen Aufruf von *TCommHandler::ReceiveMessage()* einzeln übernommen und bearbeitet. Der Zyklus des Einplanens eines Logischen Prozesses, der Ereignisbearbeitung und des Empfangens von Nachrichten wird bis zum Ende der aktuellen Simulationsphase wiederholt.

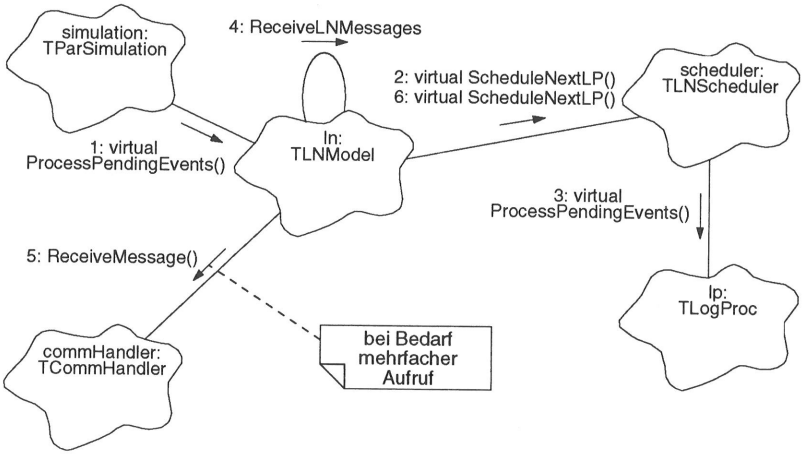

Bild 5.8: Scheduling der Logischen Prozesse in einem Logischen Knoten: Ablauf

5.3 Kapselung der Kommunikationsmechanismen

5.3.1 Nachrichten

5.3.1.1 Problemstellung

Während Instanzen aller bis jetzt beschriebenen Klassen nach ihrer Erzeugung lokal in einem Logischen Knoten und damit innerhalb eines Betriebssystemprozesses verbleiben, handelt es sich bei Nachrichten um Objekte, die zwischen Logischen Knoten austauschbar sein müssen. An dieser Stelle entsteht ein Bruch in der sonst durchgängigen objektorientierten Sicht. Die zugrundeliegende Rechenplattform kann in der Regel nur einfache Datentypen übertragen. Daraus ergibt sich die Notwendigkeit, daß ein zu versendendes Nachrichtenobjekt in einen Strom einfacher Datentypen umgewandelt wird und auf der Empfängerseite eine Kopie des Objekts wieder erstellt wird (s. a. Unterkapitel 4.6).

Auch dieser Bruch läßt sich mit Hilfe objektorientierter Methoden weitestgehend verbergen. Dazu müssen alle Nachrichten-Klassen von der abstrakten Basisklasse *TParMessage* aus Bild 5.9 abgeleitet sein. Diese Klasse enthält u. a. virtuelle Methoden für das Packen und Entpacken ihrer Felder in unstrukturierte Byte-Ströme. Außerdem enthält sie die virtuelle Funktion *HandleSending()* und die reine virtuelle Funktion *HandleReceiving()*, die von der Kommunikationsschicht vor dem Senden bzw. nach dem Empfangen der betreffenden Nachricht aufgerufen werden. Letztere muß in einer abgeleiteten Klasse implementiert sein, denn sie bestimmt, was mit einer empfangenen Nachricht zu geschehen hat. Jede Nachricht

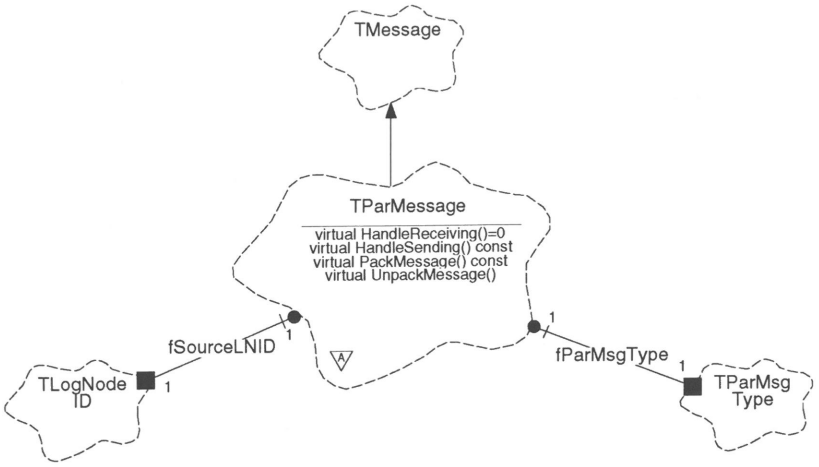

Bild 5.9: Basisklasse für Nachrichten

„weiß“ also selbst, wie sie ihre eigene Umwandlung in/aus Byte-Ströme(n) durchführen muß und was sie am Empfängerort zu tun hat. *TParMessage* ist abgeleitet von *TMessage*, der in [Kocher,1994] für das dort beschriebene sequentielle Simulationswerkzeug eingeführten Nachrichtenklasse. Dadurch ist gewährleistet, daß alle Nachrichten weiterhin von den dort entwickelten Komponenten gehandhabt werden können.

5.3.1.2 Steuernachrichten

Eine Vielzahl von Steuermechanismen des verteilten Systems ist auf den Informationsaustausch zwischen Logischen Knoten mittels Nachrichten angewiesen. Es sind dies z. B. die Ablaufsteuerung für die Synchronisation der einzelnen Ablaufphasen, die Erzeugung von Proxy-Objekten, die Berechnung der GVT und viele mehr. Jeder dieser Mechanismen kann sehr einfach beliebige *Steuernachrichten* von *TParMessage* ableiten (in Bild 5.10 angedeutet durch die Klasse *TParAnyCtrlMsg*) und durch eine geeignete Implementation von *HandleReceiving()* die gewünschten Aktionen beim Empfänger auslösen (s. a. Unterabschnitt 4.6.4.1).

5.3.1.3 Kanalnachrichten

Neben den Steuernachrichten, die mehr oder weniger zwischen beliebigen Objekten in verschiedenen Logischen Knoten ausgetauscht werden können, gibt es die wichtige Klasse der

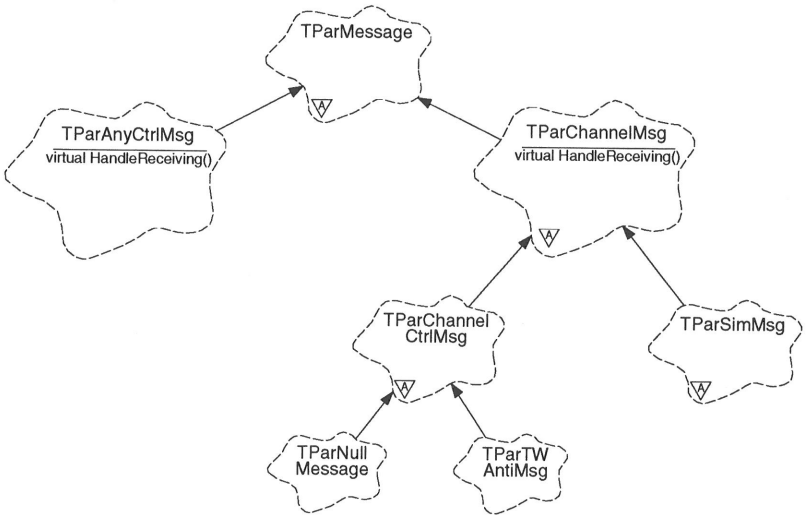

Bild 5.10: Klassenhierarchie für Steuer- und Kanalnachrichten

Kanalnachrichten (*TParChannelMsg*), die nur zwischen miteinander verbundenen Eingangs- und Ausgangskanälen ausgetauscht werden (ebenfalls zu sehen in Bild 5.10). Dabei spielt es keine Rolle, ob beide Kanäle sich auf demselben Logischen Knoten befinden oder nicht. *TParChannelMsg* enthält Informationen über den sendenden Ausgangskanal, und die virtuelle Funktion *HandleReceiving()* ist derart implementiert, daß eine über die Kommunikationsschicht versandte Nachricht nach Empfang automatisch an den zuständigen Kanal-Proxy weitergeleitet wird.

Kanalnachrichten teilen sich auf in von *TParChannelCtrlMsg* abgeleitete *Kanalsteuernachrichten* und die von *TParSimMsg* abgeleiteten *Simulationsnachrichten*. Erstere werden für den Anwender unsichtbar nur zu Steuerungszwecken zwischen verbundenen Kanälen ausgetauscht. Hierzu zählen z. B. NULL-Nachrichten (*TParNullMessage*) und Anti-Nachrichten (*TParTWAntiMsg*). Simulationsnachrichten dagegen sind die eigentlichen Nutznachrichten, die der Kommunikation der Simulationsmodellkomponenten untereinander dienen.

Simulationsnachrichten, deren Klassenhierarchie in Bild 5.11 dargestellt ist, bestehen aus zwei Teilen: der Nachrichten-Repräsentation, die den Nutzinhalt enthält, und einem zusätzlichen Handle, über das dieser Inhalt zugänglich ist. Dies entspricht dem in Unterab-

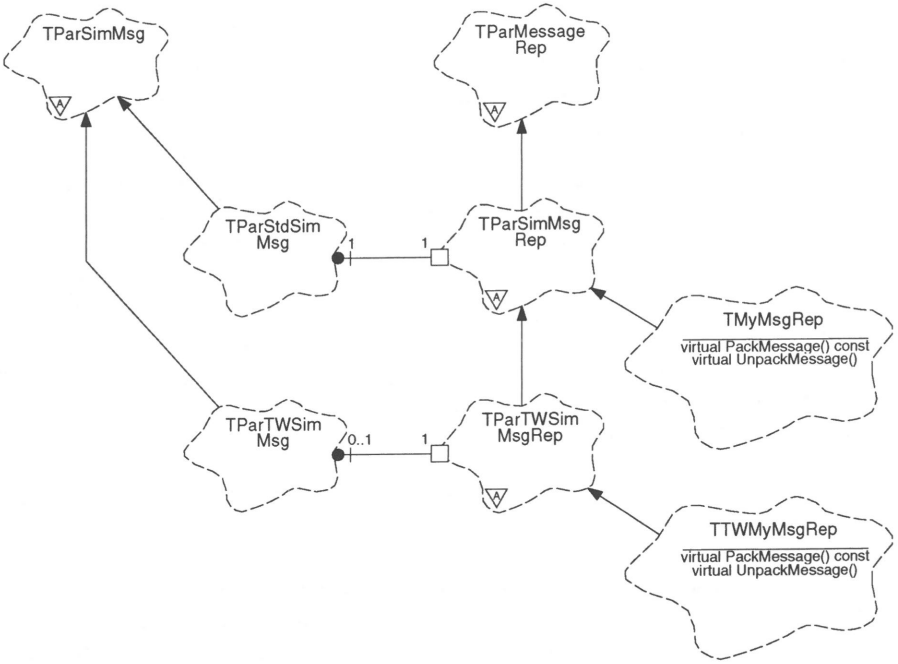

Bild 5.11: Klassenhierarchie für Simulationsnachrichten

schnitt 3.2.5.2 beschriebenen Entwurfsmuster der *Handle-Klasse*. Der Grund für diese Aufspaltung liegt in der Zustandssicherung von Nachrichteninhalten bei optimistischen Verfahren. In Abschnitt 5.4.5 wird näher erläutert, daß dabei Nachrichten-Repräsentationen auch ohne zugehörige Nachricht existieren können. Im Bild ist dies dadurch ersichtlich, daß Objekte der Klasse *TParTWSimMsgRep* durch 0 oder 1 Objekt der Klasse *TParTWSimMsg* referenziert werden. Diese beiden Klassen sind anzuwenden, wenn die Möglichkeit bestehen soll, alle vorhandenen Synchronisationsverfahren inklusive der optimistischen einzusetzen. Möchte man sich auf konservative Verfahren beschränken, so können stattdessen die einfacheren Klassen *TParSimMsgRep* und *TParStdSimMsg* verwandt werden.

Dem Simulationsanwender bleibt diese Zweiteilung jeder Nachricht im wesentlichen verborgen. Er leitet seine eigenen Nachrichten einfach von *TParSimMsgRep* bzw. *TParTWSimMsgRep* ab (angedeutet durch *TParMyMsgRep* und *TParTWMyMsgRep*) und stellt dort Funktionen für das Packen und Entpacken zur Verfügung.

5.3.2 Kommunikationsschicht

5.3.2.1 Nachrichtenverwaltung

Kommt beim Empfänger einer Nachricht ein unstrukturierter Byte-Strom an, so muß er diesen identifizieren und ein entsprechendes Nachrichtenobjekt dafür erzeugen. Damit er dazu in der Lage ist, wurde die in Bild 5.12 dargestellte Nachrichtenverwaltung entwickelt.

Nachrichten werden durch sog. *Nachrichten-Pools* erzeugt, wobei für jede Nachrichtenklasse *TMsg* in jedem Logischen Knoten mindestens eine zentrale Quelle existiert (mehrere sind erlaubt, sofern es sich um Nachrichten gleichen Typs aber unterschiedlichen Inhalts handelt). Da aber auf der anderen Seite Nachrichten desselben Typs an verschiedenen Punkten einer Applikation generiert werden müssen, teilt sich ein Nachrichten-Pool entsprechend dem Entwurfsmuster der Handle-Klasse (siehe Unterabschnitt 3.2.5.2) auf in die eigentliche zentrale Nachrichtenquelle vom Typ *TParStdMsgPoolRep<TMsg>*[4] und mehrere verteilte Handles vom Typ *TParStdMsgPool<TMsg>*, über die auf die Quelle zugegriffen werden kann und die an den benötigten Stellen einer Applikation eingesetzt werden.

Die Quellen vom Typ *TParStdMsgPoolRep<TMsg>* wiederum werden in jedem Logischen Knoten von jeweils einem zentralen *Nachrichten-Pool-Manager* vom Typ *TParMsgPoolMgr* verwaltet, der zur Sicherung seiner Einzigartigkeit in jedem LN dem Singleton-Entwurfsmuster (siehe Unterabschnitt 3.2.5.7) folgt (existieren mehrere Quellen für Nachrichten desselben Typs aber unterschiedlichen Inhalts, ist jeweils nur eine registriert). Zusammen mit der Tatsache, daß alle Nachrichtentypen und somit auch alle Typen von Nachrichten-Pools durch

[4] *A<B>* ist die in *C++* verwandte Schreibweise dafür, daß Klasse *A* mit Klasse *B* parametrisiert ist.

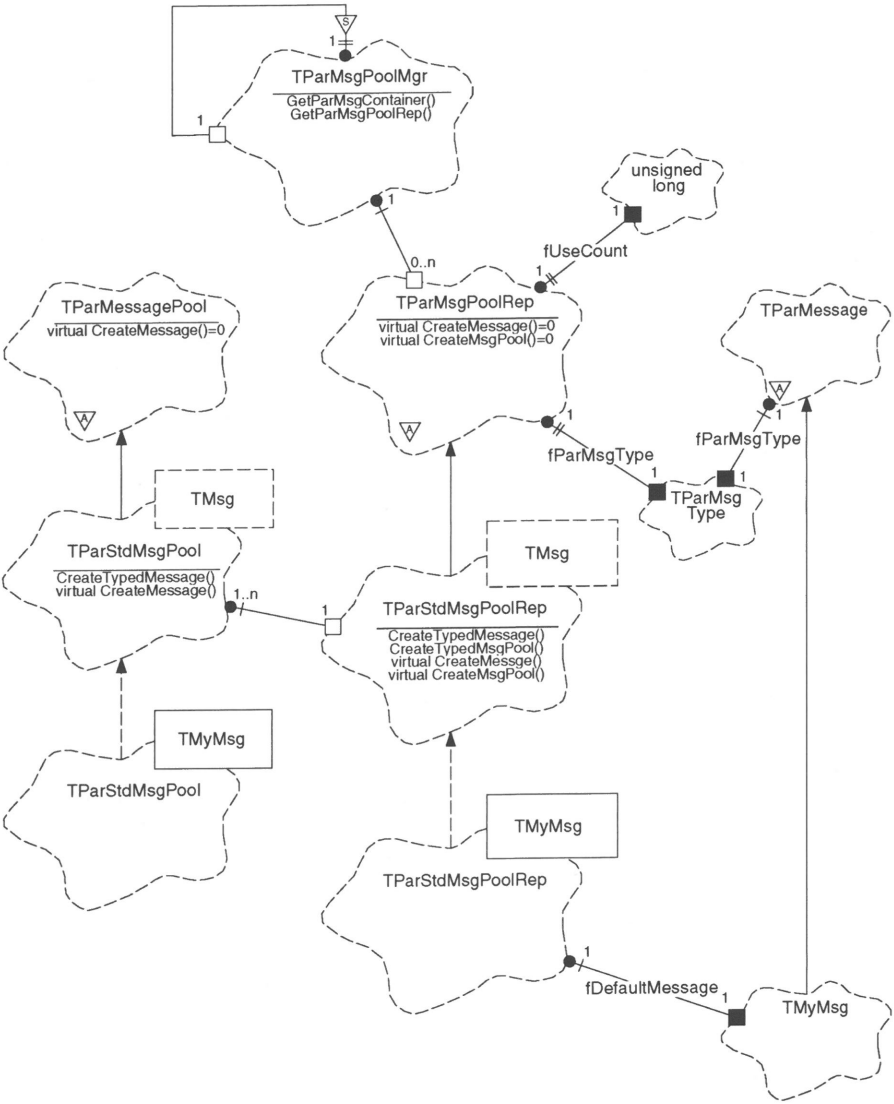

Bild 5.12: Nachrichtenverwaltung

einen global eindeutigen Bezeichner vom Typ *TParMsgType* unterscheidbar sind, kann *TParMsgPoolMgr* unter Angabe dieses Bezeichners über die Methode *GetParMsgContainer()* den zugehörigen Nachrichten-Pool identifizieren und das benötigte Nachrichtenobjekt erzeugen lassen. Deswegen wird der Nachrichten-Bezeichner mit jedem versandten Byte-Strom mitgeschickt, und es muß für jede Nachrichtenklasse *TMsg* in jedem Logischen Knoten eine Instanz von *TParStdMsgPoolRep<TMsg>* vorhanden sein.

Die Erzeugung einer Nachricht über die Handles geschieht durch Aufruf der reinen virtuellen Methode *TParMsgPool::CreateMessage()*, die den Aufruf über die abgeleitete und mit dem Typ der durch sie erzeugten Nachricht parametrisierten Klasse *TParStdMsgPool<TMsg>* schließlich an die Quelle *TParStdMsgPoolRep<TMsg>* selbst weiterleitet. Zur Erzeugung der Nachricht wird dort nach dem Prototypen-Entwurfsmuster (siehe Unterabschnitt 3.2.5.4) ein bei der Konstruktion des Pools initialisiertes Nachrichtenobjekt kopiert.

Obwohl der Gesamtmechanismus der Nachrichtenerzeugung recht komplex ist, ist er für den Anwender einfach zu benutzen. Er muß zur Erzeugung von Nachrichten des Typs *TMsg* lediglich einen Handle vom Typ *TParStdMsgPool<TMsg>* generieren. Die zugehörige Repräsentation wird dann automatisch referenziert, falls schon vorhanden, bzw. erzeugt und gegebenenfalls registriert, falls noch nicht vorhanden.

5.3.2.2 Datenströme

Wie schon mehrfach erwähnt, erfolgt das Versenden von Nachrichten zwischen Logischen Knoten mittels unstrukturierter Byte-Ströme. Die von einer Nachricht dafür zur Verfügung gestellten Pack- und Entpackmethoden bringen ihren inhaltlichen Kern in eine kompakte Form. Da dies u. U. auch für andere Mechanismen außer dem Versenden nützlich sein kann (z. B. zur Realisierung persistenter Nachrichten), wurde die Abstraktion des *Datenstroms* eingeführt.

Ein Datenstrom wird zunächst repräsentiert durch die abstrakte Basisklasse *TDataStream* (siehe Bild 5.13). Er bietet die Möglichkeit, über die reinen virtuellen Funktionen *WriteData()* und *ReadData()* beliebige Folgen von Bytes zu schreiben oder zu lesen. Datenströme können entweder beide Operationen oder auch nur eine von beiden zulassen. Was sich tatsächlich hinter einem Datenstrom verbirgt, wird erst in abgeleiteten Klassen festgelegt. *TFIFODataStream* stellt z. B. einen FIFO-Puffer dar, während *TMPSSReceiveBuffer* und *TMPSSSendBuffer* zur MPSS-Anpassungsschicht gehören und das Senden und Empfangen von in Byte-Ströme gewandelten Nachrichtenobjekten übernehmen. Letztere stellen dazu die zusätzlichen Methoden *SendStream()* bzw. *ReceiveStream()* (für nicht-blockierendes Empfangen) und *BlockedReceiveStream()* (für blockierendes Empfangen) zur Verfügung. *TMPSSReceiveBuffer::GetMessageInfo()* liefert Informationen über den zuletzt empfange-

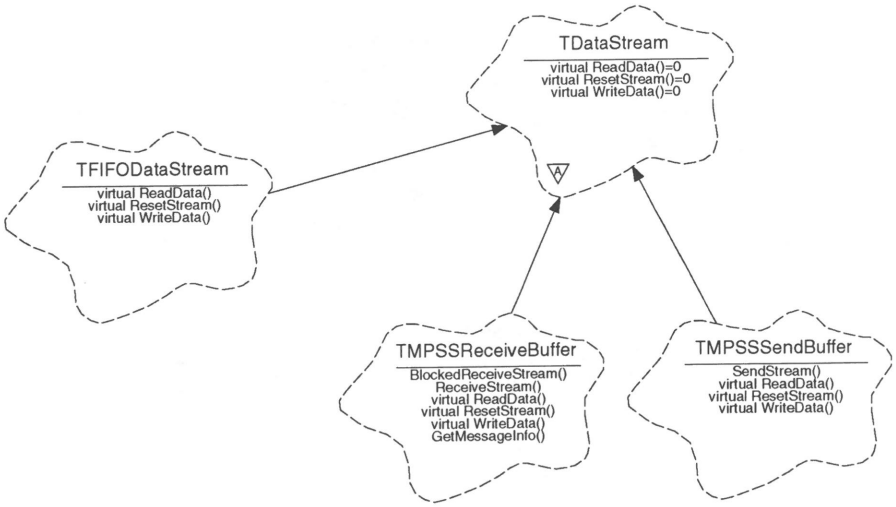

Bild 5.13: Klassen für Datenströme

nen Byte-Strom. Dazu gehört beispielsweise dessen Bezeichner, der zur Erzeugung eines geeigneten Nachrichtenobjekts benötigt wird.

Das Packen und Entpacken der Nachrichten geschieht in bzw. aus diesen abstrakten Datenströmen, ohne daß dazu deren Funktion bekannt sein muß. Zu bemerken ist noch, daß Datenströme immer unstrukturiert sind und insbesondere die gelesenen Daten richtig interpretiert werden müssen.

5.3.2.3 Management

Die zentrale Klasse für das Management der Kommunikationsschicht ist die Klasse *TCommHandler*, die u. a. Methoden für Senden und Rundsenden sowie blockierendes und nicht blockierendes Empfangen von Nachrichtenobjekten bietet (siehe Bild 5.14). Es handelt sich dabei um eine abstrakte Basisklasse, von der für den Manager-Knoten die Klasse *TMNCommHandler* und für alle übrigen Logischen Knoten die Klasse *TLNCommHandler* abgeleitet wird. Die abgeleiteten Klassen stellen zusätzliche Funktionalität für das Starten, Stoppen und die Steuerung der Logischen Knoten bereit.

Diese Klassen stellen zusammen mit dem Nachrichten-Pool-Manager *TParMsgPoolMgr* aus Unterabschnitt 5.3.2.1 und den Pack-, Entpack- und Handle-Methoden der Nachrichten selbst die Nachrichtenwandlungsschicht dar. Sie greifen auf die Klasse *TMPSSInterface* bzw. die davon abgeleiteten Klassen *TMNMPSSInterface* für den Manager-Knoten oder

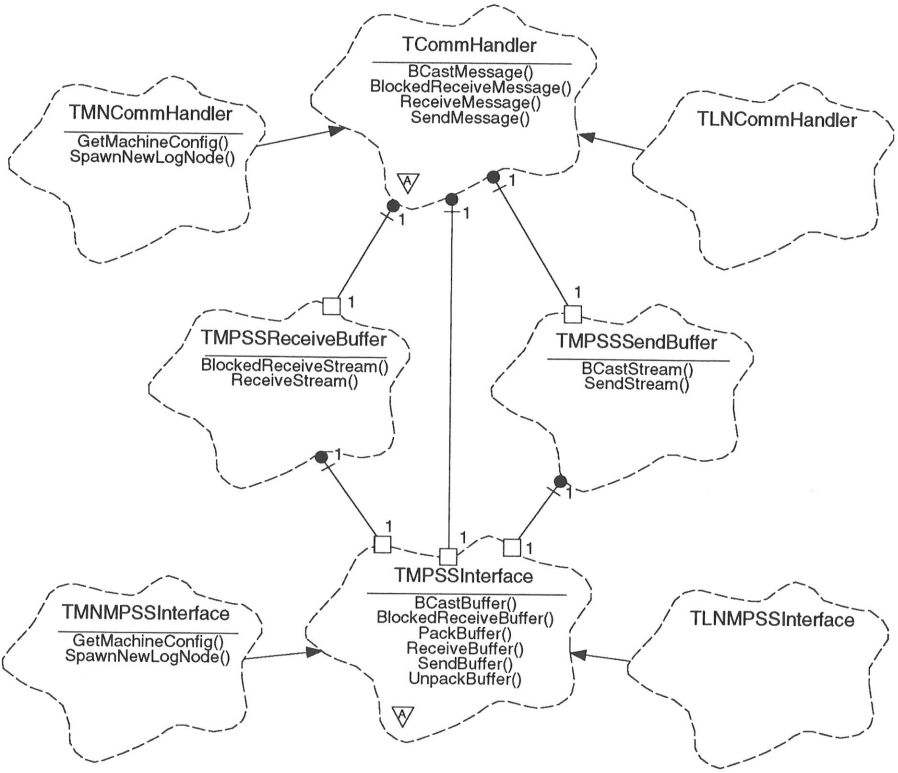

Bild 5.14: Management der Kommunikationsschicht

TLNMPSSInterface für die übrigen Logischen Knoten zu. Letztere bilden zusammen mit *TMPSSReceiveBuffer* und *TMPSSSendBuffer* die MPSS-Anpassungsschicht und stellen Methoden für das Senden und Empfangen von unstrukturierten Byte-Strömen und im Falle von *TMNMPSSInterface* auch für das Starten neuer Prozesse zur Verfügung. In jedem Logischen Knoten gibt es jeweils genau eine Instanz einer von *TCommHandler* und *TMPSSInterface* abgeleiteten Klasse.

5.3.3 Kommunikation zwischen Simulationsmodellkomponenten

5.3.3.1 Lokale Kommunikation

Simulationsmodellkomponenten, die sich im selben Logischen Prozeß befinden, benutzen zur Kommunikation untereinander das in [Kocher,1994] vorgestellte Port-Konzept, dessen

Bild 5.15: Handshake-Protokoll zwischen Ports

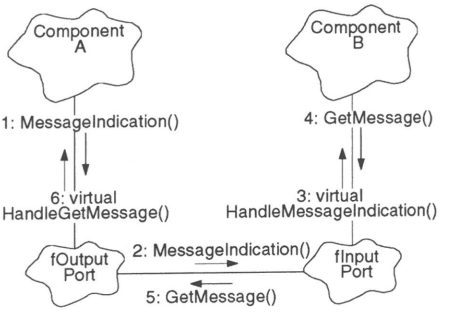

Bild 5.16: Kommunikation zwischen lokalen Komponenten

Grundzüge in Bild 5.15 dargestellt sind. Miteinander kommunizierende Komponenten bauen unidirektionale Punkt-zu-Punkt-Verbindungen über Ein- und Ausgangsports auf. Zwischen diesen Ports läuft ein Handshake-Protokoll für den Nachrichtenaustausch ab, wodurch sich unterschiedliche Komponenten einfach nach dem Baukastenprinzip kombinieren lassen. Die vollständige Übernahme dieses Konzepts kommt einem einfachen Übergang von sequentieller zu paralleler Simulation entgegen.

In Bild 5.16 sind beispielhaft die Abläufe dargestellt für den Fall, daß Komponente A eine Nachricht an Komponente B sendet. Komponente A zeigt den Sendewunsch mittels *MessageIndication()* ihrem Ausgangsport an. Dieser meldet dies dem mit ihm verbundenen Eingangsport, welcher wiederum Komponente B davon durch Aufruf von deren *HandleMessageIndication()*-Methode in Kenntnis setzt. Im in Bild 5.16 vorliegenden Fall holt Komponente B daraufhin die Nachricht sofort durch Anstoßen der *GetMessage()*-Aufruffolge. Könnte dagegen Komponente B die Nachricht momentan nicht aufnehmen (z. B. weil es sich um eine Bedieneinheit handelt, die gerade belegt ist), würde sie erst später, wenn dies wieder möglich ist, ihren Eingangsport veranlassen, mittels *IsMessageAvailable()* bei Komponente A anzufragen, ob eine Nachricht verfügbar ist, und diese dann holen.

5.3.3.2 Kommunikation über Kanäle

Simulationsmodellkomponenten, die sich in unterschiedlichen Logischen Prozessen befinden, können nur über Kanäle miteinander kommunizieren (siehe Unterkapitel 4.5). Bild 5.17 zeigt wiederum die Verhältnisse, wenn Komponente A Nachrichten an Komponente B versendet. Komponente A befindet sich dabei zusammen mit dem Ausgangskanal *OutChannel* in einem Logischen Prozeß und Komponente B zusammen mit dem Eingangskanal *InChannel* in einem anderen. Es ist zu sehen, daß die Kanäle auch über Ports mit den Komponenten verbunden sind, wodurch ihr Zwischenschalten für die Komponenten transparent geschieht.

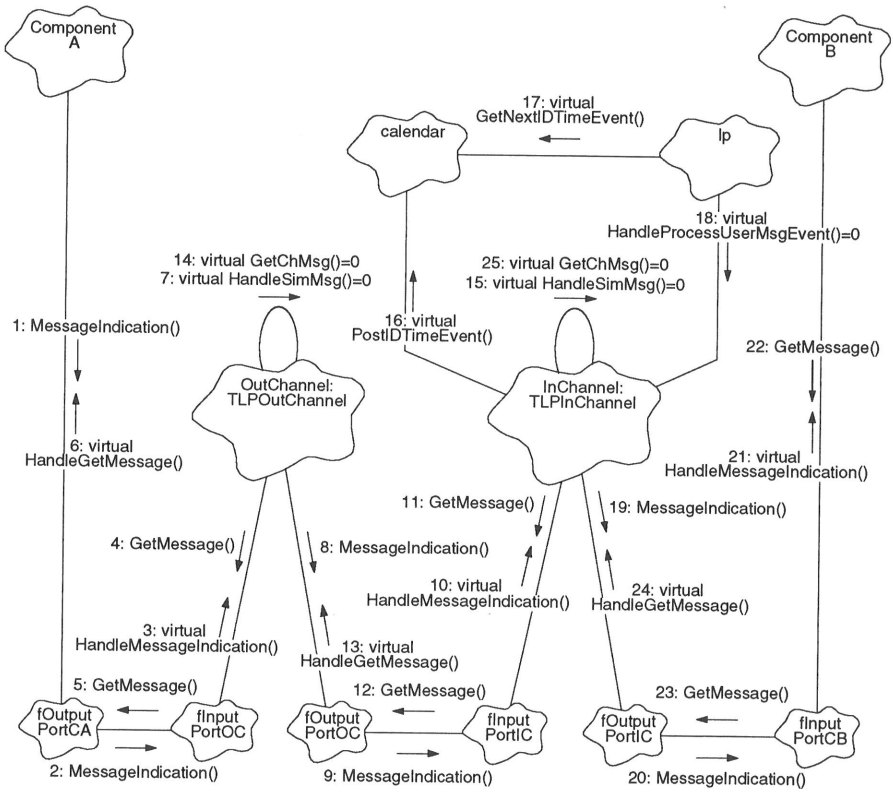

Bild 5.17: Kommunikation zwischen Komponenten über Kanäle

Für den Ausgangsport von Komponente A ist es irrelevant, ob er direkt mit dem Eingangsport von Komponente B oder mit dem des Ausgangskanals verbunden ist – er wickelt seinen Nachrichtenaustausch über das gleiche Handshake-Protokoll ab.

Wie in Abschnitt 5.2.3 schon erwähnt, wird beim Eintritt einer Nachricht in den Kanal die reine virtuelle Methode *HandleSimMsg()* und beim Verlassen *GetChMsg()* aufgerufen. Wäre *OutChannel* in Bild 5.17 z. B. ein Kanal für optimistische Synchronisation, so wäre die *HandleSimMsg()*-Methode derart überschrieben, daß alle versandten Nachrichten gespeichert und bei einem eventuellen Zurücksetzen des Logischen Prozesses geeignete Anti-Nachrichten versandt werden. Außerdem würden Nachrichten, die während einer Coast-Forward-Phase erzeugt werden, nicht versandt (siehe Paragraph 2.5.2.3.1).

Nach Bearbeitung der Nachricht durch den Ausgangskanal *OutChannel* wird diese an den Eingangskanal *InChannel* weitergeleitet. Auch hier werden die reinen virtuellen Methoden

HandleSimMsg()[5] und *GetChMsg()* aufgerufen. Die durch einen Eingangskanal für optimistische Synchronisation überschriebene *HandleSimMsg()*-Methode würde z. B. die Zeitstempel der eingehenden Simulationsnachrichten prüfen und bei Bedarf ein Zurücksetzen des empfangenden LPs auslösen. In der *HandleSimMsg()*-Methode werden die Simulationsnachrichten in der Regel in eine Warteschlange eingereiht und anschließend ein Ankunftsereignis mit dem Zeitstempel der Nachricht in den Kalender eingetragen (nur falls die Nachricht die erste in der Warteschlange ist). Der LP wird dieses Ereignis bearbeiten, sobald die entsprechende lokale Zeit erreicht ist. Die Bearbeitung des Ereignisses veranlaßt *InChannel*, die zugehörige Nachricht aus der Eingangswarteschlange zu holen und an Komponente B weiterzuleiten.

Für die beiden Komponenten A und B besteht kein Unterschied, ob sie direkt oder über Kanäle miteinander kommunizieren. Mit einer Einschränkung: Da die einzelnen Logischen Prozesse asynchron sind, läßt sich bei Empfang einer Nachricht im Eingangskanal nicht feststellen, ob die Zielkomponente zum entscheidenden Zeitpunkt diese Nachricht auch aufnehmen kann. Könnte sie dies nicht, müßte eine Rückmeldung an den Sender erfolgen und dieser seinerseits u. U. wieder zurücksetzen. Ein derartiges Zurücksetzen des Senders würde aber die Leistungsfähigkeit des Gesamtsystems nachhaltig negativ beeinflussen. Da bei konservativen Verfahren kein Zurücksetzen vorgesehen ist, müßte der Sender hier sogar warten, bis die lokale Zeit des empfangenden Logischen Prozesses den Zeitstempel der Nachricht erreicht hat. Dies würde zu einer unnötigen und ebenfalls leistungsmindernden Synchronisation der beiden Logischen Prozesse führen. Aus diesen Gründen ist das Blockieren durch die empfangende Komponente bei Kommunikation über Kanäle nicht erlaubt. Bei geeigneter Partitionierung des Simulationsmodells stellt dies aber kein Problem dar.

Durch die Anbindung der Simulationsmodellkomponenten an die Kanäle über Ports und die Abwicklung des Nachrichtenaustauschs mit Hilfe des gewohnten Handshake-Protokolls wird die Transparenz von lokaler und nicht-lokaler Kommunikation erreicht. Die Mechanismen zur Kommunikation zwischen Logischen Prozessen sind vom Simulationsmodell getrennt. Die spezifischen Mechanismen zur Synchronisation verbergen sich in den virtuellen Methoden der Kanäle – ein sehr gutes Beispiel der Anwendung des Entwurfsmusters der Schablonen-Methode (siehe Unterabschnitt 3.2.5.8). Durch einfache Ableitung geeigneter Kanal-Klassen können verschiedene Synchronisationsverfahren unterstützt werden, ohne daß sich an den in Bild 5.17 gezeigten Abläufen etwas ändert – womit auch die Kommunikations- von den Synchronisationsmechanismen entkoppelt sind. Somit werden durch die beschriebenen Abläufe einige der Hauptforderungen aus Abschnitt 4.6.1 erfüllt.

In diesem Abschnitt wurden die Verhältnisse beschrieben, wenn sich beide Logische Prozesse im selben Logischen Knoten befinden. In diesem Fall lassen sich die Ports der beiden be-

[5] Sofern es sich, wie in diesem Fall, um eine Simulationsnachricht handelt. Handelt es sich dagegen um eine Kanalsteuernachricht, wird die hier nicht weiter vorgestellte virtuelle Methode *HandleCtrlMsg()* aufgerufen.

teiligten Kanäle direkt verbinden. Geht die Verbindung dagegen zusätzlich über die Grenze eines Logischen Knotens, so müssen entsprechende Proxies erzeugt werden.

5.3.3.3 Kommunikation zwischen Logischen Knoten

Kommunikation zwischen Logischen Knoten wird mit Hilfe von Proxies abgewickelt (siehe Abschnitt 4.6.2). Bild 5.18 zeigt deren Klassenhierarchie. Der Zugriff auf die Proxies erfolgt

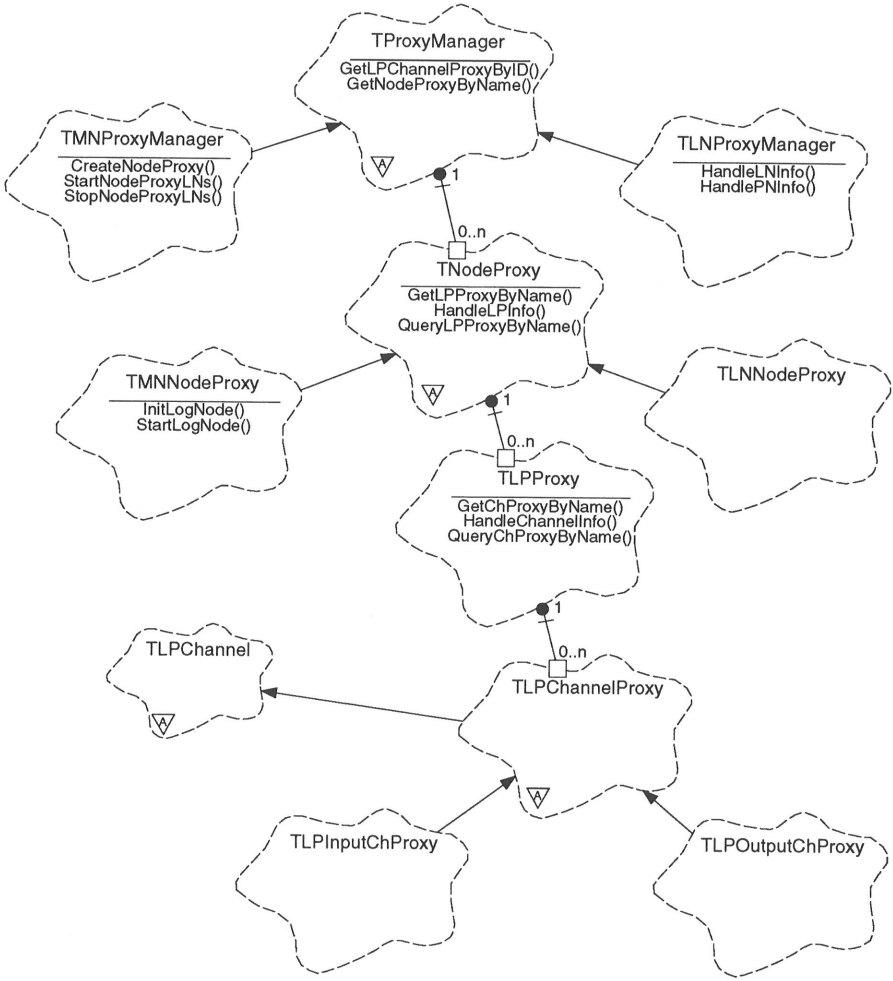

Bild 5.18: Hierarchie der Proxy-Klassen

in jedem LN über einen zentralen Proxy-Manager. Im Manager-Knoten ist dies eine Instanz der Klasse *TMNProxyManager*, in den übrigen Logischen Knoten eine Instanz von *TLNProxyManager*. Beide sind dabei abgeleitet von der abstrakten Basisklasse *TProxyManager*.

Zu Beginn wird im Manager-Knoten durch Aufruf von *TMNProxyManager::CreateNodeProxy()* für jeden Logischen Knoten im System eine Instanz der Klasse *TMNNodeProxy* erzeugt. Diese enthält z. B. dessen Namen, welches ausführbare Programm bei seinem Start geladen werden soll, einen systemweit eindeutigen Bezeichner und andere charakteristische Merkmale. Sind alle diese Proxies erzeugt, werden die entsprechenden Logischen Knoten durch Aufruf von *TMNProxyManager::StartNodeProxyLNs()* auf die vorgesehenen Physikalischen Knoten verteilt, jeweils durch Aufruf von *TMNNodeProxy::StartLogNode()* gestartet und mittels *TMNNodeProxy::InitLogNode()* initialisiert. Dazu werden Initialisierungsnachrichten mit Informationen über alle Physikalischen und Logischen Knoten im System versandt, die von den Methoden *HandlePNInfo()* und *HandleLNInfo()* bearbeitet werden.

Die Proxies für die Logischen Prozesse (*TLPProxy*) sind über die Proxies der Logischen Knoten zugänglich, in denen sich die jeweils entsprechenden Logischen Prozesse tatsächlich befinden. Auf die gleiche Weise sind die Kanal-Proxies (*TLPChannelProxy*) über die Proxies ihrer jeweiligen Logischen Prozesse erreichbar. Durch diese Hierarchie der Proxies wird die tatsächliche Hierarchie im System in jedem Logischen Knoten nachgebildet. *TProxyManager::GetLPChannelProxyByID()* stellt eine „Abkürzung“ dar, durch die Kanal-Proxies direkt, unter Umgehung der Proxies für LNs und LPs, über den Proxy-Manager erreicht werden können. Dazu hält dieser zur beschleunigten Bearbeitung empfangener Kanalnachrichten Referenzen auf die Kanal-Proxies in einem Pufferspeicher.

Während sich in jedem Logischen Knoten für jeden anderen LN immer ein Proxy befindet, werden Proxies für Logische Prozesse und Kanäle nur bei Bedarf erzeugt, d. h. wenn eine entsprechende Kommunikationsbeziehung unterhalten wird. Damit ist gewährleistet, daß der Ansatz auch für große Systeme mit vielen LPs skalierbar ist. Mit *TNodeProxy::QueryLPProxyByName()* und *TLPProxy::QueryChProxyByName()* werden benötigte Proxies erzeugt und alle hierzu erforderlichen Informationen automatisch vom jeweiligen Original angefordert (s. a. Abschnitt 4.6.2 für ein Beispiel). Mittels *TNodeProxy::GetLPProxyByName()* und *TLPProxy::GetChProxyByName()* kann auf bereits erzeugte Proxies zugegriffen werden.

Bei der Übertragung von Nachrichten sind nun die Kanäle nicht direkt mit ihrem Gegenüber, sondern mit dessen Proxy verbunden. Bild 5.19 zeigt die Abläufe beim Senden einer Simulationsnachricht. Für den Ausgangskanal ist überhaupt nicht erkennbar, daß er nicht mit einem Eingangskanal sondern mit dessen Proxy verbunden ist. Die zu sendende Nachricht *msg* wird vom Eingangskanal-Proxy durch Aufruf von *SendMessage()* an *gCommHandler* übergeben. Dieser setzt zuerst den Sendepuffer zurück und bereitet ihn damit auf die Aufnahme einer neuen Nachricht vor. Diese wird in den Schritten 4 bis 9 in einen unstrukturierten Byte-Strom umgewandelt und in den Puffer geschrieben.

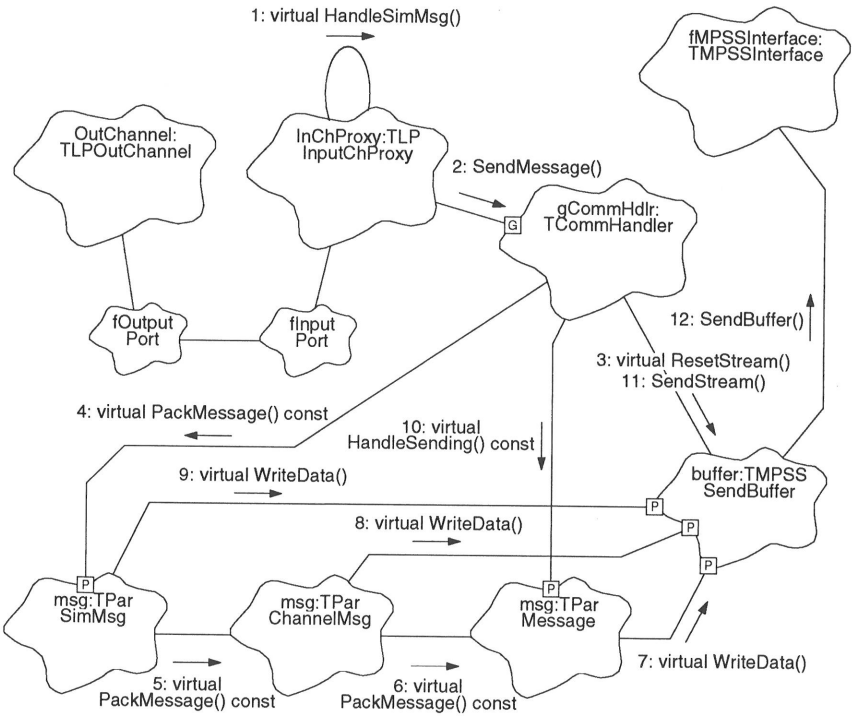

Bild 5.19: Versenden einer Simulationsnachricht über einen Kanal-Proxy

Dazu wird der Aufruf der virtuellen Methode *msg.PackMessage()*[6] von der am weitesten abgeleiteten Klasse stufenweise an ihre Basisklassen zurückgereicht[7]. Jede dieser Klassen schreibt dann mittels *buffer.WriteData()* der Reihe nach ihre Daten in den Puffer. Nach Aufruf der virtuellen *HandleSending()*-Methode von *TParMessage* wird schließlich der im Sendepuffer gespeicherte Byte-Strom durch *buffer.SendStream()* und *fMPSSInterface.SendBuffer()* über das Message-Passing-System der Rechenplattform an den Empfänger versandt.

Die Abläufe auf Empfängerseite zeigt Bild 5.20. In regelmäßigen Abständen empfängt der Logische Knoten durch Aufruf von *gCommHandler.ReceiveMessage()* Nachrichten (s. a. Unterabschnitt 5.2.5.2). Dabei wird über den Empfangspuffer ein evtl. am Rechenknoten angekommener und durch das Message-Passing-System zwischengespeicherter Byte-Strom

[6] *o.m()* ist die in *C++* verwandte Schreibweise für den Aufruf der Methode *m()* des Objekts *o*.

[7] Im Bild ist die am weitesten abgeleitete Klasse *TParSimMsg*. Da es sich dabei um eine abstrakte Basisklasse handelt, kann kein Objekt dieses Typs existieren. Aus Übersichtlichkeitsgründen wurde aber die Aufruffolge für weiter abgeleitete Klassen nicht eingezeichnet.

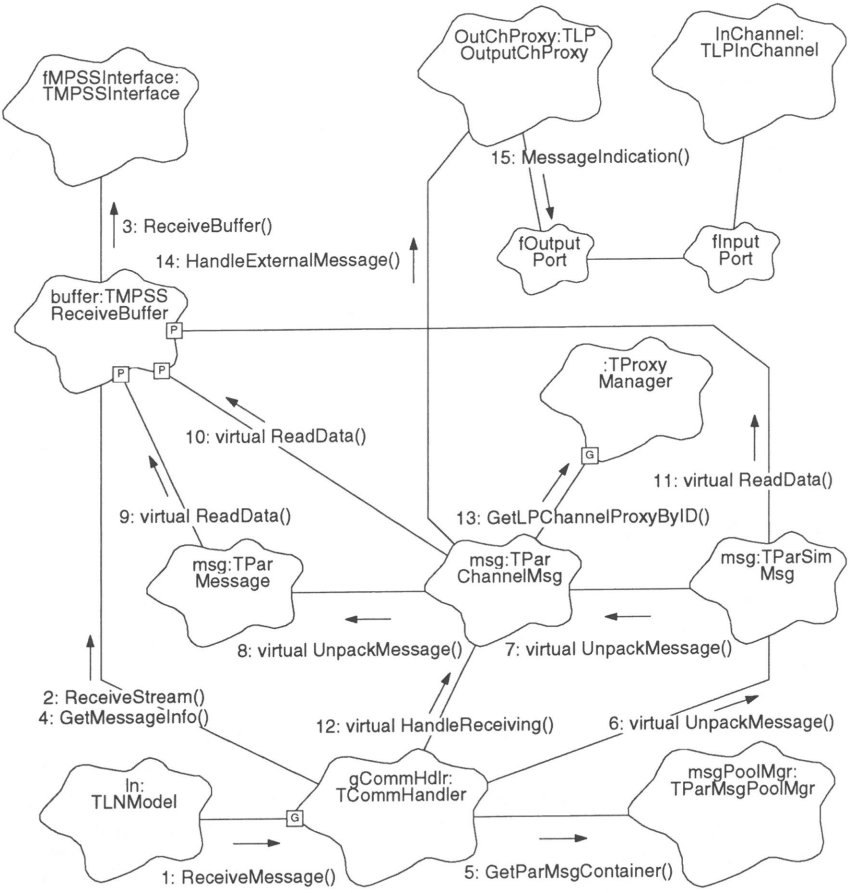

Bild 5.20: Empfangen einer Simulationsnachricht über einen Kanal-Proxy

empfangen. *buffer.GetMessageInfo()* liefert den Bezeichner des zuletzt empfangenen Byte-Stroms, mit dessen Hilfe *TParMsgPoolMgr::GetParMsgContainer()* ein geeignetes (leeres) Nachrichtenobjekt *msg* erzeugt. In den Schritten 6 bis 11 wird dieses Nachrichtenobjekt entsprechend der Vorgehensweise beim Senden durch hierarchisch geschachtelten Aufruf von *UnpackMessage()* mit den auf Senderseite verpackten Inhalten gefüllt. Die Daten werden dazu durch *buffer.ReadData()* aus dem Empfangspuffer ausgelesen und anschließend den entsprechenden Datenfeldern von *msg* zugewiesen.

Nachdem das Nachrichtenobjekt auf diese Weise auf der Empfängerseite wiederhergestellt ist, überläßt *gCommHandler* durch Aufruf der virtuellen Methode *msg.HandleReceiving()*

der Nachricht selbst, was weiter mit ihr zu geschehen hat. Im Falle der in Bild 5.20 gezeigten Simulationsnachricht wird dabei *TParChannelMsg::HandleReceiving()* ausgeführt, die über den Proxy-Manager den geeigneten Kanal-Proxy bestimmt, welcher die Nachricht schließlich nach Aufruf von *OutChProxy.HandleExternalMessage()* an seinen Ausgangsport weiterleitet. Von da an läuft wieder das übliche Handshake-Protokoll zwischen Ausgangs- und Eingangsport ab, so daß der an den Kanal-Proxy angeschlossene Eingangskanal nicht bemerkt, daß die Nachricht nicht direkt von einem Ausgangskanal sondern von dessen Proxy stammt.

Im Falle von Steuer- und Kanalsteuernachrichten laufen die oben beschriebenen Vorgänge der Nachrichtenwandlung sowie des Sendens und Empfangens gleich ab. Lediglich die Behandlung durch den Kanal-Proxy ist bei Steuernachrichten nicht vorhanden. Stattdessen werden die nach dem Empfang auszuführenden Aktionen in der unterschiedlich implementierten *HandleReceiving()*-Methode festgelegt.

5.3.4 Kanalfilter

Einige Verfahren für die verteilte Simulation, wie beispielsweise das in Unterabschnitt 2.5.2.4 erwähnte Verfahren zur Approximation der GVT aus [Mattern,1993], benötigen Informationen über Simulationsnachrichten, die über die Kanäle versandt werden, oder müssen mit jeder dieser Nachrichten Zusatzinformationen von Kanal zu Kanal versenden. Zu diesem Zweck können an den Kanälen dynamisch *Kanalfilter* angebracht werden, durch die die Nachricht, wie in Bild 5.21 gezeigt, zusätzlich geleitet wird (s. a. Unterabschnitt 4.6.4.2). Diese Filter können die Nachrichten bearbeiten und Zusatzinformationen in Form von *Nachrichtenmarkierungen* anheften. Zusätzlich zu den in Bild 5.21 gezeigten Filtern auf der jeweils vom Logischen Prozeß abgehenden externen Seite des Kanals können auch noch Filter auf seiner den Simulationsmodellkomponenten zugewandten internen Seite angebracht werden (aus Übersichtlichkeitsgründen nicht dargestellt). Kanalfilter stellen wiederum eine Umsetzung des Entwurfsmusters der „Kette der Verantwortlichkeit“ dar (siehe Unterabschnitt 3.2.5.5).

An jede Kanalnachricht können beliebig viele, von der abstrakten Basisklasse *TChMsgMarker* abgeleitete Markierungen (siehe Bild 5.22) angeheftet werden, die wiederum beliebige Daten enthalten können. Sie werden automatisch mit der Nachricht mitversandt, wozu ähnliche Mechanismen wie für die Nachricht selbst verwandt werden. D. h. die reinen virtuellen Methoden *PackThisMsgMarker()* und *UnpackThisMsgMarker()* müssen in abgeleiteten Klassen überschrieben werden und führen die Umwandlung des Objekts in einen unstrukturierten Byte-Strom und umgekehrt durch. Ähnlich der Klasse *TParMsgPoolMgr* existiert eine zentrale Instanz der Klasse *TChMsgMarkerManager*, die bei Bedarf eine (leere) Markierung erzeugen kann, die danach mit Teilen des empfangenen Byte-Stroms aufgefüllt wird.

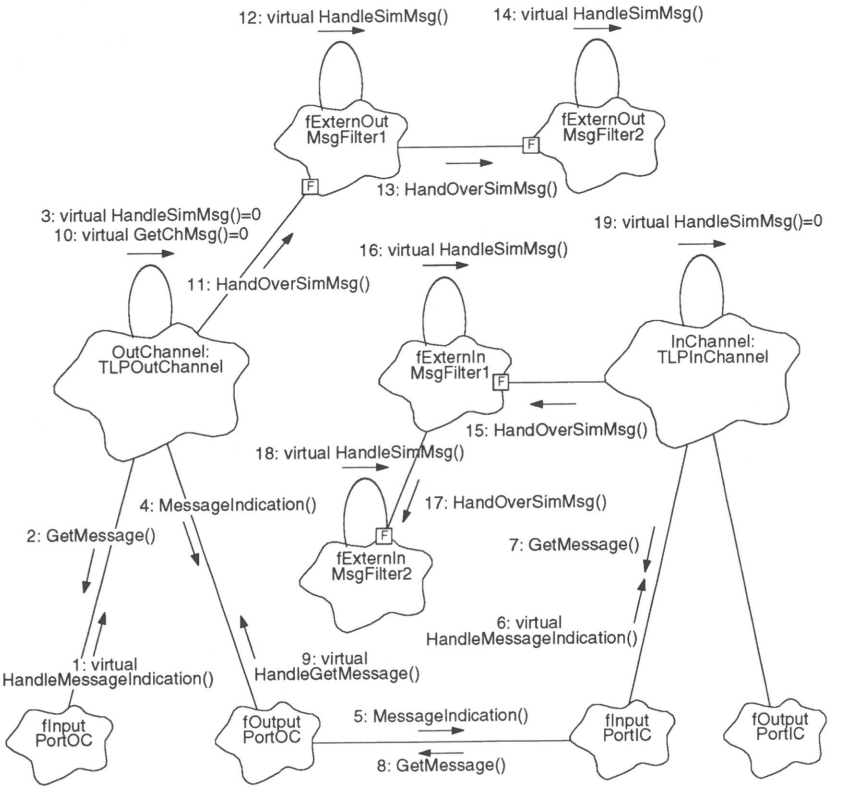

Bild 5.21: Bearbeiten versandter Nachrichten durch Kanal-Filter

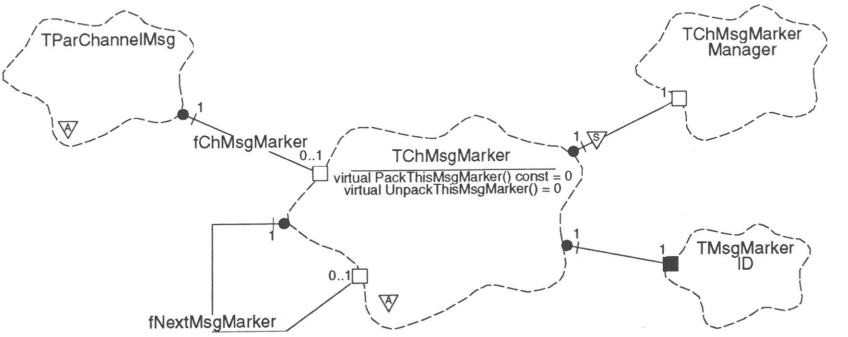

Bild 5.22: Übertragung von Zusatzinformation mittels Nachrichtenmarkierungen

Durch das einfache Anbringen von Kanalfiltern an den Ausgangs- und Eingangskanälen, die über das Anheften von Markierungen an Kanalnachrichten Informationen austauschen können, sind beliebige Mechanismen der parallelen Simulation in der Lage, auf den Austausch von Simulationsnachrichten zu reagieren, ohne in diesen eingreifen oder die Kanäle oder Nachrichten verändern zu müssen.

5.4 Kapselung der Zustandssicherungsmechanismen

5.4.1 Konzept

Optimistische Synchronisationsverfahren erfordern Mechanismen für die Zustandssicherung. Die Grundgedanken des im hier vorgestellten Werkzeug realisierten Konzepts wurden in Unterkapitel 4.7 bereits erläutert. Die Grundidee ist es, daß jede Simulationsmodellkomponente selbst verantwortlich ist für die Sicherung ihres eigenen Zustands, diese jedoch gesteuert wird von einem zentralen Manager. Durch diese dezentrale Sicherung bei gleichzeitiger zentraler Steuerung können unterschiedliche Sicherungsverfahren flexibel eingesetzt werden und im selben Modell koexistieren. Die Zustandssicherung kann leider nicht vollständig unabhängig von der jeweiligen Simulationsmodellkomponente erfolgen, da nur diese selbst „weiß", welche ihrer Daten zeitveränderlich sind und deshalb gesichert werden müssen. Bei dem vorgestellten Konzept wurde aber darauf geachtet, daß die Eingriffe in die Komponenten so gering wie möglich gehalten werden können.

5.4.2 Klassenhierarchie

Bild 5.23 zeigt die wesentlichen Bausteine des Konzepts anhand eines Klassendiagramms. Jede Simulationsmodellkomponente, die an der Zustandssicherung teilnehmen möchte – im Bild symbolisiert durch die Klasse *TMyTWComponent* – muß von der abstrakten Basisklasse *TStateSaveCtrl* abgeleitet sein. Diese wiederum enthält eine Referenz auf ein „Sicherungsobjekt", eine Instanz einer von der abstrakten Basisklasse *TStateSaver* abgeleiteten Klasse. Diese Referenz wird bei der Konstruktion der Komponente initialisiert, und die Art der Zustandssicherung für diese Komponente wird durch den Typ dieses Sicherungsobjekts bestimmt. Im Bild sind die Klassen *TCopyStateSaver* für die Erstellung von Zustandskopien und *TIncStateSaver* für inkrementelle Zustandssicherung dargestellt. Diese beiden Klassen bedienen sich der abstrakten Basisklassen *TStateCopy* und *TStateInc*, um die gesicherten Daten abzuspeichern. Eine von *TStateCopy* abgeleitete Klasse (*TMyStateCopy* im Beispiel) enthält dabei den kompletten zu sichernden Zustand einer Komponente zu einem Zeitpunkt, während von *TStateInc* abgeleitete Klassen Zustandsänderungen der Komponente speichern.

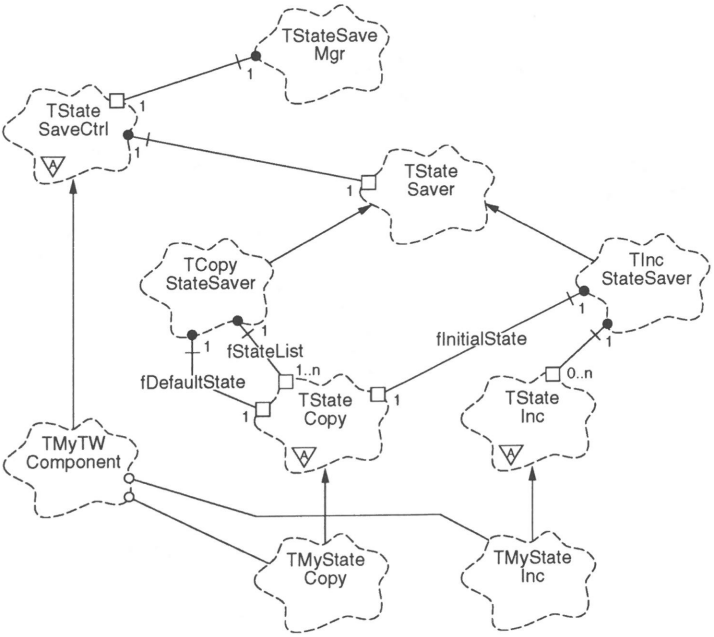

Bild 5.23: Klassenhierarchie für Zustandssicherung

TCopyStateSaver hält eine Liste von Zustandskopien samt der Zeitspannen, während derer sie jeweils gültig sind, sowie einen „Basiszustand“ *fDefaultState*, von dem durch Kopieren und anschließende Sicherung die jeweiligen Zustandskopien erstellt werden (entsprechend dem Prototypen-Entwurfsmuster aus Unterabschnitt 3.2.5.4). *TIncStateSaver* hält eine Liste von Zustandsänderungen mit den Zeitpunkten, zu denen sie aufgetreten sind, und einen „Initialzustand“ für den Ausgangszustand der Komponente bei deren Erzeugung.

Alle Objekte der von *TStateSaveCtrl* abgeleiteten Klassen werden in jedem Logischen Prozeß von einem zentralen Sicherungs-Manager, einem Objekt der Klasse *TStateSaveMgr* verwaltet und gesteuert. Ungeachtet der Art und Weise wie die einzelne Komponente ihren Zustand sichert, muß die zentrale Steuerung sicherstellen, daß alle Komponenten in einem Logischen Prozeß ihren Zustand zu von ihr bestimmten Zeitpunkten („Sicherungsaufpunkte“) wiederherstellen können. Es muß zum einen eine regelmäßige Zustandssicherung durchgeführt werden, zum anderen können aber auch andere Ereignisse, wie z. B. der Beginn eines neuen Teiltests, eine Sicherung auslösen. Aufgrund der unterschiedlichen Ursachen, die das Auslösen einer Sicherung haben kann, und aufgrund der vielfältigen Möglichkeiten, das Zustandssicherungsintervall zu bestimmen, werden Sicherungen durch sog. *Trigger* ausgelöst (siehe Bild 5.24). Diese Trigger sind von der abstrakten Basisklasse *TStateSaveTrigger*

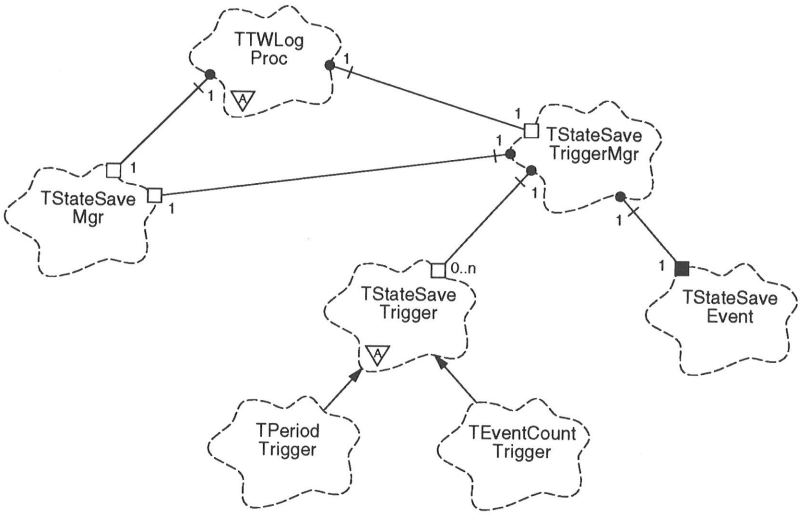

Bild 5.24: Klassenhierarchie für Auslösen einer Zustandssicherung

abgeleitet und werden vom Trigger-Manager *TStateSaveTriggerMgr* verwaltet. Sie registrieren sich dort dynamisch und können fortan Zustandssicherungen zu von ihnen gewünschten Zeitpunkten anfordern. Der Trigger-Manager koordiniert die von verschiedenen Triggern eingehenden Anforderungen und plant seinerseits entsprechende Ereignisobjekte vom Typ *TStateSaveEvent* im Kalender des Logischen Prozesses ein. Die im Bild gezeigten Klassen *TPeriodTrigger* und *TEventCountTrigger* stehen für eine zeitperiodische Sicherung alle x Zeiteinheiten bzw. alle y bearbeitete Ereignisse. Weitere Trigger, die z. B. das Zustandssicherungsintervall adaptiv anpassen, lassen sich einfach einbinden.

Eine Zustandssicherung zum Zeitpunkt T_{SS} wird vereinbarungsgemäß immer durchgeführt, bevor irgendein anderes Ereignis zu diesem Zeitpunkt bearbeitet wird. Veranlaßt der Trigger-Manager eine Zustandssicherung, so wird dies über *TStateSaveMgr* allen registrierten Objekten des Typs *TStateSaveCtrl* und deren angegliederten Objekten vom Typ *TStateSaver* bekanntgemacht. Wie diese dann gewährleisten, daß sie den Zustand der durch sie gesicherten Komponente zu diesem Zeitpunkt wiederherstellen können, bleibt ihnen überlassen. Details zu den Abläufen finden sich in Abschnitt 5.4.4.

5.4.3 Schnittstelle zur Simulationsmodellkomponente

Eine Simulationsmodellkomponente, die an der Zustandssicherung teilnehmen möchte, muß dazu folgende Voraussetzungen erfüllen:

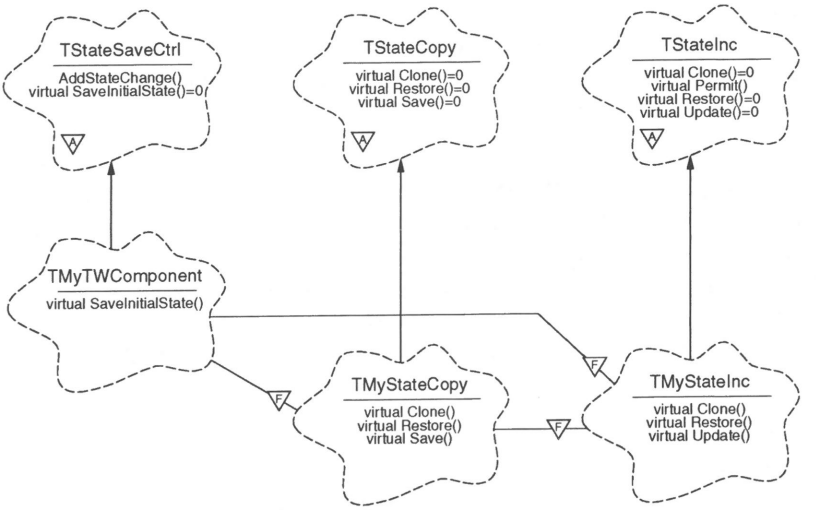

Bild 5.25: Schnittstelle der Zustandssicherung zu den Modellkomponenten

- Sie muß von der Klasse *TStateSaveCtrl* abgeleitet werden und dieser ein Sicherungsobjekt übergeben, das den gewünschten Sicherungsalgorithmus implementiert. In der Regel wird dazu ein bereits vorhandener Mechanismus herangezogen, so daß der Anwender diesen nicht extra implementieren muß.
- Sie muß von *TStateCopy* und *TStateInc* abgeleitete Klassen für die Erstellung von Zustandskopien und die Erfassung von Zustandsänderungen zur Verfügung stellen.
- Sie muß Änderungen ihres internen Zustands an das Sicherungsobjekt weiterleiten.

Bild 5.25 zeigt die für den Simulationsanwender wichtige Schnittstelle zwischen Modellkomponente und Zustandssicherungsmechanismus. Ein Objekt der vom Anwender zur Verfügung zu stellenden Klasse *TMyStateCopy* speichert den gesamten Zustand der Komponente *TMyTWComponent* ab. Es überschreibt die virtuellen Methoden *Save()* und *Restore()*, mit denen alle zu sichernden Daten von *TMyTWComponent* in die entsprechenden Felder von *TMyStateCopy* übertragen bzw. von dort zurückgeschrieben werden. Jeder Zustand ist also gewissermaßen für seine eigene Sicherung und Wiederherstellung selbst verantwortlich und stellt somit in Erweiterung des passiven Memento-Entwurfsmusters aus Unterabschnitt 3.2.5.6 eine Art „aktives Memento“ dar.

Für jede mögliche Änderung des Zustands der Komponente – z. B. für die Ankunft einer Nachricht – muß der Anwender außerdem eine von *TStateInc* abgeleitete Klasse zur Verfügung stellen. Diese implementiert zum einen die reine virtuelle Methode *Restore()*, bei deren Aufruf die Zustandsänderung, die durch das referenzierte Objekt repräsentiert

wird, rückgängig gemacht wird, und zum anderen *Update()*, mit der ein Zustand vom Typ *TMyStateCopy* aufgrund der Änderung aktualisiert wird. Auch die Zustandsänderungen sind also selbst für die entsprechenden Änderungen in der Komponente beim Zurücksetzen verantwortlich.

Durch die Verlagerung der meisten Sicherungs- und Wiederherstellungsmechanismen in externe Klassen werden die Eingriffe in die Simulationsmodellkomponente selbst minimiert. Solche Eingriffe sind lediglich an den Stellen erforderlich, an denen sich der interne Zustand der Komponente ändert. Dort muß ein der Änderung entsprechendes Objekt vom Typ *TMyStateInc* erzeugt und durch Aufruf von *TStateSaveCtrl::AddStateChange()* an das Sicherungsobjekt weitergeleitet werden. Dieses kann diese Änderung je nach Sicherungsverfahren in einer Liste abspeichern, eine intern gehaltene Zustandskopie aktualisieren oder einfach ignorieren.

Aufgrund der sehr engen Kopplung der Klassen für die Zustandskopie und die Zustandsänderungen mit der Komponente selbst haben diese in der Regel über *Friend*-Deklarationen direkten Zugriff auf deren interne Daten. Zu erwähnen ist auch noch, daß die Komponente mittels *SaveInitialState()* ihren Initialzustand speichert, von *TStateCopy* und *TStateInc* abgeleitete Klassen die sehr einfache *Clone()*-Methode zur Erstellung einer exakten Kopie implementieren müssen und Zustandsänderungsobjekte durch Überschreiben der *TStateInc::Permit()*-Methode sich davon in Kenntnis setzen lassen können, wann die Änderung „sicher“ ist, d. h. eine evtl. spekulative Simulation sich endgültig als richtig erwiesen hat.

Vereinfachungen des Konzepts lassen sich dann erreichen, wenn eine Komponente nur für ein ganz bestimmtes Zustandssicherungsverfahren vorbereitet werden soll. Ist z. B. auf jeden Fall nur die Verwendung von Zustandskopien erwünscht, so kann die Implementierung der Klassen für die Zustandsänderungen rudimentär erfolgen. Ist nur inkrementelle Sicherung erwünscht, gilt das gleiche für die Implementierung einer Klasse *TMyStateCopy*.

Im folgenden Abschnitt werden einige ausgewählte Abläufe bei der Zustandssicherung näher erläutert.

5.4.4 Abläufe

5.4.4.1 Sichern von Zuständen

Für die flexible Sicherung von Zuständen sind zwei Mechanismen wichtig. Zum einen die von *TStateSaveMgr* gesteuerte explizite Anweisung an die Komponenten, ihren Zustand zu einem bestimmten Zeitpunkt wiederherstellen zu können, zum anderen die von der Komponente ausgehende Mitteilung an ihr Sicherungsobjekt über aufgetretene Zustandsänderungen. Je nach Ausprägung des Sicherungsobjekts sind die Abläufe dabei unterschiedlich.

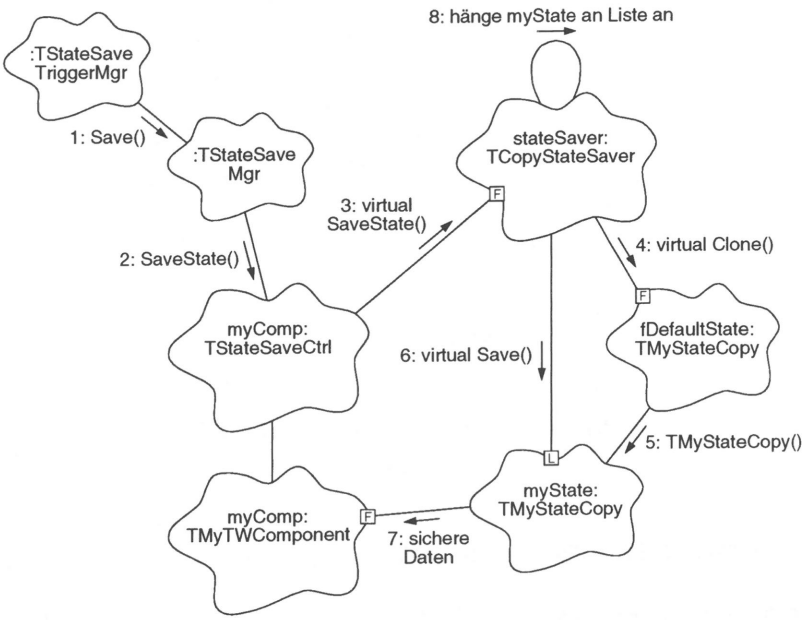

Bild 5.26: Sichern von Zuständen mittels Zustandskopien

Der erste Mechanismus ist für den Fall der Sicherung mittels Zustandskopien in Bild 5.26 gezeigt. Die vom Trigger-Manager ausgehende Aufforderung zur Sicherung wird über den Sicherungs-Manager an die von diesem gesteuerten Objekte (*myComp* im Beispiel) und das jeweilige Sicherungsobjekt (*stateSaver*) weitergleitet. Das Sicherungsobjekt vom Typ *TCopyStateSaver* implementiert die virtuelle Methode *SaveState()* entsprechend der für die Komponente gewählten Sicherungsstrategie. Es sei angenommen, daß sich der Zustand von *myComp* seit Erstellung der letzten Zustandskopie geändert hat, es also nicht genügt, nur die „Gültigkeitsdauer" der letzten Kopie zu verlängern. Zuerst wird der im Sicherungsobjekt vorhandene „Basiszustand" *fDefaultState* durch Aufruf seiner virtuellen Methode *Clone()* kopiert. Das dadurch erhaltene Zustandsobjekt *myState* wird dann mittels seiner virtuellen Methode *Save()* veranlaßt, den Zustand der Komponente *myComp* intern abzuspeichern. Anschließend wird es von *stateSaver* an die Zustandsliste *fStateList* angefügt, womit der Zustand von *myComp* gesichert ist.

Wäre das Sicherungsobjekt *stateSaver* vom Typ *TIncStateSaver*, würde für *myComp* eine inkrementelle Zustandssicherung durchgeführt. Dabei ist *TIncStateSaver::SaveState()* derart implementiert, daß die Aufforderung zur Sicherung einfach ignoriert wird. *TIncStateSaver* kann durch die im folgenden beschriebene Speicherung aller Zustandsänderungen den Zu-

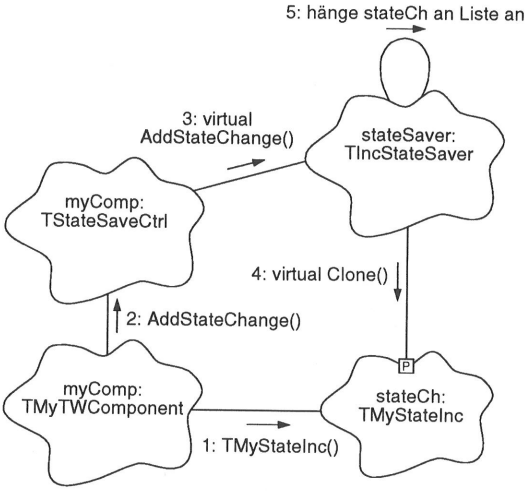

Bild 5.27: Behandlung von Zustandsänderungen bei inkrementeller Sicherung

stand der Komponente auf jeden Fall zu jeder beliebigen Zeit – und damit natürlich auch zu allen vom Sicherungs-Manager geforderten Sicherungsaufpunkten – wiederherstellen und muß deshalb keine explizite Sicherung vornehmen.

Der zweite wichtige Mechanismus, die Mitteilung von Zustandsänderungen, ist in Bild 5.27 für inkrementelle Zustandssicherung gezeigt. Sobald sich der Zustand von *myComp* ändert, wird ein Objekt *stateCh* erzeugt, das diese Änderung festhält. Dieses Objekt wird danach mittels der virtuellen Methode *AddStateChange()* dem Sicherungsobjekt *stateSaver* vom Typ *TIncStateSaver* übergeben. Diese Klasse implementiert *AddStateChange()* derart, daß *stateCh* mittels *Clone()* kopiert und diese Kopie an eine Liste mit Änderungen angefügt wird.

Wäre das Sicherungsobjekt *stateSaver* dagegen vom Typ *TCopyStateSaver*, würde für *myComp* eine Sicherung mittels Zustandskopien durchgeführt und die Änderung deswegen ignoriert[8].

Beide beschriebenen Mechanismen sind für die Simulationsmodellkomponente bezüglich des Typs des eingesetzten Sicherungsobjekts transparent[9].

[8] Genaugenommen stellt *TCopyStateSaver* durch die Mitteilung über eine Zustandsänderung fest, ob beim nächsten, vom Sicherungs-Manager festgelegten Sicherungsaufpunkt eine Sicherung erfolgen oder nur die „Gültigkeitsdauer" der vorhergegangenen Sicherung verlängert werden muß (siehe Unterkapitel 4.7).

[9] Zur Optimierung kann eine Komponente vor Mitteilung einer Zustandsänderung erst „anfragen", ob eine solche überhaupt notwendig ist. Die entsprechende Methode *TStateSaveCtrl::HasToCallAddStateChange()* berücksichtigt dazu vom Sicherungsobjekt dynamisch gelieferte Informationen und ist als „inline"-Funktion sehr schnell.

5.4.4.2 Wiederherstellen von Zuständen

Die Wiederherstellung von Zuständen wird wiederum vom Sicherungs-Manager gesteuert. Die Bilder 5.28 und 5.29 zeigen, daß dieser die Methode *RestoreState()* der von ihm verwalteten Objekte vom Typ *TStateSaveCtrl* aufruft (unter Angabe des wiederherzustellenden Zeitpunkts), und diese ihr angegliedertes Sicherungsobjekt mittels der virtuellen Methode *RestoreState()* auffordern, den Zustand der Komponente wiederherzustellen. Wie dies gemacht wird, hängt vom Typ des Sicherungsobjekts ab. Bei der in Bild 5.28 gezeigten Sicherung mittels Zustandskopien sucht *stateSaver* in seiner Liste *fStateList* die dem gewünschten Zeitpunkt entsprechende Zustandskopie *myState* (löscht dabei alle Kopien höherer Zeitpunkte) und ruft dessen virtuelle Methode *Restore()* auf. Diese transferiert dann alle gespeicherten Daten zurück in die Komponente, womit deren Zustand wiederhergestellt ist.

Bei inkrementeller Zustandssicherung (Bild 5.29) geht *stateSaver* in absteigender Zeitstempelfolge alle in seiner Liste gespeicherten Zustandsänderungen bis zum wiederherzustellenden Zeitpunkt durch. Jedes gefundene Änderungsobjekt (*stateCh* im Bild) wird durch den Aufruf seiner virtuellen *Restore()*-Methode veranlaßt, die in ihm gespeicherte Änderung in der Komponente rückgängig zu machen. Anschließend wird es aus der Liste entfernt und gelöscht. Durch das schrittweise Rückgängigmachen aller Änderungen, die nach dem wiederherzustellenden Zeitpunkt erfolgt sind, ist der Zustand der Komponente schließlich restauriert. Auch dieser Mechanismus ist also für die Simulationsmodellkomponente bezüglich des eingesetzten Sicherungsobjekts transparent.

5.4.5 Zustandssicherung von Nachrichteninhalten

Bei der Sicherung von Simulationsnachrichten müssen zwei Aspekte betrachtet werden. Zum einen muß festgehalten werden, wo sich die Nachricht zu einem bestimmten Zeitpunkt befand, und zum anderen, welchen Inhalt sie hatte. Der erste Punkt liegt in der Verantwortlichkeit der Komponente, in welcher die Nachricht sich befindet. Eine Nachricht, die durch das Simulationsmodell läuft, kann sehr häufig an verschiedenen Stellen von der Zustandssicherung erfaßt werden. Müßte die Nachricht, deren Inhalt u. U. recht umfangreich sein kann, jedesmal komplett kopiert werden, könnte dies zu erheblichen Ineffizienzen bezüglich Zeit- und Speicherplatzaufwand führen. Aus diesem Grund wurde der zweite Punkt in die Verantwortlichkeit der Nachricht selbst gelegt.

In Unterabschnitt 5.3.1.3 wurde beschrieben, daß eine Simulationsnachricht aus zwei Teilen besteht: einer Nachrichten-Repräsentation, die den eigentlichen Nutzinhalt enthält, und einem Nachrichten-Handle. Diese Zweiteilung erweist sich für die Zustandssicherung als sehr vorteilhaft. Bild 5.30 zeigt die Verhältnisse für die Sicherung einer Nachricht bestehend aus der Repräsentation *myMsgRep* und dem Handle *myMsg*.

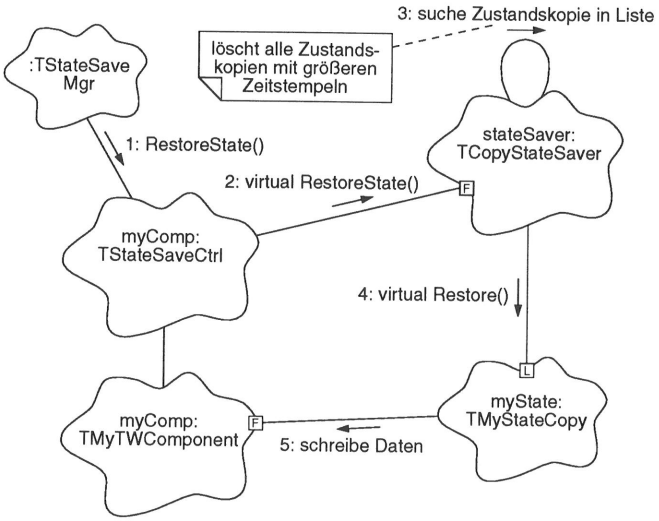

Bild 5.28: Wiederherstellen eines Zustands bei Sicherung mittels Zustandskopien

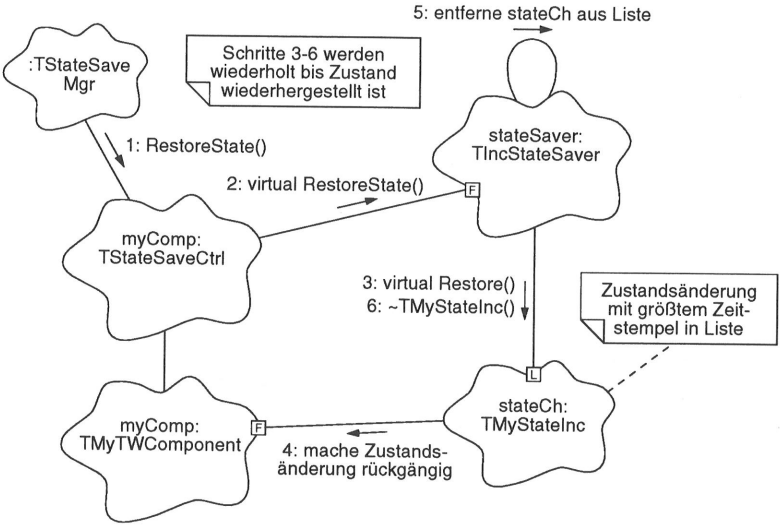

Bild 5.29: Wiederherstellen eines Zustands bei inkrementeller Sicherung

Es gibt im Simulationsmodell immer nur maximal einen Nachrichten-Handle für eine Repräsentation, entsprechend der Tatsache, daß eine bestimmte Nachricht nur einmal vorhanden sein kann. Im Beispiel befindet die Nachricht sich zur aktuellen LVT in der Bedieneinheit *phase2*. Zuvor hat sie aber schon die Bedieneinheit *phase1* und die Warteschlange *queue* durchlaufen und wurde von deren Sicherungsobjekten abgespeichert. Dazu wurde aber jeweils nicht die Nachricht selbst kopiert, sondern lediglich ein Sicherungs-Verweisobjekt vom Typ *TStateSaveMsgHandle* angelegt, welches eine Referenz auf die Repräsentation enthält. Durch das Anlegen solcher Verweisobjekte können die Komponenten auf sehr effiziente Art und Weise speichern, welche Nachricht sie zu einem bestimmten Zeitpunkt enthielten.

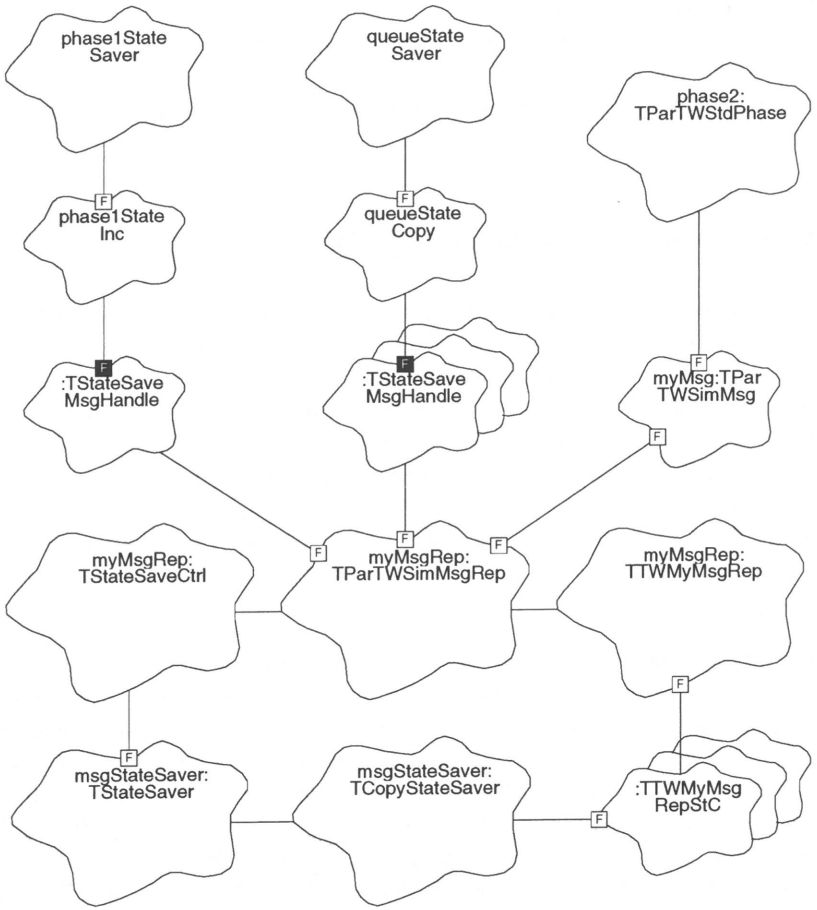

Bild 5.30: Zustandssicherung von Nachrichten

Die Sicherung des Nachrichteninhalts wird von der Nachrichten-Repräsentation eigenständig durchgeführt. Dazu ist die Klasse *TParTWSimMsgRep* von *TStateSaveCtrl* abgeleitet und dadurch wie die Modellkomponenten in den Zustandssicherungsmechanismus eingebunden. Für die vom Benutzer für eigene Nachrichten abzuleitende Klasse *TTWMyMsgRep* muß er die üblichen Klassen für Zustandskopien (*TTWMyMsgRepStC*) und Zustandsänderungen (nicht gezeigt) zur Verfügung stellen. Durch geeignete Wahl des Sicherungsobjekts ist auch für den Nachrichteninhalt die flexible Wahl verschiedener Sicherungsmechanismen möglich.

Die Lebensdauer einer Nachrichten-Repräsentation wird bestimmt durch die sie referenzierenden Objekte (Nachrichten-Handle und Sicherungs-Verweisobjekte). Durch automatische Referenz-Zählung wird sie gelöscht, sobald keine Referenz mehr auf sie verweist. Eine Nachrichten-Repräsentation kann aufgrund der Zustandssicherung auch dann noch existieren, wenn die entsprechende Nachricht (und damit der Nachrichten-Handle) zur aktuellen LVT nicht mehr existiert.

5.4.6 Zustandssicherung des Ereigniskalenders

Beim Zurücksetzen muß selbstverständlich auch der Zustand des Ereigniskalenders wiederhergestellt werden. Dazu müssen alle Ereignisse, die er enthalten hatte, wieder eingetragen und solche, die in der Zwischenzeit eingetragen und noch nicht bearbeitet worden waren, wieder entfernt werden. Zur Realisierung dieser Aufgabe wurden zwei Alternativen in Betracht gezogen: Eine erste Möglichkeit besteht darin, daß grundsätzlich die Simulationsmodellkomponenten für ihre Ereigniseinträge selbst verantwortlich sind. Das hieße, daß beim Zurücksetzen jede Komponente ihre Ereignisse ein- und austrägt. Da dies dazu führen würde, daß Komponenten Informationen über eingetragene Ereignisse speichern müßten, und dies die Zustandssicherung aus deren Sicht komplizieren würde, wurde ein anderer Weg gewählt.

Der Kalender führt eine eigene Zustandssicherung durch. Aufgrund der Ereigniszeit und des Zeitpunkts des Eintrags kann er bestimmen, welche Ereignisse er zu einem bestimmten Zeitpunkt enthielt. Um seinen Zustand später wiederherstellen zu können, muß er eingetragene Ereignisse vor deren Bearbeitung sichern. Erschwert wird dies dadurch, daß die Lebensdauer von Ereignisobjekten (d. h. Objekten, die zum Eintrag von Ereignissen in den Kalender benutzt werden, und die z. B. die auszuführenden Aktionen bei der Bearbeitung des Ereignisses definieren) nicht festgelegt ist und diese selbst wiederum Daten enthalten können. Da es deswegen i. allg. nicht reicht, einfach eine Referenz auf das Ereignisobjekt zu speichern, wurde das in Bild 5.31 gezeigte Konzept zur Sicherung von Ereignisobjekten entwickelt.

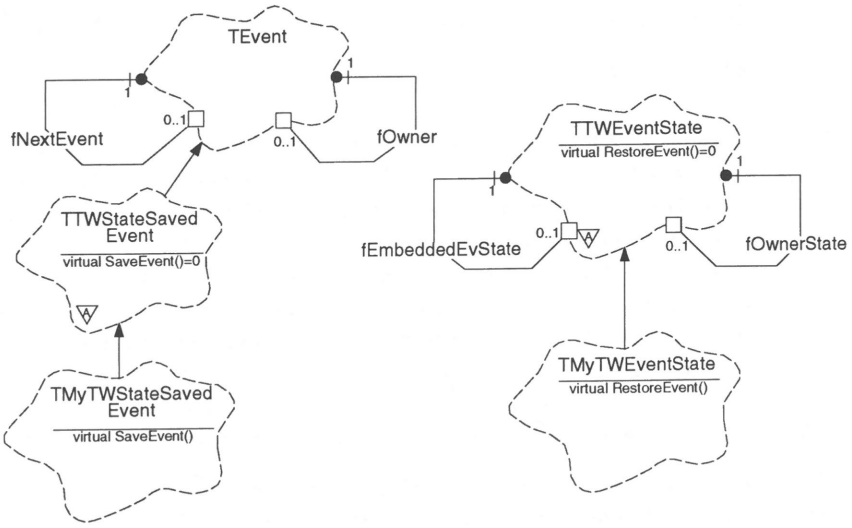

Bild 5.31: Zustandssicherung von Ereignisobjekten

Ereignisobjekte, die nicht während der gesamten Simulationsdauer unverändert Bestand haben, müssen dabei von der abstrakten Basisklasse *TTWStateSavedEvent* abgeleitet werden (*TMyTWStateSavedEvent*) und die virtuelle Methode *SaveEvent()* implementieren. Diese muß bei einem Aufruf eine Instanz einer von *TTWEventState* abgeleiteten Klasse liefern (*TMyTWEventState*), die den momentanen Zustand des Ereignisobjektes beinhaltet. Bevor ein Ereignisobjekt zur Bearbeitung aus dem Kalender geholt wird, sichert dieser dessen Inhalt durch Aufruf der *SaveEvent()*-Methode. Trägt der Kalender das Ereignis später beim Zurücksetzen wieder ein, wird die in *TMyTWEventState* implementierte Methode *RestoreEvent()* aufgerufen, welche das Ereignisobjekt mit dem gesicherten Inhalt wiederherstellt.

TTWStateSavedEvent ist von der Klasse *TEvent* aus [Kocher,1994] abgeleitet, wodurch die entsprechenden Ereignisobjekte auch von den dort entwickelten Kalendern gehandhabt werden können. Dies kommt der Forderung nach einem einfachen Übergang von sequentieller zu paralleler Simulation entgegen. Eingebettete Ereigniszustände (*fEmbeddedEvState*) wurden zur Unterstützung des dort beschriebenen Konzepts eingebetteter Ereignisse eingeführt.

Aus Effizienzgründen besteht bei optimistischen Kalendern nach wie vor die Möglichkeit, einfache Ereignisobjekte vom Typ *TEvent* einzutragen, sofern diese während der gesamten Simulationsdauer unverändert bleiben. In diesem Fall speichert der Kalender vor der Bearbeitung lediglich eine Referenz auf das entsprechende Objekt ab.

5.5 Kapselung der Approximation der Globalen Virtuellen Zeit

In Unterabschnitt 2.5.2.4 wurden verschiedene Methoden zur Approximation der GVT kurz vorgestellt. Diese soll möglichst genau, schnell und ohne wesentliche Leistungseinbußen der Simulation erfolgen. Diese Attribute eines bestimmten Verfahrens werden sich in der Regel von Simulationsmodell zu Simulationsmodell nicht wesentlich ändern, sondern eher vom Simulationssystem (z. B. FIFO-Übertragung von Nachrichten oder nicht) oder der verwandten Rechenplattform abhängen. Aus Sicht des Simulationsanwenders ergibt sich deswegen weniger das Bedürfnis, den Mechanismus zur GVT-Approximation häufig auszutauschen. Nichtsdestoweniger sollte dieser Austausch, z. B. bei einem Wechsel der Rechenplattform, einfach möglich sein. Aus diesem Grund wurde auch die GVT-Approximation durch das in Bild 5.32 angedeutete System aus abstrakten Basisklassen gekapselt.

Genau eine Instanz einer von *TGVTGlobalMgr* abgeleiteten Klasse existiert im Manager-Knoten für zentrale Aufgaben, wie z. B. der Bearbeitung von GVT-Anforderungen oder

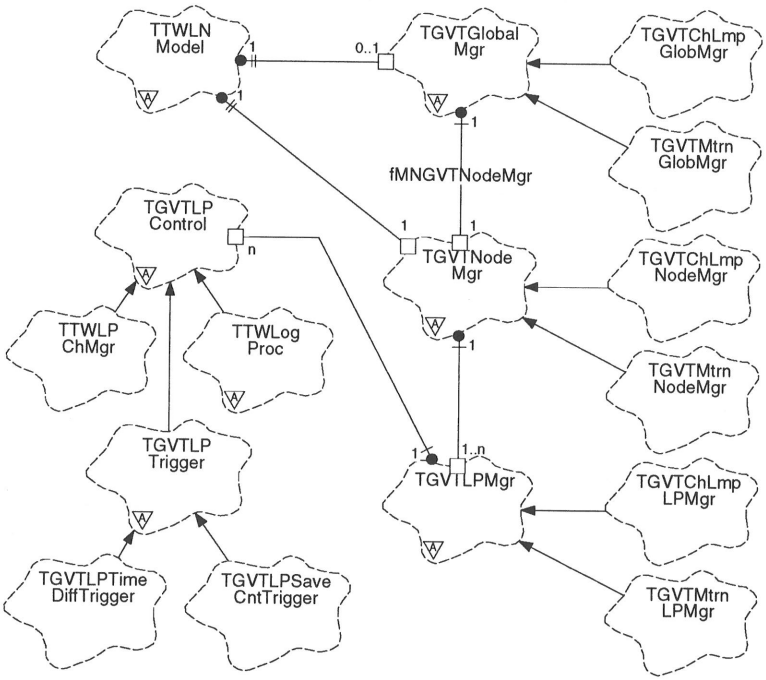

Bild 5.32: Basisklassen für die Approximation der GVT

dem Verbreiten von approximierten GVT-Werten. In jedem Logischen Knoten (inklusive dem Manager-Knoten) gibt es eine Instanz einer von *TGVTNodeMgr* und für jeden Logischen Prozeß einer von *TGVTLPMgr* abgeleiteten Klasse. Alle erwähnten Klassen sind abstrakte Basisklassen, von denen für die verschiedenen GVT-Approximationsalgorithmen jeweils zueinander passende Klassen abgeleitet werden müssen. In Bild 5.32 sind diese Spezialisierungen für die implementierten Algorithmen nach [Chandy&Lamport,1985] und [Mattern,1993] dargestellt (siehe Unterabschnitt 2.5.2.4). Die Instanzen der zum jeweiligen Algorithmus gehörenden Klassen können dann über spezielle Steuernachrichten (siehe Unterabschnitt 5.3.1.2) miteinander kommunizieren und durch die Installation von Kanalfiltern an den Kanälen der Logischen Prozesse die Zeitstempel der ausgetauschten Simulationsnachrichten überwachen. Der in [Mattern,1993] vorgestellte Algorithmus hängt dabei beispielsweise jeder versandten Simulationsnachricht eine Nachrichtenmarkierung mit einer entsprechenden „Farbe“ an.

Durch Ableitung von *TGVTLPControl* können verschiedene Klassen zu Verwaltungszwecken unter die Kontrolle des GVT-Managers für Logische Prozesse gestellt werden. Im Bild sind dies die Verwaltungsstruktur für den jeweiligen Logischen Prozeß, der Kanal-Manager sowie die Auslöser einer GVT-Anforderung. Letztere sind von der abstrakten Basisklasse *TGVTLPTrigger* abgeleitet und fordern von Zeit zu Zeit aufgrund bestimmter Kriterien eine neue GVT-Approximation an. Diese Kriterien können z. B. die Differenz zwischen lokaler Zeit und der zuletzt approximierten GVT (*TGVTLPTimeDiffTrigger*) oder die Anzahl durchgeführter Zustandssicherungen sein (*TGVTLPSaveCntTrigger*). Eine solche Anforderung wird dann zusammen mit der lokalen Zeit an den globalen GVT-Manager im Manager-Knoten gesandt, welcher sie in eine nach Anforderungszeitpunkten geordnete Warteschlange einreiht. Ist eine GVT-Approximation im Gange, wird auf deren Abschluß gewartet. Ist dieser erfolgt, werden alle Anforderungen mit einer kleineren Zeit als der approximierten GVT entfernt, da sie obsolet sind. Die erste verbleibende Anforderung in der Warteschlange wird anschließend, falls vorhanden, dazu benutzt, eine neue Approximation zu starten. Dieses System erlaubt die Synchronisation der Logischen Prozesse, indem sie zu bestimmten Simulationszeitpunkten blockieren und eine GVT-Anforderung absetzen. Es ist in diesem Fall gewährleistet, daß keine Anforderung verlorengeht, bevor die GVT-Approximation nicht diesen Zeitpunkt erreicht hat. Ungeachtet eines etwaigen zwischenzeitlichen Zurücksetzens wird der Zeitpunkt kommen, an dem der letzte LP im System den Synchronisationspunkt erreichen und eine GVT-Anforderung senden wird. Spätestens die dadurch ausgelöste Approximation wird als Ergebnis diese Zeit liefern, so daß die Synchronisation in jedem Fall erfolgreich und verklemmungsfrei durchgeführt werden kann.

Kapitel 6

Realisierung als objektorientierte Bibliothek

6.1 Ziele der Bibliothek

In Unterkapitel 4.1 wurden die wichtigsten bisher verfügbaren Werkzeuge für parallele ereignisgesteuerte Simulation aufgeführt und die Notwendigkeit neuer Ansätze begründet. Für die Realisierung der im letzten Kapitel beschriebenen Konzepte wurde die Form einer in einer Standard-Programmiersprache implementierten Bibliothek gewählt. Diese Lösung ist flexibel, leicht anpaß- und erweiterbar und stellt bei geeigneter Architektur einen guten Kompromiß zwischen Nähe zum Simulationssystem (Synchronisationsverfahren, Rechenplattform usw.) für eine hohe Effizienz und Nähe zum Simulationsmodell für eine einfache Anwendbarkeit dar.

Die grundsätzlichen Anforderungen, denen diese Bibliothek genügt, wurden schon in Unterkapitel 4.2 aufgeführt. Im Mittelpunkt steht dabei die einfache Anwendbarkeit paralleler Simulationstechniken. Aus diesem Grund wurde die parallele Bibliothek wo immer möglich abwärtskompatibel zu der in [Kocher,1994] beschriebenen sequentiellen Bibliothek gehalten. Einem Simulationsanwender, der mit dieser sequentiellen Bibliothek vertraut ist, soll der Umstieg auf die parallele Bibliothek dadurch erleichtert werden, daß den Teilen, mit denen er in Berührung kommt, ähnliche Konzepte wie im sequentiellen Fall zugrundeliegen. Insbesondere soll es möglich sein, Modellkomponenten, die für die sequentielle Bibliothek entwickelt wurden, mit möglichst geringem Aufwand auf parallele Verhältnisse anzupassen. Der Einstieg in die Welt der parallelen Simulation soll also über bereits bekanntes Terrain erfolgen und dadurch erleichtert werden. Simulationsmodelle, die für die sequentielle Bibliothek entwickelt wurden, sollen bei steigendem Bedarf an Rechenleistung einfach parallelisiert werden können.

Ein weiterer Punkt stellt die einfache Erweiterbarkeit der Bibliothek dar. Kapitel 5 hat Konzepte vorgestellt, mit deren Hilfe die Implementierung neuer Synchronisationsverfahren ohne Eingriffe in die Basisfunktionalität (z. B. Nachrichtenaustausch, Einplanen und Ausführen von Ereignissen etc.) vorgenommen werden kann. Bei der Realisierung von Varianten existierender Verfahren kann dabei zudem leicht auf bestehenden Code aufgebaut werden. Aus Sicht des Programmierers, der an der Implementierung von Verfahren zur parallelen ereignisgesteuerten Simulation interessiert ist, bildet die Bibliothek ein Framework (siehe Abschnitt 3.2.7), in das er die von ihm gewünschten Mechanismen einbetten kann. Aus Sicht des Simulationsanwenders stellt sie dagegen einen Toolkit (siehe Abschnitt 3.2.6) mit vorgefertigten Bausteinen dar, mit deren Hilfe er sein Simulationsmodell implementieren und parallel simulieren kann.

Als Programmiersprache wurde, wie schon erwähnt, *C++* gewählt (siehe Unterkapitel 3.3). Zum einen, um die oben erwähnte Kompatibilität zur sequentiellen Bibliothek, die ebenfalls in *C++* implementiert ist, sicherzustellen. Zum anderen ist *C++* die einzige objektorientierte Programmiersprache, die auch auf Parallelrechnern eine gewisse Verbreitung besitzt.

Im folgenden Abschnitt soll kurz auf die wichtigsten Eigenschaften der erwähnten sequentiellen Bibliothek eingegangen werden. Danach wird der Aufbau der parallelen Bibliothek beschrieben und näher auf den Übergang von sequentieller zu paralleler Simulation eingegangen.

6.2 Objektorientierte sequentielle Bibliothek

Die schon mehrfach erwähnte objektorientierte Bibliothek zur sequentiellen ereignisgesteuerten Simulation aus [Kocher,1994] wurde für die Behandlung komplexer hierarchischer Systeme entwickelt. Sie ermöglicht den hierarchischen Aufbau von Simulationsmodellkomponenten aus anderen Komponenten und deren strikte Kapselung. Modellkomponenten kommunizieren untereinander mittels Nachrichten, für deren Austausch sie ein Port-Konzept verwenden (s. a. Unterabschnitt 5.3.3.1). Die Bibliothek ermöglicht auf einfache Weise die Wiederverwendung von standardisierten Grundkomponenten, aus denen komplexere Modelle nach dem Baukastenprinzip aufgebaut werden können.

Aufgrund des modularen Aufbaus der mit dieser Bibliothek entwickelten Simulationsmodelle und der Kapselung der Simulationsmodellkomponenten eignet sie sich gut als Basis für eine Parallelisierung. Der Aufwand hierfür wurde in [Kocher,1994] bereits abgeschätzt. Jedoch wurde dort von weitaus geringeren Anforderungen an die Flexibilität ausgegangen und der Einsatz optimistischer Synchronisationsverfahren nicht betrachtet. Dadurch haben sich die Abschätzungen als zu optimistisch und die Vorschläge als nicht ausreichend erwiesen.

6.3 Aufbau der parallelen Bibliothek

Bild 6.1 zeigt den Aufbau der parallelen Bibliothek. Wie zu sehen ist, handelt es sich dabei nicht um einen monolithischen Block, sondern um schichtenweise angeordnete Einzelbibliotheken. Diese fußen auf dem Betriebssystem der zugrundeliegenden Rechenplattform, das auf den für diese Arbeit benutzten Plattformen *UNIX* war. Für den Nachrichtenaustausch zwischen den Logischen Knoten ist außerdem noch ein *Message-Passing-Subsystem* erforderlich, wobei hierfür *PVM* auf einem Verbund von HP-Workstation-Rechnern und die *NX*-Bibliothek auf dem Parallelrechner *Paragon* von Intel verwandt wurden (s. a. Unterabschnitt 3.1.3.3.2). Prinzipiell können aber beliebige Systeme benutzt werden, solange sie den Austausch von unstrukturierten Byte-Strömen zwischen den einzelnen Rechenknoten sowie das Starten und Stoppen von Prozessen erlauben.

Oberhalb der sequentiellen Bibliothek befinden sich in der untersten Schicht der parallelen Bibliothek, der Basisbibliothek, u. a. die Mechanismen zur Kommunikation und zur Steuerung der verteilten Anwendung. Des weiteren sind hier viele der in Kapitel 5 beschriebenen abstrakten Basisklassen sowie weitere grundlegende Datenstrukturen beheimatet. In den beiden darüberliegenden Schichten der Bibliothek befinden sich spezielle Klassen für konservative und optimistische Simulation. Zuoberst schließlich ist eine Einzelbibliothek für die Verteilung und Konfiguration größerer Simulationsmodelle sowie die einfache Wahl der

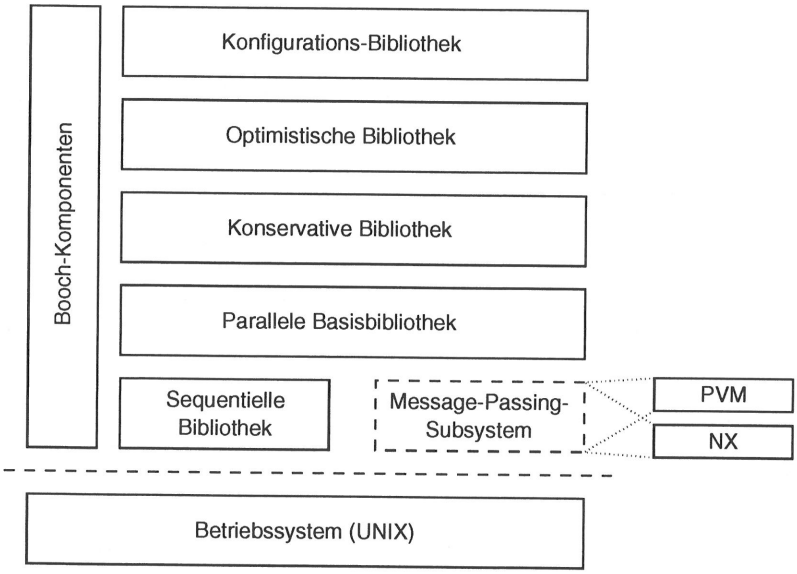

Bild 6.1: Aufbau der Bibliothek

spezifischen Mechanismen für parallele Simulation und deren Parametern angesiedelt. Auf diesen Teil wird in Unterkapitel 6.6 näher eingegangen.

In allen Schichten der Bibliothek werden die *Booch-Komponenten* [Booch,1993] verwandt, eine Sammlung überwiegend parametrisierter Klassen, die häufig benötigte Strukturen wie Listen, Warteschlangen etc. zur Verfügung stellen.

6.4 Kategorisierung der Klassen

Wenn man die Gesamtheit aller Klassen der parallelen Bibliothek betrachtet, lassen sich die für die Umsetzung des Konzepts aus den Kapiteln 4 und 5 benötigten Klassen in die in Bild 6.2 gezeigten Kategorien einordnen. Essentiell für eine verteilte Anwendung sind zum einen die in *CCommunication* zusammengefaßten Klassen für die Kommunikation der einzelnen Logischen Knoten untereinander und die in *CExec* enthaltenen Klassen für die Ablaufsteuerung, die die einzelnen Phasen des Programmablaufs steuert und synchronisiert. *CLogNode*, *CLogProcess* und *CCalendar* enthalten die Klassen zur Steuerung des Logischen Knotens, der einzelnen Logischen Prozesse und für die unterschiedlichen Ereignis-

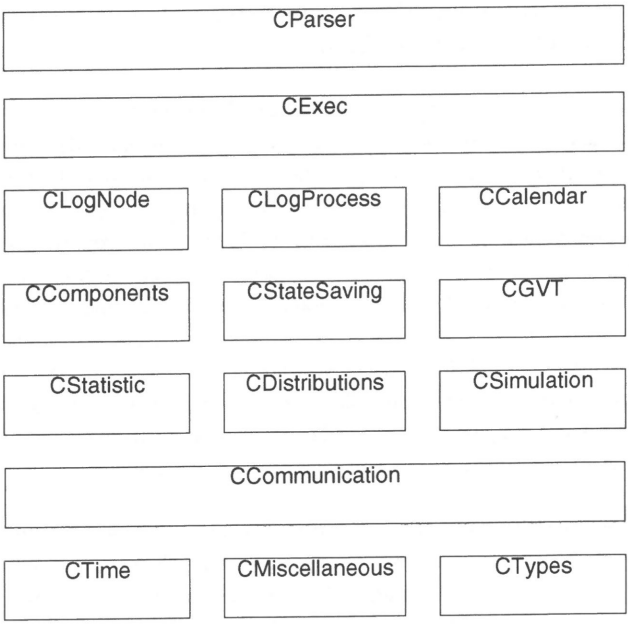

Bild 6.2: Klassenkategorien

kalender. Die Klassen für die Zustandssicherung und die Approximation der GVT finden sich in *CStateSaving* und *CGVT*. *CSimulation* steuert den eigentlichen Simulationslauf mit der Warmlaufphase und den einzelnen Teiltests. Die Klassen für statistische Auswertung, Verteilungsfunktionen und Erzeugung von Zufallszahlen befinden sich in *CStatistic* und *CDistributions*. *CComponents* enthält Klassen für die Erzeugung der Simulationsmodellkomponenten. Die in Unterkapitel 2.6 beschriebene Erweiterung des Zeitbegriffs wird in *CTime* realisiert. Daneben gibt es noch *CTypes* mit Definitionen globaler Typen und *CMiscellaneous* mit weiteren Klassen, die nicht anderweitig zugeordnet werden können. Schließlich sind die Klassen für das in Unterkapitel 6.6 näher beschriebene Parserkonzept in *CParser* zusammengefaßt.

6.5 Fehlerbehandlung

Die Behandlung von Fehlern in einer verteilten Anwendung ist schwieriger als in einer zentralisierten. Zum einen ist die Zahl der Fehlerquellen höher, zum anderen ist die Stelle, an der der Fehler aufgetreten ist, oft nicht einfach zu lokalisieren. Schließlich muß gewährleistet sein, daß ein schwerwiegender Fehler zu einer zentralen Meldung an den Benutzer und zum Abbruch der gesamten Anwendung führt.

Zur Behandlung von Fehlern innerhalb eines Logischen Knotens wurde das Konzept der Ausnahmebehandlung von *C++* verwandt (siehe z. B. [Stroustrup,1991]). Tritt ein Fehler auf, wird eine Ausnahme (*Exception*) ausgeworfen und von einem Ausnahme-Manager aufgefangen, von dem es in jedem Logischen Knoten genau einen gibt (siehe Unterabschnitt 5.2.5.1). Trat der Fehler im Manager-Knoten auf, wird die Fehlerursache zusammen mit der Stelle, an der er aufgetreten ist, direkt ausgegeben. Bevor die Ausführung des Managers abgebrochen wird, werden alle anderen Logischen Knoten benachrichtigt und beendigen ihrerseits die Ausführung. Trat der Fehler außerhalb des Manager-Knotens auf, so sendet der betroffene Logische Knoten eine Nachricht mit Fehlerursache und Fehlerort an den Manager und stoppt anschließend. Nach Empfang einer solchen Nachricht gibt der Manager diese Information zusammen mit der Angabe des verursachenden Logischen Knotens aus und veranlaßt alle übrigen Logischen Knoten, ihre Ausführung ebenfalls zu beenden.

Dieses Konzept gewährleistet, daß Fehler zentral über den Manager-Knoten ausgegeben und alle Logischen Knoten automatisch gestoppt werden. Er ist außerdem auch anwendbar, wenn das *C++*-Konzept der Ausnahmebehandlung von einem Compiler, wie im Fall der Intel-*Paragon*, nicht zur Verfügung gestellt wird. In diesem Fall wird das Auswerfen und Auffangen einer Ausnahme über ein Makro nachgebildet.

6.6 Konfiguration

6.6.1 Aufbau von Netztopologien

Der Aufbau von komplexen verteilten Simulationsmodellen und die Verbindung der einzelnen Simulationsmodellkomponenten untereinander ist auf *C++*-Quelltextebene trotz Bibliotheksunterstützung zeitraubend, fehlerträchtig und unflexibel. Um dem Benutzer einen einfachen Aufbau seines Modells und eine einfache Wahl der Synchronisations- und Steuermechanismen zu ermöglichen, wurde eine Beschreibungssprache und ein zugehöriges Parserkonzept für das Einlesen der entsprechenden Parameter über Dateien eingeführt. Dieses basiert auf einem in [Lang,1997] vorgestellten Konzept und wurde für die Belange der parallelen Simulation erweitert.

Jedes Simulationsmodell besteht für die Zwecke der Topologiebeschreibung aus einzelnen *Netz-Knoten*, die durch *Netz-Verbindungen* verbunden sind (siehe Bild 6.3). Jeder Netz-Knoten kann intern beliebig hierarchisch aufgebaut sein, er kann externe Verbindungen jedoch nur auf höchster Hierarchieebene unterhalten. Netz-Knoten werden durch einen Namen und einen Bezeichner identifiziert, die beide netzweit eindeutig sind. Des weiteren besitzen sie eine Zuordnung zu einem bestimmten Logischen Prozeß und Logischen Knoten.

Bei Netz-Verbindungen kann es sich entweder um direkte Verbindungen ohne eigene Funktionalität oder um eine detaillierte Modellierung des Übertragungsabschnitts handeln. Da Netz-Verbindungen auch hierarchisch aufgebaut sein und die Simulation komplexer Vorgänge erforderlich machen können, müssen sie wie alle anderen Simulationsmodellkomponenten einem Logischen Prozeß zugeordnet sein. Sie werden deshalb grundsätzlich charakterisiert durch die Bezeichner der beiden Netz-Knoten, die sie verbinden, und die Zuordnung zu einem Logischen Prozeß und Logischen Knoten.

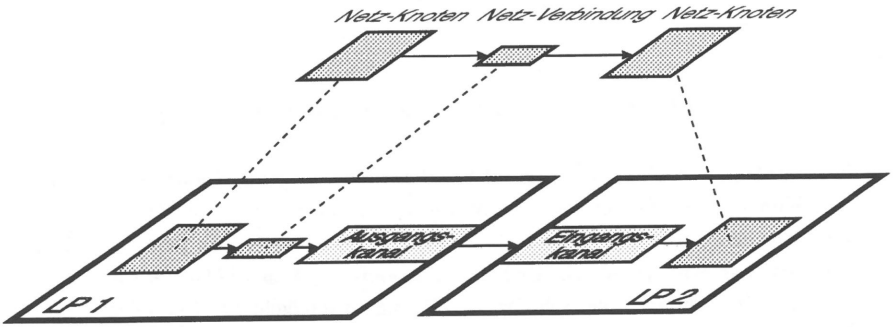

Bild 6.3: Automatische Verbindung von Netz-Knoten über LP-Grenzen hinweg

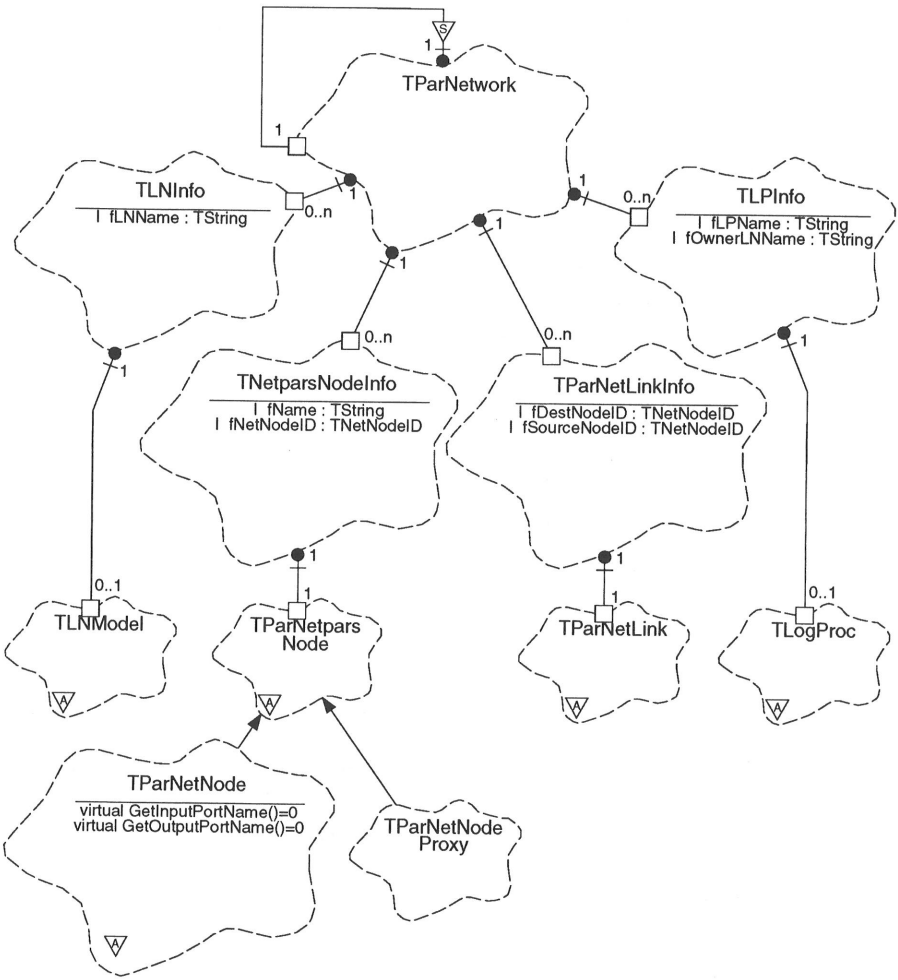

Bild 6.4: Klassendiagramm für Repräsentation der Netztopologie

In jedem Logischen Knoten werden Informationen über Logische Knoten, Logische Prozesse, Netz-Knoten und Netz-Verbindungen von einer zentralen Instanz der Klasse *TParNetwork* verwaltet (siehe Bild 6.4). Für Netz-Knoten, die nicht im lokalen Logischen Knoten angesiedelt sind, wird jeweils ein Proxy, eine Instanz der Klasse *TParNetNodeProxy*, angelegt, um Verbindungen über LN-Grenzen hinweg zu ermöglichen. Netz-Knoten und Netz-Verbindungen müssen von den abstrakten Basisklassen *TParNetNode* und *TParNetLink* abgeleitet sein.

Die Informationen über die Netztopologie befinden sich in einer zentralen Simulationsparameter-Eingabedatei, die von allen Logischen Knoten gelesen wird. Dadurch erhalten alle LNs eine globale Sicht der Topologie und sind in der Lage, Verbindungen über ihre Grenzen hinweg aufzubauen. Die Alternative des Einlesens dieser Datei nur durch den Manager-Knoten, der die Informationen dann anschließend an die übrigen Logischen Knoten verteilt, wurde verworfen. Der Grund dafür liegt darin, daß sich in dieser zentralen Eingabedatei auch Konfigurationsdaten von benutzerspezifizierten Simulationsmodellkomponenten befinden können. Das Einlesen dieser Daten wird vom Parser-Konzept der sequentiellen Bibliothek unterstützt, so daß hier kein zusätzlicher Aufwand für den Benutzer entsteht. Würde die Datei zentral nur vom Manager-Knoten eingelesen, müßte der Benutzer zusätzliche Routinen für die Verteilung der Daten bereitstellen.

Die Simulationsparameter-Eingabedatei wird während der Ausführungsphase „Aufbau des Simulationsmodells“ (siehe Unterkapitel 4.4) von allen Logischen Knoten eingelesen, und die dort definierten Netz-Knoten (bzw. Proxies hierfür) und Netz-Verbindungen werden erzeugt. Während der nachfolgenden Phase „Herstellen der Verbindungen der LPs untereinander“ werden alle durch Netz-Verbindungen spezifizierten Verbindungen zwischen den Netz-Knoten hergestellt. Befinden sich beide Netz-Knoten im selben LP, werden die Ports einer Netz-Verbindung mit den Ports der anzuschließenden Netz-Knoten verbunden. Letztere werden durch Aufruf der reinen virtuellen Methoden *GetInputPortName()* und *GetOutputPortName()* von *TParNetNode* ermittelt. Abgeleitete Simulationsmodellkomponenten implementieren diese Methoden und können dadurch aufgrund eines Verbindungswunsches entsprechende Ports zur Verfügung stellen.

Befindet sich ein anzuschließender Netz-Knoten in einem anderen Logischen Prozeß (wie dies in Bild 6.3 der Fall ist), werden automatisch die notwendigen Kanäle erzeugt (entsprechend den für das gewählte Synchronisationsverfahren im LP vorhandenen Prototypen, siehe Abschnitt 5.2.4) und die Ports des Netz-Knotens und der Netz-Verbindung jeweils an die internen Ports der Kanäle angeschlossen. Befinden sich die beiden Logischen Prozesse dabei in unterschiedlichen Logischen Knoten, werden dazu Steuernachrichten ausgetauscht. Die Verbindung der beiden Kanäle miteinander erfolgt dann entweder direkt, falls sie sich im selben Logischen Knoten befinden, oder mit Hilfe des in Abschnitt 4.6.2 vorgestellten Ablaufs zur Erzeugung von Kanal-Proxies und deren Verbindung mit den Kanälen falls nicht.

Das Parserkonzept erlaubt es, sowohl einzelne Netz-Knoten zu erzeugen und zu verbinden, als auch größere Strukturen, wie Ringe, Gitter etc., durch eine kompakte Beschreibung einfach zu realisieren. Diese Strukturen können sich auch über mehrere Logische Prozesse und Logische Knoten hinweg erstrecken. Ihre Verbindung erfolgt für den Anwender völlig transparent.

6.6.2 Konfiguration von Steuerparametern

Eine Forderung an die parallele Simulationsbibliothek ist die einfache Austauschbarkeit der Synchronisationsmechanismen. Aus diesem Grund werden auch die einzusetzenden Algorithmen und ihre Parameter über die zentrale Simulationsparameter-Eingabedatei eingelesen und können so jederzeit flexibel geändert werden. Auch weitere Steuerparameter, wie beispielsweise Anzahl und Dauer der Teiltests, werden auf diese Weise konfiguriert. Darüber hinaus bietet das Konzept die Möglichkeit, zusätzliche Schlüsselworte einzuführen und somit auch die Konfiguration neu implementierter Algorithmen ohne Probleme vornehmen zu können.

Da die zentrale Simulationsparameter-Eingabedatei die Konfiguration der Steuerparameter, die Beschreibung der Netztopologie und die Konfiguration der Simulationsmodellkomponenten enthält, kann sie sehr umfangreich und komplex werden. Aus diesem Grund wurde die Möglichkeit der Vorverarbeitung durch einen Präprozessor geschaffen. Beim Öffnen einer Datei wird diese auf Wunsch automatisch vor dem Einlesen zuerst einem Präprozessor übergeben. Dadurch können beispielsweise Makros definiert und externe Dateien eingefügt werden.

Die Möglichkeit der Vorverarbeitung besteht auch für alle übrigen Eingabedateien der parallelen Bibliothek. Bei diesen handelt es sich zum einen um eine Maschinenkonfigurations-Datei, die nur vom Manager-Knoten gelesen wird und Informationen darüber enthält, welche Logischen Knoten auf welchen Physikalischen Knoten angesiedelt werden sollen, welchen Namen diese haben und welche Programm-Datei geladen werden soll. Des weiteren kann es mehrere Dateien mit den Druckformaten für die Simulationsergebnisse (siehe [Kocher,1994]) sowie eine unbestimmte Anzahl anwendungsspezifischer Eingabedateien geben.

6.7 Übergang auf parallele Simulation

6.7.1 Allgemeine Richtlinien

Eine wichtige Eigenschaft der parallelen Bibliothek ist der einfache Übergang von der sequentiellen Simulation. Was ist bei einem Einsatz der parallelen Bibliothek verglichen mit der sequentiellen zu beachten? Anders gefragt: Welche Voraussetzungen muß ein sequentielles Simulationsprogramm erfüllen, um möglichst einfach parallelisiert werden zu können?

Die wichtigste Voraussetzung ist, daß keine zeitveränderlichen globalen Daten benutzt werden dürfen. Bei einer Verteilung der Simulation auf mehrere Logische Knoten erforderte deren Verwendung den Einsatz aufwendiger Zugriffs- und Synchronisationsverfahren (siehe z. B. [Mehl&Hammes,1993]). Da zudem die unüberlegte Verwendung globaler Daten zu

erheblichen Leistungseinbußen führen kann und diese bei entsprechender Modellierung verzichtbar sind, wurde auf die Implementierung solcher Mechanismen verzichtet. Unberührt davon bleibt selbstverständlich die Benutzung zeitkonstanter globaler Daten, da diese auf einfache Weise in den einzelnen Logischen Knoten repliziert werden können. Ein Beispiel hierfür ist die Definition der Verkehrsszenarien der in Abschnitt 7.4.3 beschriebenen Simulation des Signalisiersystems Nr. 7. Bei den Abläufen der einzelnen Szenarien handelt es sich zwar um globale Daten, sie werden aber zu Beginn der Simulation einmalig definiert und können deswegen in allen Logischen Knoten lokal gespeichert werden.

Um eine einfache Partitionierung des Simulationsmodells zu ermöglichen, sollte außerdem der Informationsaustausch zwischen Modellkomponenten, die auf unterschiedliche Logische Prozesse verteilt werden sollen, ausschließlich über Nachrichten erfolgen. Da in der Architektur der sequentiellen Bibliothek dies aber ohnehin bevorzugt wird, entsteht hierdurch für den Anwender kein Nachteil.

6.7.2 Konservative Methoden

Sind obige Voraussetzungen erfüllt, müssen noch einige Anpassungen vorgenommen werden, auf die im folgenden eingegangen werden soll. Ist von Anfang an klar, daß nur konservative Synchronisationsverfahren eingesetzt werden sollen, genügen wenige, verwaltungstechnische Änderungen des sequentiellen Programms, damit es mit der parallelen Bibliothek zusammenarbeitet.

6.7.2.1 Simulationssteuerung

Die erste Ergänzung betrifft die Simulationssteuerung, die den Ablauf der einzelnen Simulationsphasen (Warmlaufphase und Teiltests) steuert. Komponenten der sequentiellen Bibliothek müssen von der Klasse *TSimulationControl* abgeleitet sein, wenn sie über Beginn oder Ende dieser Phasen informiert werden wollen. Dies gilt sowohl für verschiedene Simulationsmodellkomponenten als auch für sämtliche Statistiken und einige Meßgeräte. Alle Objekte der Klasse *TSimulationControl* werden von einem zentralen Managerobjekt, einem Objekt der Klasse *TSimulationControlManager*, verwaltet, bei dem sie sich bei ihrer Generierung automatisch registrieren.

Da sich bei der parallelen Simulation mehrere asynchron laufende Logische Prozesse in einem Logischen Knoten befinden können, müssen nun die Simulationsphasen in jedem LP getrennt gesteuert werden. Aus diesem Grund besitzt jeder LP ein eigenes Managerobjekt vom Typ *TLPSimCtrlMgr*, an dem sich die im jeweiligen LP befindlichen Komponenten registrieren

müssen. Objekte vom Typ *TSimulationControl* registrieren sich aber immer an einem zentralen Managerobjekt, das sich aufgrund der Architektur der sequentiellen Bibliothek nur für den gesamten Logischen Knoten setzen läßt.

Eine Möglichkeit zur Lösung dieses Problems wäre die Abänderung des Konstruktors von *TSimulationControl*, so daß dieser das Managerobjekt, an dem er sich registrieren soll, explizit als Übergabeparameter verlangt. Dies hätte allerdings einen erheblichen Eingriff in die bestehende sequentielle Bibliothek bedeutet, der zudem bei rein sequentiellen Simulationen von geringem Nutzen gewesen wäre. Aus diesem Grund wurde diese Möglichkeit verworfen. Stattdessen wurde das zentrale Managerobjekt so gestaltet, daß es Registrierungen automatisch an die richtigen Managerobjekte in den LPs weiterleitet. Hierzu wurde das Konzept der Mixin-Klasse verwandt (siehe Paragraph 3.2.4.3.2). Die Mixin-Klasse *TLPSimulationControl* erwartet bei ihrer Generierung eine Referenz auf den lokalen Simulations-Manager, welche sie umgehend an das zentrale Managerobjekt meldet. Letzteres leitet dann alle Registrierungen, die fortan eintreffen, an den zuletzt gemeldeten lokalen Simulations-Manager weiter.

Konkret bedeutet dies, daß eine existierende Komponente, die von *TSimulationControl* abgeleitet ist oder die auf einer niedrigeren Hierarchiestufe davon abgeleitete Komponenten enthält, zusätzlich als erstes von *TLPSimulationControl* abgeleitet werden muß. Da in *C++* Konstruktoren von Basisklassen in der Reihenfolge deren Auflistung bei der Deklaration der abgeleiteten Klasse ausgeführt werden, wird auf diese Weise der Konstruktor von *TLPSimulationControl* zuerst bearbeitet und der lokale Simulations-Manager an das zentrale Managerobjekt gemeldet. Dadurch registrieren sich Objekte vom Typ *TSimulationControl* automatisch an ihrem lokalen Simulations-Manager, und es muß in die u. U. sehr komplexen Komponenten nur auf oberster Hierarchieebene eingegriffen werden.

6.7.2.2 Simulationsmodellkomponenten

Ein weiterer Aspekt betrifft die Erzeugung von Zufallsverteilungen. Diese sind in der sequentiellen Bibliothek von der Klasse *TDistribution* abgeleitet und besitzen eine Referenz auf einen Zufallszahlengenerator. Letztere wird im Konstruktor mit der in der Klasse *TRandomNumberGenerator* enthaltenen statischen Variablen *pSystemRNG* initialisiert. Es handelt sich dabei um einen statischen Zeiger, der im Normalfall auf einen zentralen Zufallszahlengenerator zeigt.

Da bei der parallelen Simulation jeder Logische Prozeß einen eigenen Zufallszahlengenerator besitzt und sich mehrere LPs in einem Logischen Knoten befinden können, entsteht hier ein ähnliches Problem wie oben bei der Simulationssteuerung. Und ebenso wurde ein ähnlicher Lösungsansatz gewählt. Die Mixin-Klasse *TParEntity* setzt bei ihrer Erzeugung *pSystemRNG* auf den Zufallszahlengenerator des lokalen Logischen Prozesses. Alle fortan

erzeugten Verteilungen bekommen damit den lokalen Generator zugewiesen. *TParEntity* ermöglicht, ähnlich wie *TEntity* in der sequentiellen Bibliothek, außerdem den Zugriff auf einige Verwaltungsobjekte des Logischen Prozesses, wie z. B. den lokalen Simulations-Manager oder den Zustandssicherungs-Manager, sowie die Ermittlung der erweiterten lokalen Zeit und das Eintragen von Ereignissen mit erweiterten Zeiten. Simulationsmodellkomponenten, die in der sequentiellen Bibliothek von *TEntity* abgeleitet sind, werden für die parallele Simulation einfach zusätzlich zuerst von *TParEntity* abgeleitet.

6.7.2.3 Nachrichten

Eine letzte wesentliche Änderung für die Benutzung konservativer Verfahren betrifft die Nachrichten. Wie in Unterabschnitt 5.3.1.3 dargestellt, bestehen Simulationsnachrichten in der parallelen Bibliothek aus zwei Teilen: einem Handle-Objekt und einer Nachrichten-Repräsentation. Letztere enthält die Nutzinformation und ist aus Anwendersicht die „eigentliche" Nachricht. In der sequentiellen Bibliothek dagegen leitet der Anwender üblicherweise eigene Nachrichten von der Klasse *TMessage* ab, welche in der parallelen Bibliothek Basisklasse der Handle-Klassen ist. Bei einer Parallelisierung muß der Anwender also eine Abspaltung der Nutzinformation in eine separate Repräsentationsklasse vornehmen. Normalerweise kann dies dadurch geschehen, daß eigene Nachrichten statt von *TMessage* von *TParSimMsgRep* bzw. *TParTWSimMsgRep* abgeleitet und Zugriffe auf die Nutzinformation entsprechend umgelenkt werden.

6.7.3 Optimistische Methoden

Waren die Änderungen für konservative Simulation verwaltungstechnischer Natur und „schematisch" nach festgelegten Regeln vorzunehmen, erfordert die Anpassung an optimistische Techniken ein tieferes Verständnis der Vorgänge im Simulationsmodell selbst. Der Anwender muß in der Lage sein, die zu sichernden Daten und die Stellen, an denen diese geändert werden, zu identifizieren.

6.7.3.1 Voraussetzungen

Eine wichtige Voraussetzung für eine einfache Anpassung ist eine saubere Datenkapselung. Erlaubt ein Objekt anderen Objekten weitgehend freien Zugriff auf die eigenen Daten (sei es über *Friend*-Deklarationen oder gar öffentlich zugelassenen Zugriff), ist es schwierig, alle Stellen der möglichen Änderungen dieser Daten aufzufinden und dort Modifikationen für die Zustandssicherung anzubringen. Erfolgt der Datenzugriff dagegen ausschließlich über

wohldefinierte Methoden des Objekts selbst, wie es bei sauberer Umsetzung des objektorientierten Paradigmas der Normalfall sein sollte, sind die Änderungen der Daten eines Objekts in dessen Methoden konzentriert. Dadurch lassen sie sich wesentlich einfacher auffinden, und die Modifikationen beschränken sich auf diese Methoden.

6.7.3.2 Grundsätzliche Arbeiten

Ein wichtiger erster Schritt, um die in Abschnitt 5.4.3 aufgeführten Bedingungen für die Teilnahme einer Komponente an der Zustandssicherung zu erfüllen, ist die Aufteilung der Daten in während des Simulationslaufs veränderliche und unveränderliche. Alle Klassen, die veränderliche Daten enthalten, müssen anschließend zusätzlich von der Klasse *TStateSaveCtrl* abgeleitet werden.

6.7.3.3 Zustandskopien oder inkrementelle Zustandssicherung?

Als nächstes ist die Frage zu klären, welche Arten der Zustandssicherung für die einzelnen Komponenten verwandt werden soll. Ist von vornherein klar, daß nur eine Sicherung mittels Zustandskopien Sinn macht (beispielsweise bei Komponenten mit kleinen Datenmengen, die sehr häufigen Änderungen unterliegen), genügt es, eine von *TStateCopy* abgeleitete Klasse zur Verfügung zu stellen, mit deren Hilfe diese Daten gesichert und restauriert werden können (s. a. Abschnitt 5.4.2).

Da das Sicherungsobjekt für Zustandskopien derart optimiert ist, daß eine Zustandskopie nur erstellt wird, falls der Zustand sich seit der letzten Erstellung geändert hat (ansonsten wird lediglich die „Gültigkeitsdauer" der letzten Kopie verlängert, siehe Unterkapitel 4.7), müssen zusätzlich zur Identifizierung der veränderlichen Daten auch noch die Stellen, an denen die Änderungen vorgenommen werden, herausgefunden und die Änderungen dem Sicherungsobjekt angezeigt werden. Es ist in diesem Fall aber ausreichend, eine von *TStateInc* abgeleitete Klasse sehr rudimentär zu halten, da sie nur den Zweck der Meldung an das Sicherungsobjekt erfüllen muß (s. a. Abschnitt 5.4.4).

Soll nur inkrementelle Zustandssicherung implementiert werden, müssen von *TStateInc* abgeleitete Klassen zur Verfügung gestellt werden, die die Änderungen vollwertig aufnehmen und wieder rückgängig machen können. Eine von *TStateCopy* abgeleitete Klasse muß auf jeden Fall für die Sicherung des Initialzustands auch vorhanden sein. Sie kann aber bei ausschließlich inkrementeller Sicherung u. U. einfach gehalten werden. Bei einer unvollständigen Kapselung der Daten kann es gegebenenfalls günstig sein, den Zugriff auf die Daten zuerst einzuschränken und die Änderungen in wenigen Methoden zu konzentrieren.

Für eine größtmögliche Flexibilität bei der Wahl des Zustandssicherungsverfahrens muß eine Vorbereitung sowohl auf eine Sicherung mittels Zustandskopien als auch auf eine inkrementelle Sicherung erfolgen.

6.7.3.4 Nachrichten

Um Simulationsnachrichten an die optimistische Simulation anzupassen, gelten im Prinzip die gleichen Richtlinien wie bei konservativer Simulation (s. o.). Ist die Nutzinformation in eine separate Repräsentationsklasse abgespalten, muß diese auf die Zustandssicherung vorbereitet werden. Dies geschieht nach den gleichen Regeln wie bei den Modellkomponenten. Eine Besonderheit entsteht dadurch, daß Nachrichten während der Simulation dynamisch erzeugt werden und sich danach zuerst am lokalen Sicherungs-Manager des Logischen Prozesses anmelden müssen. Dies geschieht nicht automatisch bei der Erzeugung, da Nachrichten z. B. auch in der Nachrichtenwandlungsschicht nach Empfang eines Byte-Stroms generiert werden können. Von dort werden sie an die Eingangskanäle der LPs weitergeleitet und in Warteschlangen eingereiht. Im Eingangskanal eines LPs nimmt die Nachricht jedoch nicht an der lokalen Zustandssicherung teil, für sie steht die Simulationszeit quasi still.

6.7.3.5 Statistiken

Auch die verwandten Statistiken müssen bei optimistischer Simulation eine Zustandssicherung durchführen. In der sequentiellen Bibliothek werden alle Statistiken von einem zentralen Statistik-Manager erzeugt. Durch dessen Austausch in der parallelen Bibliothek werden geeignete Statistiken automatisch erzeugt, ohne daß der Anwender in seinem Simulationsprogramm irgendwelche diesbezüglichen Änderungen vornehmen muß. Einzige Bedingung ist, daß die die Statistik erzeugende Komponente von *TStateSaveCtrl* abgeleitet ist und daß die Ausführung des Konstruktors von *TStateSaveCtrl* vor der Erzeugung der Statistik erfolgt. Dieser ist nämlich für die korrekte Zuweisung des lokalen Sicherungs-Managers verantwortlich, was den Vorteil hat, daß die Schnittstelle der die Statistiken erzeugenden Methoden nicht um eine Referenz auf diesen Manager ergänzt werden muß.

6.7.4 Klassenhierarchie einer Simulationsmodellkomponente am Beispiel einer Bedieneinheit

Bild 6.5 zeigt beispielhaft die Klassenhierarchie einer einfachen Bedieneinheit, deren Bediendauern für Nachrichten statistisch verteilt sind. Die aus der sequentiellen Bibliothek stammende Klasse *TStdPhase* wird für konservative parallele Simulation durch die beiden Mixin-Klassen *TParEntity* und *TLPSimulationControl* zur Klasse *TParStdPhase* erweitert. Weitere Maßnahmen außer diesen Ableitungen sind nicht erforderlich, um die

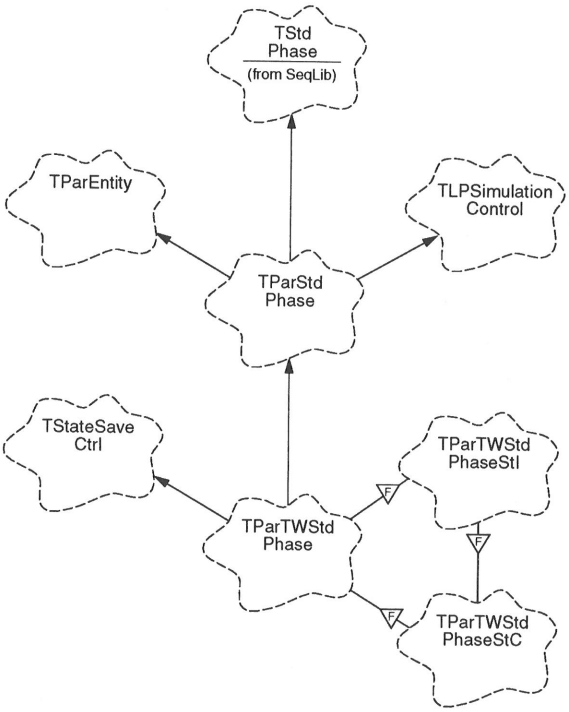

Bild 6.5: Klassenhierarchie der Bedieneinheit

für sequentielle Simulation entwickelte Komponente für konservative parallele Simulation anzupassen.

Um die Bedieneinheit auch für optimistische Synchronisationsverfahren einsetzen zu können, ist die Klasse *TParTWStdPhase* zusätzlich von *TStateSaveCtrl* abgeleitet. Außerdem sind die beiden Klassen *TParTWStdPhaseStC* für eine Kopie des Zustands der Bedieneinheit (wird eine Nachricht bedient und wenn ja welche?) und *TParTWStdPhaseStI* für die Sicherung einer Zustandsänderung (Ankunft einer neuen Nachricht bzw. Bedienende einer Nachricht) zusätzlich implementiert. Entsprechend werden auch einige Methoden durch *TParTWStdPhase* überschrieben, um Zustandsänderungen anzuzeigen.

Komponenten lassen sich also sehr einfach zuerst ausschließlich für konservative Simulation entwickeln bzw. anpassen und später für optimistische Verfahren erweitern. Letztere können dann, da von den konservativen Klassen abgeleitet, trotzdem weiterhin problemlos in einer konservativen Simulation eingesetzt werden.

Kapitel 7

Anwendung und Leistungsnachweis der Konzepte

7.1 Vorbemerkungen

In diesem Kapitel sollen die zuvor vorgestellten Konzepte auf ihre Leistungsfähigkeit hin untersucht werden. Dazu werden zuerst künstliche Modelle mit generischen Knoten- und Verbindungselementen zur Gewinnung allgemeiner Erkenntnisse untersucht und anschließend die Simulation eines Modells des Signalisiersystems Nr. 7 als reale Anwendung der Bibliothek betrachtet. Bei letzterem soll nicht nur auf Leistungsaspekte sondern auch auf die Erfahrungen bei der Anwendung der parallelen Bibliothek bei der Parallelisierung eines existierenden sequentiellen Simulationsprogramms eingegangen werden. Speziellere Untersuchungen zur Leistungsfähigkeit des entwickelten Kommunikationssystems (siehe Unterkapitel 4.6) finden sich in [Necker,1995].

7.2 Allgemeine Leistungsparameter

Die Leistungsfähigkeit einer parallelen Simulation hängt allgemein von verschiedenen Parametern ab:

1. Verhältnis Rechenzeit zu Kommunikationszeit,
2. Parallelität, die dem Simulationsmodell innewohnt,
3. Synchronisationsverfahren (inklusive z. B. Verfahren der Zustandssicherung),
4. räumliche und zeitliche Lokalität der Auswirkungen eines Ereignisses,
5. Verteilung Logischer Prozesse auf Rechenknoten (*Mapping*),

6. Leistungsfähigkeit des Simulationswerkzeugs,
7. Rechenplattform.

Die ersten vier Punkte sind Gegenstand mehrerer Parameter-Untersuchungen, auf die im folgenden Unterkapitel näher eingegangen wird, während der fünfte Punkt in der vorliegenden Arbeit nicht näher untersucht wird. Getestet werden soll vor allem die Leistungsfähigkeit des in den vorangegangenen Kapiteln vorgestellten Werkzeugs und der beschriebenen Softwarekonzepte (Punkt 6). Als Rechenplattform (Punkt 7) für die Untersuchungen wurde ein Parallelrechner *Intel Paragon XP/S-5* verwandt (siehe Abschnitt 3.1.4).

Neben den oben aufgeführten allgemeinen Punkten gibt es verschiedene, für spezielle Synchronisationsverfahren wichtige Parameter, wie z. B.:

- Häufigkeit der Zustandssicherung (optimistische Synchronisation),
- Häufigkeit der GVT-Approximation (optimistische Synchronisation),
- Größe des verwertbaren Lookahead (konservative Synchronisation).

7.3 Künstliche Modelle

7.3.1 Generische Knoten- und Verbindungselemente

Zur Durchführung verschiedener Leistungsuntersuchungen wurden künstliche Modelle mit eigens dafür entwickelten generischen Elementen verwandt. Diese Modelle bestehen aus Knotenelementen (Bild 7.1), die als Quelle und als Senke für Nachrichten dienen und beliebig viele Ein- und Ausgänge haben können. Diese Knotenelemente werden über Verbindungselemente (Bild 7.2) miteinander verbunden.

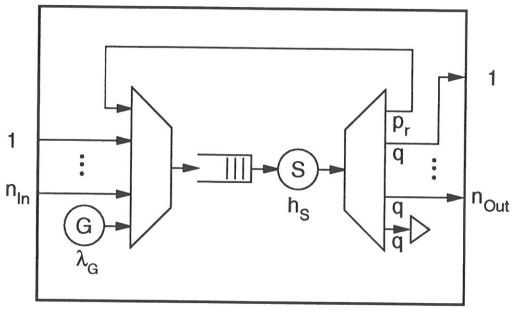

Bild 7.1: Generisches Knotenelement

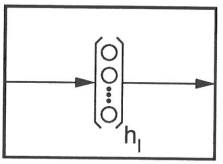

Bild 7.2: Generisches Verbindungselement

Ein Knotenelement enthält einen Generator G, der eine Folge von Nachrichten erzeugt, deren zeitliche Abstände eine Folge von unabhängigen, identisch verteilten Zufallsvariablen bilden. Die mittlere Erzeugungsrate der Nachrichten wird mit λ_G bezeichnet. Die Nachrichten werden zusammen mit Nachrichten der n_{In} externen Eingänge des Knotenelements über einen Multiplexer an eine Warteschlange mit unbegrenzter Zahl an Warteplätzen weitergereicht, hinter der die Bedieneinheit S folgt. Die Bedienzeiten der Nachrichten durch S sind unabhängig und identisch verteilt mit der mittleren Bediendauer h_S. Der hinter S folgende Demultiplexer leitet die Nachrichten mit der Rückkoppelwahrscheinlichkeit p_r zurück an den Multiplexer und damit zur erneuten Bedienung durch S. Mit der Wahrscheinlichkeit $1 - p_r$ geht sie dagegen entweder an die Senke oder an einen der n_{Out} Ausgänge des Knotenelements, wobei gilt: $1 - p_r = (n_{Out} + 1)q$.

Ein Verbindungselement modelliert eine Verzögerung auf dem Übertragungsabschnitt zwischen zwei Knotenelementen. Es enthält einen „Infinite Server", ein Bediensystem bestehend aus einer unendlichen Anzahl unabhängig, parallel arbeitender Bedieneinheiten. Die Bedienzeiten der Nachrichten durch den „Infinite Server" sind ebenfalls unabhängig und identisch verteilt mit der mittleren Bediendauer h_I.

Mit den vorgestellten Komponenten lassen sich einige wichtige Leistungsparameter für die parallele Simulation untersuchen. So läßt sich über die Rückkoppelwahrscheinlichkeit p_r die mittlere Anzahl von Durchläufen einer Nachricht durch das System aus Multiplexer, Warteschlange, Bedieneinheit und Demultiplexer vor dem Verlassen Richtung Senke oder Ausgänge einstellen und damit letztlich das Verhältnis Rechenzeit zu Kommunikationszeit in jedem Knoten. Diese mittlere Anzahl von Durchläufen, n_{M_S}, ist geometrisch verteilt und läßt sich berechnen nach: $n_{M_S} = 1/(1 - p_r)$. Die Art der Verteilungsfunktion im Verbindungselement und die Größe von h_I beeinflussen die zeitliche Lokalität der Auswirkungen eines Ereignisses auf andere Knotenelemente und insbesondere den verfügbaren Lookahead bei konservativen Verfahren (s. a. Unterabschnitt 2.5.1.2).

7.3.2 Untersuchte Modellparameter und Topologien

Tabelle 7.1 zeigt die Modellparameter, deren Einfluß auf die Leistungsfähigkeit des parallelen Simulationssystems untersucht wurde. Die Bedeutung der Rückkoppelwahrscheinlichkeit, p_r, und der Bedienzeit des „Infinite Server", h_I, wurden im letzten Abschnitt schon erläutert. Da alle LPs am Ende einer Simulationsphase (d. h. am Ende der Warmlaufphase und jedes Teiltests) synchronisiert werden, ist die Teiltestlänge, T_{PTL}, aufgrund der unvermeidlichen Synchronisationsverluste ein weiterer untersuchter Modellparameter. Die Verluste rühren einerseits daher, daß die einzelnen LPs am Ende einer Simulationsphase blockiert werden, und andererseits daher, daß das Gesamtsystem nach dem Beginn einer Simulationsphase jedesmal eine gewisse Zeit benötigt, bis es wieder eingeschwungen ist. Bei optimistischen

Modellparameter	Variable	untersuchter Einfluß
Anzahl generischer Knotenelemente[1]	n_{Nd}	Modellgröße
Teiltestlänge	T_{PTL}	Synchronisationsaufwand am Teiltestende und Dauer des Einschwingens nach der Synchronisation
Rückkoppelwahrscheinlichkeit	p_r	Verhältnis Rechenzeit zu Kommunikationszeit in jedem Knoten
Bedienzeit des „Infinite Server“	h_I	Größe des Lookahead
Zustandssicherungs-Intervall	$n_{E_{SS}}$	Anzahl bearbeiteter Ereignisse zwischen zwei Zustandssicherungen (optimist.)
GVT-Anforderungs-Intervall	$n_{S_{GVT}}$	Anzahl Zustandssicherungen zwischen zwei GVT-Anforderungen (optimist.)
Maximale Anzahl bearbeiteter Ereignisse pro Zeitschlitz	$n_{E_{Sl}}$	Häufigkeit des Empfangs externer Nachrichten

Tabelle 7.1: Untersuchte Modellparameter

Verfahren kommt hinzu, daß bei jedem Erreichen des Simulationsphasenendes eine GVT-Berechnung angefordert wird, woraus eine Erhöhung der Anforderungsrate resultiert.

Die einzelnen Logischen Prozesse erhalten von einem Scheduler in jedem Logischen Knoten Rechenzeit zugeteilt (siehe Unterkapitel 4.4 und Unterabschnitt 5.2.5.2). Der Modellparameter $n_{E_{Sl}}$ gibt an, wieviele Ereignisse ein Logischer Prozeß maximal pro Zeitschlitz bearbeiten darf, bevor er die Kontrolle an den Logischen Knoten zurückgeben muß. Dieser Wert darf nicht zu groß sein, da auf externe Nachrichten sonst zu spät reagiert wird. Dies kann zum einen zu einer Verstopfung des Message-Passing-Subsystems führen, zum anderen bei optimistischen Verfahren zu häufigerem Zurücksetzen. $n_{E_{Sl}}$ darf aber auch nicht zu klein gewählt werden, da das Zurückgeben der Kontrolle, das Empfangen von Nachrichten und das erneute Einplanen eines LPs mit einem gewissen Aufwand verbunden sind.

Zwei Parameter, die speziell für optimistische Synchronisationsverfahren von Bedeutung sind, sind das Zustandssicherungs-Intervall, $n_{E_{SS}}$, und das GVT-Anforderungs-Intervall, $n_{S_{GVT}}$. Ersteres gibt die Anzahl der bearbeiteten Ereignisse zwischen zwei Zustandssicherungen an. Ist $n_{E_{SS}}$ zu groß, dann ergeben sich beim Zurücksetzen Leistungseinbußen aufgrund großer Coast-Forward-Phasen und damit einer hohen Anzahl unnötigerweise nochmals simulierter Ereignisse. Ist dieser Parameter zu klein, wirkt sich der Aufwand für die häufige Erstellung von Zustandskopien negativ aus. Bei den Messungen wurde für die Simulationsmodellkomponenten die in Unterkapitel 4.7 beschriebene optimierte Erstellung von Zustandskopien als Zustandssicherungsverfahren angewandt.

[1]Entspricht bei der für die Messungen gewählten Aufteilung des Modells der Anzahl eingesetzter Rechenknoten (ausgenommen Messungen in Abschnitt 7.3.7).

Der zweite Parameter, das GVT-Anforderungs-Intervall, $n_{S_{GVT}}$, gibt die Anzahl durchgeführter Zustandssicherungen zwischen zwei GVT-Anforderungen an. Jeder Logische Prozeß fordert in diesen Intervallen eine neue GVT-Berechnung am Manager-Knoten an (siehe Unterkapitel 5.5). Es hat sich als günstig erwiesen, dabei die absolute Zahl an Zustandssicherungen zu betrachten, ungeachtet eines evtl. Zurücksetzens. Auch bei diesem Parameter muß wieder abgewogen werden: Ein großes Intervall bedeutet selteneres Aufräumen nicht mehr benötigter Zustandssicherungen (*Fossil Collection*), was wiederum zu erhöhtem Speicherplatzverbrauch und einer aufwendigeren Listenverwaltung führt. Häufig durchgeführte GVT-Approximationen dagegen erhöhen das Nachrichtenaufkommen und belasten die einzelnen LPs übermäßig mit dem Aufwand für ihre Bearbeitung. Für alle folgenden Messungen wurde das Verfahren nach [Chandy&Lamport,1985] zur GVT-Approximation verwandt.

Die beschriebenen Modellparameter erlauben die Untersuchung der wichtigsten in Unterkapitel 7.2 beschriebenen allgemeinen Leistungsparameter für die parallele Simulation. Ihre Auswirkungen wurden für verschiedene Modellgrößen (repräsentiert durch die Anzahl der generischen Knotenelemente, n_{Nd}) und Topologien untersucht. Für letztere wurden die folgenden Anordnungen gewählt:

- Tandem-Anordnung,
- Ring-Anordnung,
- Gitter-Anordnung,

jeweils mit uni- und bidirektionalen Verbindungen zwischen den einzelnen Knotenelementen. Soweit nicht anders angegeben[2], war in jedem Logischen Prozeß genau ein generisches Knotenelement und für jede von diesem abgehende Verbindung genau ein generisches Verbindungselement enthalten. Jeder Logische Prozeß war in einem eigenen Logischen Knoten angesiedelt, der wiederum auf einem eigenen Physikalischen Knoten ausgeführt wurde.

Der oben nicht aufgeführte Modellparameter der mittleren Bedienzeit der Bedieneinheit (h_S) beeinflußt die mittlere Länge der Warteschlange. Da die Knoten im System unterschiedliche Gesamt-Ankunftsraten von Nachrichten haben, andererseits aber die Modellparameter aller Knoten die gleichen Werte haben sollen, ist der Knoten mit der höchsten Gesamt-Ankunftsrate maßgebend für die maximale mittlere Warteschlangenlänge im System. Ausführliche Betrachtungen zu deren Berechnung finden sich in [ten Brink,1995]. Für die folgenden Messungen wurde h_S jeweils so gewählt, daß diese maximale mittlere Warteschlangenlänge bei jedem Modell ungefähr 2 betrug.

Mit Hilfe der Erzeugungsrate für Nachrichten durch den Generator, λ_G, lassen sich T_{PTL}, h_I und h_S skalieren. Für alle hier gezeigten Messungen galt $\lambda_G = 1$. Alle genannten Simulationszeit-Größen sind normiert auf 1 SZE (Simulationszeiteinheit) und die Raten entsprechend auf $1/SZE$, so daß diese Größen dimensionslos sind.

[2] Ausnahme: Messungen in Abschnitt 7.3.7.

7.3.3 Vorgehensweise bei den Untersuchungen

Als Synchronisationsverfahren für die folgenden Untersuchungen wurden jeweils ein konservatives und ein optimistisches Verfahren verwandt. Ersteres bediente sich der Vermeidung von Verklemmungen durch NULL-Nachrichten (Unterabschnitt 2.5.1.2), bei letzterem handelte es sich um „reines" Time Warp mit Aggressive Cancellation (Unterabschnitt 2.5.2.2).

Sequentielle Vergleichsmessungen wurden mit dem entwickelten Werkzeug mit nur einem Logischen Knoten und einem Logischen Prozeß durchgeführt, wobei die eigens zu diesem Zweck entwickelten Klassen *TIsolatedLP* und *TFastIsolatedLPCal* (siehe Abschnitte 5.2.2 und 5.2.4) eingesetzt wurden. *TIsolatedLP* steht für einen „isolierten" Logischen Prozeß, der keine Außenbeziehungen über Kanäle unterhalten kann und der deswegen auch keinen Synchronisationsbedarf hat. Mit der Klasse *TFastIsolatedLPCal* steht ein speziell für sequentielle Simulation optimierter Ereigniskalender zur Verfügung, wie er in [Brown,1988] vorgestellt wurde. Er vermeidet auch die Verwendung des in Unterabschnitt 5.2.2.1 vorgestellten Konzepts für einen erweiterten Zeitbegriff. Beide Klassen stellen somit eine einfache Möglichkeit dar, ein parallelisiertes Simulationsmodell auch sequentiell zu simulieren.[3]

In die Speedup-Messungen floß nur die Zeit für den reinen Simulationslauf ein (siehe Bild 4.3), Zeiten für das Starten und Initialisieren der Applikation sowie für den Aufbau und die Initialisierung des Simulationsmodells wurden nicht berücksichtigt. In diesen Punkten schneidet die parallele Simulation aufgrund ihrer Verteilung meist schlechter ab als die sequentielle. Andererseits hängen die Lade-, Aufbau- und Initialisierungszeiten zu einem Großteil von der Anzahl eingesetzter Physikalischer Knoten ab und nicht von der Länge der Simulation selbst. Da ihr Einfluß daher mit zunehmender Simulationsdauer schwindet und sie zudem stark von der aktuellen und oft schwer nachvollziehbaren Belastung der E/A-Einheiten des Rechensystems abhängen, wurde ihr Einfluß nicht berücksichtigt.

Um die statistische Aussagesicherheit der gemessenen Speedup-Werte zu ermitteln, wurden die Simulationsläufe in jeweils 10 Teiltests unterteilt, der Speedup für jeden Teiltest getrennt bestimmt und schließlich der Mittelwert und mit Hilfe der Stichprobentheorie dessen 95 %-Konfidenzintervall bestimmt (siehe z. B. [Kühn,1995]). Um sicherzustellen, daß das System sich für die Messungen in einem eingeschwungenen Zustand befand, wurde vorab eine Warmlaufphase mit der Länge eines Teiltests durchgeführt.

Die Bediendauer der Bedieneinheit des generischen Knotenelements h_S war in allen folgenden Messungen negativ-exponentiell verteilt, die Bediendauer des „Infinite Server" des generischen Verbindungselements h_I war konstant.

[3] Gegenüber der neuesten Version der sequentiellen Bibliothek war die Laufzeit dadurch ca. 30 % länger. Ein direkter Vergleich wäre allerdings verzerrt, da deren Implementierung in mehrjähriger Arbeit sehr stark optimiert wurde und das parallele Werkzeug z. T. auf einer älteren Version basiert. Zudem bietet es auch für sequentielle Simulation einen höheren Funktionsumfang (z. B. zur Konfiguration) und ist deshalb deutlich umfangreicher. Hier kommen u. a. Einflüsse wie effiziente Code-Erzeugung und -Bearbeitung zum Tragen.

7.3.4 Tandem-Anordnungen

7.3.4.1 Unidirektional

Bild 7.3 zeigt eine Tandem-Anordnung der generischen Knotenelemente aus Abschnitt 7.3.1, die unidirektional mittels der ebenfalls dort beschriebenen generischen Verbindungselemente verbunden sind.

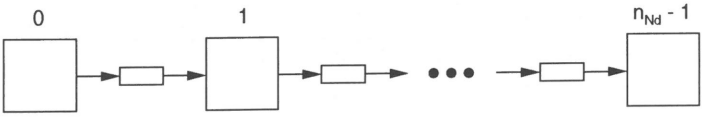

Bild 7.3: Unidirektionale Tandem-Anordnung

In einem ersten Schritt wurde für dieses Modell der Einfluß der Teiltestlänge T_{PTL} und der Rückkoppelwahrscheinlichkeit p_r auf den erzielbaren Speedup untersucht. Bild 7.4 zeigt letzteren bei einer Modellgröße von $n_{Nd} = 10$ Knoten aufgetragen über T_{PTL} für das konservative und das optimistische Verfahren jeweils für $p_r = 0{,}0$ ($n_{M_S} = 1$), $p_r = 0{,}9$ ($n_{M_S} = 10$)

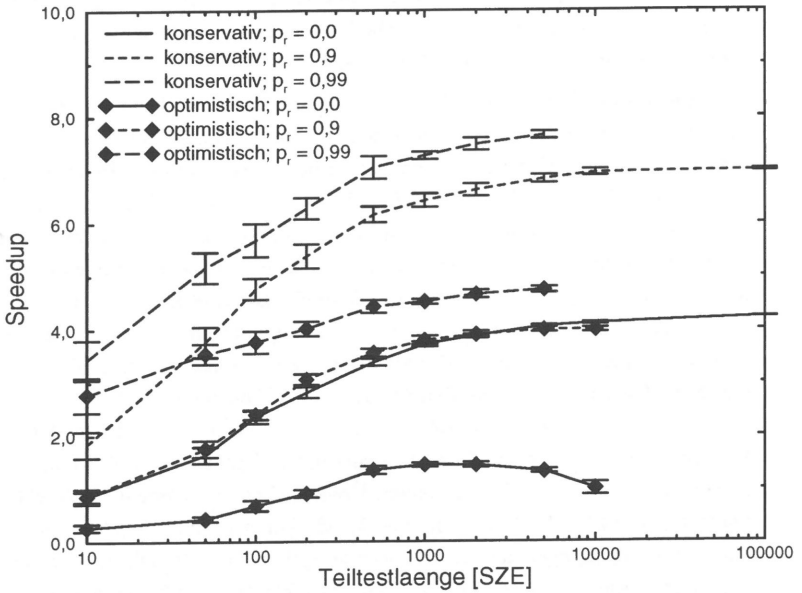

Bild 7.4: Speedup für unidirektionale Tandem-Anordnung in Abhängigkeit der Teiltestlänge und der Rückkoppelwahrscheinlichkeit (10 Knoten)

und $p_r = 0{,}99$ ($n_{M_S} = 100$). Die übrigen Parameter hatten die in der folgenden Tabelle aufgelisteten Werte:

	λ_G	h_S	h_I	$n_{E_{SI}}$	$n_{E_{SS}}$	$n_{S_{GVT}}$
konservativ, $p_r = 0{,}0$	1	0,366	0,5	10000	–	–
konservativ, $p_r = 0{,}9$	1	0,0366	0,5	10000	–	–
konservativ, $p_r = 0{,}99$	1	0,00366	0,5	1000	–	–
optimistisch, $p_r = 0{,}0$	1	0,366	0,5	50	300	30
optimistisch, $p_r = 0{,}9$	1	0,0366	0,5	50	300	30
optimistisch, $p_r = 0{,}99$	1	0,00366	0,5	50	300	1

Es lassen sich folgende Schlüsse aus dem Ergebnis ziehen:

- Das konservative Verfahren ist bei diesem vorwärtsgerichteten Modell grundsätzlich besser als das optimistische. Ersteres wird im eingeschwungenen Zustand durch die nur in eine Richtung bestehenden Abhängigkeiten kaum blockiert, so daß hier vor allem die Kosten für den Nachrichtenaustausch zum Tragen kommen. Beim optimistischen Verfahren gibt es wenig Spielraum für eine fehlerfreie spekulative Voraussimulation, so daß der zusätzliche Aufwand für die Zustandssicherung und die Approximation der GVT nicht kompensiert werden kann.
- Mit wachsender Rückkoppelwahrscheinlichkeit p_r vergrößert sich das Verhältnis von Rechenzeit zu Kommunikationszeit. Die Kosten für die Kommunikation fallen weniger ins Gewicht und der Speedup verbessert sich.
- Mit zunehmender Teiltestlänge fallen die Synchronisationsverluste am Teiltestende und die Einschwingphase zu Beginn eines neuen Teiltests weniger ins Gewicht, so daß sich beim konservativen Verfahren grundsätzlich der Speedup erhöht. Beim optimistischen Verfahren tritt mit zunehmender Teiltestlänge folgendes Problem auf: Durch den unidirektionalen Nachrichtenfluß müssen nachgeordnete Knoten nicht nur die durch ihren eigenen Generator erzeugten Nachrichten bearbeiten sondern auch einen Teil der in den vorderen Knoten erzeugten (ausführliche Betrachtungen hierzu finden sich in [ten Brink,1995]). Die vorderen Knoten sind dadurch weniger belastet als die übrigen und können somit schneller in der Simulationszeit voranschreiten als die hinteren. Da eine Synchronisation erst am Teiltestende erfolgt, laufen die lokalen Uhren der einzelnen Logischen Prozesse umso weiter auseinander, je länger ein solcher Teiltest dauert. Dies resultiert in einem erhöhten Speicherplatzaufwand für die Zustandssicherungen in den vorauseilenden LPs und einem Überangebot an externen Nachrichten in den nachfolgenden LPs. Dadurch kann der Speedup bei zunehmendem T_{PTL} wieder sinken oder aufgrund mangelnden Speichers eine optimistische Simulation unmöglich werden. Der Einsatz eines Zeitfensters (siehe Unterabschnitt 2.5.3.2) könnte hier Abhilfe schaffen. Mangelnder Speicher der einzelnen Rechenknoten ist auch der Grund, weswegen die optimistischen Messungen für höhere Werte von T_{PTL} nicht mehr durchgeführt

werden konnten. Die Kurve für konservative Simulation und $p_r = 0{,}99$ bricht dagegen bei $T_{PTL} = 5000$ ab, da die sequentielle Vergleichsmessung für höhere Werte nicht mehr in der auf der Rechenplattform am Stück verfügbaren Rechenzeit durchgeführt werden konnte.

Den Speedup der unidirektionalen Tandem-Anordnung in Abhängigkeit der Modellgröße zeigt Bild 7.5. Die Parameter für diese Messungen hatten die Werte: $T_{PTL} = 5000$, $\lambda_G = 1$, $h_S = 0{,}0366$, $h_I = 0{,}5$, $p_r = 0{,}9$, $n_{E_{SS}} = 300$, $n_{S_{GVT}} = 30$, $n_{E_{SI}} = 10000$ (konservativ) bzw. $n_{E_{SI}} = 50$ (optimistisch).

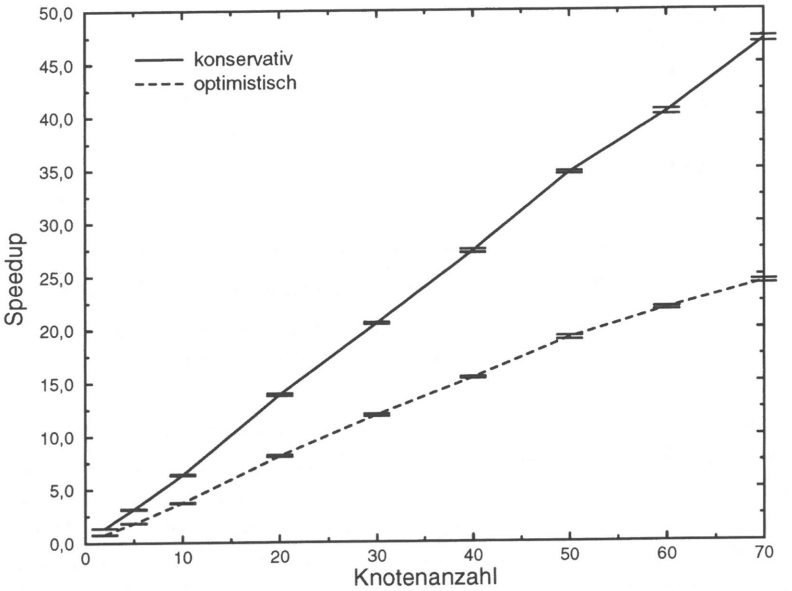

Bild 7.5: Speedup für unidirektionale Tandem-Anordnung in Abhängigkeit der Modellgröße

7.3.4.2 Bidirektional

In der in diesem Unterabschnitt getesteten Tandem-Anordnung sind die Knotenelemente im Gegensatz zum letzten Unterabschnitt jeweils bidirektional verbunden (Bild 7.6).

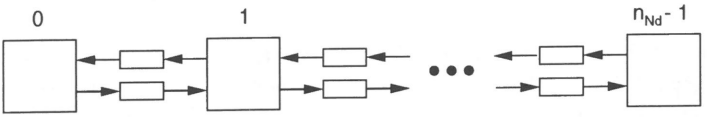

Bild 7.6: Bidirektionale Tandem-Anordnung

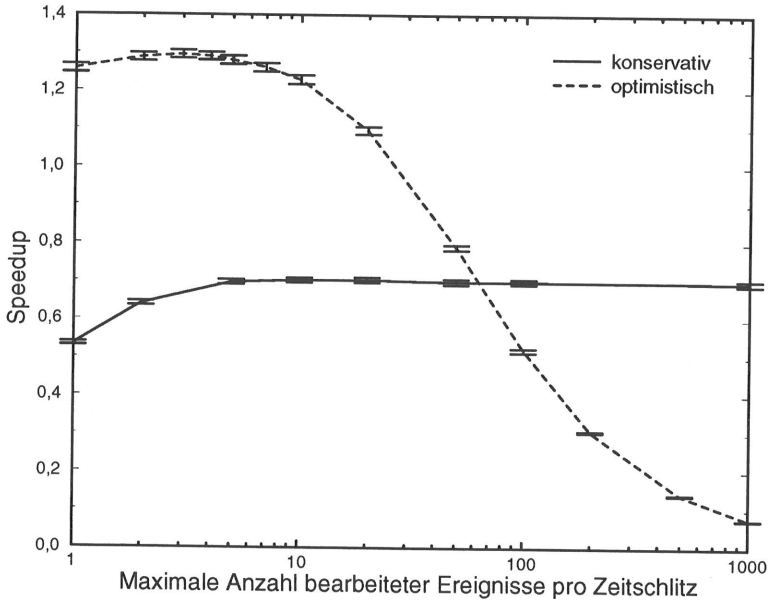

Bild 7.7: Speedup für bidirektionale Tandem-Anordnung in Abhängigkeit der Anzahl maximal bearbeiteter Ereignisse pro Zeitschlitz (10 Knoten)

In einem ersten Versuch mit diesem Modell wurde die Abhängigkeit des Speedup von der maximalen Anzahl bearbeiteter Ereignisse pro Zeitschlitz, $n_{E_{SI}}$, untersucht. Bild 7.7 zeigt die Ergebnisse für $n_{Nd} = 10$ Knoten (übrige Parameter: $T_{PTL} = 5000$, $\lambda_G = 1$, $h_S = 0{,}0244$, $h_I = 0{,}5$, $p_r = 0{,}9$, $n_{E_{SS}} = 10$, $n_{S_{GVT}} = 250$). Das konservative Verfahren ist relativ unempfindlich gegenüber der Variation von $n_{E_{SI}}$. Bei kleinen Werten macht sich der Aufwand für das häufige Zurückgeben der Kontrolle durch den Logischen Prozeß, das Empfangen von Nachrichten und das erneute Einplanen eines LPs bemerkbar. Bei größeren Werten geht der Speedup in die Sättigung, da die maximale Anzahl bearbeitbarer Ereignisse aufgrund von Blockierungen der LPs überhaupt nicht erreicht wird (bei konservativer Simulation wird die Kontrolle bei Blockierung vorzeitig zurückgegeben). Das optimistische Verfahren hat ein Maximum bei $n_{E_{SI}} = 3$. Bei kleineren Werten überwiegt wie beim konservativen Verfahren der Verwaltungsaufwand, bei größeren wird zu spät auf externe Nachrichten reagiert, was zu häufigerem Zurücksetzen und unnötigen, später zurückgesetzten Falschberechnungen führt.

In einem zweiten Versuch mit der bidirektionalen Tandem-Anordnung (Bild 7.8) wurde die Abhängigkeit des Speedup von der Modellgröße und der Größe des Lookahead untersucht. Die Werte der festen Parameter waren folgende: $T_{PTL} = 5000$, $\lambda_G = 1$, $h_S = 0{,}0244$, $p_r = 0{,}9$, $n_{E_{SS}} = 10$, $n_{S_{GVT}} = 350$, $n_{E_{SI}} = 1000$ (konservativ) bzw. $n_{E_{SI}} = 3$ (optimi-

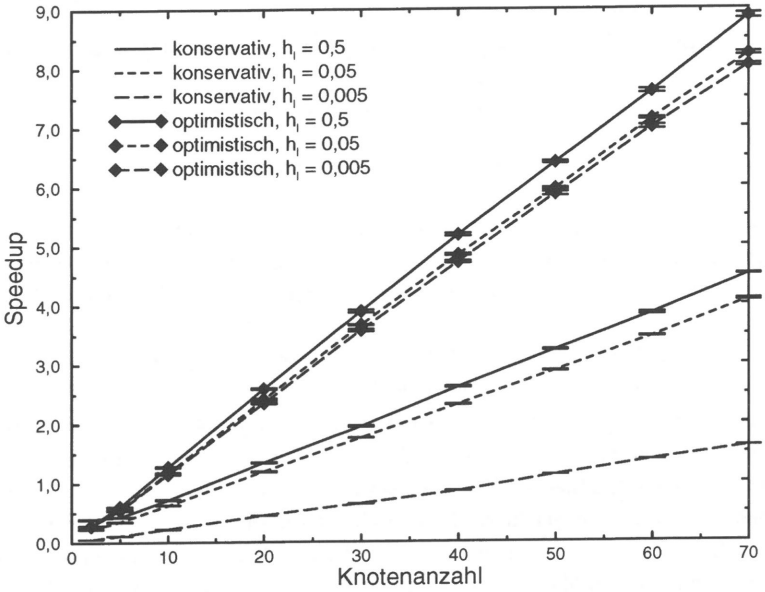

Bild 7.8: Speedup für bidirektionale Tandem-Anordnung in Abhängigkeit der Modellgröße

stisch). Variiert wurde neben der Anzahl der Knotenelemente, n_{Nd}, auch die Bediendauer des „Infinite Server", h_I. Aufgrund der Ergebnisse erkennt man folgendes:

- Das optimistische Verfahren ist bei dieser direkten gegenseitigen Abhängigkeit benachbarter Logischer Prozesse schneller als das konservative.
- Während das optimistische Verfahren weitgehend unempfindlich gegenüber Änderungen von h_I ist, hängt die Leistung des konservativen von diesem Parameter und damit vom verfügbaren Lookahead stark ab. Letzteres deckt sich mit Ergebnissen aus der Literatur (z. B. [Fujimoto,1988]). Besonders auffällig wird das Problem durch eine kleine Änderung am Modell: Wäre die Bediendauer des „Infinite Server" nicht konstant sondern negativ-exponentiell verteilt, dann wäre der verfügbare Lookahead Null und das konservative Verfahren gar nicht ohne weiteres anwendbar (s. a. Unterabschnitt 2.5.1.2).

7.3.5 Ring-Anordnungen

7.3.5.1 Unidirektional

Als nächstes wurde die unidirektionale Tandem-Anordnung zu einem Ring erweitert (siehe Bilder 7.9 und 7.10) und mit den Parametern $T_{PTL} = 5000$, $\lambda_G = 1$, $h_S = 0{,}0366$, $h_I = 0{,}5$,

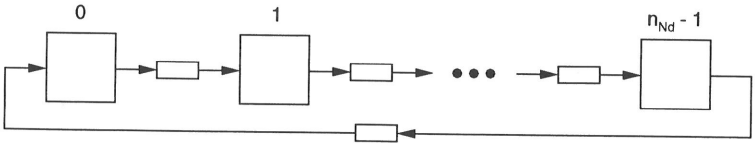

Bild 7.9: Unidirektionale Ring-Anordnung

$p_r = 0{,}9$, $n_{E_{SS}} = 20$, $n_{S_{GVT}} = 30$ bei variabler Knotenanzahl n_{Nd} simuliert. Die maximale Anzahl pro Zeitschlitz zu bearbeitender Ereignisse war beim konservativen Verfahren $n_{E_{Sl}} = 10000$ und beim optimistischen $n_{E_{Sl}} = 10$ für $n_{Nd} \leq 40$. Bei größeren Knotenanzahlen wurde das optimistische Verfahren aufgrund kaskadierten Zurücksetzens der Logischen Prozesse instabil (s. a. Abschnitt 2.5.3). Für $n_{Nd} = 50$ und $n_{Nd} = 60$ ließ sich dieses Problem durch Reduzierung der maximalen Anzahl pro Zeitschlitz zu bearbeitender Ereignisse auf $n_{E_{Sl}} = 5$ bzw. $n_{E_{Sl}} = 4$ unter Inkaufnahme von Leistungseinbußen noch weitgehend in den Griff bekommen, für $n_{Nd} = 70$ gelang dies nicht mehr. Durch die zusätzliche großräumige Rückkopplung gegenüber der Tandem-Anordnung sank die Leistung des konservativen Verfahrens deutlich auf ungefähr ein Fünftel ab. Das optimistische Verfahren büßte zwar nur stark die Hälfte seiner Leistung ein und liegt damit knapp vor dem konservativen, eignet sich aufgrund der erwähnten Instabilitäten aber nur für kleine Knotenanzahlen.

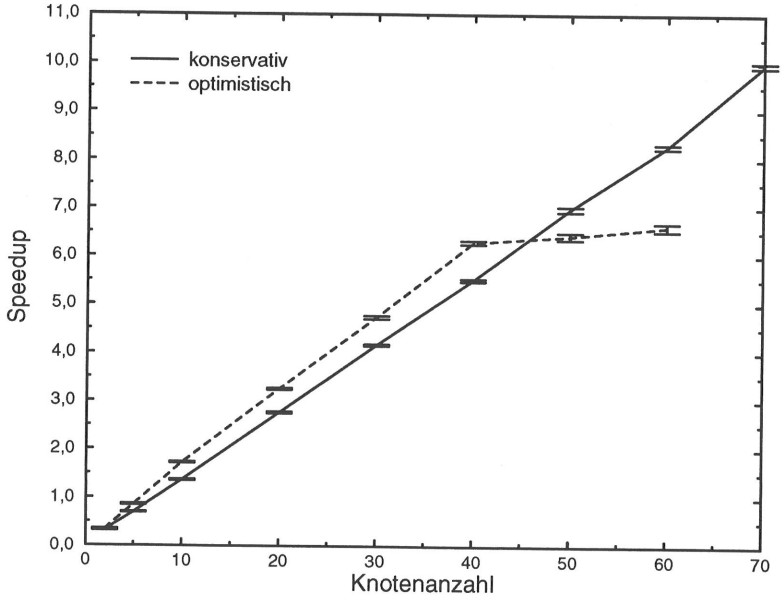

Bild 7.10: Speedup für unidirektionale Ring-Anordnung in Abhängigkeit der Modellgröße

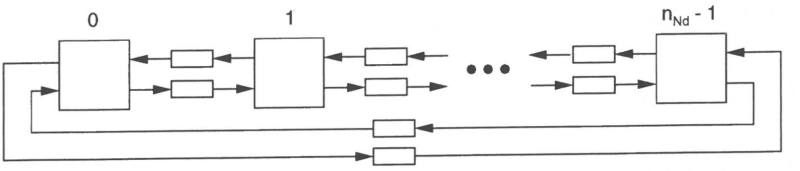

Bild 7.11: Bidirektionale Ring-Anordnung

7.3.5.2 Bidirektional

Auch die bidirektionale Tandem-Anordnung wurde zu einem Ring erweitert (siehe Bild 7.11) und wiederum der Speedup gemessen (siehe Bild 7.12, Parameter: $T_{PTL} = 5000$, $\lambda_G = 1$, $h_S = 0{,}0244$, $h_I = 0{,}5$, $p_r = 0{,}9$, $n_{E_{SS}} = 10$, $n_{S_{GVT}} = 400$, $n_{E_{SI}} = 1000$ (konservativ) bzw. $n_{E_{SI}} = 3$ (optimistisch)). Beim optimistischen Verfahren sind geringfügige Leistungseinbußen festzustellen. Es liegt aber immer noch deutlich vor dem konservativen Verfahren, das nahezu unverändert gegenüber der bidirektionalen Tandem-Anordnung abschneidet. Bei der engen Kopplung benachbarter Knoten durch die bidirektionalen Verbindungen fällt die zusätzliche großräumige Rückkopplung offenbar kaum ins Gewicht.

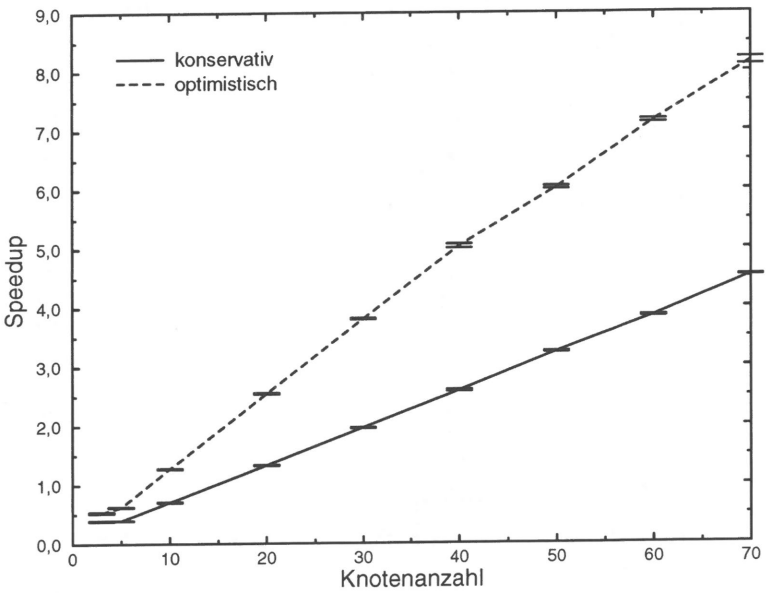

Bild 7.12: Speedup für bidirektionale Ring-Anordnung in Abhängigkeit der Modellgröße

7.3.6 Gitter-Anordnungen

7.3.6.1 Unidirektional

Als letzte Topologie mit generischen Knoten- und Verbindungselementen wurde eine Gitterstruktur untersucht. Bild 7.13 zeigt diese mit unidirektionalen Verbindungen, und in Bild 7.14 sind die zugehörigen Ergebnisse für variable Knotenanzahl n_{Nd} zu sehen (Parameter: $T_{PTL} = 5000$, $\lambda_G = 1$, $h_S = 0{,}01464$, $h_I = 0{,}5$, $p_r = 0{,}9$, $n_{E_{SS}} = 200$, $n_{S_{GVT}} = 5$, $n_{E_{Sl}} = 10000$ (konservativ) bzw. $n_{E_{Sl}} = 50$ (optimistisch)). Das Gitter war dabei „möglichst quadratisch", d. h. 30 Knoten waren z. B. in einer 5×6-Struktur angeordnet. Auch bei dieser unidirektionalen Anordnung ist das konservative Verfahren dem optimistischen überlegen, wenn auch bei beiden Leistungseinbußen gegenüber der unidirektionalen Tandem-Anordnung zu verzeichnen sind. Diese lassen sich zum einen mit den zusätzlichen Abhängigkeiten zwischen den Knoten und damit einer häufigeren Blockierung beim konservativen Verfahren bzw. häufigerem Zurücksetzen beim optimistischen Verfahren erklären. Zum anderen hat die Gitter-Anordnung eine ungünstigere Lastverteilung als die Tandem-Anordnung (ausführlichere Betrachtungen hierzu finden sich in [ten Brink,1995]).

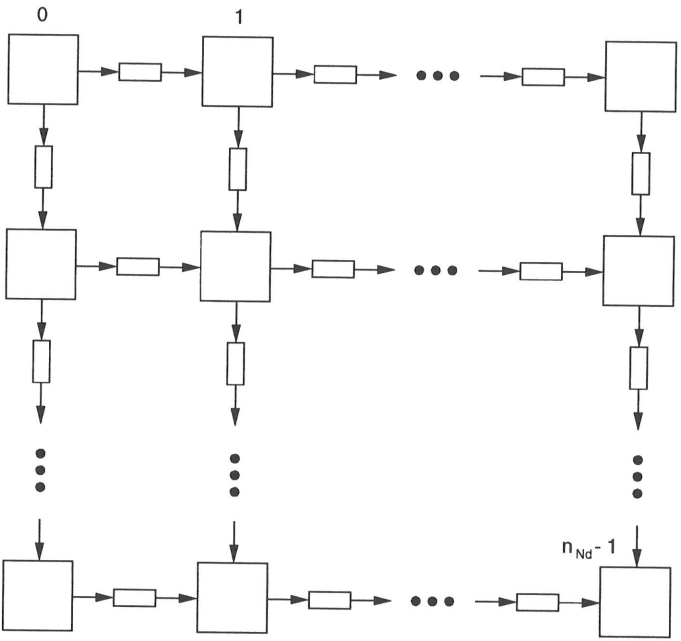

Bild 7.13: Unidirektionale Gitter-Anordnung

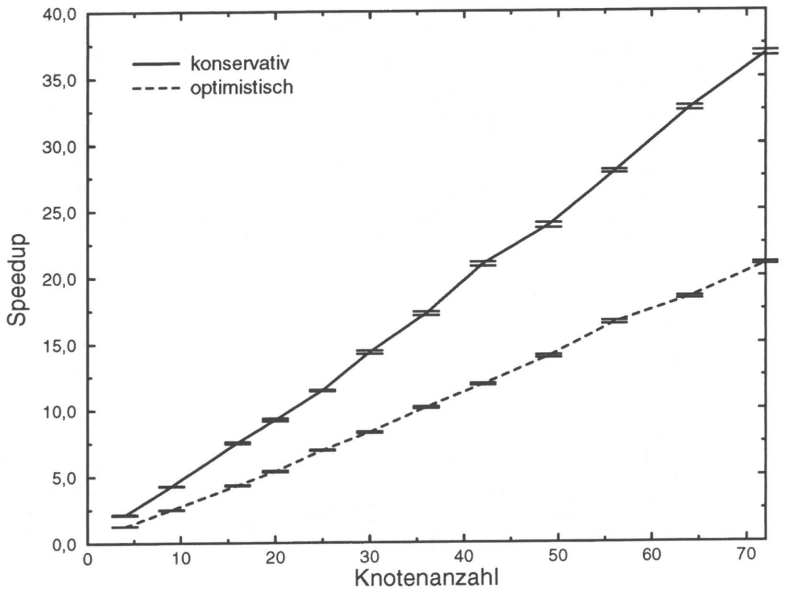

Bild 7.14: Speedup für unidirektionale Gitter-Anordnung in Abhängigkeit der Modellgröße

7.3.6.2 Bidirektional

Eine Gitter-Anordnung mit bidirektionalen Verbindungen schließlich zeigt Bild 7.15. Bei den durchgeführten Untersuchungen mit variabler Knotenzahl n_{Nd} (Parameter: $T_{PTL} = 5000$, $\lambda_G = 1$, $h_S = 0{,}01464$, $h_I = 0{,}5$, $p_r = 0{,}9$, $n_{E_{SS}} = 10$, $n_{S_{GVT}} = 350$, $n_{E_{Sl}} = 1000$ (konservativ) bzw. $n_{E_{Sl}} = 3$ (optimistisch)) waren die Knoten wieder „möglichst quadratisch" angeordnet (s. o.). In Bild 7.16 ist zu sehen, daß das konservative Verfahren bei dieser Anordnung gegenüber der Tandem-Anordnung deutliche Leistungseinbußen um 50 % erleidet, während das optimistische nur etwa 20 % schwächer abschneidet.

7.3.7 Mehrere Logische Prozesse in einem Logischen Knoten

Bei den vorangegangenen Untersuchungen war immer genau ein generisches Knotenelement in jedem Logischen Prozeß vorhanden. Ein Logischer Knoten enthielt genau einen Logischen Prozeß und wurde exklusiv auf einem Physikalischen Knoten bearbeitet. Sollen mehrere generische Knotenelemente auf einen Physikalischen Knoten gebracht werden (z. B. weil nicht genügend Rechenknoten zur Verfügung stehen), gibt es hierzu verschiedene Möglichkeiten.

Die erste Möglichkeit, die Zuweisung zu Logischen Prozessen und Logischen Knoten zu belassen und einfach mehrere LNs vom Betriebssystem der Rechenplattform auf einem Physi-

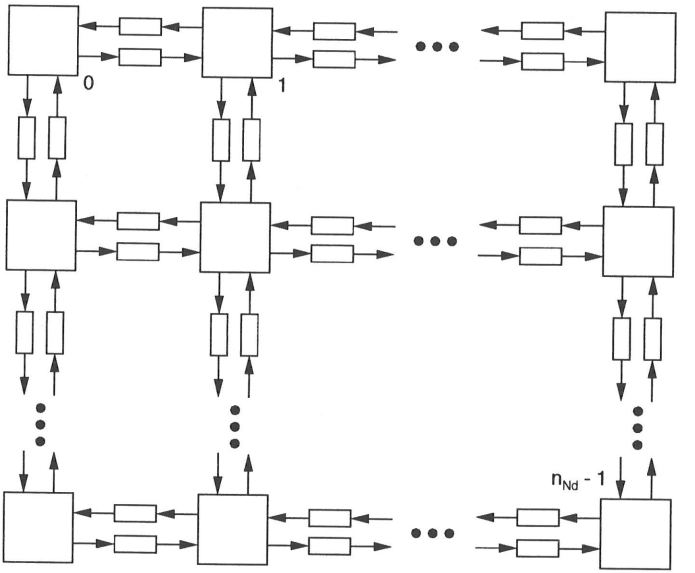

Bild 7.15: Bidirektionale Gitter-Anordnung

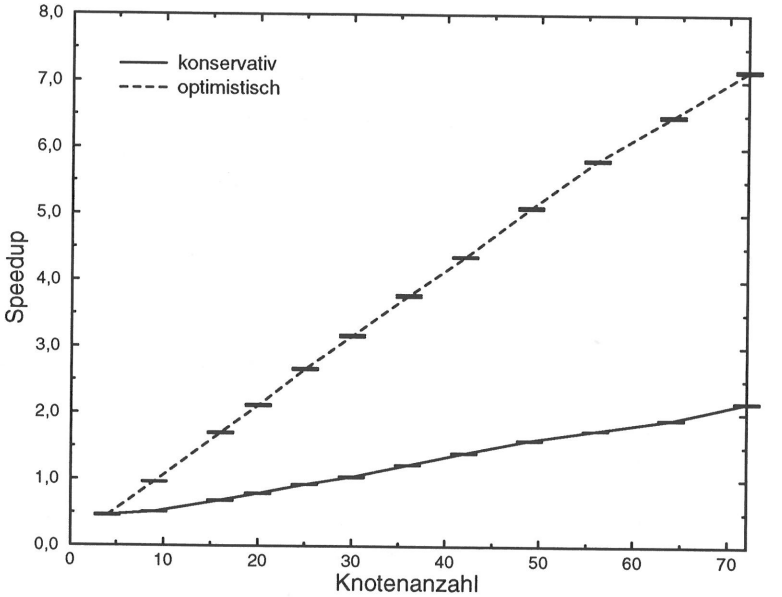

Bild 7.16: Speedup für bidirektionale Gitter-Anordnung in Abhängigkeit der Modellgröße

kalischen Knoten verwalten zu lassen, wurde nicht weiter untersucht. Vor allem aufgrund des begrenzten Speicherplatzes ist auf der Paragon die quasiparallele Bearbeitung nur weniger Prozesse auf jedem Rechenknoten möglich. Auch ist der Prozeßwechsel aufwendig, so daß die beiden folgenden Varianten höhere Speedup-Werte erwarten lassen.

Diese beiden anderen Möglichkeiten bestehen darin, entweder jedem generischen Knotenelement weiterhin einen eigenen Logischen Prozeß zuzuordnen und mehrere LPs in einem Logischen Knoten anzusiedeln, oder wie in den vorherigen Abschnitten weiterhin nur einen LP pro Logischem Knoten zu haben und dafür mehrere generische Knotenelemente in diesem unterzubringen. Auf den ersten Blick scheint die erste Variante nicht sehr vielversprechend. Ein zentraler Ereigniskalender ist Garant dafür, daß immer das Ereignis mit dem kleinsten Zeitstempel im Logischen Knoten bearbeitet wird; wieso sollte die im Vergleich dazu relativ aufwendige Synchronisation mehrerer Logischer Prozesse im selben Logischen Knoten Vorteile bringen? Daß dies durchaus der Fall sein kann, soll im folgenden durch Vergleich beider Varianten anhand der unidirektionalen Gitter-Anordnung bei Verwendung des konservativen Verfahrens exemplarisch vorgeführt werden. Nicht zuletzt läßt sich auch so die im Werkzeug realisierte Konzeption von Logischen Prozessen und Logischen Knoten motivieren.

Hierzu wurde von der Verfügbarkeit von 16 Physikalischen Knoten, auf denen jeweils ein Logischer Knoten plaziert war, ausgegangen, und es wurden quadratische Gitter-Anordnungen mit Kantenlängen von 8, 12 und 16 generischen Knoten simuliert. Um die Anzahl der über die Grenzen eines Logischen Knotens bestehenden Verbindungen klein zu halten, wurden jeweils Teil-Quadrate mit einer Kantenlänge von 2, 3 und 4 generischen Knotenelementen in jedem LN gruppiert. Die übrigen Parameter hatten die Werte $T_{PTL} = 1000$, $\lambda_G = 1$, $h_S = 0{,}01464$, $h_I = 0{,}5$, $p_r = 0{,}9$ und $n_{Esl} = 10$. Die Rechenzeitvergabe im Logischen Knoten erfolgte im Falle von mehreren LPs zyklisch.

In Bild 7.17 sind die Werte für den erzielten Speedup zusammen mit 95 %-Konfidenzintervallen eingetragen für den Fall, daß alle generischen Knotenelemente eines Logischen Knotens einem einzigen LP zugewiesen wurden und für den Fall, daß jedes generische Knotenelement einem eigenen LP zugewiesen wurde. Im ersten Fall befand sich somit nur ein Logischer Prozeß in jedem Logischen Knoten, im zweiten befanden sich 4, 9 bzw. 16 Logische Prozesse in jedem Logischen Knoten.

Bei der Lösung mit nur einem Logischen Prozeß in einem Logischen Knoten ist ein abnehmender Speedup bei zunehmender Anzahl generischer Knotenelemente zu erkennen. Im Gegensatz dazu steigt er bei mehreren Logischen Prozessen pro LN mit zunehmender Anzahl an generischen Knotenelementen und ist außerdem größer als bei der Lösung mit nur einem LP. Dies läßt sich zum einen mit einer schnelleren Kalenderverwaltung aufgrund kürzerer Ereignislisten erklären, zum anderen wirkt sich vor allem die geringere Anzahl an Eingangskanälen pro LP positiv aus. Bei der 8×8-Gitter-Anordnung sind dies bis zu vier Kanäle, bei der 16×16-Anordnung bis zu acht. Da beim konservativen Verfahren in jedem Logischen Prozeß

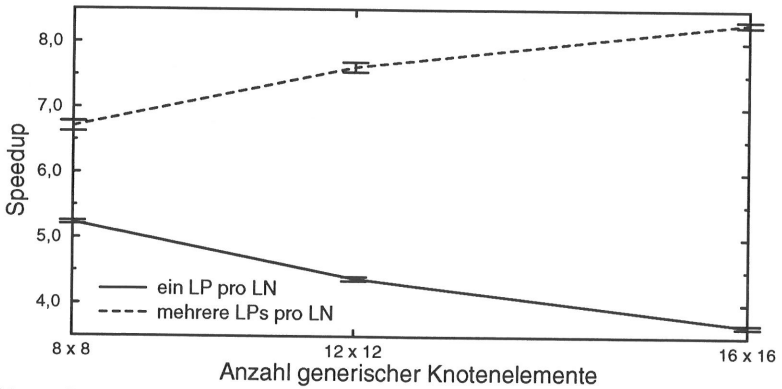

Bild 7.17: Speedup bei mehreren generischen Knotenelementen pro Logischem Knoten

die Bearbeitung von Ereignissen mit Zeitstempeln, die größer als die kleinste Eingangskanalzeit sind, nicht möglich ist und bei steigender Eingangskanalzahl die Abhängigkeiten von vorgeschalteten LPs zunehmen, wird der Logische Prozeß und damit der gesamte Logische Knoten auch häufiger blockiert. Im Gegensatz dazu kann bei mehreren Logischen Prozessen pro LN im Falle einer Blockierung meist auf einen anderen, nicht blockierten LP ausgewichen werden, so daß die Rechenkapazität des Physikalischen Knotens nicht brachliegt.

Anders ausgedrückt, kann bei der ersten Lösung ein einzelner Eingangskanal die Ereignisbearbeitung des gesamten monolithischen Blocks und damit des Logischen Knotens zum Erliegen bringen, während er bei der zweiten nur einen Teil davon blockiert, der dann bei der Rechenzeitvergabe einfach übergangen wird. Die Zunahme des Speedups bei der zweiten Lösung läßt sich mit der Verbesserung des Verhältnisses von Rechenzeit zur Anzahl der über die Grenzen eines Logischen Knotens hinweg versandten Nachrichten erklären.

7.3.8 Schlußfolgerungen

Aus den Untersuchungen mit den künstlichen Modellen der obigen Abschnitte lassen sich einige Schlußfolgerungen ziehen:

- Bei rein vorwärtsgerichteten Modellen schneidet das konservative Verfahren besser ab als das optimistische. Blockierungen treten hier selten auf, und der zusätzliche Aufwand für die Zustandssicherung beim optimistischen Verfahren kann nicht durch fehlerfreie Voraussimulation wettgemacht werden.
- Für Modelle mit bidirektionalen Verbindungen ist das optimistische Verfahren besser geeignet als das konservative. Die enge Kopplung benachbarter Knoten führt zu häufigem Blockieren und einer hohen Empfindlichkeit des konservativen Verfahrens

gegenüber der Größe des Lookahead und damit der Größe und Verteilung der Bedienzeiten von Bedieneinheit und „Infinite Server".

- Das Verhältnis von Rechenzeit pro Knoten zu Kommunikationszeit sollte möglichst groß sein, um eine hohe Leistungssteigerung zu erzielen.
- Großräumige Rückkopplungen können beim optimistischen Verfahren zu Stabilitätsproblemen führen.
- Die Aufteilung des Simulationsmodells eines Logischen Knotens in mehrere Logische Prozesse kann unter bestimmten Umständen vorteilhaft sein.

7.4 Signalisiersystem Nr. 7

7.4.1 Vorbemerkungen

Um Erfahrungen mit der entwickelten Bibliothek an einer realen Anwendung aus dem Bereich der Kommunikationstechnik zu sammeln, wurde ein Modell des vom *ITU-T* (*Telecommunication Sector of the International Telecommunication Union*) standardisierten Signalisiersystems Nr. 7 (kurz: SS7) untersucht [ITU-T,1993]. Eine Reihe von Arbeiten haben sich bereits mit der Modellierung und Analyse dieses Signalisiersystems beschäftigt (siehe z. B. [Willmann,1989], [Willmann&Kühn,1990], [Bafutto et al.,1994], [Bafutto,1995]). Simulationen des Systems finden sich z. B. in [Unger et al.,1994] oder [Durchdewald,1995]. SS7 bietet sich aus verschiedenen Gründen für eine Untersuchung der Möglichkeiten einer Parallelisierung an:

- Reale Signalisiersysteme bestehen aus einer großen Anzahl von Signalisierknoten und bieten dadurch eine ausreichende Problemgröße für eine Parallelisierung.
- Durch die Problemgröße ergeben sich bei sequentieller Simulation erhebliche Schwierigkeiten bezüglich Speicherplatzanforderungen und Rechenzeitbedarf.
- Jeder Signalisierknoten hat bei entsprechender Modellierung eine hohe Komplexität, so daß ein gutes Verhältnis zwischen Rechen- und Kommunikationszeit erwartet werden kann.
- Aufgrund der Struktur des Systems läßt sich eine natürliche Partitionierung des Modells zwischen Signalisierknoten finden.

Bevor auf die Parallelisierung und die Leistungsuntersuchungen näher eingegangen wird, zuerst ein Überblick über Aufbau und Arbeitsweise des SS7 und die verwandte Modellierung.

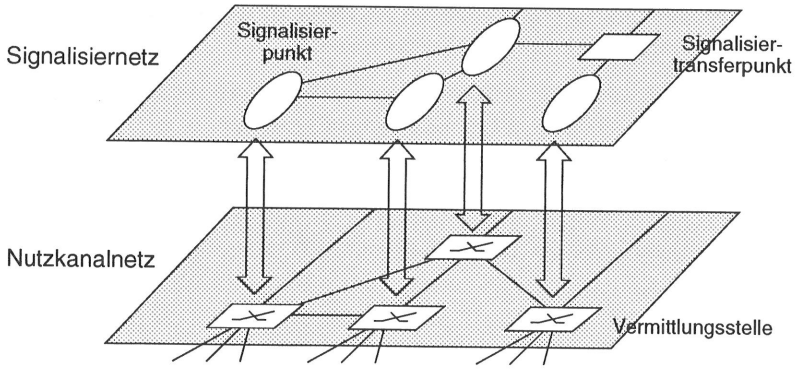

Bild 7.18: Struktur des Signalisiersystems Nr. 7

7.4.2 Grundstruktur und Architektur

Für die Abwicklung der unterschiedlichen Funktionen in einem Telekommunikationsnetz, wie beispielsweise dem *ISDN* (*Integrated Services Digital Network*), ist eine Signalisierung sowohl zwischen dem Teilnehmer und seinem Anschlußpunkt im Netz als auch zwischen den einzelnen Knoten im Netz selbst erforderlich. Diese Signalisierung läuft in einem dem Nutzkanalnetz überlagerten, separaten Signalisiernetz ab (siehe Bild 7.18). Die Signalisierknoten des Signalisiernetzes lassen sich aufteilen in *Signalisierpunkte* (*Signalling Point, SP*) und *Signalisiertransferpunkte* (*Signalling Transfer Point, STP*). Jeder Vermittlungsstelle des Nutzkanalnetzes ist ein Signalisierpunkt zugeordnet, während Signalisiertransferpunkte nur interne Aufgaben des Signalisiernetzes erfüllen und keine Entsprechung im Nutzkanalnetz haben.

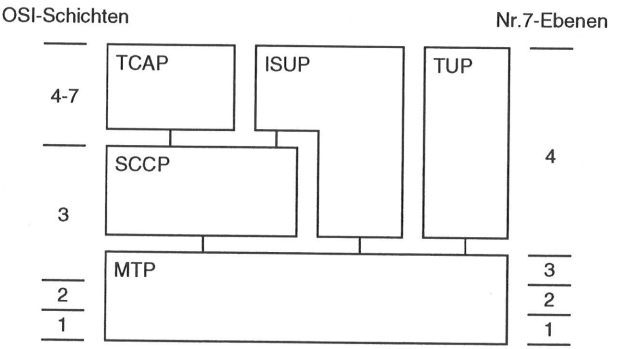

Bild 7.19: Protokollarchitektur des Signalisiersystems Nr. 7

Die Protokollarchitektur des Signalisiersystems Nr. 7 ist in Bild 7.19 zusammen mit einer Einordnung in die entsprechenden Schichten des OSI-Referenzmodells [ITU-T,1994] dargestellt. Die einzelnen Blöcke haben die folgenden Funktionen:

MTP	*Message Transfer Part*	Stellt einen einfachen, paketorientierten, verbindungslosen Datagrammdienst mit abschnittsweiser Reihenfolgesicherung und Fehlererkennung zur Verfügung.
SCCP	*Signalling Connection Control Part*	Erweitert den *MTP* um zur OSI-Schicht 3 konforme verbindungslose und -orientierte Dienste, erweitert Adressierungsmöglichkeiten des *MTP* und ermöglicht durch Adreßumwandlung (*Global Title Translation*) Name-Server-Funktionalität (wichtig z. B. für die Mobilkommunikation).
ISUP	*ISDN User Part*	Dient der Steuerung von *ISDN*-Verbindungen samt der dazugehörigen Dienstmerkmale, wie Anrufweiterleitung etc.
TCAP	*Transaction Capabilities Application Part*	Ermöglicht den Austausch von Operationsaufrufen und Quittungen inklusive des Datenaustauschs zwischen Anwendungsprozessen in Vermittlungsstellen, Datenbanken oder Managementeinrichtungen.
TUP	*Telephone User Part*	Dient dem Auf- und Abbau herkömmlicher Fernsprechverbindungen.

7.4.3 Modellierung

Die im letzten Abschnitt kurz angerissene Architektur des Signalisiersystems Nr. 7 wurde in verschiedenen Arbeiten zur Durchführung einer Analyse mittels eines Warteschlangennetzes modelliert (siehe [Willmann,1989], [Willmann&Kühn,1990], [Bafutto et al.,1994] oder [Bafutto,1995]). Auf der Basis dieses Modells wurde in [Durchdewald,1995] ein Simulationsmodell zur Verifikation der Analyseergebnisse erstellt und implementiert. Dieses Modell soll im folgenden kurz vorgestellt werden, detailliertere Beschreibungen finden sich in den erwähnten Literaturstellen.

Ein Signalisierabschnitt zwischen zwei Signalisierpunkten deckt die *MTP*-Ebenen 1 und 2 ab und stellt eine bidirektionale digitale Übertragungsstrecke mit Fehlerkorrektur und Reihenfolgesicherung zur Verfügung. Entgegen der in [Bafutto,1995] verwandten Modellierung dieser Ebenen gemäß der ITU-T-Empfehlung Q.706, wurden diese beiden Ebenen vereinfachend durch ein Bediensystem bestehend aus einer unendlichen Anzahl unabhängig, parallel arbeitender Bedieneinheiten („Infinite Server“) mit jeweils konstanter Bedienzeit modelliert.

Die *MTP*-Ebene 3 wird einem Signalisierpunkt zugerechnet und gemäß Bild 7.20 modelliert. Zu sehen ist ein Prozessormodell mit drei Prozessorphasen: *Message Discrimination*

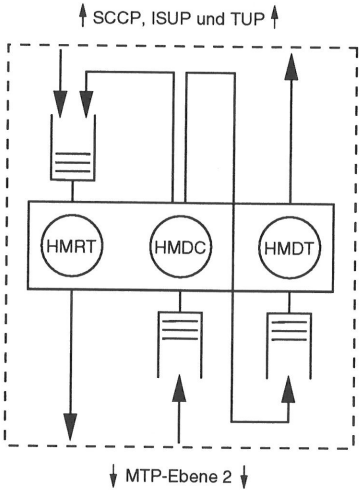

Bild 7.20: Modell für die *MTP*-Ebene 3

(*HMDC*) untersucht, ob eine aus Ebene 2 ankommende Meldung für den SP bestimmt ist oder nicht. Im ersten Fall wird die Meldung an die *Message Distribution* (*HMDT*) übergeben, von wo aus sie an höhere Ebenen weitergeleitet wird. Im zweiten Fall verläßt die Nachricht über *Message Routing* (*HMRT*) den SP wieder, genauso wie von höheren Ebenen kommende Nachrichten.

Die Modellierung des *SCCP* ist in Bild 7.21 dargestellt. Aus dem Modell aus [Bafutto,1995] wurden dabei zur Vereinfachung die Mechanismen zur Segmentierung und Flußkontrolle entfernt. Zu sehen sind sechs Bedienphasen in einem Prozessormodell, wobei jeweils eine Empfangs- und eine Sendephase zu einem internen Block des *SCCP* gehören. Der Auswertung der Adreßinformation und der Adreßumsetzung dienen die Phasen *SCCP Routing Control Transmission* (*SCRT*) und *SCCP Routing Control Reception* (*SCRR*). Die Phasen *SCCP Connection-Oriented Control Transmission* (*SCOT*) und *SCCP Connection Oriented Control Reception* (*SCOR*) bzw. die Phasen *SCCP Connectionless Control Transmission* (*SCLT*) und *SCCP Connectionless Control Reception* (*SCLR*) stellen jeweils Funktionen für verbindungsorientierte und verbindunglose Dienste in verschiedenen Dienstklassen bereit.

Der modellierte *ISUP* enthält vier Bedienphasen (Bild 7.22). Die *Message Distribution Control* (*MDSC*) leitet eine empfangene Nachricht an die zuständige Stelle innerhalb des *ISUP*s weiter, während die *Message Sending Control* (*MSDC*) für die Übergabe einer Nachricht an *MTP* oder *SCCP* zuständig ist. Der Block *Call Processing Control* steuert den Auf- und Abbau von ISDN-Verbindungen und wird durch die Phasen *Call Processing Control*

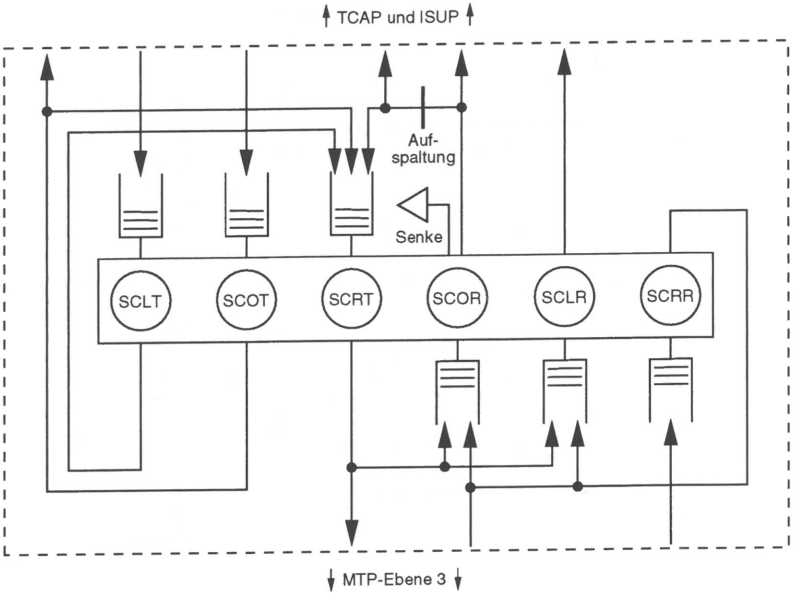

Bild 7.21: Modell für den *SCCP*

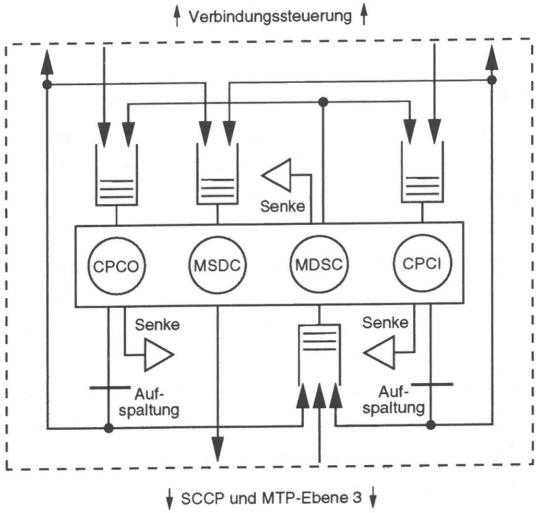

Bild 7.22: Modell für den *ISUP*

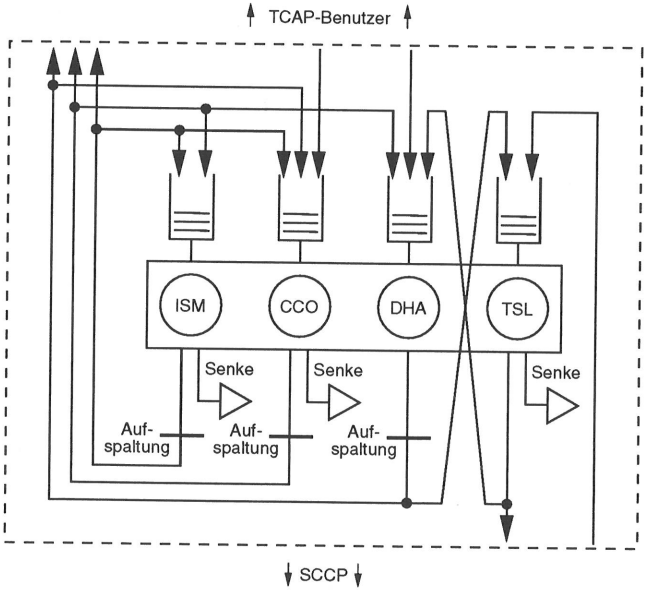

Bild 7.23: Modell für den *TCAP*

Incoming (*CPCI*) für ankommende Rufe und *Call Processing Control Outgoing* (*CPCO*) für abgehende Rufe modelliert.

Der *TCAP* schließlich wird nach Bild 7.23 modelliert und enthält die vier Bedienphasen *Dialog Handling* (*DHA*), *Component Coordinator* (*CCO*), *Invocation State Machine* (*ISM*) und *Transaction Sublayer* (*TSL*). Der *TUP* wurde in [Durchdewald,1995] nicht implementiert und wird deshalb hier auch nicht näher beschrieben.

Das Gesamtmodell eines Signalisierpunktes wird aus den oben beschriebenen Teilmodellen für *MTP*-Ebene 3, *SCCP*, *ISUP* und *TCAP* zusammengesetzt, wobei die Prozessormodelle der einzelnen Blöcke noch einem oder mehreren Prozessoren zur Bearbeitung zugeteilt werden. Zusätzlich beinhaltet jeder SP eine Verkehrssteuerung (*Call Control*, *CC*), die für den Ablauf verschiedener Verkehrsszenarien verantwortlich ist. Ein Szenario besteht aus einer Folge von zwischen SPs ausgetauschten Meldungen und ist Teil eines Dienstes. Solche Szenarien können z. B. sein: „Erfolgreicher ISDN-Verbindungsaufbau“, „ISDN-Teilnehmer besetzt“ oder „ISDN-Teilnehmer meldet sich nicht“. Ein Szenario beschreibt nicht nur die Abfolge der ausgetauschten Meldungen, sondern auch, wie sie in den verschiedenen Teilblöcken der SPs selbst bearbeitet werden. Auch kann die Einbeziehung eines dritten Knotens neben Ursprungs- und Zielknoten, z. B. für Datenbankabfragen beim 130-Dienst, beinhaltet sein.

7.4.4 Parallele Simulation

7.4.4.1 Vorgehensweise

Aus dem oben beschriebenen Modell für das Signalisiersystem Nr. 7 wurde ein paralleles Simulationsmodell entwickelt. Genauer gesagt wurde das in [Durchdewald,1995] mit Hilfe der sequentiellen *C++*-Simulationsbibliothek aus [Kocher,1994] implementierte Simulationsprogramm als Basis für eine Parallelisierung genommen. Es sollten dadurch Erfahrungen bei der Parallelisierung von bestehenden sequentiellen Simulationsprogrammen gewonnen und der Aufwand hierfür bestimmt werden.

Andere Arbeiten auf dem Gebiet der parallelen Simulation von SS7-Systemen finden sich zum einen in [Kalantery et al.,1993]. Dort wurde allerdings ein sehr vereinfachtes Modell für die SPs und nur ein kleines Netz unter Anwendung eines konservativen Verfahrens untersucht. Zum anderen berichten [Unger et al.,1994] von der Simulation eines ausführlichen Modells mittels eines optimistischen Verfahrens auf Multiprozessor-Systemen mit (virtuell) gemeinsamem Speicher. Es finden sich aber keine direkten Vergleiche zwischen sequentieller und paralleler Simulation, und es wurde auch nicht auf den Aufwand der Parallelisierung des Simulationsmodells eingegangen.

7.4.4.2 Aufwand der Parallelisierung

Bezüglich des Aufwands der Parallelisierung haben sich im wesentlichen die in Unterkapitel 6.7 gemachten Aussagen bestätigt. Es war innerhalb von zwei bis drei Tagen möglich, das Programm mittels eines konservativen Synchronisationsverfahrens parallel lauffähig zu machen, ohne daß dabei ein tieferes Verständnis der internen Abläufe notwendig gewesen wäre. Die Änderungen für optimistische Verfahren nahmen dagegen, wie erwartet, mehr Zeit in Anspruch. Der Hauptaufwand bestand darin, die veränderlichen Daten und die Stellen, an denen sie geändert werden, zu identifizieren. Die erforderlichen Ergänzungen ließen sich dann dank Bibliotheksunterstützung relativ einfach realisieren.

Eine weitere Erfahrung, die bei der Parallelisierung gemacht wurde, betrifft die Datenausgabe und -eingabe über Dateien. Bei ersterer multiplizierten sich die Anzahl der Ergebnis- und Log-Dateien mit der Anzahl der Logischen Knoten. Die Dateneingabe dagegen kann prinzipiell für alle Logischen Knoten aus derselben Datei erfolgen. Bei einer größeren Anzahl Logischer Knoten ließen es Zugriffs- und Speicherplatzprobleme aber ratsam erscheinen, auch die Eingabedateien aufzuspalten. Hierzu waren zusätzliche Änderungen am Programm erforderlich.

7.4.4.3 Testläufe

Um einen Eindruck von der Leistungsfähigkeit der Bibliothek zu gewinnen, wurde als Beispiel eine ISDN-Signalisierung gewählt, die sich bei der Wahl der Parameter und des Verkehrs an den hypothetischen Daten aus [Bafutto,1995] und [Durchdewald,1995] orientiert. Die für die Testläufe gewählten Netztopologien bestanden aus einzelnen Sub-Netzen, von denen eines in Bild 7.24 gezeigt ist. Ein Sub-Netz enthält jeweils sieben Signalisierpunkte und zwei Signalisiertransferpunkte. Optional kann noch ein weiterer Signalisierpunkt mit angeschlossener Datenbank (gestrichelt gezeichnet) vorhanden sein, um beispielsweise Szenarien für einen 130-, Kreditkarten- oder Mobiltelefon-Dienst zu ermöglichen. Ursprungs- und Zielknoten der einzelnen Verkehrsszenarien sind jeweils die vier Signalisierpunkte der untersten Ebene in Bild 7.24. Jedes Sub-Netz ist über genau einen Signalisierpunkt mit weiteren Sub-Netzen verbunden.

Alle im Netz eingesetzten Signalisierpunkte und Signalisiertransferpunkte sind gleich und ihre Parameter ähnlich dem ISDN-Beispiel aus [Bafutto,1995] gewählt. Jeder der funktionalen Blöcke *MTP*, *SCCP*, *ISUP* und *TCAP* wird auf jeweils einen simulierten Prozessor abgebildet. Die Prioritäten der einzelnen Bedienphasen (hoher Wert $\triangleq$ hohe Priorität) und ihre (konstanten) Bediendauern sind in Tabelle 7.2 aufgelistet.

Die Signalisierabschnitte zwischen den SPs sind, wie schon erwähnt, in jede der beiden Richtungen als „Infinite Server" mit einer konstanten Bediendauer von 5 ms (Simulationszeit) modelliert. Die drei simulierten Verkehrsszenarien eines ISDN-Beispieldienstes vom Ursprungsknoten A zum Zielknoten B sind in Bild 7.25 dargestellt, die einzelnen Nachrichten samt ihren Längen in Tabelle 7.3 aufgeführt.

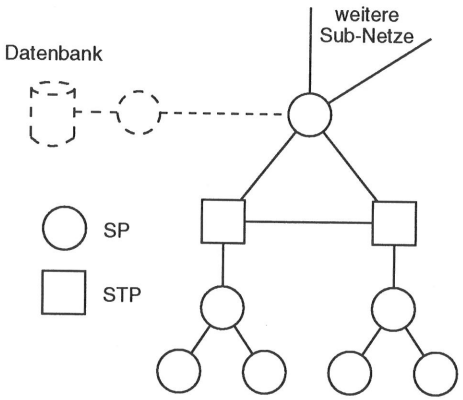

Bild 7.24: Sub-Netz für einen Logischen Prozeß

Funktionaler Block	Prozessor	Bedienphase	Priorität	Bediendauer
MTP	1	*HMDT*	2	1,0 ms
		HMDC	1	0,5 ms
		HMRT	3	1,0 ms
SCCP	2	*SCRR*	3	1,0 ms
		SCRT	4	1,0 ms
		SCLR	1	1,0 ms
		SCLT	2	1,0 ms
		SCOR	6	1,0 ms
		SCOT	5	1,0 ms
ISUP	3	*MDSC*	4	0,5 ms
		MSDC	1	0,5 ms
		CPCI	3	2,0 ms
		CPCO	2	2,0 ms
TCAP	4	*DHA*	2	2,0 ms
		CCO	3	2,0 ms
		TSL	1	1,0 ms
		ISM	4	0,5 ms

Tabelle 7.2: Parameter eines Signalisierpunktes

Meldung	Name	Länge (Oktett)
IAM	Initial Address Message	60
ACM	Address Complete Message	20
ANM	Answer Message	15
REL	Release Message	20
RLC	Release Complete Message	15

Tabelle 7.3: Meldungen des ISDN-Beispieldienstes

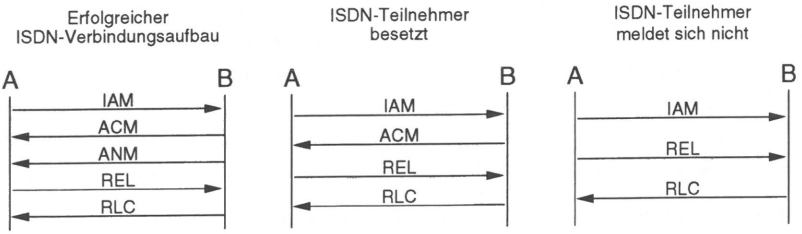

Bild 7.25: Verkehrsszenarien des ISDN-Beispieldienstes

Zur Durchführung der Testläufe wurde jeweils ein Sub-Netz einem Logischen Prozeß zugeordnet. Genau ein LP war in jedem Logischen Knoten angesiedelt, von denen wiederum jeweils einer pro Physikalischem Knoten vorhanden war. Jedes Sub-Netz hatte vier Verbindungen zu benachbarten Sub-Netzen, die Gesamtanordnung der Sub-Netze war torusförmig (d. h. gitterförmig mit zusätzlichen Verbindungen gegenüberliegender Randpunkte)[4]. Jeder Ursprungsknoten erzeugte eine negativ-exponentiell verteilte Anzahl Szenarien pro Sekunde (Simulationszeit) mit dem Mittelwert 10, deren Zielknoten sich mit 75 % Wahrscheinlichkeit im selben Sub-Netz wie der Ursprungsknoten befanden. Der dadurch erzeugte Verkehr fand bezüglich der parallelen Simulation also rein lokal innerhalb der jeweiligen Logischen Prozesse und damit der Logischen Knoten statt. Die übrigen 25 % der Szenarien teilten sich gleichmäßig auf alle Zielknoten der übrigen Sub-Netze auf. Die Anteile der jeweils angestoßenen Szenarien aus Bild 7.25 am Gesamtaufkommen betrugen: 70 % „Erfolgreicher ISDN-Verbindungsaufbau“, 20 % „ISDN-Teilnehmer besetzt“ und 10 % „ISDN-Teilnehmer meldet sich nicht“.

Bei einer variablen Anzahl von Physikalischen Knoten und damit einer variablen Anzahl von Sub-Netzen wurden wiederum Simulationen mit einem konservativen Synchronisationsverfahren mit NULL-Nachrichten (siehe Unterabschnitt 2.5.1.2) und einem optimistischen mit Aggressive Cancellation (siehe Unterabschnitt 2.5.2.2) durchgeführt. Das Vorgehen bei den Untersuchungen und den jeweiligen sequentiellen Vergleichsmessungen wurde schon in Abschnitt 7.3.3 beschrieben. Die Anzahl der Teiltests jedes Simulationslaufs war fünf, ihre Länge betrug jeweils 50 s (Simulationszeit) bei einer Warmlaufphase gleicher Länge.

Die Ergebnisse der Testläufe sind in Bild 7.26 zusammengefaßt. Dort ist die reale Rechendauer für die Durchführung eines Teiltests über der Anzahl der Sub-Netze aufgetragen (was bei den beiden parallelen Verfahren der Anzahl eingesetzter Rechenknoten entspricht). Die sequentiellen Vergleichsmessungen auf einem Rechenknoten der *Paragon* konnten nur bis zu einer Zahl von 25 Sub-Netzen problemlos durchgeführt werden, da für größere Modelle der zur Verfügung stehende Speicher nicht mehr ausreichte – dies obwohl speziell für diese Messungen ein Rechenknoten mit 64 MB Halbleiterspeicher (statt sonst üblicher 32 MB) benutzt wurde. Bei den gestrichelt eingezeichneten Meßwerten bei 30 und 36 Sub-Netzen erhöhten Paging-Vorgänge der virtuellen Speicherverwaltung die reine Laufzeit erheblich. Um dennoch einen Anhaltspunkt für die Ergebnisse zu geben, falls genügend Halbleiterspeicher zur Verfügung gestanden hätte, wurde für diese beiden Messungen nicht die Laufzeit sondern die benötigte CPU-Zeit, d. h. die Zeit, die tatsächlich für die Bearbeitung des Programms selbst verbraucht wurde, eingezeichnet. Bei mehr als 36 Sub-Netzen konnte aus Gründen zu langer Rechenzeit auch diese Näherung nicht mehr durchgeführt werden.

Bei den Messungen ist zu erkennen, daß das optimistische Synchronisationsverfahren durchweg besser abschneidet als das konservative. Dies entspricht den bei den Untersuchungen

[4] Ausgenommen im Fall von vier Sub-Netzen: hier ging die Anordnung in ein Gitter über.

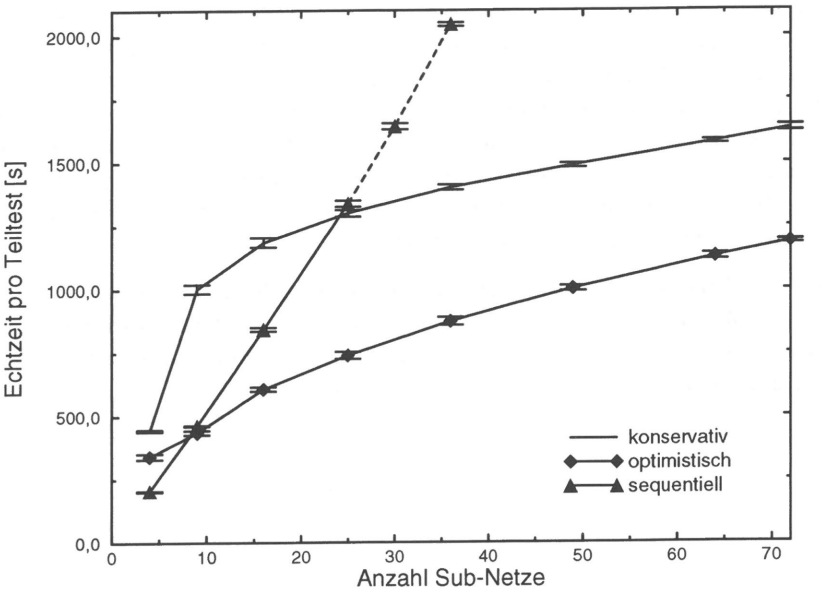

Bild 7.26: Laufzeit in Abhängigkeit der Modellgröße für SS7-Beispiel

der künstlichen Modelle mit bidirektionalen Verbindungen gemachten Erfahrungen. Ab einer Anzahl von neun Physikalischen Knoten (entsprechend neun Sub-Netzen) ist die (optimistische) parallele Simulation schneller als die sequentielle, d. h. ab dieser Modellgröße ließ sich ein Speedup größer als Eins erzielen.

7.4.4.4 Ergebnis

Die oben beschriebene Parallelisierung eines komplexen sequentiellen Programms zur Simulation des Signalisiersystems Nr. 7 ließ sich mit Hilfe der entwickelten parallelen Bibliothek weitgehend problemlos durchführen. Die implementierten Konzepte haben sich als vorteilhaft und flexibel erwiesen. Die eingesetzten Synchronisationsverfahren ließen sich nach erfolgter Anpassung des Simulationsprogramms problemlos austauschen.

Das getestete Beispiel zeigt, daß eine Leistungssteigerung der SS7-Simulation durch eine Parallelisierung möglich ist. Die diesbezüglichen Aussagen beschränken sich allerdings auf dieses Beispiel. Andere Netzstrukturen und Verkehrsbelastungen können die Ergebnisse in beide Richtungen verändern, da insbesondere die derzeit verfügbaren Synchronisationsverfahren für parallele Simulation empfindlich auf solche Änderungen reagieren. Es wurde darauf verzichtet, ausführlichere Studien hierzu durchzuführen, da die Fülle möglicher Parameterände-

rungen bei diesem Simulationsmodell es unrealistisch erscheinen läßt, allgemeingültige Aussagen zu finden. Erst anhand eines konkreten Modells läßt sich nach durchgeführter Parallelisierung deren Effizienz bewerten. Aber gerade bezüglich dieses, der parallelen ereignisgesteuerten Simulation derzeit leider noch innewohnenden Problems, haben die vorgestellten Konzepte einen deutlichen Fortschritt gebracht: Der Zusatzaufwand, ein implementiertes sequentielles Simulationsprogramm zu parallelisieren und dadurch auf mögliche Leistungssteigerungen hin mit verschiedenen Synchronisationsverfahren zu testen, hat wesentlich abgenommen und ist dadurch überschaubar geworden.

Ein Vorteil der parallelen Simulation hat sich in jedem Fall ergeben: Es lassen sich Simulationsmodelle simulieren, die in sequentieller Form aufgrund von Speicherplatzmangel nicht mehr realisierbar sind. Die Grenze der simulierbaren Komplexität eines Simulationsmodells läßt sich also durch eine Parallelisierung weiter hinausschieben.

Kapitel 8

Zusammenfassung und Ausblick

In dieser Arbeit wurde das Konzept eines flexiblen und universellen Werkzeugs für parallele ereignisgesteuerte Simulation vorgestellt. Die Implementierung der Schlüsselkonzepte unter Zuhilfenahme objektorientierter Methoden wurde erläutert und Ergebnisse von Untersuchungen zum Nachweis der Leistungsfähigkeit gezeigt.

Das wichtigste Ziel der Arbeit war, parallele Simulation einfacher anwendbar zu machen. Durch konsequente Trennung von Simulationsmodell und Synchronisationsverfahren kann das Modell ohne Festlegung auf ein bestimmtes Verfahren entwickelt und implementiert werden. Da die Synchronisationsverfahren zudem einfach austauschbar sind, kann ihre Eignung anhand des konkreten Modells nach einer Art Baukastenprinzip getestet werden. Der Übergang von einem häufig angewandten sequentiellen Werkzeug wurde so einfach wie möglich gestaltet. Dadurch können konventionell sequentiell entwickelte Simulationen einfach parallelisiert werden, wenn die Rechen- oder Speicherkapazität eines Einzelrechners nicht mehr ausreichend ist.

Durch die hierarchische Aufteilung der parallelen Einheiten in Logische Prozesse, Logische Knoten und Physikalische Knoten ist eine flexible Anpassung an die dem Modell innewohnende Parallelität möglich. Das entwickelte Kommunikationssystem ist ebenfalls strikt vom Simulationsmodell getrennt und erlaubt eine transparente Kommunikation mit nicht-lokalen Simulationsmodellkomponenten. Die Kommunikation basiert auf dem Austausch von Nachrichtenobjekten, die beliebige Datentypen enthalten und hierarchisch aufgebaut sein dürfen. Es werden auch verschiedene Synchronisationsverfahren unterstützt, ohne daß in die Basisfunktionalität eingegriffen werden muß. Der Austausch von Simulationsnachrichten kann überwacht und zusätzliche Informationen können an diese Nachrichten angeheftet werden, ohne die Nachrichten selbst verändern zu müssen. Durch Kapselung der von der Rechenplattform abhängigen Teile ist das Kommunikationssystem zudem portabel gehalten.

Für die Zustandssicherung bei optimistischer Simulation erlaubt das Konzept den Einsatz verschiedener Verfahren, wie die Erstellung von Zustandskopien oder inkrementelle Siche-

rung. Unterschiedliche Verfahren können im selben Simulationsmodell koexistieren und einfach ausgetauscht werden. Hierdurch ist es möglich, für jede Modellkomponente das jeweils am besten geeignete Verfahren zu verwenden. Die für die Zustandssicherung notwendigen Eingriffe in die Komponenten wurden minimal gehalten. Es wurde außerdem eine Optimierung der Zustandssicherung mittels Zustandskopien, sowie ein kombiniertes Verfahren aus Zustandskopien und inkrementeller Sicherung vorgestellt.

Die entworfenen Konzepte des Werkzeugs wurden als Bibliothek in der Standard-Programmiersprache *C++* implementiert, was vorteilhaft ist für eine leichte Einarbeitung des Anwenders sowie die gute Portierbarkeit auf verschiedene Rechenplattformen. Um den Aufbau und die Konfiguration auch großer Simulationsmodelle komfortabel zu gestalten, ist ein Parserkonzept für das Einlesen der zugehörigen Parameter über Dateien implementiert.

Das Werkzeug wurde mit Hilfe eigens hierfür entworfener künstlicher und bezüglich der wichtigsten Leistungsparameter der parallelen Simulation parametrisierter Modelle auf seine Leistungsfähigkeit hin untersucht. Es hat sich dabei gezeigt, daß sich bei geeigneten Simulationsmodellen und passend gewähltem Synchronisationsverfahren der Simulationslauf stark beschleunigen läßt. Da die Größe der Beschleunigung von diesen beiden Einflüssen abhängt, hat sich das Design des entwickelten Werkzeugs, namentlich die Kapselung und Austauschbarkeit der Synchronisationsverfahren, als sehr flexibel und geeignet erwiesen.

Um die Verwirklichung des Anspruchs eines einfachen Übergangs von sequentieller zu paralleler Simulation zu testen, wurde anschließend eine existierende sequentielle Simulation des Signalisiersystems Nr. 7 parallelisiert. Dies war dank der Unterstützung durch die im parallelen Werkzeug verwirklichten Konzepte weitgehend problemlos und schnell durchführbar. Die Parallelisierung konnte ohne Kenntnis des einzusetzenden Synchronisationsverfahrens erfolgen und dieses erst später in den Testläufen dynamisch gewählt werden. Die Testläufe haben exemplarisch an einem Beispiel gezeigt, daß durch die Parallelisierung größere Simulationsmodelle simulierbar wurden und sich auch hier eine Beschleunigung des Simulationslaufs ergab.

Die konsequente Anwendung objektorientierter Entwurfstechniken hat die Beherrschung der Komplexität, die Kapselung der Synchronisations-, Kommunikations- und Zustandssicherungsmechanismen sowie die einfache Erweiterbarkeit und Anwendbarkeit des Konzepts wesentlich erleichtert oder sogar erst möglich gemacht. Die Vorschläge aus [Kocher,1994] zu einer Parallelisierung der sequentiellen Bibliothek wurden nicht aufgegriffen, da sich mit dem Einsatz optimistischer Verfahren und der o. a. geforderten Flexibilität deutlich höhere Anforderungen an das parallele Werkzeug ergaben. Daraus resultierte schließlich auch ein wesentlich höherer Aufwand als ursprünglich abgeschätzt.

Alles in allem haben die entwickelten und implementierten Konzepte die in sie gestellten Anforderungen erfüllt, und es konnte gezeigt werden, daß der Einsatz von Parallelverarbeitung im Bereich der ereignisgesteuerten Simulation durchaus lohnenswert sein kann.

Zukünftig sollten Untersuchungen an weiteren parallelen Applikationen durchgeführt werden. Hierfür wären zum einen weitere Modelle des Signalisiersystems Nr. 7 denkbar. Zum anderen scheinen aufgrund der Ergebnisse für künstliche Modelle vor allem vorwärtsgerichtete Simulationsmodelle, wie z. B. ATM-Koppelnetzstrukturen, äußerst vielversprechend. Die Erfahrungen und Bedürfnisse, die dabei erkennbar werden, sollten in die Weiterentwicklung des Werkzeugs einfließen. Einige mögliche Richtungen einer solchen Weiterentwicklung sind schon heute erkennbar.

Weitere Verfahren der parallelen ereignisgesteuerten Simulation sollten implementiert und ihre Eignung bezüglich verschiedener Simulationsmodelle untersucht werden. Des weiteren könnte die Unterstützung konservativer Synchronisationsverfahren durch eine automatische Erkennung des verfügbaren Lookahead verbessert werden. Dieser wird derzeit vom Anwender zu Beginn eines Simulationslaufs für jeden Logischen Prozeß angegeben. Eine automatische Erkennung könnte auch dynamisch unter Zuhilfenahme vorausbestimmter Bedienzeiten erfolgen. Die Realisierung dieser Ideen unter Berücksichtigung der für das Werkzeug aufgestellten Forderungen, wie Trennung von Simulationsmodell und Synchronisationsverfahren, einfache Anwendbarkeit und einfache Parallelisierbarkeit, scheint dabei eine besondere Herausforderung darzustellen.

Weitere Verbesserungen sind bei der Eingabe und Konfiguration denkbar. In einem ersten Schritt könnte durch eine Erweiterung der Einlesemodule die Vorverarbeitung der Eingabedateien ersetzt werden. Änderungen bei der Eingabe sollten aber konsistent mit entsprechend angedachten Änderungen des sequentiellen Werkzeugs erfolgen, um den Übergang von sequentieller zu paralleler Simulation nicht zu erschweren. In einem zweiten Schritt wäre dann die Implementierung einer grafischen Eingabe zu erwägen, eventuell weiterhin mit der Schnittstelle über Dateien. Letzteres würde die einfache Portierung auf verschiedene Rechenplattformen nicht durch die zusätzliche Notwendigkeit der Portierung der grafischen Oberfläche belasten – diese könnte nach wie vor auf einer lokalen Workstation laufen und es müßten nur die Dateien zum Parallelrechner transferiert werden.

Zudem wäre noch eine automatische Partitionierung und Verteilung des Simulationsmodells möglich, was durch eine Vorverarbeitung realisiert werden könnte, um die Komplexität des Werkzeugs nicht unnötig zu steigern. In diesem Zusammenhang wäre auch über die Möglichkeit des Einsatzes von Lastbalancierungsverfahren (statisch oder dynamisch) nachzudenken.

Literaturverzeichnis

[Agha,1990] G. Agha: *Concurrent Object-Oriented Programming*, Communications of the ACM, Vol. 33, No. 9, September 1990, pp. 125-141.

[Alexander et al.,1977] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel: *A Pattern Language*, Oxford University Press, NY, USA, 1977.

[Ayani,1989] R. Ayani: *A Parallel Simulation Scheme Based on Distances Between Objects*, Distributed Simulation 1989, SCS Simulation Series, Vol. 21, No. 2, March 1989, pp. 113-118.

[Ayani,1993] R. Ayani: *Parallel Simulation*, Tutorial Proceedings of the 16th IFIP W.G.7.3. International Symposium on Computer Performance Modelling, Measurement and Evaluation (PERFORMANCE'93), Rome, October 1993.

[Baezner et al.,1990] D. Baezner, G. Lomow, B. W. Unger: *Sim++: The Transition to Distributed Simulation*, Distributed Simulation 1990, SCS Simulation Series, Vol. 22, No. 1, January 1990, pp. 211-218.

[Bafutto et al.,1994] M. Bafutto, P. J. Kühn, G. Willmann: *Capacity and Performance Analysis of Signaling Networks in Multivendor Environments*, IEEE Journal on Selected Areas in Communications, Vol. 12, No. 3, April 1994, pp. 490-500.

[Bafutto,1995] M. Bafutto: *Modellierung, Verkehrsanalyse und Planung von Zentralkanal-Signalisiernetzen im ISDN mit besonderer Berücksichtigung neuer Dienste der Mobilkommunikation und des Intelligenten Netzes*, 63. Bericht über verkehrstheoretische Arbeiten, Institut für Nachrichtenvermittlung und Datenverarbeitung, Universität Stuttgart, 1995 (Dissertation).

[Bagrodia&Liao,1994] R. L. Bagrodia, W.-T. Liao: *Maisie: A Language for the Design of Efficient Discrete-Event Simulations*, IEEE Transactions on Software Engineering Vol. 20, No. 4, April 1994, pp. 225-238.

[Bauer,1994] H. Bauer: *Verteilte diskrete Simulation komplexer Systeme*, Berichte aus der Informatik, Shaker, Aachen, 1994 (Dissertation).

[Bauer&Sporrer,1993] H. Bauer, C. Sporrer: *Reducing Rollback Overhead in Time-Warp Based Distributed Simulation with Optimized Incremental State Saving*, Proceedings of the 26th Annual Simulation Symposium (ASS'93), Arlington, VA, USA, March 29 - April 1, 1993, pp. 12-20.

[Bellenot,1990] S. Bellenot: *Global Virtual Time Algorithms*, Distributed Simulation, SCS Simulation Series, Vol. 22, No. 1, January 1990, pp. 122-127.

[Bellenot,1992] S. Bellenot: *State Skipping Performance with the Time Warp Operating System*, 6th Workshop on Parallel and Distributed Simulation (PADS'92), SCS Simulation Series, Vol. 24, No. 3, January 1992, pp. 53-61.

[Bellenot,1993] S. Bellenot: *Performance of a Riskfree Time Warp Operating System*, 7th Workshop on Parallel and Distributed Simulation (PADS'93), San Diego, CA, USA, May 16-19, 1993, pp. 155-158.

[Beraldi&Nigro,1995] R. Beraldi, L. Nigro: *Distributed Object-Oriented Simulation Environment: An Implementation of Time Warp Using PVM*, Proceedings of the 1995 EUROSIM Conference (EUROSIM'95), Session "Software Tools and Products", Vienna, Austria, September 11-15, 1995, pp. 151-154.

[Bershad,1991] B. N. Bershad: *The PRESTO Users Manual*, University of Washington, Department of Computer Science, Seattle, WA, USA, October 1991.

[Bodin et al.,1993] F. Bodin, P. Beckman, D. Gannon, S. Narayana, S. X. Yang: *Distributed pC++: Basic Ideas for an Object Parallel Language*, Scientific Programming, Vol. 2, No. 3, Wiley & Sons, NY, USA, Fall 1993.

[Booch,1993] *The C++ Booch Components*, Version 2.3, Rational, 3320 Scott Boulevard, Santa Clara, CA, USA, 1993.

[Booch,1994] G. Booch: *Object-Oriented Analysis and Design with Applications*, 2nd Edition, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, USA, 1994.

[Booch&Vilot,1993] G. Booch, M. Vilot: *Simplifying the Booch Components*, C++ Report, SIGS Publications, NY, USA, June 1993, pp. 41-53.

[Bräunl,1993] T. Bräunl: *Parallele Programmierung: Eine Einführung*, Vieweg, Braunschweig/Wiesbaden, 1993.

[Brown,1988] R. Brown: *Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem*, Communications of the ACM, Vol. 31, No. 3, October 1988, pp. 1220-1227.

[Butler&Lusk,1993] R. Butler, E. Lusk: *Monitors, Messages, and Clusters: The P4 Parallel Programming System*, Technical Report Preprint MCS-P362-0493, Argonne National Laboratory, Argonne, IL, USA, 1993.

[C++-Draft,1995] *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++*, ANSI-X3J16 committee, ISO-WG-21, April 28, 1995.

[Cardelli&Wegner,1985] L. Cardelli, P. Wegner: *On Understanding Types, Data Abstraction, and Polymorphism*, ACM Computing Surveys, Vol. 17, No. 4, December 1985, pp. 471-522.

[Chandy&Lamport,1985] K. M. Chandy, L. Lamport: *Distributed Snapshots: Determining Global States of Distributed Systems*, ACM Transactions on Computer Systems, Vol. 3, No. 1, February 1985, pp. 63-75.

[Chandy&Misra,1979] K. M. Chandy, J. Misra: *Distributed Simulation: A Case Study in Design and Verification of Distributed Programs*, IEEE Transactions on Software Engineering, Vol. 5, No. 5, September 1979, pp. 440-452.

[Chandy&Misra,1981] K. M. Chandy, J. Misra: *Asynchronous Distributed Simulation via a Sequence of Parallel Computations*, Communications of the ACM, Vol. 24, No. 11, April 1981, pp. 198-206.

[Chandy&Sherman,1989] K. M. Chandy, R. Sherman: *The Conditional Event Approach to Distributed Simulation*, Distributed Simulation 1989, SCS Simulation Series, Vol. 21, No. 2, March 1989, pp. 93-99.

[Coplien,1992] J. O. Coplien: *Advanced C++ – Programming Styles and Idioms*, Addison-Wesley, Reading, MA, USA, 1992.

[Coulaud&Dillon,1995] O. Coulaud, E. Dillon: *PARA++: C++ Bindings for Message Passing Libraries*, 2nd European PVM Users' Group Meeting (EuroPVM'95), Lyon, France, September 13-15, 1995.

[De Vries,1990] R. C. De Vries: *Reducing Null Messages in Misra's Distributed Discrete Event Simulation Method*, IEEE Transactions on Software Engineering, Vol. 16, No. 1, January 1990, pp. 82-91.

[Dijkstra,1965] E. W. Dijkstra: *Solution of a Problem in Concurrent Programming Control*, Communications of the ACM, Vol. 8, No. 9, September 1965, p. 569.

[D'Souza et al.,1994] L. M. D'Souza, X. Fan, P. A. Wilsey: *pGVT: An Algorithm for Accurate GVT Estimation*, 8th Workshop on Parallel and Distributed Simulation (PADS'94), Edinburgh, Scotland, U. K., July 6-8, 1994, pp. 102-109.

[Duden,1993] *Duden „Informatik": Ein Sachlexikon für Studium und Praxis*, Dudenverlag, Mannheim, Leipzig, Wien, Zürich, 1993.

[Durchdewald,1995] T. Durchdewald: *Objektorientierte Simulation des Protokoll-Stacks im Signalisiersystem Nr. 7*, Diplomarbeit, Nr. 1361, Institut für Nachrichtenvermittlung und Datenverarbeitung, Universität Stuttgart, 1995.

[Ferscha,1995] A. Ferscha: *Probabilistic Adaptive Direct Optimism Control in Time Warp*, 9th Workshop on Parallel and Distributed Simulation (PADS'95), June 14-16, 1995, Lake Placid, NY, USA, pp. 120-129.

[Fichman&Kemerer,1992] R. G. Fichman, C. F. Kemerer: *Object-Oriented and Conventional Analysis and Design Methodologies*, IEEE Computer, Vol. 25, No. 10, October 1992, pp. 22-39.

[Fleischmann&Wilsey,1995] J. Fleischmann, P. A. Wilsey: *Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators*, 9th Workshop on Parallel and Distributed Simulation (PADS'95), June 14-16, 1995, Lake Placid, NY, USA, pp. 50-58.

[Flynn,1966] M. J. Flynn: *Very High-Speed Computing Systems*, Proceedings of the IEEE, Vol. 54, No. 12, December 1966, pp. 1901-1909.

[Fujimoto,1988] R. M. Fujimoto: *Lookahead in Parallel Discrete Event Simulation*, Proceedings of the International Conference on Parallel Processing, 1988, pp. 34-41.

[Fujimoto,1989] R. M. Fujimoto: *Performance Measurements of Distributed Simulation Strategies*, Transactions of the Society for Computer Simulation, Vol. 6, No. 2, 1989, pp. 89-132.

[Fujimoto,1990] R. M. Fujimoto: *Parallel Discrete Event Simulation*, Communications of the ACM, Vol. 33, No. 10, October 1990, pp. 30-53.

[Fujimoto,1993] R. M. Fujimoto: *Parallel Discrete Event Simulation: Will the Field Survive ?*, ORSA Journal on Computing, Vol. 5, No. 3, 1993, pp. 213-230.

[Gafni,1988] A. Gafni: *Rollback Mechanisms for Optimistic Distributed Simulation Systems*, Distributed Simulation 1988, SCS Simulation Series, Vol. 19, No. 3, February 1988, pp. 61-67.

[Gamma et al.,1994] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, USA, 1994.

[Geiger,1991] A. Geiger: *Parallelrechner – Architektur und Anwendung*, Skript zur Vorlesung, Rechenzentrum der Universität Stuttgart, Anwendungen der Informatik im Maschinenwesen, Oktober 1991.

[Geist et al.,1994] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*, MIT

Press, Massachusetts Institute of Technology, Cambridge, MA, USA, 1994.

[Gonauser&Mrva,1989] M. Gonauser, M. Mrva (Hrsg.): *Multiprozessor-Systeme*, Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo, 1989.

[Grimshaw,1993] A. S. Grimshaw: *Easy-to-Use Object-Oriented Parallel Processing with Mentat*, IEEE Computer, Vol. 26, No. 5, May 1993, pp. 39-51.

[Habermann,1992] R. Habermann: *Die Simulation von Vermittlungssystemen unter Anwendung von Techniken der parallelen Datenverarbeitung*, Dissertation, Universität Hannover, 1992.

[Händel et al.,1994] R. Händel, M. N. Huber, S. Schröder: *ATM Networks: Concepts, Protocols, Applications*, 2nd Edition, Addison-Wesley, Reading, MA, USA, 1994.

[Heiss,1994] H. U. Heiss: *Prozessorzuteilung in Parallelrechnern*, Reihe Informatik, Band 98, BI Wissenschaftsverlag, Mannheim, Leipzig, Wien, Zürich, 1994.

[Hoare,1974] C. A. R. Hoare: *Monitors: An Operating System Structuring Concept*, Communications of the ACM, Vol. 17, No. 10, October 1974, pp. 549-557.

[Hwang,1993] K. Hwang: *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, New York, USA, 1993.

[Intel,1995] *Paragon System User's Guide*, Intel Corporation Supercomputer Systems Division, Beaverton, OR, USA, 1995.

[ITU-T,1993] ITU-T: *Specifications of Signalling System No. 7*, ITU-T Recommendations Q.700 Series, International Telecommunication Union, Geneva, Switzerland, 1994.

[ITU-T,1994] ITU-T: *Data Communications Networks: Open Systems Interconnection (OSI)*, ITU-T Recommendations X.200 Series, International Telecommunication Union, Geneva, Switzerland, 1994.

[Jefferson,1985] D. R. Jefferson: *Virtual Time*, ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, July 1985, pp. 405-425.

[Jefferson et al.,1987] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, S. Bellenot: *Distributed Simulation and the Time Warp Operating System*, ACM Proceedings of the Symposium on Operating Systems Principles, Austin, TX, USA, November 1987, pp. 70-75.

[Johnson&Foote,1988] R. E. Johnson, B. Foote: *Designing Reusable Classes*, Journal of Object-Oriented Programming, Vol. 1, No. 2, June/July 1988, pp. 22-35.

[Kalantery et al.,1993] N. Kalantery, S. C. Winter, D. R. Wilson, A. P. Redfern: *Fast Parallel Simulation of SS7 Telecommunication Networks*, International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'93), SCS Simulation Series, Vol. 25, No. 1, 1993, pp. 171-175.

[Kale&Krishnan,1993] L. V. Kale, S. Krishnan: *CHARM++: A Portable Concurrent Object Oriented System Based on C++*, 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93), September - October 1993, Washington, DC, USA, pp. 91-108.

[Kernighan&Ritchie,1988] B. W. Kernighan, D. M. Ritchie: *The C Programming Language*, 2nd Edition, ANSI C, Prentice Hall, Englewood Cliffs, NJ, USA, 1988.

[Kocher,1994] H. Kocher: *Entwurf und Implementierung einer Simulationsbibliothek unter Anwendung objektorientierter Methoden*, 59. Bericht über verkehrstheoretische Arbeiten, Institut für Nachrichtenvermittlung und Datenverarbeitung, Universität Stuttgart, 1994 (Dissertation).

[Kühn,1995] P. J. Kühn: *Nachrichtenverkehrstheorie*, Skript zur Vorlesung, Institut für Nachrichtenvermittlung und Datenverarbeitung, Universität Stuttgart, 1995.

[Lang,1997] M. Lang: *Effizienz von Verfahren zur adaptiven und verteilten Verkehrslenkung in Paketvermittlungsnetzen*, 67. Bericht über verkehrstheoretische Arbeiten, Institut für Nachrichtenvermittlung und Datenverarbeitung, Universität Stuttgart, 1997 (Dissertation).

[Lehnert,1993] R. Lehnert: *Distributed Simulation for Performance Evaluation of Broadband Communication Networks*, AEÜ, Archiv für Elektronik und Übertragungstechnik, International Journal of Electronics and Communications, Vol. 47, No. 5/6, September/November 1993, pp. 420-425.

[Lewis,1991] T. G. Lewis: *Data Parallel Computing: An Alternative for the 1990s*, IEEE Computer, Vol. 24, No. 9, September 1991, pp. 110-111.

[Lin,1994] Y.-B. Lin: *Parallel Independent Replicated Simulation on a Network of Workstations*, 8th Workshop on Parallel and Distributed Simulation (PADS'94), Edinburgh, Scotland, U. K., July 6-8, 1994, pp. 73-80.

[Lin&Lazowska,1990] Y.-B. Lin, E. D. Lazowska: *Exploiting Lookahead in Parallel Simulation*, IEEE Transactions on Parallel and Distributed

Systems, Vol. 1, No. 4, October 1990, pp. 457-469.

[Lin et al.,1993] Y.-B. Lin, B. R. Preiss, W. M. Loucks, E. D. Lazowska: *Selecting the Checkpoint Interval in Time Warp Simulation*, 7th Workshop on Parallel and Distributed Simulation (PADS'93), San Diego, CA, USA, May 16-19, 1993, pp. 3-10.

[Lubachevsky,1988] B. D. Lubachevsky: *Bounded Lag Distributed Discrete Event Simulation*, Distributed Simulation 1988, SCS Simulation Series, Vol. 19, No. 3, February 1988, pp. 183-191.

[Lubachevsky,1989a] B. D. Lubachevsky: *Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks*, Communications of the ACM, Vol. 32, No. 1, January 1989, pp. 111-123.

[Lubachevsky,1989b] B. D. Lubachevsky: *Scalability of the Bounded Lag Distributed Discrete Event Simulation*, Distributed Simulation 1989, SCS Simulation Series, Vol. 21, No. 2, March 1989, pp. 100-107.

[Madisetti et al.,1988] V. Madisetti, J. Walrand, D. Messerschmitt: *WOLF: A Rollback Algorithm for Optimistic Distributed Simulation Systems*, Proceedings of the 1988 Winter Simulation Conference, 1988, pp. 296-305.

[Mattern,1993] F. Mattern: *Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation*, Journal of Parallel and Distributed Computing, Vol. 18, 1993, pp. 423-434.

[Mehl,1991a] H. Mehl: *Speed-Up of Conservative Distributed Discrete Event Simulation Methods by Speculative Computing (Short Version)*, Advances in Parallel and Distributed Simulation, SCS Simulation Series, Vol. 23, No. 1, January 1991, pp. 163-166

[Mehl,1991b] H. Mehl: *Breaking Ties Deterministically in Distributed Simulation Schemes*, Technischer Bericht, Fachbereich Informatik, Universität Kaiserslautern, 1991.

[Mehl,1992] H. Mehl: *A Deterministic Tie-Breaking Scheme for Sequential and Distributed Simulation*, 6th Workshop on Parallel and Distributed Simulation (PADS'92), SCS Simulation Series, Vol. 24, No. 3, January 1992, pp. 199-200.

[Mehl&Hammes,1993] H. Mehl, S. Hammes: *Shared Variables in Distributed Simulation*, 7th Workshop on Parallel and Distributed Simulation (PADS'93), San Diego, CA, USA, May 16-19, 1993, pp. 68-75.

[Meyer,1988] B. Meyer: *Object-Oriented Software Construction*, Prentice Hall, Englewood Cliffs, NJ, USA, 1988.

[Meyer,1992] B. Meyer: *Applying "Design by Contract"*, IEEE Computer, Vol. 25, No. 10, October 1992, pp. 40-51.

[Misra,1986] J. Misra: *Distributed Discrete-Event Simulation*, ACM Computing Surveys, Vol. 18, No. 1, March 1986, pp. 39-65.

[MPI-Draft,1995] *MPI: A Message-Passing Interface Standard*, Version 1.1, Message Passing Interface Forum, June 1995.

[Necker,1995] T. Necker: *Object-Oriented Communication for Distributed Discrete Event Simulation*, Proceedings of the 1995 EUROSIM Conference (EUROSIM'95), Vienna, Austria, September 11-15, 1995, pp. 333-338.

[Nevison,1990] C. Nevison: *Parallel Simulation of Manufacturing Systems: Structural Factors*, Distributed Simulation 1990, SCS Simulation Series, Vol. 22, No. 1, January 1990, pp. 17-19.

[Nicol,1988] D. Nicol: *Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks*, ACM SIGPLAN Notices, Vol. 23, No. 9, September 1988, pp. 124-137.

[Nicol,1992] D. Nicol: *Conservative Parallel Simulation of Priority Class Queueing Networks*, IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 3, May 1992, pp. 294-303.

[Nicol&Heidelberger,1995] D. M. Nicol, P. Heidelberger: *On Extending Parallelism to Serial Simulators*, 9th Workshop on Parallel and Distributed Simulation (PADS'95), June 14-16, 1995, Lake Placid, NY, USA, pp. 60-67.

[OMG,1995] *The Common Object Request Broker: Architecture and Specification*, 2.0 (Draft), Object Management Group, Framingham, MA, USA, May 1995.

[OSF^{TM}DCE,1992] Open Software Foundation: *Introduction to OSF^{TM}DCE*, Prentice Hall, NJ, USA, 1992.

[Palaniswamy&Wilsey,1993] A. C. Palaniswamy, P. A. Wilsey: *An Analytical Comparison of Periodic Checkpointing and Incremental State Saving*, 7th Workshop on Parallel and Distributed Simulation (PADS'93), San Diego, CA, USA, May 16-19, 1993, pp. 127-134.

[Peacock et al.,1979] J. K. Peacock, J. W. Wong, E. G. Manning: *Distributed Simulation Using a Network of Processors*, Computer Networks 3, February 1979, pp. 44-56.

[Phillips&Cuthbert,1991] C. I. Phillips, L. G. Cuthbert: *Concurrent Discrete Event-Driven Simulation Tools*, IEEE Journal on Selected Areas in Communications, Vol. 9, No. 3, April 1991, pp. 477-485.

[Prakash&Subramanian,1991] A. Prakash, R. Subramanian: *Filter: An Algorithm for Reducing Cascaded Rollbacks in Optimistic Distributed Simulations*, Proceedings of the 24th Annual Simulation Symposium, New Orleans, LA, USA, April 1991, pp. 123-132.

[Preiss,1989] B. R. Preiss: *The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environments*, Distributed Simulation 1989, SCS Simulation Series, Vol. 21, No. 2, March 1989, pp. 139-144.

[Preiss et al.,1992] B. R. Preiss, I. D. MacIntyre, W. M. Loucks: *On the Trade-off between Time and Space in Optimistic Parallel Discrete-Event Simulation*, 6th Workshop on Parallel and Distributed Simulation (PADS'92), SCS Simulation Series, Vol. 24, No. 3, January 1992, pp. 33-42.

[Reed et al.,1988] D. A. Reed, A. D. Malony, B. D. McCredie: *Parallel Discrete Event Simulation Using Shared Memory*, IEEE Transactions on Software Engineering, Vol. 14, No. 4, April 1988, pp. 541-553.

[Reiher et al.,1989] P. L. Reiher, F. Wieland, D. Jefferson: *Limitation of Optimism in the Time Warp Operating System*, Proceedings of the 1989 Winter Simulation Conference, Washington, DC, USA, December 4-6, 1989, pp. 765-770.

[Reiher et al.,1990] P. Reiher, R. Fujimoto, S. Bellenot, D. Jefferson: *Cancellation Strategies in Optimistic Execution Systems*, Distributed Simulation 1990, SCS Simulation Series, Vol. 22, No. 1, January 1990, pp. 112-121.

[Rönngren&Ayani,1994] R. Rönngren, R. Ayani: *Adaptive Checkpointing in Time Warp*, 8th Workshop on Parallel and Distributed Simulation (PADS'94), Edinburgh, Scotland, U. K., July 6-8, 1994, pp. 110-117.

[Rothermel,1993] K. Rothermel: *Verteilte Systeme*, Skript zur Vorlesung, Lehrstuhl Verteilte Systeme, Institut für Parallele und Verteilte Höchstleistungsrechner, Universität Stuttgart, 1993.

[Schmidt,1994] D. C. Schmidt: *The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software*, 12th Sun User Group Conference, San Jose, CA, USA, June 14-17, 1994.

[Schmidt,1995] D. C. Schmidt: *Design Patterns for Initializing Network Services: Introducing the Acceptor and Connector Patterns*, C++ Report, Vol. 7, No. 8, SIGS Publications, NY, USA, November/December 1995.

[Stein,1993] W. Stein: *Objektorientierte Analysemethoden - ein Vergleich*, Informatik Spektrum, Band 16, 1993, S. 317-332.

[Steinman,1992a] J. S. Steinman: *SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete Event Simulation*, The International Journal for Computer Simulation, Vol. 2, No. 3, 1992, pp. 251-286.

[Steinman,1992b] J. S. Steinman: *SPEEDES: A Unified Approach to Parallel Simulation*, 6th Workshop on Parallel and Distributed Simulation (PADS'92), SCS Simulation Series, Vol. 24, No. 3, January 1992, pp. 75-84.

[Stoutamire,1995] D. Stoutamire: *The pSather 1.0 Manual*, International Computer Science Institute, Berkeley, CA, USA, 1995.

[Stroustrup,1991] B. Stroustrup: *The C++ Programming Language*, 2nd Edition, Addison-Wesley, Reading, MA, USA, 1991.

[Stroustrup,1994] B. Stroustrup: *The Design and Evolution of C++*, Addison-Wesley, Reading, MA, USA, 1994.

[Teale,1993] S. Teale: *C++ IOStreams Handbook*, Addison-Wesley, Reading, MA, USA, 1993.

[ten Brink,1995] S. ten Brink: *Untersuchungen zur optimistischen parallelen Simulation von Warteschlangensystemen auf einem Parallelrechner*, 2. Semesterarbeit, Nr. 1385, Institut für Nachrichtenvermittlung und Datenverarbeitung, Universität Stuttgart, 1995.

[Tinker&Agre,1989] P. A. Tinker, J. R. Agre: *Object Creation, Messaging, and State Manipulation in an Object Oriented Time Warp System*, Distributed Simulation 1989, SCS Simulation Series, Vol. 21, No. 2, March 1989, pp. 79-84.

[Tomlinson&Garg,1993] A. I. Tomlinson: *An Algorithm for Minimally Latent Global Virtual Time*, 7th Workshop on Parallel and Distributed Simulation (PADS'93), San Diego, CA, USA, May 16-19, 1993, pp. 35-42.

[Turner&Xu,1992] S. J. Turner, M. Q. Xu: *Performance Evaluation of the Bounded Time Warp Algorithm*, 6th Workshop on Parallel and Distributed Simulation (PADS'92), SCS Simulation Series, Vol. 24, No. 3, January 1992, pp. 117-126.

[Unger et al.,1994] B. W. Unger, D. J. Goetz, S. W. Maryka: *Simulation of SS7 Common Channel Signaling*, IEEE Communications Magazine, Vol. 32, No. 3, March 1994, pp. 52-62.

[Vinoski,1993] S. Vinoski: *Distributed Object Computing with CORBA*, C++ Report, SIGS Publications, NY, USA, July/August 1993.

[Waldschmidt,1995] K. Waldschmidt (Hrsg.): *Parallelrechner: Architekturen – Systeme – Werkzeuge*, B. G. Teubner, Stuttgart, 1995.

[Wegner,1990] P. Wegner: *Concepts and Paradigms of Object-Oriented Programming*, OOPS Messenger, Vol. 1, No. 1, ACM Special Interest Group on Programming Languages, August 1990, pp. 7-87.

[Wegner,1992] P. Wegner: *Dimensions of Object-Oriented Modeling*, IEEE Computer, Vol. 25, No. 10, October 1992, pp. 12-20.

[West&Mullarney,1988] J. West, A. Mullarney: *ModSim: A Language for Distributed Simulation*, Distributed Simulation 1988, SCS Simulation Series, Vol. 19, No. 3, February 1988, pp. 155-159.

[Willmann,1989] G. Willmann: *Modelling and Performance Evaluation of Multi-Layered Signalling Networks Based on the CCITT*

No. 7 Specification, North-Holland Studies in Telecommunication, ITC-12: Teletraffic Science for New Cost-Effective Systems, Networks and Services, Part 2, M. Bonatti, ed., Amsterdam: North-Holland, 1989, pp. 930-940.

[Willmann&Kühn,1990] G. Willmann, P. J. Kühn: *Performance Modeling of Signaling System No. 7*, IEEE Communications Magazine, Vol. 28, No. 7, July 1990, pp. 44-56.

Anhang A

Notation für objektorientierten Entwurf

Zur Unterstützung des objektorientierten Entwurfs nach [Booch,1994], wie in Unterkapitel 3.2 vorgestellt, existiert eine zugehörige Notation. Im folgenden werden die Teile der Notation, die in dieser Arbeit verwandt wurden, kurz erläutert. Ausführlichere Informationen, insbesondere auch zu den hier nicht gebrauchten Interaktions-, Modul-, Prozeß- und Zustands-Übergangsdiagrammen können [Booch,1994] entnommen werden.

A.1 Klassendiagramme

Bild A.1: Klassen und Klassen-Kategorien

Das grafische Grundelement von *Klassendiagrammen* ist eine „Wolke" mit gestrichelter Umrandung (siehe Bild A.1). Jede „Wolke" repräsentiert eine Klasse, wobei der Name der Klasse in ihrem Innern steht. Optional können zusätzlich noch wichtige Methoden und Felder der Klasse angegeben werden, wobei die Felder in der Form *Name : Typ* geschrieben werden. Falls notwendig, kann vor jeder Methode bzw. jedem Feld durch senkrechte Linien die Sichtbarkeit angegeben werden. Keine Linie bedeutet dabei, die Methode oder das Feld ist öffentlich zugänglich (public), bei einer Linie ist sie geschützt (protected), und bei zwei Linien ist sie privat (private). Handelt es sich bei der Klasse um eine abstrakte Basisklasse, wird dies durch ein „A" in einem auf der Spitze stehenden Dreieck innerhalb

der Wolke dargestellt. Um eine weitergehende Strukturierung zu ermöglichen, können zusammengehörende Klassen in *Klassen-Kategorien* gruppiert werden. Eine Klassen-Kategorie wird durch ein Rechteck dargestellt, das den Namen der Kategorie enthält und optional die enthaltenen Klassen auflistet (siehe Bild A.1 rechts).

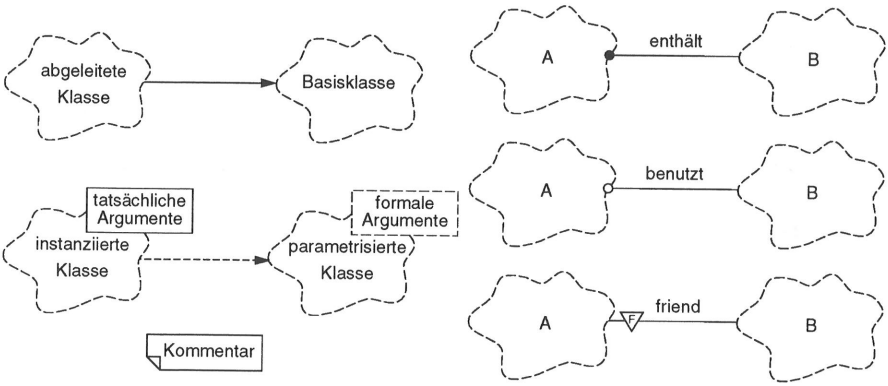

Bild A.2: Beziehungen zwischen Klassen

Die Notation für die grundlegenden Beziehungen zwischen Klassen ist in Bild A.2 gezeigt. Vererbungsbeziehungen werden durch einen Pfeil von der abgeleiteten Klasse zur Basisklasse repräsentiert, eine Enthaltensein-Beziehung durch eine Linie mit gefülltem Kreis, eine Benutzen-Beziehung durch eine Linie mit leerem Kreis und eine Friend-Beziehung durch eine Linie mit einem „F“ in einem auf der Spitze stehenden Dreieck. Im Bild ist Klasse B in Klasse A enthalten, wird Klasse B von Klasse A benutzt und ist Klasse A ein „Friend“ von Klasse B, d. h. Klasse A hat Zugriff auf die privaten und geschützten Methoden und Felder von Klasse B. Im Bild ist außerdem noch die Notation für eine parametrisierte Klasse, eine davon instanziierte Klasse sowie einen Kommentar, der zum besseren Verständnis an beliebigen Stellen angebracht werden kann, gezeigt.

Die Vererbungs- und Enthaltensein-Beziehungen können noch mit verschiedenen Attributen versehen werden. Bild A.3 zeigt die Notation für öffentliche (public), geschützte (protected) und private (private) Ableitung sowie für virtuelle Basisklassen. Bezüglich der Sichtbarkeitsregeln gilt diese Notation auch für die Enthaltensein-Beziehung (siehe Bild A.4). Bei dieser kann auch noch angegeben werden, ob eine Klasse physikalisch als Instanz in einer anderen enthalten ist („has by value“) oder durch eine Referenz („has by reference“). Angebrachte Zahlen sagen etwas über die Anzahl der auf jeder Seite der Beziehung beteiligten Objekte aus. Im Bild hat 1 Instanz der linken Klasse 0 bis n Instanzen der rechten. Ist schließlich die Beziehung statisch, d. h. ist für alle Instanzen einer Klasse nur eine gemeinsame Instanz der enthaltenen Klasse vorhanden, wird dies durch ein „S“ in einem auf der Spitze stehenden Dreieck gekennzeichnet.

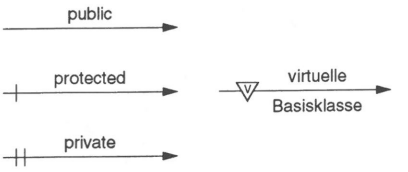

Bild A.3: Attribute der Vererbungs-Beziehung

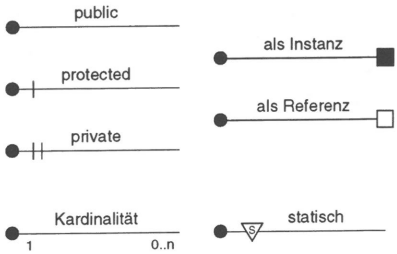

Bild A.4: Attribute der Enthaltensein-Beziehung

A.2 Objektdiagramme

Objektdiagramme zeigen Objekte des Systems und ihre Beziehungen zueinander. Mit ihrer Hilfe können exemplarisch wichtige Abläufe im System dargestellt werden. Objekte werden durch eine „Wolke" mit durchgezogener Umrandung symbolisiert (siehe Bild A.5). Die Bezeichnung des Objekts steht im Innern und hat die Form *Objektname : Klasse*, wobei optional auch nur der Objektname oder die Klasse angegeben werden können. Soll zum Ausdruck kommen, daß es sich um mehrere Instanzen einer Klasse handelt, so können mehrere Wolken hintereinandergelegt werden (Bild A.5 rechts).

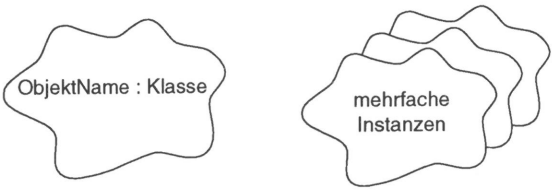

Bild A.5: Objekte

Beziehungen zwischen Objekten werden durch Linien gekennzeichnet (siehe Bild A.6), wobei zusätzlich die gegenseitige Sichtbarkeit der Objekte angegeben werden kann, wo dies notwendig ist. Dies geschieht durch kleine Rechtecke mit Buchstaben an den jeweiligen Enden der Linien. Im Bild steht der Buchstabe „F" beispielsweise dafür, daß Objekt 2 als Feld in Objekt 1 enthalten ist. Weitere Möglichkeiten, die Sichtbarkeit zu bezeichnen, sind ein „P", um zu kennzeichnen, daß das Objekt 2 ein Parameter einer Methode des Objekts 1 ist, ein „L" dafür, daß Objekt 2 ein lokales Objekt von Objekt 1 ist und „G" schließlich für ein globales Objekt. Ist das Rechteck ausgefüllt, kann Objekt 2 exklusiv Objekt 1 zugeordnet werden, ist das Rechteck leer, handelt es sich um eine Referenz.

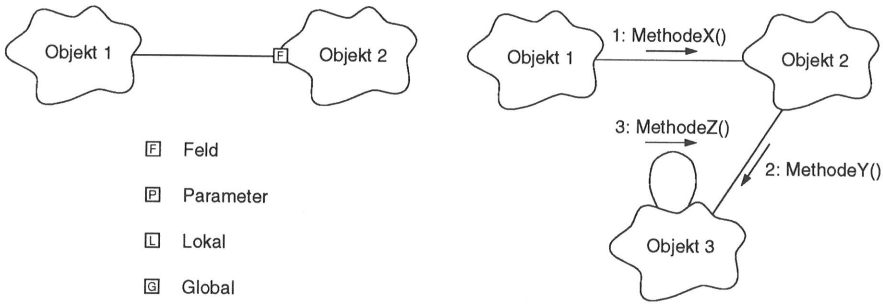

Bild A.6: Beziehungen zwischen Objekten

Dynamische Abläufe werden durch Pfeile an den Beziehungen dargestellt. Diese Pfeile symbolisieren einen Methoden-Aufruf und werden mit dem Namen der aufgerufenen Methode gekennzeichnet. Der Ablauf der Aufrufe geht aus zusätzlich angebrachten Reihenfolgenummern hervor (Bild A.6 rechts).

Anhang B

Laufzeit-Typüberprüfung

In Paragraph 3.2.4.3.2 wurde schon ausführlich auf die Nutzung des dynamischen Bindens beim objektorientierten Entwurf eingegangen. Dieses ermöglicht es, (virtuelle) Methoden einer gemeinsamen Basisklasse von Objekten aufzurufen, wobei die ausgeführte Methode dann vom tatsächlichen Typ des Objekts abhängt. Bei einem sauberen objektorientierten Entwurf kann das dynamische Binden die meisten expliziten Abfragen nach dem Typ eines Objekts (meistens in größeren *switch-case*-Anweisungsfolgen) ersetzen. Dies geht allerdings nicht immer. So benutzt die parallele Bibliothek z. B. das Port-Konzept der sequentiellen, das mit Objekten des Typs *TMessage* arbeitet. Die meisten Komponenten der parallelen Bibliothek können aber nur Nachrichten der abgeleiteten Klasse *TParMessage* handhaben bzw. müssen sicherstellen, z. B. nur Nachrichten des Typs *TParTWSimMsg* für optimistische Simulation zu erhalten.

Zur Lösung solcher Probleme wurde 1993 das Konzept der Laufzeit-Typüberprüfung (*Run-Time Type Information*, *RTTI*) in den Entwurf des *C++*-Standards aufgenommen (siehe [Stroustrup,1994] oder [C++-Draft,1995]). Es sieht u. a. einen neuen Operator *dynamic_cast* vor, mit dessen Hilfe ein Zeiger (oder eine Referenz) des Typs einer abgeleiteten Klasse aus einem Zeiger (oder einer Referenz) des Typs einer Basisklasse gewonnen werden kann. Die Typumwandlung ist im Gegensatz zum alten Cast-Mechanismus in *C/C++* sicher, da sie nur durchgeführt wird, wenn der Zeiger auch wirklich auf ein Objekt des abgeleiteten Typs zeigt. Im anderen Fall wird entweder Null zurückgeliefert (bei Zeigern) oder eine Ausnahme ausgeworfen (bei Referenzen). Das RTTI-Konzept beinhaltet außerdem noch den Operator *static_cast*, dessen Zweck die nicht zur Laufzeit überprüfte Umwandlung eines Zeigers auf den Typ einer Basisklasse in einen Zeiger auf den Typ einer abgeleiteten Klasse ist, den Operator *reinterpret_cast*, der einen Zeiger eines Typs in einen Zeiger eines anderen, nicht abgeleiteten Typs umwandelt und den Operator *const_cast*, der ein *const*-Attribut entfernt. Schließlich gibt es noch einen Operator *typeid*, mit dem der exakte Typ des Objekts festgestellt werden kann, und die Struktur *type_info*, die Informationen über diesen Typ enthält.

Bei den für diese Arbeit zur Verfügung stehenden *C++*-Compilern war das RTTI-Konzept aufgrund der relativ späten Aufnahme in den Entwurf des Standards leider noch nicht implementiert. Um trotzdem eine Möglichkeit einer überprüften Typumwandlung ähnlich des *dynamic_cast*-Operators zu haben, gibt es verschiedene Möglichkeiten. Eine besteht im Einfügen eines Bezeichner-Feldes in der Basisklasse. Dieses kann bei der Erzeugung von Objekten abgeleiteter Klassen gesetzt und später abgefragt werden. Ergibt die Abfrage, daß das Objekt tatsächlich vom gewünschten abgeleiteten Typ ist, kann nachfolgend eine sichere Typumwandlung mittels des alten Cast-Mechanismus erfolgen. Bei tieferen Ableitungs-Hierarchien versagt diese Lösung allerdings, da sie immer nur auf den Typ der am weitesten abgeleiteten Klasse abprüfen kann (falls nicht in jeder abgeleiten Klasse wiederum ein Bezeichner-Feld vorhanden ist – dies erfordert allerdings geschachtelte Abfragen und wird sehr unhandlich und unflexibel).

Eine zweite, in der parallelen Bibliothek realisierte Möglichkeit ist die (teilweise) Nachbildung der RTTI-Funktionalität. Das implementierte Konzept ist eine bezüglich Geschwindigkeit optimierte Variante eines Vorschlags aus [Stroustrup,1991]. Die gefundene Lösung ist schnell, einfach anwendbar und unterstützt einfache Vererbung, bzw. bei mehrfacher Vererbung einen Pfad im Ableitungsbaum. Zudem ist der zusätzliche Speicherplatzbedarf gering.

Zentrales Element des in Bild B.1 gezeigten Konzepts ist die Klasse *TRTTIInfo*, die die Typ-Information für eine Klasse enthält. Es handelt sich dabei um den Namen der Klasse,

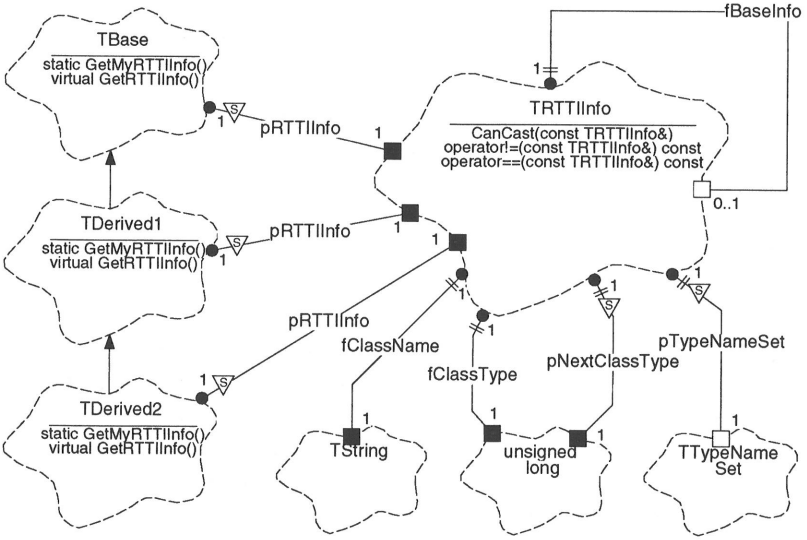

Bild B.1: Laufzeit-Typüberprüfung

fClassName, sowie einen innerhalb eines Logischen Knotens eindeutigen Bezeichner, *fClassType*, vom einfachen Datentyp *unsigned long*. Letzterer wird jeder neuen Instanz der Klasse *TRTTIInfo* automatisch mit Hilfe des statischen Feldes *pNextClassType* zugewiesen. Dieses wird nach jeder Vergabe inkrementiert, womit die erforderliche Eindeutigkeit gewährleistet ist. Das statische Feld vom Typ *TTypeNameSet* enthält alle Namen der an der Typüberprüfung teilnehmenden Klassen (für die deswegen eine Instanz der Klasse *TRTTIInfo* existiert) und verhindert doppelte Namenseinträge. *TRTTIInfo* enthält außerdem noch einen Zeiger *fBaseInfo* auf die Typ-Information der nächsthöheren Basisklasse. Die Methode *TRTTIInfo::CanCast()* stellt entlang dieses Zeigers rekursiv fest, ob die als Argument übergebene Typ-Information mit der Typ-Information einer Basisklasse übereinstimmt. Ist dies der Fall, sind die entsprechenden Klassen voneinander abgeleitet und es kann ein Cast einer Referenz (oder eines Zeigers) der Basisklasse auf eine Referenz (oder einen Zeiger) der abgeleiteten Klasse durchgeführt werden. Der Vergleich beider Typ-Informationen findet dabei durch den überladenen Gleichheits-Operator anhand eines Vergleichs des Feldes *fClassType* statt. Da es sich dabei um einen einfachen Datentyp handelt, ist diese Operation nicht aufwendig – im Gegensatz zum Originalvorschlag aus [Stroustrup,1991], der einen Vergleich der Zeichenketten beider Klassennamen vorsieht.

In Bild B.1 ist ein Beispiel dargestellt, bei dem die Laufzeit-Typüberprüfung für den aus den Klassen *TBase*, *TDerived1* und *TDerived2* bestehenden Ableitungsbaum implementiert ist. Jede dieser Klassen enthält ein statisches Feld der Klasse *TRTTIInfo* und stellt die beiden Methoden *GetMyRTTIInfo()* und *GetRTTIInfo()* bereit. Erstere ist eine statische Funktion und liefert eine Referenz auf das RTTI-Feld der Klasse, auf die sie angewandt wurde, zurück, letztere ist eine virtuelle Funktion und liefert eine Referenz auf das RTTI-Feld der Klasse des Objekts zurück, auf das sie angewandt wurde.

Soll z. B. festgestellt werden, ob sich hinter einer Referenz auf *TBase* ein Objekt des Typs *TDerived1* verbirgt (und handelt es sich z. B. tatsächlich um ein Objekt des Typs *TDerived2*), erreicht man dies durch Aufruf der Methode *GetRTTIInfo()*, die die Typ-Information des Objekts zurückliefert (hier von *TDerived2*). Durch Aufruf von *TDerived1::GetMyRTTIInfo().CanCast()* mit der erhaltenen Typ-Information als Parameter, kann nun festgestellt werden, ob *TDerived1* eine Basisklasse ist und falls ja, ein entsprechender Cast durchgeführt werden (in diesem Fall möglich, da *TDerived2* von *TDerived1* abgeleitet ist).

Um die Anwendung des beschriebenen Konzepts einfach zu gestalten, wurden verschiedene Makros definiert. *RTTIDECLARATION* wird einfach in die (öffentliche) Schnittstelle einer an der Typüberprüfung teilnehmenden Klasse eingefügt und definiert dort die Methoden *GetRTTIInfo()*, *GetMyRTTIInfo()* und das statische Feld *pRTTIInfo*. Ein weiteres Makro, *RTTIDEFINITION*, muß für jede teilnehmende Klasse im Quelltext plaziert werden. Ihm werden als Parameter der Name der Klasse und der Name der direkten Basisklasse (falls vorhanden) übergeben. Damit wird dann das statische *pRTTIInfo*-Feld initialisiert.

Für die Typumwandlung selbst wurden wie in [Stroustrup,1991] zwei Makros definiert: *ptr_cast(T, p)* wandelt den Zeiger p falls möglich in einen Zeiger des Typs T um. Falls dies nicht möglich ist, ergibt die Umwandlung Null. *ref_cast(T, r)* wandelt die Referenz r falls möglich in eine Referenz des Typs T um, sonst wird eine Ausnahme ausgeworfen.

Ein konkretes Anwendungsbeispiel nach Bild B.1 aus Sicht des Programmierers ist das folgende:

```
class TBase {
public:
   RTTIDECLARATION
};

class TDerived1 : public TBase {
public:
   RTTIDECLARATION
};

class TDerived2 : public TDerived1 {
public:
   RTTIDECLARATION
};

RTTIDEFINITION0(TBase)
RTTIDEFINITION1(TDerived1,TBase)
RTTIDEFINITION1(TDerived2,TDerived1)

main()
{
   TBase     b;
   TDerived2 d;
   TBase     *b_ptr = &d;
   TBase     &b_ref = d;
   TDerived1 *d1_ptr = ptr_cast(TDerived1,b_ptr); // d1_ptr zeigt auf d
             d1_ptr = ptr_cast(TDerived1,&b);     // d1_ptr ist 0
   TDerived1 &d1_ref = ref_cast(TDerived1,b_ref); // d1_ref referenziert d
   TDerived1 &d2_ref = ref_cast(TDerived1,b);     // wirft Ausnahme aus
}
```

Da RTTI nur sehr zurückhaltend eingesetzt werden sollte, hält sich auch die Zahl der Klassen, die mittels der Makros darauf vorbereitet werden müssen, und damit der erforderliche Zu-

satzaufwand in Grenzen. Die Makros *ptr_cast(T, p)* und *ref_cast(T, r)* zur Typumwandlung selbst sind dagegen genauso einfach anzuwenden wie der im Entwurf des *C++*-Standards vorgesehene Operator *dynamic_cast*. Steht RTTI durch den Compiler später zur Verfügung, können die Makros problemlos gegen diesen Operator ausgetauscht werden.