Technische Universität Braunschweig

Institut für Betriebssysteme und Rechnerverbund

simuLan

Ein Werkzeug zur Simulation lokaler Netze

Diplomarbeit

cand. inform. Stefan Petri

Aufgabenstellung und Betreuung :
Prof. Dr. H. Langendörfer

Braunschweig
22. Juli 1993

Zusammenfassung

Durch den Preisverfall der Hardware und die Zunahme der Leistungsfähigkeit bekommt die Vernetzung von Personal Computern und Workstations untereinander und mit Spezialrechnern immer größere Bedeutung. simuLan ist ein Werkzeug zur effizienten und flexiblen Modellierung und Simulation solcher Netzwerke. Modellkomponenten für neue Protokolle und Anwendungen lassen sich mit wenig Aufwand implementieren. Die Konzepte von simuLan werden vorgestellt, die Bedienung des Programms erläutert und Erweiterungsmöglichkeiten beschrieben.

Abstract

With decreasing hardware costs and increasing computing power the interconnection of Personal Computers, Workstations and special purpose hardware becomes more and more important. simuLan is a tool for efficient and flexible modelling and simulation of such networks. Models for new protocols and applications can be programmed with little effort. The concepts of simuLan are shown, the usage of the program is explained and the possibilities for customizing are discussed.

Inhaltsverzeichnis

1 Einleitung und Motivation **1**
1.1 Überblick über die Eigenschaften von simulan 1

2 Netzwerke **3**
2.1 Das OSI - Referenzmodell 3
2.2 Protokollschichten im ARPA Internet 6
2.3 Lokale Netze 7

3 Modellieren und Simulation **9**
3.1 Modellierungs - Verfahren 9
3.2 Simulationstechniken 10
3.2.1 Der Zeitablauf 10
3.2.2 Lastparameter 11
3.2.3 Verkehrsmatrizen 11
3.2.4 Wiederholung von Simulationsläufen mit variierten Parametern . . 12
3.3 Instrumentierung 12
3.4 Dynamische Kontrolle der Simulationslänge 12
3.5 Bestimmung von Konfidenzintervallen 13
3.5.1 Das BMVA-Verfahren in simulan 14
3.5.2 Der Ablauf des implementierten BMVA-Verfahrens 15
3.6 Dynamische Aggregierung 18

4 Modellieren mit simulan **19**
4.1 Darstellbare Topologien 19
4.2 Protokollschichten in simulan 19
4.3 Baukastenprinzip 20

5 Aufbau einer Konfigurationsdatei **22**
5.1 Notation 24
5.2 Zerlegung der Konfigurationsdatei in Worte 24
5.3 Der Vorspann 24
5.3.1 Definition von Ausgabekanälen 25
5.3.2 Definition von Laufvariablen 25
5.4 Der Hauptteil der Konfiguration 26
5.4.1 Name - Wert - Paare 26
5.4.2 Globale Simulationsparameter 27
5.4.3 Beschreibung der Netzknoten (Schicht $2-5$) 28
5.4.4 Beschreibung des Netzmodells (Schicht 1) 29
5.4.5 Beschreibung der Instrumentierung 29
5.4.6 Beschreibung der statistischen Absicherung 31
5.5 Der Abschluß der Konfiguration 32

6 Bedienung von simula 33
6.1 Aufruf . . . 33
6.2 Signale . . . 34
6.3 Fehlermeldungen . . . 34

7 Implementation von Modellkomponenten 35
7.1 Grundkomponenten für Protokollschichtmodelle . . . 36
7.1.1 Die Eventliste . . . 36
7.1.2 Zufallsverteilungen . . . 36
7.1.3 Verkehrsverteilungen . . . 37
7.1.4 Warteschlangen . . . 37
7.1.5 Speicherverwaltung . . . 38
7.2 Zusammenspiel der Module . . . 38
7.3 Globale Datenstrukturen . . . 39
7.4 Implementation von Einlesefunktionen . . . 41
7.4.1 Einlesefunktionen für Schicht 5 . . . 42
7.4.2 Einlesefunktionen für die Schichten 2,3,4 . . . 42
7.4.3 Einlesefunktionen für Schicht 1 . . . 43
7.5 Initialisierungsfunktionen . . . 43
7.6 Aufräumfunktionen . . . 43
7.7 Dienstzugangs- und Anzeigefunktionen . . . 43
7.8 Debugging-Funktionen . . . 45
7.9 Implementation von Meßpunkten . . . 45

8 Beispiele zur Realisierung von Protokollschichtmodulen 47
8.1 Ein Netzmodul (Schicht 1) : PTP . . . 47
8.1.1 Datenstrukturen und Variablen des PTP-Moduls . . . 47
8.1.2 `ptp_rc()` . . . 50
8.1.3 `ptp_init()` . . . 52
8.1.4 `ptp_cleanup()` . . . 52
8.1.5 `ptp_data_encapsulation()` . . . 52
8.1.6 `ptp_end_transmit()` . . . 54
8.1.7 `ptp_transmit_error()` . . . 55
8.1.8 Die Instrumentierung des PTP-Modells . . . 55
8.1.9 `ptp_print_frame()` . . . 57
8.2 Eine lasterzeugende Anwendung : MandelSpawn . . . 57
8.2.1 Das mslaved - Modell . . . 58
8.2.2 Parameter des mslaved - Modells . . . 58
8.2.3 Datenstrukturen und Variablen . . . 59
8.2.4 `mslaved_rc()` . . . 62
8.2.5 `mslaved_gwc42()` . . . 64
8.2.6 `mslaved_pdu_received()` . . . 64
8.2.7 `mslaved_endofservice()` . . . 66
8.2.8 `mslaved_timeout()` . . . 66
8.2.9 `mslaved_pdu_sended()` . . . 67
8.2.10 `mslaved_print_info()` . . . 68
8.2.11 `print_ms_answer()` . . . 68
8.3 Protokollschichtmodule für die Schichten 2, 3, 4 . . . 68

9 Ausblicke 70

A simuLan Programmierhandbuch **71**
A.1 Allgemeine Datentypen . . . 71
A.1.1 Simulierte Zeit: `TIME_T` . . . 71
A.1.2 Zeiger auf unbekannte Objekte: `VOID_P` . . . 71
A.2 Allgemeine Variablen . . . 71
A.3 Allgemeine Makros . . . 72
A.4 Auswertung der Kommandozeile . . . 72
A.5 Globale Simulationsparameter . . . 73
A.6 Speicherverwaltung . . . 74
A.6.1 `myalloc()`, `myrealloc()`, `myfree()` . . . 74
A.6.2 `mystrdup()` . . . 75
A.6.3 Generische Freilisten: `DEFUN_GET_REL_OBJECT()` . . . 75
A.7 Die Eventliste . . . 76
A.7.1 Die `EVENT` - Struktur . . . 76
A.7.2 `insert_event()`, `CHANGE_EVENT`, `CANCEL_EVENT()` . . . 76
A.7.3 `get_next_event()` . . . 77
A.7.4 `print_event()` . . . 78
A.8 Warteschlangen . . . 78
A.8.1 `make_nqeue()`, `nq_push`, `nq_pop()`, `print_nqueue()` . . . 78
A.8.2 `make_tqeue()`, `tq_push`, `tq_pop()`, `print_tqueue` . . . 79
A.9 Wahrscheinlichkeitsverteilungen . . . 79
A.9.1 `ZUFALL_INFO`, `USE_ZUFALL()` . . . 79
A.9.2 Als Funktion implementierte Wahrscheinlichkeitsverteilungen . . . 80
A.9.3 `UNIFORM()`, `INT_UNIFORM()`, `EXPNTL()`, `INT_EXPNTL()` . . . 80
A.10 Verkehrsmatrizen . . . 81
A.11 Knotenbeschreibungen . . . 82
A.11.1 Die `NODE` - Struktur . . . 82
A.11.2 `lookupnodename()` . . . 83
A.12 Netzbeschreibung . . . 83
A.13 Schnittstellen zwischen den Protokollschichtmodulen . . . 84
A.13.1 Die `SADU` - Struktur . . . 84
A.13.2 Der Service Access Point `Ln_sap` . . . 85
A.13.3 Nachrichten empfangen: `Ln_pdu_received` . . . 85
A.13.4 Sendeauftrag abschließen: `Ln_pdu_sended` . . . 86
A.14 Instrumentierung . . . 86
A.14.1 Ausgabeformatierung: `probe_printf()` . . . 86
A.14.2 Die `PROBE` - Struktur . . . 87
A.14.3 Die Schnittstelle zur statistischen Absicherung . . . 89
A.15 Einlesen der Konfiguration . . . 89
A.15.1 Fehlerbehandlung:`RC_WARNING`,`RC_SYNTAX_ERROR`,`RC_ERROR`, `RC_FATAL` 89
A.15.2 Worte Einlesen : `xxgetword()`, `xxungetword()` . . . 90
A.15.3 Worte ausgeben: `xxprintword()` . . . 90
A.15.4 Ausgabeumlenkung: `xxgetfd()` . . . 90
A.15.5 Zahlenangaben einlesen:`xxgetint()`,`xxgetulong()`, `xxgetfloat()`, `xxgetdouble()` . . . 91
A.15.6 Einheitenangaben: `xxgetbits()`, `xxgettime_t()`, `xxgetmeter()` 91
A.15.7 Funktionen auswählen: `xxgetfkt()` . . . 92
A.15.8 Instrumentierung einlesen: `rcprobepoint()` . . . 92
A.15.9 `copy_probepoint()` . . . 93
A.15.10 `rcreadkeywds()` . . . 93
A.15.11 Einlesen von Verteilungsfunktionen . . . 95
A.15.12 Einlesen von Verkehrsverteilungen : `traffic_rc()` . . . 95
A.15.13 Einlesen der globalen Simulationsparameter . . . 96

A.15.14 Einlesen der Knotenparameter 96
A.15.15 Einlesen der Netzparameter 97
A.16 Debugging 97
A.16.1 Debugflags 97
A.16.2 Datenstrukturen und Funktionen 98
A.16.3 Interaktive Fehlersuche 99

B Übersicht über Paketaufbau – Protokolloverheads 100

C Programmquellen 102

Literaturverzeichnis 103

Begriffsdefinitionen 115

Gesamtindex 116

Typenindex 121

Variablenindex 122

Funktionenindex 123

Kapitel 1

Einleitung und Motivation

> At the beginning of the 1990s, it is becoming clear that the computing environment will continue to undergo fundamental changes, driven primarily by technological progress. Roughly speaking, we can divide the history of modern computing into eras:
>
> - 1970s: Timesharing (1 computer with many users)
> - 1980s: Personal computing (1 computer per user)
> - 1990s: Parallel computing (many computers per user)
>
> (A.S. Tanenbaum, im Intro zur Dokumentation des verteilten Betriebssystems "Amoeba")

Durch den Preisverfall der Hardware bei gleichzeitiger Zunahme der Leistungsfähigkeit geht der Trend immer stärker weg vom zentralen "Großrechner" hin zu Personal Computern und Workstations. Die Vernetzung solcher "kleinen" Rechner untereinander und mit Spezialrechnern (File Server, Number Cruncher) bekommt eine immer größere Bedeutung. Dadurch entsteht das Bedürfnis nach Leistungsbewertung und Leistungsvorhersage solcher Netzwerke. simuLan ist ein Werkzeug, mit dem das Verhalten von existierenden oder geplanten Netzwerken (einschließlich der Netzknoten) simuliert und Leistungsvorhersage durchgeführt werden kann.

Diese Arbeit beschreibt das Grundgerüst von simuLan und die Möglichkeiten, die zur Modellierung konkreter Netzwerkbestandteile geboten werden.

1.1 Überblick über die Eigenschaften von simuLan

- Baukastensystem – Modelle von Rechnernetzen können einfach aus vordefinierten Modellkomponenten zusammengesetzt werden.
- Hierarchische Modellierung von Protokollschichten.
- Flexible Instrumentierung – die Modellkomponenten enthalten Meßpunkte, an denen auf Wunsch Meßwerte ermittelt und in wählbarem Format ausgegeben werden.
- Ausgabeumlenkung – verschiedene Meßspuren können in verschiedene Dateien geschrieben oder direkt an Nachbereitungsprogramme übergeben werden.
- Statistische Absicherung – für die ermittelten Meßwerte können Vertrauensgrenzen automatisch ermittelt werden.

- Dynamische Kontrolle der Länge von Simulationsläufen – die Simulation wird beendet, wenn für ausgewählte Meßwerte die vorgegebenen Vertrauensgrenzen erreicht sind.
- Wiederholung von Simulationsläufen mit variierten Parametern.
- Einfache Erweiterbarkeit – ein Skelett für neue Modellkomponenten läßt sich nach vorgegebenem Schema erzeugen, für die Ausformulierung in C steht eine Anzahl von Grundkomponenten zur Verfügung.
- Ereignisorientierte Simulation.
- Hohe Performanz und Flexibilität, da alle Komponenten in C implementiert sind.
- Selbsterläuternd – simuLAN kann eine kommentierte Schablone für eine Modellbeschreibung ausgeben, in der alle verfügbaren Modellkomponenten und ihre Parameter angegeben sind.
- Alphanumerische Ein- und Ausgabe – dadurch unabhängig von Hardware, Betriebssystem oder Graphikoberfläche.
- Checkpointing – bei langen Programmläufen können Aufsetzpunkte erzeugt werden, von denen aus nach einem Rechnerausfall weitergerechnet werden kann.
- Interaktives Debugging.
- Gezieltes Verfolgen der Ereignisse in bestimmten Modulen (Event Tracing) – für Debugging und Verifikation.
- Gezieltes Verfolgen bestimmter Nachrichten (Packet Tracing) – funktionale Analyse.
- Möglichkeiten zur interaktiven Variation von Parametern.
- Die Dokumentation ist online verfügbar.

Kapitel 2

Netzwerke

> Lokale Netze sind Datenkommunikationssysteme, welche einer Anzahl von unabhängigen Einrichtungen eine zumeist partnerschaftlich orientierte Kommunikation hoher Datenrate auf relativ begrenztem geographischen Gebiet ermöglicht. (aus [Lan83])

2.1 Das OSI - Referenzmodell

Um die Komplexität von Netzwerkentwürfen zu verringern, sind die meisten Netzwerke in Protokollschichten (Layer, im Folgenden kurz Schichten) organisiert.

Die ISO (International Standardization Organisation) hat dazu das sogenannte "OSI Reference Model" vorgeschlagen [ISO88]. Das Modell soll dazu beitragen, den Informationsaustausch zwischen Systemen zu reglementieren und damit technologische Unterschiede verschiedener Systeme zu überwinden.

Die Funktionen der Schicht N werden durch eine oder mehrere identische *N-Instanzen* (*N-Entities*) geleistet. Eine N-Instanz bietet ihren Funktionsumfang der Schicht *N+1* in Form von Diensten (Services) an und benutzt ihrerseits Dienste der Schicht *N-1*. Die Dienste sind nur über *Dienstzugangspunkte* (*Service Access Point*, *SAP*) zugänglich.

Der Ablauf der Kommunikation zwischen N-Instanzen, die zu verwendenden Datenformate und die Zuordnung Dienstprimitiv $\leftrightarrow$ Dateneinheit werden in einem *N-Protokoll* beschrieben. Der äußere Rahmen eines N-Protokolls ist durch die Schnittstellen bzw. Dienstprimitive vorgegeben.
Die physikalische Verbindung zweier Systeme ermöglicht eine direkte Kommunikation in der untersten Schicht. Durch den hierarchischen Aufbau kann auch eine N-Instanz mit ihrer *Partnerinstanz* (*Peer-Entity*) – nach einem festgelegten *Schichtenprotokoll* – logisch kommunizieren, indem sie die direkte Kommunikation mit der Schicht *N-1* benutzt. Diese Interaktion zwischen benachbarten Schichten wird durch *Dienstprotokolle* geregelt. Dienstbenutzer und Diensterbringer verkehren dabei über *Dienstprimitive* (*Service Primitives*):

`Request`	`: Anforderung`	`Schicht`	$N{+}1 \rightarrow N$,
`Indication`	`: Anzeige`	`Schicht`	$N{+}1 \leftarrow N$,
`Response`	`: Antwort`	`Schicht`	$N{+}1 \rightarrow N$,
`Confirm`	`: Bestätigung`	`Schicht`	$N{+}1 \leftarrow N$.

Die Datenübertragung zwischen Peer-Entities erfolgt über *Protokolldateneinheiten* (*Protocol Data Unit*, *PDU*), die von den tieferliegenden Schichten nicht interpretiert werden. Die Datenübertragung ist also transparent.

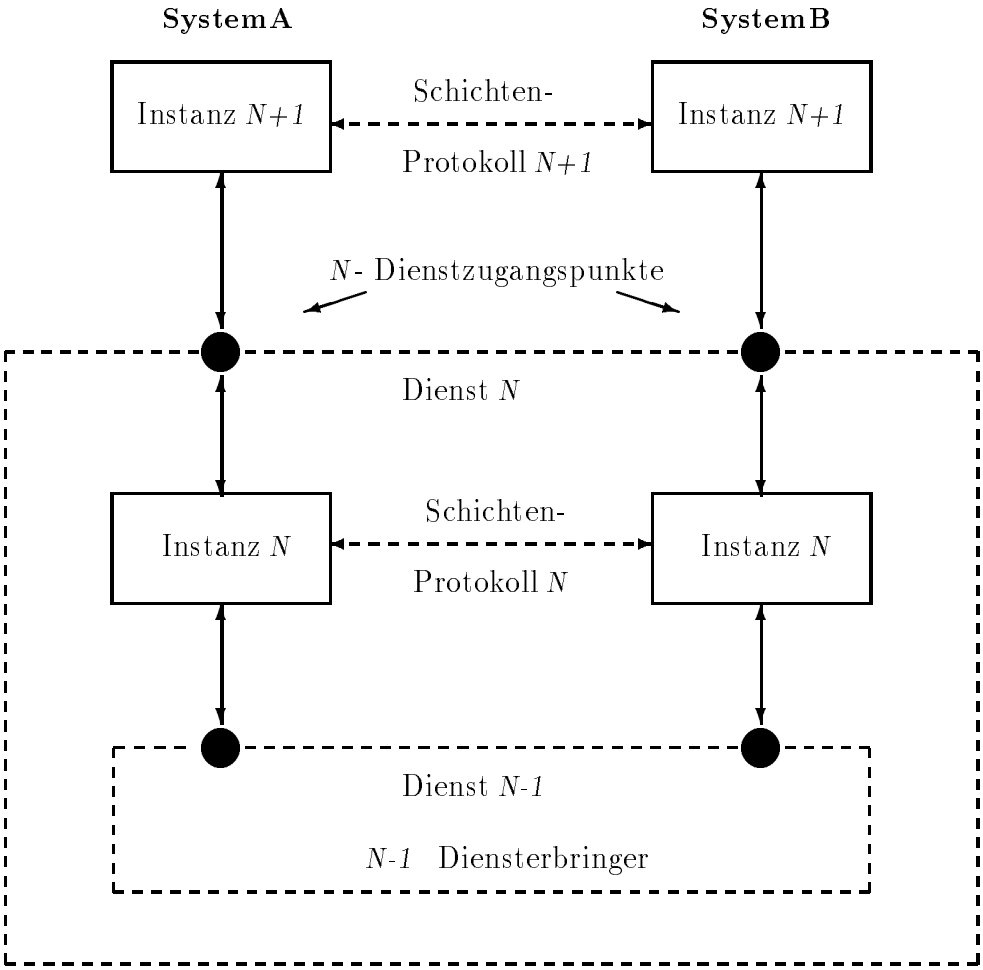


Abbildung 2.1: Modell einer Protokollschicht und ihrer Interaktion [GKS+85]

Das Ende einer Verbindung zwischen Partner-Instanzen bildet innerhalb der SAPs ein *Verbindungsendpunkt* (*Connection Endpoint*, *CEP*).

Das OSI Reference Model ordnet jeder der vorgeschlagenen 7 Schichten bestimmte Aufgaben zu. Es liefert weder eine exakte Beschreibung der Dienste dieser Schicht, noch eine genaue Definition der zu verwendenden Protokolle zwischen diesen.

Den einzelnen Schichten werden folgende Aufgaben zugeordnet :

1) *Physical Layer* **:** Festlegung der mechanischen und elektrischen Charakteristiken der Verbindung. Es wird eine transparente, ungesicherte Übertragung von Bitfolgen zur Verfügung gestellt.

2) *Data Link Layer* **:** Aufteilung des zwischen zwei benachbarten Netzknoten übertragenen Bitstromes in *Frames* und Absicherung gegen physikalische Übertragungsfehler.

3) *Network Layer* **:** Routing vom Senderknoten zum Empfängerknoten, einschließlich Behandlung von Überlastsituationen.

4) *Transport Layer* **:** Zuverlässige Ende-zu-Ende - Kommunikation, einschließlich Flußkontrolle.

5) *Session Layer* **:** Die Sitzungsschicht bietet Dienste zur Eröffnung, geordneten Durchführung und Beendigung einer Sitzung; sie synchronisiert die miteinander kommunizierenden Instanzen.

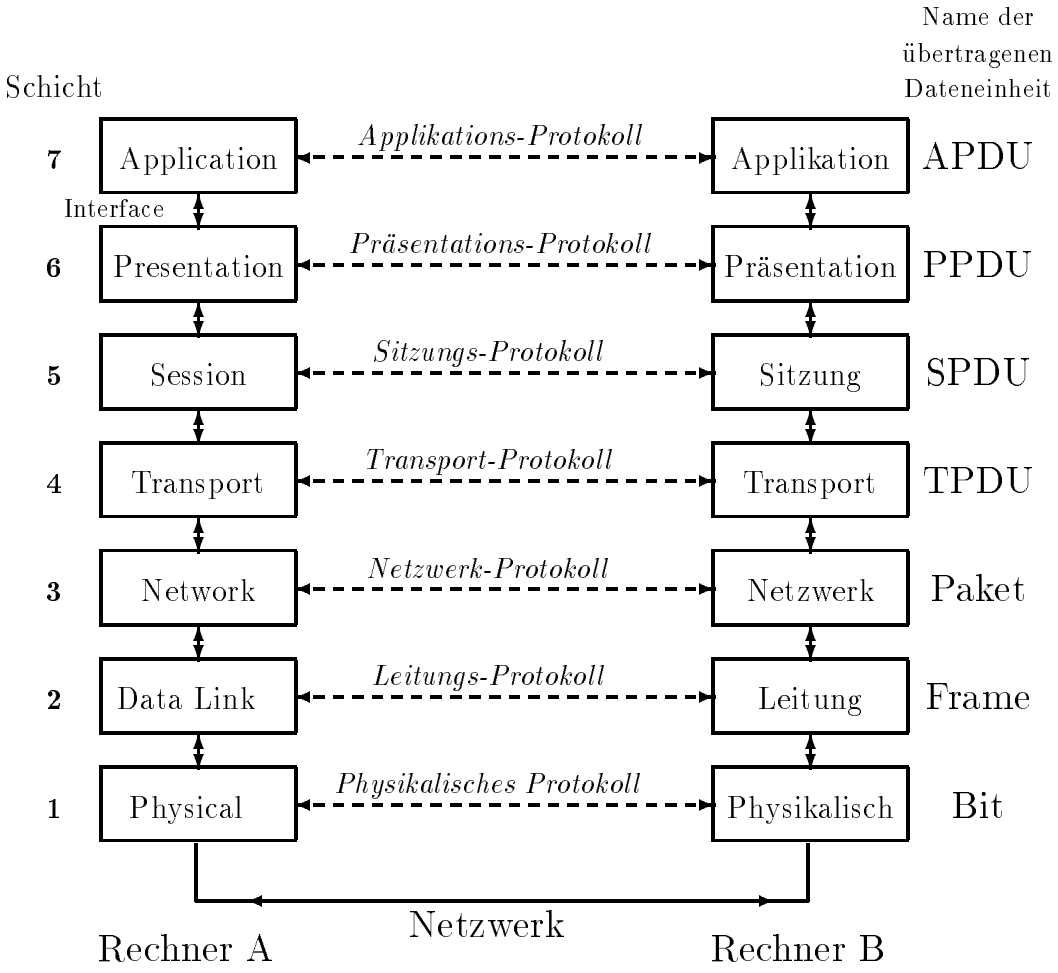


Abbildung 2.2: Das ISO/OSI-Referenzmodell

6) *Presentation Layer* : Es werden Aufgaben der Gerätesteuerung und Anpassungen an gerätespezifische Eigenschaften realisiert. (Codeumwandlung EBCDIC $\leftrightarrow$ASCII, Verschlüsselung von Daten.)

7) *Application Layer* : Hier sind Dienste wie elektronische Post, Dateitransfer und virtuelle Terminals angesiedelt. Diese Dienste werden von den Anwenderprozessen direkt genutzt.

Weitergehende Literatur über Netzwerkarchitekturen und Protokollschichten findet sich z.B. in [GKS$^+$85], [Com88], [Tan89].

2.2 Protokollschichten im ARPA Internet

Das im ARPA Internet verwendete Schichtenmodell wurde nicht von einem Standardisierungskommitee vorgeschlagen, sondern es entwickelte sich aus den Anforderungen der Praxis.

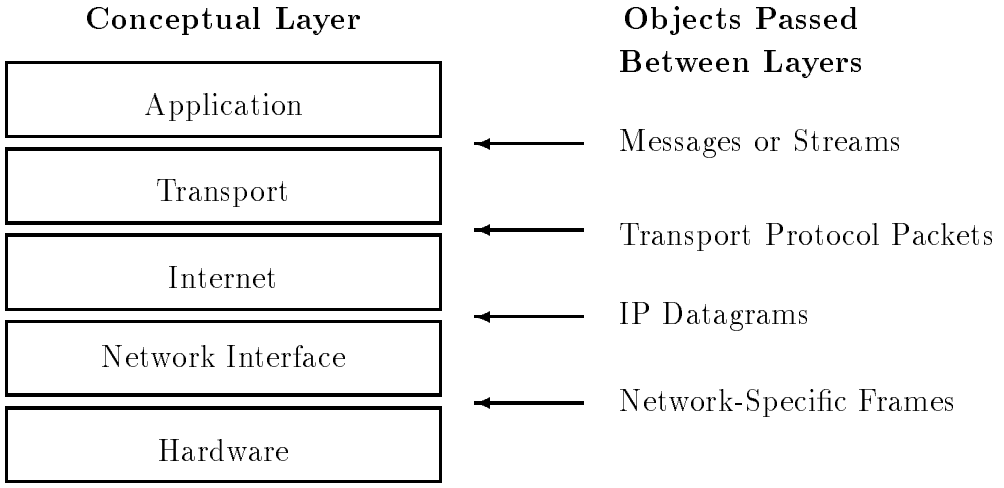


Abbildung 2.3: The 5 conceptual layers of hardware and protocol software used in the Internet, and the form of objects passed between layers. The layer labeled *Network Interface* is sometimes called the *data link* layer. [Com88], pg. 108

Application Layer : Die Benutzerprogramme benutzen die Transportschicht um Daten zu senden und zu empfangen. Jedes Benutzerprogramm benutzt ein eigenes Datenformat, entweder als Bytestrom oder als Folge von einzelnen Nachrichten.

Transport Layer : Die Transportschicht ermöglicht den Austausch von Daten zwischen Benutzerprogrammen. Sie kann zuverlässige Kommunikation anbieten. Die Benutzerdaten werden in Pakete zerlegt und an die nächst tiefere Schicht übergeben.

Internet Layer : Diese Schicht transportiert Datenpakete vom Senderknoten zum Empfängerknoten. Sie umfaßt Routing und Behandlung von Überlastsituationen.

Network Interface Layer : Diese Schicht regelt die Übertragung zwischen direkt verbundenen Knoten, spezifisch für die Art der Verbindung.

Hardware : Die direkte physikalische Verbindung der Netzknoten.

Ein Unterschied zwischen dem ISO- und dem Internet- Modell liegt in der Realisierung der zuverlässigen Datenübertragung. Im ISO - Modell wird auf allen Protokollschichten eine zuverlässige Übertragung mit Prüfsummen, Acknowledgements, Timeouts und Retransmits angestrebt.
Im Internet - Modell wird Zuverlässigkeit als Aufgabe der Ende-zu-Ende- Übertragung in der Transportschicht angesehen. Dadurch vereinfacht sich das Verständnis und die Implementation der Protokollsoftware erheblich.

In [Com88] vergleicht D. Comer die beiden Schichten- Konzepte (Kapitel 10). Kapitel 1 enthält einen Überblick über Geschichte und Organisation des ARPANET / Internet.

2.3 Lokale Netze

Lokale Netze (*Local Area Networks*, *LAN*s) zeichnen sich aus durch :

- eine geographische Ausdehnung von einigen 10 Metern bis zu einigen Kilometern.
- einen Kommunikationskanal mittlerer oder hoher Datenübertragungsrate (1 Mbit/s bis zu wenigen 100 Mbit/s).
- eine niedrige Fehlerrate.
- einfache Möglichkeiten, das Netz um weitere Rechner und Peripherie zu erweitern.
- den Besitz und Gebrauch durch eine einzelne Organisation.
- einfache und flexible Erweiterbarkeit.
- einfacher Netzzugang für den Benutzer.

Die am weitesten verbreitete Norm für lokale Netze ist der IEEE- Standard 802 (Institute of Electrical and Electronics Engineers), der weitgehend von der OSI übernommen wurde (OSI 8802). In ihm sind verschiedene Netzzugriffsverfahren normiert. Er umfaßt die Teile 802.1 bis 802.5 und 802.7, der Standard 802.6 über *Metropolitan Area Networks* (MAN) ist in Vorbereitung :

802.1 gibt eine Übersicht über den Aufbau des Dokuments und motiviert die Unterteilung der ISO- Schicht 2 (Data Link Layer, DLL) in die Teilschichten 2a (*Logical Link Layer*, *LLC*) und 2b (*Medium Access Layer*, *MAC*). Diese Unterteilung ist notwendig, da sich einerseits für verschiedene Netztopologien verschiedene Zugriffsverfahren etabliert haben, andererseits die Funktionalität des DLL unabhängig vom verwendeten Zugriffsprotokoll sein soll (siehe Abb. 2.4, Seite 8). *Diese Unterteilung wird in* s[i]mu[l]a[t]n *in der Unterteilung der Protokollschichten nachvollzogen.*

802.2 spezifiziert für die LLC - Schicht drei verschiedene Protokolltypen durch die Angabe von Funktionsumfang und Dienstprimitiven :

- LLC-Typ 1 realisiert einen verbindungslosen Dienst, ohne Acknowledgements oder Garantie, daß die Daten beim Empfänger ankommen.
- LLC-Typ 2 realisiert einen verbindungsorientierten Dienst. Vor Übertragungsbeginn muß eine Verbindung zwischen Sender und Empfänger aufgebaut werden. Die Ankunft der Daten und die Einhaltung ihrer Reihenfolge ist garantiert. Nach Ende der Übertragung muß die Verbindung explizit abgebaut werden.
- LLC-Typ 3 realisiert einen verbindungslosen Dienst mit Acknowledgements.

802.3 spezifiziert das *Carrier Sense Multiple Access / Collision Detection* - Zugriffsverfahren (*CSMA/CD*). Die Stationen sind passiv an ein einziges Medium, den *Bus*, gekoppelt. Stationen greifen nur aktiv auf den Bus zu, wenn sie eine Nachricht senden wollen und ihn vorher als unbelegt erkannt haben (Carrier Sense). Da alle sendewilligen Stationen gleichberechtigt sind (Multiple Access), werden hier Verfahren zur Konflikterkennung (Collision Detection) und Konfliktbehebung definiert für den Fall, daß mehrere Stationen gleichzeitig auf den Bus zugreifen. Die bekannteste Implementation dieses Verfahrens ist das *Ethernet*.

802.4 beschreibt das *Token-Bus*- Zugriffsverfahren, das als Übertragungsmedium ebenfalls einen allgemein zugänglichen Bus benutzt. Der Zugriff erfolgt im Gegensatz zu CSMA/CD allerdings deterministisch. Die angeschlossenen Stationen werden logisch in einem Ring angeordnet, indem jeder Station ein Nachfolger zugewiesen wird. Auf diesem logischen Ring kreist ein spezielles Bitmuster, das Token, dessen Besitz einer Station die Sendeberechtigung zuteilt. Liegt bei Empfang des Tokens ein Sendewunsch vor, so beginnt die Station zu senden, andernfalls wird das Token sofort an den Nachfolger weitergegeben. Nach Beendigung der Übertragung wird ein neues Token generiert und an den Nachfolger weitergegeben.

802.5 beschreibt das *Token-Ring*- Zugriffsverfahren. Es arbeitet ähnlich dem Token-Bus. Die Stationen sind durch Ring- Segmente verbunden, so daß der Nachfolger einer Station nicht explizit bekannt sein muß. Im Gegensatz zum Token-Bus arbeitet jede Station "passiv", indem sie ankommende Signale auf das nächste Ringsegment weiterleitet. Nur bei Empfang des Tokens und vorliegendem Sendewunsch sendet eine Station eigene Daten und "vernichtet" sie nach einem Ringumlauf wieder. Ein weiterer Unterschied ist, daß das Token nicht vom Ring genommen und anschließend neu generiert wird, sondern ständig als Frei- oder Besetzt-Token auf dem Ring kreist. Diese Unterscheidung wird durch Invertierung eines Bits im Token durchgeführt.

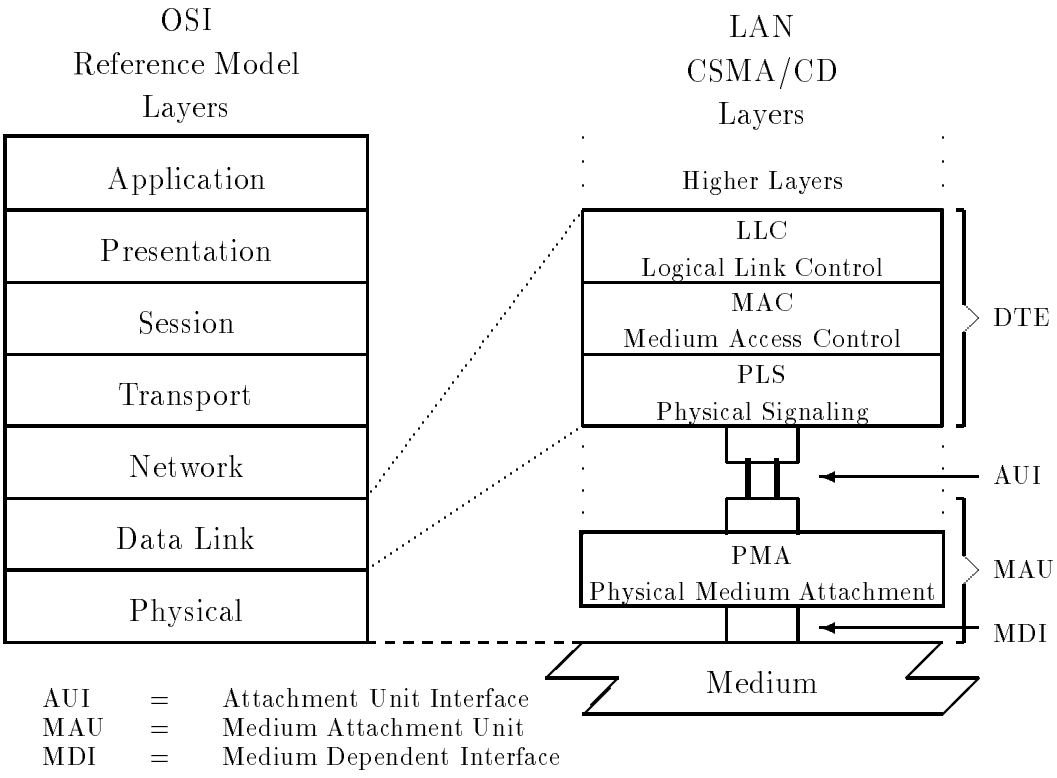


Abbildung 2.4: Service Specification Relations to the LAN Model (aus [IEE88c])

Kapitel 3

Modellieren und Simulation

Um komplexe Systeme zu untersuchen ist es häufig sinnvoll, Modelle zu konstruieren, die die interessierenden Aspekte des zu untersuchenden Systems nachbilden. Einige Gründe für die Verwendung von Modellen sind:

- Ein existierendes System steht nicht für Experimente zur Verfügung (Kernreaktor, Rechnernetz im Produktionseinsatz).
- Es sollen Vorhersagen über ein geplantes System gemacht werden.
- Es sollen Meßdaten gewonnen werden, die in einem existierenden System nicht zur Verfügung stehen (Programme zur Leistungsmessung verbrauchen auch selber Rechenzeit und I/O- Bandbreite, wodurch die Meßdaten verfälscht werden.)
- Simulationen können mit Zeitlupen- oder Zeitraffereffekt durchgeführt werden, um sehr schnelle oder sehr langsame Vorgänge zu betrachten (funktionale Analyse).

3.1 Modellierungs - Verfahren

Grundsätzlich sind zwei Methoden der Modellierung zu unterscheiden :

- *Analytische Modelle* : Das zu untersuchende System wird durch Gleichungen beschrieben, aus denen die interessierenden Kenngrößen abgeleitet werden. Oftmals kann man Systeme in Komponenten zerlegen, für die die Lösung der beschreibenden Gleichungen bekannt ist.
- *Simulative Methoden* : Das Verhalten der Komponenten des zu untersuchenden Systems und ihr Zusammenwirken werden nachgeahmt und daraus die interessierenden Kenngrößen gemessen.

Hybrid-Modelle kombinieren die Vorteile der analytischen Modelle mit denen der Simulation, indem einzelne Komponenten durch ihre Kenngrößen bzw. deren Verteilungsfunktionen beschrieben werden, ohne näher auf ihr "Innenleben" einzugehen.

Für die Leistungsanalyse lassen sich Rechensysteme als *Warteschlangennetz* modellieren, in dem "Jobs" (Kunden) um "Resourcen" (Bediener) konkurrieren, die sie zur Bearbeitung ihrer Aufgaben benötigen. Solange ein Bediener mit einem Kunden beschäftigt ist, müssen die neu ankommenden Kunden warten. Die interessierenden Leistungsmerkmale solcher System sind u.a. Auslastung und Durchsatz der einzelnen Bediener, mittlere Länge der

Warteschlangen, mittlere Wartezeiten der Kunden, Gesamtzeit für die Bearbeitung eines Jobs. Diese Kenngrößen lassen sich analytisch bestimmen, wenn die Verteilungsfunktionen für die Zwischen-Ankunftszeiten der Kunden und für deren Resourcen-Anforderungen sowie die Bedienstrategien der Bediener und die Verteilungsfunktionen der Bedienzeiten bekannt sind und hinreichende Vorbedingungen erfüllen. [Kle76]

An ihre Grenzen stoßen die analytischen Methoden, wenn ein Kunde mehrere verschiedene Resourcen gleichzeitig beansprucht (ein Job braucht Speicher und CPU), sowie bei Verwendung nicht-trivialer Bedienstrategien.

Großen Einfluß auf die Entwicklung von simuLAN hatte, daß sich Netze mit nicht - deterministischen Zugriffsverfahren (z.B. CSMA/CD) nur dann analytisch bewerten lassen, wenn starke Voraussetzungen in das Modell hineingesteckt werden [Bor88] [Lan88].

Der Vorteil der Simulation ist, daß die Modelle beliebig detailgetreu sein können. Zu Bedenken ist aber, daß bei Simulationsprogrammen mit der Genauigkeit auch der Aufwand an Modellierarbeit und Rechenzeit steigt. Zur Unterstützung der Modellierarbeit stehen in simuLAN vorgefertigte Protokollschichtmodelle zur Verfügung, die wie bei einem Baukasten zum Modell eines Rechnernetzes zusammengesetzt werden können. simuLAN ist so konzipert, daß für eine Protokollschicht verschiedene Protokollschichtmodelle unterschiedlicher Genauigkeit gegeneinander ausgetauscht werden können.

simuLAN wurde in C implementiert. Im Vergleich zu anderen Simulationssystemen, die in speziellen Simulationssprachen geschrieben wurden, ergibt sich dadurch ein geringerer Rechenzeitbedarf [Lan88]. Außerdem wird eine größere Flexibilität erreicht, was am Beispiel der CSMA-Verfahren deutlich wird, die sich in Sprachen wie GPSS, SIMULA oder HIT nur vereinfacht modellieren lassen [SD91].

Für die Erstellung neuer Protokollschichtmodelle (die Konstruktion neuer "Bauelemente") stellt simuLAN eine Bibliothek von Grundfunktionen zur Verfügung. (für eine Erläuterung der Konzepte siehe Abschnitt 7 [Implementation von Modellkomponenten], Seite 35; für eine detaillierte Beschreibung der Funktionen und Datenstrukturen siehe Abschnitt A [simuLAN *Programmierhandbuch*], Seite 71).

Ausführliche Literatur über Modellieren und Simulation findet sich z.B in [Lav83], [Kob78].

3.2 Simulationstechniken

3.2.1 Der Zeitablauf

> And time began seriously to pass.
> (D. Adams, Dirk Gently's Holistic Detective Agency.)

Bei der Simulation eines Systems werden die Änderungen des Zustands über der Zeit nachgebildet.

In simuLAN wird die *ereignisorientierte Simulation* angewendet (*Discrete Event Simulation*). Dabei wird angenommen, daß Zustandsänderungen des Systems durch einzelne *Ereignisse* ausgelöst werden. Jedes Ereignis tritt zu einer bestimmten Zeit ein, wodurch die Reihenfolge der Bearbeitung festgelegt ist. Die aktuelle Simulationszeit wird jeweils auf die Zeit des nächsten zu bearbeitenden Ereignisses gesetzt und die Zustandsänderungen gemäß der Ereignisbeschreibung berechnet. Dabei werden ggf. neue, zukünftige Ereignisse erzeugt. Bei der Simulation nicht-deterministischer Netzzugriffsverfahren (z.B. CSMA/CD) tritt zusätzlich der Fall auf, daß bereits erzeugte Ereignisse gestrichen oder auf einen anderen

Zeitpunkt verschoben werden müssen [SD91]. Mit den meisten Simulationssprachen ist so etwas nur stark eingeschränkt oder gar nicht möglich.

Im Gegensatz dazu steht die *Fixed Time Increment-* Methode, bei der die simulierte Zeit in festen Intervallen weitergesetzt wird. Bei jedem Schritt werden die sich ergebenden Zustandsänderungen berechnet. Diese Methode hat sich für die Simulation von lokalen Netzen als ungünstig erwiesen, da die relevanten Zustandsänderungen in ungleichmäßigen Abständen auftreten. Bei Verwendung kleiner Intervalle werden viele "leere" Zeitschritte bearbeitet, bei großen Intervallen werden Zustandsänderungen als gleichzeitig betrachtet, die eigentlich nacheinander stattfinden.

In simuLan wird als interne Zeiteinheit die *Indivisible Time Unit* (*ITU*) verwendet. Sie kennt nur ganzzahlige Werte. Die Größe einer ITU wird zur Laufzeit angegeben. Sie muß so klein gewählt werden, daß sie in der Größenordnung der Zeitauflösung des simulierten Systems liegt (siehe Abschnitt A.1.1 [*Datentyp für die simulierte Zeit*], Seite 71).

3.2.2 Lastparameter

Um das Verhalten eines Systems zu simulieren, wird eine Arbeitslast als Eingabeparameter für das Modell verwendet. Diese Arbeitslast kann mittels Zufallsverteilungen erzeugt werden, wofür in der Literatur gelegentlich der Begriff *Monte Carlo Methode* verwendet wird. Eine andere Möglichkeit ist, Meßspuren von einem realen System als Eingabe zu benutzen (*Trace Driven*). Dafür brauchbare Meßwerte sind aber sehr schwierig zu bekommen, daher wurde dieses Verfahren in simuLan bisher nicht realisiert.

Die Beschreibung der Arbeitslast wird nicht durch das Grundgerüst von simuLan festgelegt, sondern muß in die Modelle der lasterzeugenden Anwendungen eingebaut werden. Dazu steht eine Liste der wichtigsten Verteilungsfunktionen zur Verfügung, sowie ein Mechanismus, der es erlaubt, die Verteilungsfunktionen und ihre Parameter zur Laufzeit auszuwählen, statt sie zur Compilezeit vorgeben zu müssen. (siehe Abschnitt 7.1.2 [*Konzept der Zufallsverteilungen*], Seite 36, und Abschnitt A.15.11 [Einlesen von Verteilungsfunktionen], Seite 95).

Für analytische Modellkomponenten, die durch ihre Kenngrößen beschrieben werden, können die Verteilungsfunktionen der Kenngrößen mit dem gleichen Mechanismus ausgewählt werden.

Die Liste der aktuell implementierten Verteilungsfunktionen wird von simuLan in der Schablone für die Konfigurationsbeschreibung ausgegeben (siehe Abschnitt 6 [*Bedienung von* simuLan], Seite 33).

Die Zufallsfunktionen bauen auf den in der Systembibliothek vorhandenen Pseudozufallsgeneratoren auf. Sollten sich diese auf einer Maschine als schlecht erweisen, können leicht eigene Generatoren eingesetzt werden.

3.2.3 Verkehrsmatrizen

Ein wichtiger Parameter für die Modellierung von Netzwerken ist die Angabe, welche der Netzknoten miteinander kommunizieren (z.B. Untersuchung von Client - Server - Beziehungen). Es ist üblich, für jedes Paar *(S, E)* eine Wahrscheinlichkeit anzugeben, mit der eine vom Sender S erzeugte Nachricht an den Empfänger E gerichtet ist. Daraus ergibt sich zunächst eine quadratische *Verkehrsmatrix*. In der Praxis zeigt sich jedoch, daß diese Matrix meistens dünn besetzt ist. In simuLan kann daher einem Knoten eine Liste von "bevorzugten" Partnerknoten und zugehörigen Wahrscheinlichkeiten zugeordnet werden.

Falls die Summe der Wahrscheinlichkeiten kleiner als eins ist, wird für die Differenz eine Gleichverteilung über alle Knoten angenommen.

In den Fällen, in denen eine solche detaillierte Beschreibung nicht benötigt wird, kann der Empfänger der Nachrichten eines Senders gleichverteilt aus allen simulierten Netzknoten ausgewählt werden.

3.2.4 Wiederholung von Simulationsläufen mit variierten Parametern

Ein einzelner Simulationslauf reicht häufig nicht aus, um ein System zu beurteilen. simuLAN unterstützt die Variation von Eingabeparametern, so daß deren Einfluß auf das Verhalten des Systems beurteilt werden kann. In der Beschreibung eines zu simulierenden Netzwerkes können Laufvariablen definiert werden, deren Werte für Lastparameter eingesetzt werden. Für jede Wertekombination der Laufvariablen wird ein Simulationslauf durchgeführt, entsprechend der Abarbeitung verschachtelter FOR-NEXT-Schleifen in Programmiersprachen.

3.3 Instrumentierung

Um die Gewinnung, Verarbeitung und Ausgabe von Meßdaten während der Simulation möglichst flexibel und doch einheitlich zu gestalten, werden an den für "interessant" befundenen Stellen einer Modellkomponente *Meßpunkte* einprogrammiert. An diesen Meßpunkten können jeweils mehrere Messungen durchgeführt werden. Zu jeder Messung gehören eine *Meßfunktion*, Ausgabekanäle und Ausgabefunktionen sowie ggf. eine Funktion zur statistischen Absicherung. In der Beschreibung des zu simulierenden Systems wird angegeben, ob und welche Messungen an einem Meßpunkt durchgeführt werden sollen, ob und wohin die Meßspuren ausgegeben werden sollen, und wie die Ausgaben formatiert werden sollen. Außerdem kann angegeben werden, daß für einen Meßwert eine statistische Absicherung durchgeführt werden soll (siehe Abschnitt 3.5 [Bestimmung von Konfidenzintervallen], Seite 13).

Die Meßfunktionen werden für jeden Meßpunkt nach einem vorgegebenen Schema programmiert. Die definierten Meßpunkte und die wählbaren Meßfunktionen werden in der Schablone für die Beschreibung der zu simulierenden Konfiguration ausgegeben (siehe Abschnitt 6 [*Bedienung von* simuLAN], Seite 33).

3.4 Dynamische Kontrolle der Simulationslänge

> Meine Damen und Herren, noch 30 Sekunden.
> (Lufthansapilot beim Anflug auf Hannover im Nebel).

Bei der Simulation eines Systems interessieren im allgemeinen nicht die an den Meßpunkten gewonnen Meßwertreihen, sondern deren Mittelwerte als Kenngrößen des Systems (z.B. mittlere Auslastung einer Systemkomponente). Es ergibt sich also das Problem, aus einer Reihe von Stichproben auf den wahren Mittelwert zu schließen. Dazu müssen einerseits so viele Stichproben gewonnen werden, daß der errechnete Mittelwert möglichst wenig vom wahren Mittelwert abweicht, andererseits soll die Simulation nicht unnötig lange laufen.

In simuLAN kann die Länge eines Simulationslaufes auf mehrere Arten begrenzt werden :

- Durch Vorgabe eines zu simulierenden Zeitintervalls.

- Durch Vorgabe einer maximalen Anzahl zu bearbeitender Ereignisse.
- Durch Angabe einer maximalen Länge für Warteschlangen. Wird diese Länge überschritten, so deutet das daraufhin, daß das simulierte System instabil ist.
- simulan ist in der Lage selbständig zu bestimmen, ob für eine Kenngröße genügend Meßwerte gesammelt wurden. In der Beschreibung einer Messung, die an einem Meßpunkt vorgenommen werden soll, kann angegeben werden, daß jeder ermittelte Meßwert an eine *Absicherungs-Funktion* übergeben werden soll. Diese berechnet den laufenden Mittelwert sowie dessen Vertrauensgrenzen. Der Simulationslauf wird abgebrochen, wenn für alle derart angegebenen Kenngrößen das vorgegebene Konfidenzintervall erreicht wurde (siehe Abschnitt 3.5 [Bestimmung von Konfidenzintervallen], Seite 13).

Diese dynamische Bestimmung der Simulationslänge ist eine Abkürzung des sonst zu beobachtenden Benutzerverhaltens, bei dem ein Simulationslauf mit steigender, fest vorgegebener Länge wiederholt wird, bis die Vertrauensgrenzen der gesuchten Kenngrößen zufriedenstellend eng sind.

> Wie beim Roulette : Man spielt solange, bis man gewonnen hat.
> [schöni]

Bei der Anwendung dieser dynamischen Simulationslängenkontrolle ist Vorsicht geboten, da simulan nicht feststellen kann, ob für eine gegebene Modellbeschreibung die theoretischen Voraussetzungen für die Errechnung der Vertrauensgrenzen überhaupt gegeben sind.

Die Vertrauensgrenzen können auch berechnet werden, ohne Einfluß auf die Länge der Simulation zu haben.

Darüberhinaus ist in simulan keine weitere Auswertung der Meßwerte vorgesehen. Diese wird nachgeschalteten Programmen überlassen, die die Ausgaben von simulan direkt als Eingabe bekommen, oder sie aus einer Datei einlesen.

3.5 Bestimmung von Konfidenzintervallen

Bei der Auswertung einer durch Simulation ermittelten Meßwertreihe wird der Erwartungswert einer Zufallsvariablen mit unbekannter Verteilungsfunktion durch das Stichprobenmittel geschätzt. Als Maß für die Güte der Schätzung verwendet man die Breite des *Konfidenzintervall*s, das um den geschätzten Erwartungswert gelegt wird. Es ist so breit, daß es den wahren Erwartungswert mit der Wahrscheinlichkeit $(1 - \alpha)$ einschließt [Sch82c].

Die Überdeckungswahrscheinlichkeit $(1 - \alpha)$ wird in der englischsprachigen Literatur als *Confidence Level* bezeichnet.

Für die Ermittelung von Konfidenzintervallen werden in der Literatur im Wesentlichen fünf Verfahren vorgeschlagen :

- *Replikation* : Der Simulationslauf wird viele Male mit verschiedenem Startwert für den Zufallszahlengenerator wiederholt, um statistisch unabhängige Meßwertreihen zu erhalten. Die Mittelwerte der einzelnen Meßreihen sind nach dem zentralen Grenzwertsatz normalverteilt. Daraus läßt sich die Varianz schätzen und mit dieser die Vertrauensgrenzen. Außerdem läßt sich die Anzahl der benötigten Meßwerte pro

Simulationslauf bestimmen. Der Nachteil dieser Methode ist, daß bei jeder Wiederholung der Simulation die unproduktive Einschwingphase durchlaufen werden muß, und daß die Anzahl der erforderlichen Meßwerte erst berechnet werden kann, nachdem schon Simulationsläufe durchgeführt worden sind.

- *Batch Mean Value Analysis* (*BMVA*) : Der Endzustand eines Simulationslaufes wird als Anfangszustand des nächsten benutzt. Das bedeutet, daß ein langer Simulationslauf durchgeführt wird, dessen Meßwertreihe in einzelne Segmente (*Batches*) aufgeteilt wird. Dadurch wird die unproduktive Einschwingphase nur einmal durchlaufen. Die Batches müssen so viele Meßwerte umfassen, daß ihre Mittelwerte statistisch unabhängig werden. Dann läßt sich die Varianz berechnen und mit dieser wiederum die Vertrauensgrenzen.

- *Regeneration* : Es wird ein langer Simulationslauf durchgeführt. Die erzeugte Meßwertreihe wird in voneinander statistisch unabhängige Abschnitte (Zyklen) unterteilt, die jeweils in einem Systemzustand beginnen, in dem das zukünftige Verhalten von der Vergangenheit unabhängig ist. Bei der Simulation komplexer Systeme lassen sich solche Zustände im allgemeinen nur sehr schwer bestimmen.

- *Spektrum-Methode* : Statt die statistische Unabhängigkeit der Stichproben zu fordern, wird deren Autokovarianz geschätzt. Daraus läßt sich die Varianz berechnen und mit dieser wieder die Vertrauensgrenzen. Es ist jedoch sehr schwierig nachzuweisen, ob ein System die Voraussetzungen für die Anwendung der Spektrum-Methode erfüllt (B. Schmidt in [Sch82c] nach *Duket und Pritsker, Examination of Simulation Output Using Spectral Methods, 1978*).

- *Autoregression* : Eine nicht-unabhängige Meßwertreihe wird durch Darstellung als "autoregressiver Prozess" in eine Reihe statistisch unabhängiger, identisch verteilter Meßwerte transformiert. Das Verfahren stellt keine Anforderungen an das simulierte System, aber sehr hohe an das mathematische Verständnis.

B. Schmidt vergleicht in [Sch82c] diese fünf Verfahren hinsichtlich Einfachheit, Allgemeinheit und Anpaßbarkeit, und empfiehlt dann eine Kombination aus BMVA und Autoregression.
[Fis78] enthält eine ausführliche, aber schwer verständliche Beschreibung der Autoregressionsmethode sowie (unlesbare) FORTRAN-Listings dazu. Dort finden sich auch Kapitel über die anderen Verfahren.
A.M. Law und J.S. Carson geben in [LC79] einen BMVA - Algorithmus an.
B. Schmeiser untersucht in [Sch82a] für das BMVA-Verfahren den Einfluß von Batchgröße und Batchanzahl (bei vorgegebener Stichprobenanzahl) auf Breite und Überdeckungsgrad des Konfidenzintervalls.
N.R. Adam vergleicht in [Ada83] die BMVA-Algorithmen von Fishman und Law / Carson und entwickelt daraus ein modifiziertes Verfahren. Dieses Verfahren ist das Vorbild für die BMVA-Implementierung in simuLan.

3.5.1 Das BMVA-Verfahren in simuLan

Von obigen Überlegungen ausgehend, wurde in simuLan ein BMVA-Verfahren mit folgenden Eigenschaften implementiert :

- Die zu erzielende Genauigkeit (relative Breite des Konfidenzintervalls und Konfidenzlevel) werden vom Benutzer vorgegeben.

- Die erforderliche Mindestgröße der Batches wird errechnet.

- Es werden so lange Meßwerte gesammelt, bis die geforderte Genauigkeit erreicht ist.
- Geringer Platzbedarf für die Stichproben, da nur die Mittelwerte der Batches gespeichert werden brauchen. Wenn die maximale Anzahl Batches erreicht ist, werden die Mittelwerte benachbarter Batches paarweise zusammengefaßt ([Sch82c], Abschnitt 3).

Voraussetzung für die Anwendbarkeit der BMVA ist die Ergodizität des Prozesses (mit wachsender Anzahl der Meßwerte geht die Varianz gegen Null) [Fis78]. simulan kann **nicht** überprüfen, ob diese Voraussetzung erfüllt ist.
Es wird angenommen, daß das System nach einer Einschwingphase in einen stabilen Zustand übergeht. Die Länge dieser Einschwingphase kann mit dem BMVA-Verfahren nicht ermittelt werden, stattdessen gibt der Benutzer vor, daß die ersten *delete_count* Meßwerte nicht berücksichtigt werden sollen. Die Wahl eines zu kleinen *delete_count* macht sich durch einen deutlich erhöhten Bedarf an Meßwerten für das Erreichen der angestrebten Vertrauensgrenzen bemerkbar.

3.5.2 Der Ablauf des implementierten BMVA-Verfahrens

Das implementierte BMVA-Verfahren ist in drei Abschnitte unterteilt :

1. Übergehen der Einschwingphase – eine vorgegebene Anzahl von Meßwerten wird nicht berücksichtigt.
2. Ermitteln der minimalen Batchgröße, für die die Autokorrelation der Batchmittelwerte insignifikant wird. Die Batchmittelwerte werden dann als statistisch unabhängig betrachtet.
3. Solange weitere Meßwerte in Batches sammeln, bis das geforderte relative Konfidenzintervall bei vorgegebenem Überdeckungsgrad erreicht ist.

[Sch82a] zeigt für feste Stichprobenanzahlen auf, daß mit steigender Batchanzahl das Konfidenzintervall enger wird. Andererseits sinkt mit steigender Batchanzahl die Wahrscheinlichkeit, daß das Konfidenzintervall den wahren Mittelwert enthält. Aus den dort angestellten Untersuchungen folgt, daß Batchanzahlen zwischen 10 und 30 in den meisten Fällen angemessen sind, da die gesuchte Information größtenteils schon in den ersten 10 Batches enthalten ist, und mehr als 30 Batches nur noch eine geringe Steigerung der Genauigkeit bewirken.

Daraus ergibt sich nach der in [Ada83] beschriebenen "B-Method" folgender Algorithmus : (Schritt 9 und 10 der "B-Method" wurde nicht implementiert.)

0.) Vom Benutzer werden vorgegeben :

`delete_count` : Anzahl der in der Einschwingphase zu übergehenden Meßwerte (default 5000).

`batch_size` : Startwert für die Batchgröße (default 400).

`batches_needed` : Anzahl der Batches, die zunächst gesammelt werden, bevor die Autokorrelation der Batchmittelwerte auf verschwindende Signifikanz getestet wird (default 40).

`max_batches` : Anzahl der Batchmittelwerte, die maximal gespeichert werden sollen. Wenn diese Anzahl überschritten wird, werden jeweils benachbarte Werte paarweise zusammengefaßt (default 100).

`max_lags` : Anzahl der Batchmittelwerte, die für die Autokorrelationstests benutzt werden (default 10) ([Ada83] empfiehlt 10% bis 25% von `batches_needed`.)

`req_confidence_level` : Der angestrebte Konfidenzlevel (default 0.9).

`req_rel_confidence_intervall_half` : Die angestrebte relative halbe Breite des Konfidenzintervalls (default 0.05 == 5%).

1.) Die ersten `delete_count` Meßwerte werden ignoriert, um die Anfangsstörungen auszublenden.

2.a.) Die Anzahl `batch_count` bisher gesammelter Werte wird auf 0 gesetzt.

2.b.) Es werden `batches_needed` − `batch_count` Batches der Größe `batch_size` gesammelt.

2.c.) Es wird geprüft, ob die Autokorrelation der Batchmittelwerte signifikant ist.

$$overall\_mean = \frac{\sum_{i=1}^{batch\_count} batch\_mean_i}{batch\_count}$$

$$batch\_mean\_variance = \frac{\sum_{i=1}^{batch\_count} (batch\_mean_i)^2 - batch\_count \cdot overall\_mean^2}{batch\_count - 1}$$

(Gleichung (5) in [Ada83])

$$lag_p = \frac{\sum_{i=1}^{batch\_count-p} (batch\_mean_i - overall\_mean)(batch\_mean_{i+p} - overall\_mean)}{batch\_count \cdot batch\_mean\_variance}$$

$(p = 1 \ldots max\_lags)$

$$lags\_variance_p = \frac{1}{batch\_count}\left(1 + 2\sum_{i=1}^{p-1} lag_p^2\right)$$ (Gleichung (7) in [Ada83])

Die Autokorrelation ist insignifikant, wenn für alle Autokorrelationen lag_p und alle Varianzen $lags\_variance_p$ $(p = 1 \ldots max\_lags)$ gilt

$|lag_p| < z\_normal\left(\sqrt{lags\_variance_p}\right)$ (Gleichung (6) in [Ada83])

($z\_normal\,(p)$ ist das obere p-Quantil der Standardnormalverteilung.)

Ja : Die `batch_size` ist so groß, daß die Batchmittelwerte als statistisch unabhängig betrachtet werden können.
⇒ **Schritt 3**

Nein : `batch_size` wird verdoppelt, `batch_count` halbiert (`batches_needed` muß gerade sein) und die Batches zusammengeschoben, indem jeweils die Mittelwerte zweier benachbarter Batches zusammengefaßt werden.
⇒ **Schritt 2.b.**

3.a.) Es wird ein weiterer Batch von `batch_size` Meßwerten gesammelt.

3.b.) Es wird geprüft, ob das mit diesem Batch erreichte Konfidenzintervall bei vorgegebenem Konfidenzlevel die vorgegebene Breite des Konfidenzintervalls erreicht.

$$confidence\_intervall\_half = \frac{t\_normal\left(\frac{1-req\_confidence\_level}{2}, batch\_count - 1\right) \cdot \sqrt{|batch\_mean\_variance|}}{batch\_count}$$

($t\_normal(p)$ ist das obere p-Quantil der Student-t-Verteilung.)

Das erreichte Konfidenzintervall ist eng genug, falls

$$\frac{confidence\_intervall\_half}{overall\_mean} < req\_rel\_confidence\_intervall\_half$$

(Gleichung (10) in [Ada83])

Ja : **Fertig**

Nein : Es müssen noch mehr Batches gesammelt werden.
Falls `batch_count == max_batches` ist, werden die Batchmittelwerte zusammengeschoben (wie in 2.c.)
⇒ **Schritt 3.a.**

3.6 Dynamische Aggregierung

Eine Erweiterung des Konzepts der dynamischen Simulationslängenkontrolle ist die *dynamische Aggregierung*. Dabei werden für die Kenngrößen einer komplexen Modellkomponente laufend die empirischen Verteilungsfunktionen bestimmt und geprüft, ob sie gegen eine Grenzverteilung konvergieren. Wenn das der Fall ist, braucht die Modellkomponente nicht mehr explizit simuliert werden, sondern es können die ermittelten Grenzverteilungen eingesetzt werden. Dadurch kann sich eine erhebliche Verkürzung der Laufzeit ergeben.

Beispiel :

Mehrere Rechner sind durch ein CSMA/CD- Netz verbunden. Die mittlere Paketverzögerung in Abhängigkeit von der Paketlänge in der MAC-Schicht unter der angegebenen Lastanforderung erreicht (wenn überhaupt) eher eine Grenzverteilung als die Kenngrößen eines bestimmten Knotens, da die Ereignisse für einen einzelnen Knoten wesentlich seltener auftreten als die für das verbindende Netz. Das komplexe CSMA/CD - Modell kann also verhältnismäßig früh durch die Verteilungsfunktion der Paketverzögerung ersetzt werden, während die Netzknoten weiter simuliert werden.

Dieses Konzept wurde in simulAn (noch) nicht realisiert, um den Rahmen dieser Arbeit nicht zu sprengen. Es wurde jedoch bei der Implementation der Instrumentierung darauf geachtet, diese Möglichkeit offen zu halten.

Kapitel 4

Modellieren mit simuLan

Ein mit simuLan darstellbares Rechnernetz besteht aus einzelnen Netzknoten, die durch ein einheitliches Netz verbunden sind.

4.1 Darstellbare Topologien

In simuLan wird für jeden Netzknoten eine eigene Knotenbeschreibung erstellt, aber nur *eine* Netzbeschreibung für das verbindende Netz. Damit lassen sich u.a. Bus- , Ring- , Stern- und Point-to-Point- (vollvermaschte) Topologien modellieren. Daher ist simuLan vor allem zur Simulation lokaler Netze geeignet.

4.2 Protokollschichten in simuLan

simuLan unterstützt die Modellierung von Protokollschichten. Es können (derzeit) bis zu fünf Schichten simuliert werden. Dabei entsprechen die OSI - Schichten 1 und 2a, d.h. Physical Layer und Medium Access Layer [IEE88a], der simuLan - Schicht 1, die das verbindende Netz modelliert.

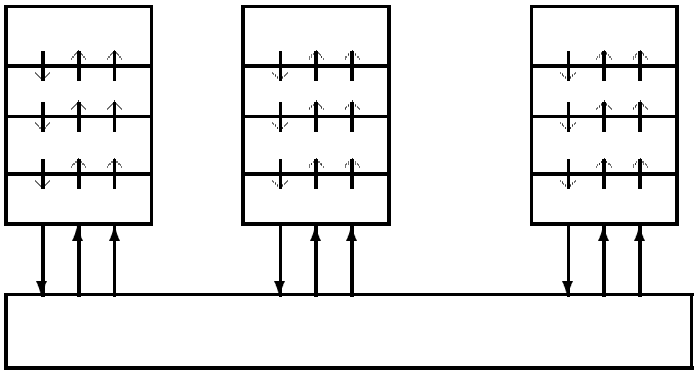


Abbildung 4.1: Aufteilung der Knoten und des verbindenden Netzes in Protokollschichten Von diesem Netzmodul existiert in einem modellierten Rechnernetz genau eine Instanz. Von den darüberliegenden Protokollschichten dagegen kann in jedem Netzknoten eine Instanz existieren.

Folgende Entsprechung zwischen dem OSI/ISO-Modell und dem s|m$_u$|L^{a}n-Modell ist vorgesehen :

```
s|mu|La n-Schicht 1 ↔ OSI-Schicht 1, 2a (Physical und
                                         Medium Access Control Layer)
                  2 ↔              2b    (Logical Link Control)
                  3 ↔              3     (Network Layer)
                  4 ↔              4     (Transport Layer)
                  5 ↔              5,6,7 (Session-, Application-,
                                         Presentation Layer)
```

Die oberste s|m$_u$|L^{a}n - Schicht 5 (auch als "*Knotenmodell*" bezeichnet) modelliert die lasterzeugende Anwendung (die Vorgänge "im Inneren des Knotens"). Eine weitere Aufteilung dieser Schicht erschien für die bisher implementierten Modelle nicht sinnvoll, könnte aber leicht vorgenommen werden.

Die s|m$_u$|L^{a}n - Schicht 2 soll sicherstellen, daß von jedem Knoten jeweils nur ein Sendeauftrag gleichzeitig an das Netzmodul übergeben wird.

Die s|m$_u$|L^{a}n - Schicht 1 (das Netzmodell) regelt den konkurrierenden Zugriff der Knoten auf das Übertragungsmedium (bzw. die Medien).

Diese Entsprechungen sind nicht in das Grundgerüst von s|m$_u$|L^{a}n einprogrammiert, aber alle Modellkomponenten sollten sich daran halten, damit sie frei austauschbar sind.

Die s|m$_u$|L^{a}n - Schicht N bietet der Schicht $N+1$ ($N = 1 \ldots 4$) drei Dienste an :

- Sendeauftrag : Eine $N+1$-PDU wird von Schicht $N+1$ angenommen, um sie an Schicht N im Empfängerknoten weiterzuleiten.
- Empfangsbenachrichtigung : Schicht N übergibt eine vollständig und korrekt empfangene $N+1$-PDU an Schicht $N+1$.
- Sendebenachrichtigung : Schicht N benachrichtigt Schicht $N+1$, daß ein zuvor abgegebener Sendeauftrag ausgeführt wurde. Falls Schicht N feststellen konnte, daß die Übertragung fehlerhaft war, oder der Sendeauftrag gar nicht durchgeführt werden konnte, so wird dies der Schicht $N+1$ ebenfalls mitgeteilt.

Eine $N+1$-PDU wird für Schicht N durch ihre Länge, den Sender, den Empfänger sowie einen Zeiger auf den Inhalt, der von Schicht N nicht interpretiert wird, beschrieben.

4.3 Baukastenprinzip

Für jede Protokollschicht werden verschiedene *Protokollschichtmodelle* implementiert, für Protokollschicht 1 beispielsweise CSMA/CD-, Token Ring- oder Point-to-Point- Modelle, für Protokollschicht 3 ein Modell des IP-Protokolls, für Protokollschicht 4 TCP und UDP, für Protokollschicht 5 verschiedene Modelle lasterzeugender Anwendungen.

In der Modellbeschreibung werden die Netzknoten aus Protokollschichtmodellen zusammengesetzt, die durch Parameter näher beschrieben werden. Auch für Schicht 1 (das verbindende Netz) wird ein Modell ausgewählt. Zu den Parametern der Protokollschichtmodelle gehören auch Angaben über die Instrumentierung.

In jedem Knoten kann für jede Protokollschicht kein oder ein Modell ausgewählt werden. Wenn in der Konfigurationsdatei für eine der Schichten 2, 3 oder 4 kein Modell angegeben wird, setzt die darüberliegende Schicht direkt auf der darunterliegenden auf.

Falls ein Protokoll oder eine Anwendung nicht mit einem der implementierten Protokollschichtmodelle dargestellt werden kann, bietet die Grundbibliothek von simuLAN dem "fortgeschrittenen" Benutzer die Möglichkeit, neue Modellkomponenten ("Bauelemente") mit wenig Aufwand zu erstellen. Um solche Erweiterungen sichtbar zu machen, kann das Programm eine Auflistung und Beschreibung aller aktuell implementierten Modellkomponenten ausgeben.

simuLAN ist nur sehr begrenzt fähig, zu überprüfen, ob die ausgewählte Kombination von Protokollschichtmodulen funktionsfähig oder gar sinnvoll ist.

In den nächsten beiden Kapiteln werden der Aufbau einer Modellbeschreibung und die Bedienung des Simulationsprogramms erläutert. Anschließend werden die Konzepte für die Programmierung neuer Modellkomponenten dargestellt und dann an Beispielen verdeutlicht.

Kapitel 5

Aufbau einer Konfigurationsdatei

simulan liest die Beschreibung des zu simulierenden Modells aus der *Konfigurationsdatei*.

Da sich die Syntax der Konfiguration mit jedem neu implementierten Protokollschichtmodell und jedem neu definierten Meßpunkt (aufwärtskompatibel) erweitert, wird hier auf eine vollständige Beschreibung der Syntax und Semantik verzichtet. Stattdessen ist simulan in der Lage, eine Schablone für eine Konfigurationsdatei mit allen verfügbaren Sprachelementen auszugeben, aus der mit einem Texteditor die gewünschte Modellbeschreibung aufgebaut werden kann (siehe Abschnitt 6 [*Bedienung von* simulan], Seite 33).

Die Konfigurationsdatei besteht aus einem Vorspann für die Definition von Ausgabeumlenkungen und Laufvariablen, einem Hauptteil für die Definition der Protokollschichten und einem kurzen Abschluß.

Ein Beispiel für eine solche Konfigurationsdatei :

```
#! simulan

## -- Vorspann --

FILE delays     "/tmp/delay-mess-spur"
FILE bmva-out   "./bmva-results"

# eine Laufvariable für die Anzahl der simulierten Knoten
FOR int k FROM 5 TO 25 STEP 5

# jeder Simulationslauf wird mit 3 verschiedenen random seeds
# wiederholt.
FOR int r FROM 1 TO 173847 STEP 81930

## --- Hauptteil ---

BEGINconf

## globale Simulations-Parameter
ITU             1e9             # Indivisible Time Units per second
random_seed     $r
max_events      1e8
max_simul_time  2e15
max_q_length    1000
max_cpu_time    200000

# Jede Sekunde (realer Zeit) soll die erreichte simul_time auf den Bildschirm
# ausgegeben werden. Die Steuerzeichen sind für vt100-artige Terminals.
itimer_intervall 1.0    # Sekunden
itimer_probept
        MEASURE NIL     null_fkt
        FORMAT  stderr  " \e[5;5Hs imul_time %t% "
```

```
NET
    CSMA
        bits_per_second 10e6

        # jede Sekunde tritt eine Übertragungsstörung auf
        error_distr     expntl 1 s

        # ein Kabelsegment von 500 m Länge, alle Stationen
        # gleichmäßig verteilt
        calc_laufzeit gleichvert 500 m

        # Meßpunkt am Ende einer Übertragung
        end_transmit_probept
                MEASURE NIL frame_delay
                FORMAT  delays "%t% \t%M% \n" # "Uhrzeit tab Framedelay"
                STATISTIC STOP bmva-out
                bmva
                        delete_count    2000
                ENDbmva
     ENDcsma
ENDnet

NODES
        # k identische Knoten vom Typ "k0"
        NAME "Knoten %d" COUNT $k
        LAYER5 k0

                # exakt alle 100 Mikrosekunden ein Sendwunsch
                zw_ank_distr konstant 100 us

                mes_size_distr konstant 128 byte

                # alle Knoten sind als Empfänger gleich wahrscheinlich
                calc_empfaenger uniform_empf
        ENDk0

        # die Schichten 3 und 4 bleiben leer

        # Schicht 2 serialisiert den Zugang der Knoten zur Schicht 1
        # (primitiver Controller)
        LAYER2 l2simple

                # nach jedem Senden oder Empfangen eine kurze "Totzeit"
                finish_time_needed 10 ms
        ENDl2simple

        # keine weiteren Knoten mehr

ENDconf # Ende des Hauptteils

## -- Abschluß --
END     # Ende der Konfigurationsdatei
```

5.1 Notation

In den nachfolgenden Syntaxbeschreibungen wird folgende Notation verwendet :

'`[ TEXT ]`' : Textstücke zwischen eckigen Klammern sind optional.

'`[...]`' : Das vorstehende Textstück kann optional wiederholt werden.

'`<name>`' : Textstücke zwischen spitzen Klammern bezeichnen einen symbolischen Namen, für den in der Konfigurationsdatei ein Wert eingesetzt werden muß (Strings, Zahlen oder Paramterlisten für Modellkomponenten).

'`{ a|b|c }`' : Eine der Alternativen ist auszuwählen.

Andere Textstücke werden so eingegeben wie sie angegeben sind.

5.2 Zerlegung der Konfigurationsdatei in Worte

Die Konfigurationsdatei wird in Worte zerlegt, die durch *Zwischenraum* (White Space) getrennt sind (Leerzeichen, Tabulator, Zeilenvorschub, Wagenrücklauf, Seitenvorschub).

Kommentare beginnen mit einem **#** (Hash, Gartenzaun) an einem Wortanfang. Danach wird alles bis zum nächsten Zeilenende überlesen.

Worte, die einen Zwischenraum enthalten, können zwischen " (Doublequotes, Gänsefüßchen) gesetzt werden. Doublequotes wirken *nicht* als Wortbegrenzung und sind nicht Bestandteil des Wortes.

Der **** (Backslash, Rückschräger) gibt dem nächsten Zeichen eine besondere Bedeutung. Folgende spezielle Zeichenfolgen sind definiert:

```
\a    Bell
\b    Backspace
\e    Escape
\f    Formfeed
\n    Newline
\r    Carriage Return
\t    Tabulator
\v    Vertical Tabulator
\ddd  Eine 1-3 stellige Oktalzahl, deren Wert als ASCII-Zeichen
      interpretiert wird.
\ \n   (Backslash am Zeilenende) : Das Zeilenende sowie
      sämtlicher White Space am Beginn der nächsten Zeile
      werden überlesen. Dadurch kann ein Wort über mehrere
      Eingabezeilen ausgedehnt werden.
```

Alle anderen nach einem Backslash auftretenden Zeichen werden unverändert in das Eingabewort übernommen. Insbesondere verlieren White Space, Doublequotes, Backslash und das Kommentarzeichen ihre spezielle Bedeutung.

5.3 Der Vorspann

Der Vorspann der Konfiguration wird von simuLan genau einmal zu Beginn eines Programmlaufes gelesen. Er enthält Angaben über Ausgabeumlenkungen und Variation von Parame-

tern (Laufvariablen).

5.3.1 Definition von Ausgabekanälen

```
[ FILE <file-identifier> <filename>
  [...]
]
```

Mit '`FILE`' wird einem logischen "Ausgabekanal" <*file-identifier*> eine Datei des Betriebssytems zugeordnet. <*filename*> kann der absolute oder relative Pfadname einer Datei sein. Wenn <*filename*> mit `>>` (Größer Größer) beginnt, wird an den vorhandenen Dateiinhalt angefügt, sonst wird der bereits vorhandene Dateiinhalt überschrieben.
Falls <*filename*> mit einem `|` (Senkrechter Strich, Pipe-Symbol) beginnt, wird der Rest von <*filename*> als eine Kommandozeile für ein Nachbereitungsprogramm interpretiert und mittels `popen(3)` eine Pipe zu einem Subprozeß eröffnet.

Folgende <*file-identifier*> sind vordefiniert :

```
NIL     das NULL-File, die Ausgaben werden unterdrückt
        (nicht zu verwechseln mit /dev/null).
stdout  Die Standard-Ausgabe.
stderr  Der Standard-Error-Kanal.
outfile Das in der Kommandozeile angegebene Ausgabefile
        (Option -o), oder identisch mit stdout.
```

Äquivalent zur '`FILE`' Definition ist die Kommandozeilenoption `-f` (siehe Abschnitt 6 [*Bedienung von* simuLAN], Seite 33).

5.3.2 Definition von Laufvariablen

```
[ FOR { int | ulong | float | double } <variablenname>
      FROM <start> TO <end> [ STEP <step> ]
  [...]
]
```

Mit '`FOR`' wird eine Laufvariable und die Werte, die die Variable annehmen soll, definiert. Für jede Wertekombination aller definierte Laufvariablen wird ein Simulationslauf durchgeführt, analog zur Abarbeitung verschachtelter FOR-Schleifen in Programmiersprachen. *Die Laufvariablen können erst im Hauptteil der Konfiguration benutzt werden.* Anstelle einer numerischen Konstanten kann dort '`$<variablenname>`' geschrieben werden.

In der '`FOR`' - Definition wird der Datentyp der Laufvariablen angegeben, entsprechend den Datentypen in C. Da numerische Konstanten grundsätzlich zunächst als Fließpunktzahl (Double) eingelesen werden, können auch ganzzahlige Laufvariablen mit Gleitkomma eingegeben werden (also z.B. '`1e6`' statt '`1000000`').
Falls '`STEP`' nicht angegeben ist, wird als Schrittweite 1 angenommen.

5.4 Der Hauptteil der Konfiguration

```
BEGINconf

[ <global-parameters> ]

[ <node-descriptions> ]

[ <net-description> ]

ENDconf
```

Der Hauptteil wird für jeden Simulationslauf erneut gelesen. Dabei werden die den angegebenen Modellkomponenten entsprechenden Datenstrukturen aufgebaut.

Er enthält Angaben über globale Parameter für diesen Simulationslauf, und über die zu verwendenden Protokollschichtmodelle für die Knoten und das verbindende Netz. Zu jedem Protokollschichtmodell können Angaben über die Bestückung der Meßpunkte und die statistische Absicherung der Meßwerte gemacht werden.

Der Hauppteil wird mit '`BEGINconf`' eingeleitet und mit '`ENDconf`' abgeschlossen.

5.4.1 Name - Wert - Paare

Die Modellparameter werden im allgemeinen in der Form

```
<Parametername> <Parameterwert>
```

angegeben, wobei die Parameter für ein Modul durch Schlüsselwörter für die Art des Moduls und eine Ende-Kennzeichnung eingerahmt werden :

```
<Protokollschicht> <Modulname>
        <Parametername> <Parameterwert>
        ...
        END<Modulname>
```

Die Parameter für ein Modul können im allgemeinen in beliebiger Reihenfolge und beliebig oft angegeben werden. Für den Simulationslauf wird jeweils der als letztes für einen Parameter angegebene Wert benutzt (siehe Abschnitt A.15.10 [`rcreadkeywds()`], Seite 93).

Wenn der Parameterwert eine Zeit, Länge oder Byteanzahl darstellt (in der Schablone für die Konfigurationsdatei las <*TIME_T*>, <*meters*> bzw. <*bits*> angegeben), kann als nächstes Wort eine Einheitenangabe folgen.

Als Zeiteinheiten werden verstanden : '`min`', '`s`', '`sec`', '`ms`', '`us`', '`ns`', '`itu`' (Default ist '`sec`').

Als Längeneinheiten werden verstanden : '`km`', '`m`', '`meter`', '`dm`', '`cm`', '`mm`', '`um`', '`nm`' (Default ist '`meter`').

Als Nachrichtenlängen werden verstanden : '`MB`', '`MByte`', '`Mb`', '`Mbit`', '`KB`', '`KByte`', '`Kb`', '`Kbit`', '`B`', '`Byte`', '`byte`', '`b`', '`Bit`', '`bit`' (Defualt ist '`bit`').

5.4.2 Globale Simulationsparameter

```
# <global-parameters>

[ ITU           <double>        # indivisible time-units per second ]
[ max_events    <ulong>         # max. number of events to simulate ]
[ max_simul_time <TIME_T>       # max time to simulate ]
[ max_cpu_time  <double>        # max cycles to succ ]
[ max_q_length  <int>           # max. allowed queuelength ]
[ random_seed   <int>           # seed for srand() ]

[ cronic_intervall <TIME_T>     # intervall for 'cronic' dummy-event ]
[ cronic_probept                # schedule cronic dummy-event
        <parameter-for-cronic_probept> ]

[ itimer_intervall <double>     # intervall for itimer, in seconds ]
[ itimer_probept                # output some data every x seconds
        <parameter-for-itimer_probept> ]
```

Mit '`ITU`' wird die Zahl der Indivisible Time Units angegegeben, in die eine Sekunde unterteilt wird. Intern werden alle Zeiten in ITUs berechnet. Da alle folgenden Zeitangaben aus der Konfiguration in ITU umgerechnet werden, sollte die ITU-Angabe nur einmal am Anfang des Hauppteiles stehen.

Mit '`max_events`' wird angegeben, wieviele Ereignisse maximal bearbeitet werden sollen. Wenn dieser Wert überschritten wird, endet der Simulationslauf und die bis dahin gesammelten Ergebnisse werden ausgewertet.

Mit '`max_simul_time`' wird die Länge des zu simulierenden Zeitintervalls vorgegeben (in ITU). Wenn dieser Wert überschritten wird, endet der Simulationslauf und die bis dahin gesammelten Ergebnisse werden ausgewertet.

Mit '`max_cpu_time`' wird die maximale Rechenzeit angegeben, die s$_{\text{j}}$m$_{\text{u}}$L^{a}n für den Programmlauf verbrauchen darf (in CPU-Sekunden). Wenn dieser Wert überschritten wird, endet der Simulationslauf und die bis dahin gesammelten Ergebnisse werden ausgewertet.

Mit '`max_q_length`' wird die maximal zulässige Länge für die Warteschlangen angegeben. Wenn dieser Wert überschritten wird, endet der Simulationslauf und die bis dahin gesammelten Ergebnisse werden ausgewertet (siehe Abschnitt A.8 [Warteschlangen], Seite 78).

Mit '`random_seed`' wird der Startwert für den Pseudozufallsgenerator angegeben. Falls `-1` angegeben ist, wird die von `time(2)` gelieferte Zeit als Startwert benutzt.

Der '`cronic_probept`' bietet die Möglichkeit, Ausgaben in regelmäßigen Abständen der *simulierten* Zeit anzustoßen. Dafür wird periodisch ein spezielles Meß-Ereignis ausgelöst (*cronic event*). Mit '`cronic_intervall`' wird das Intervall dafür in ITU angegeben (siehe Abschnitt 5.4.5 [Beschreibung der Instrumentierung], Seite 29).

Der '`itimer_probept`' ist ein Meßpunkt, dessen Meßfunktionen in regelmäßigen Abständen realer Zeit aufgerufen werden. Mit '`itimer_intervall`' wird das Intervall (in Mikrosekunden) für regelmäßig ausgelöste Interrupts angegeben, mittels derer diese Messungen angestoßen werden (siehe `setitimer(2)`).

Falls für einen dieser Parameter keine Angaben gemacht werden, wird ein Defaultwert eingesetzt. Die Defaultwerte werden ausgegeben, wenn s$_{\text{j}}$m$_{\text{u}}$L^{a}n mit der Option '`-p`' aufgerufen wird (siehe Abschnitt 6 [*Bedienung von* s$_{\text{j}}$m$_{\text{u}}$L^{a}n], Seite 33).

5.4.3 Beschreibung der Netzknoten (Schicht 2 – 5)

```
[ NODES
        [
          [ NAME <node-name> ]  [ COUNT <count> ]
          LAYER5 <l5-module>
                 <layer5-params-for-this-node>
          END<l5-module>

          [ LAYER4 <l4-module>
                <layer4-params-for-this-node>
            END<l4-module>
          ]

          [ LAYER3 <l3-module>
                <layer3-params-for-this-node>
            END<l3-module>
          ]

          [ LAYER2 <l2-module>
                <layer2-params-for-this-node>
            END<l2-module>
          ]
        ]

        [..]                    # more nodes, indicated by 'LAYER5'
]
```

Die Beschreibung der Netzknoten wird mit '`NODES`' eingeleitet. Für jeden Netzknoten muß zumindest mit '`LAYER5`' ein Modell für die oberste Schicht 5 angegeben werden. Das Einlesen der Knotenbeschreibungen wird beendet, wenn keine '`LAYER5`'- Angabe mehr gefunden wird. Wenn für eine der Protokollschichten 2, 3, 4 keine Angabe gemacht wird, setzt die darüberliegende Schicht direkt auf der darunterliegenden auf. simuLAn *kann nicht überprüfen, ob das sinnvoll ist.*

Mit '`COUNT`' kann angegeben werden, daß mehrere identische Knoten entsprechend der folgenden Parameter angelegt werden sollen. Wenn '`COUNT`' nicht angegeben wird, wird 1 angenommen.

Die konfigurierten Knoten werden intern in der Reihenfolge ihrer Definition durchnumeriert. Mit '`NAME`' kann den nachfolgend beschriebenen Knoten ein *Knotenname* gegeben werden. Falls der Name die Zeichenfolge "*%d*" enthält, wird dafür mit `sprintf(3)` die Nummer des betreffenden Knotens eingesetzt.

Es ist vorgesehen, daß zu einem Knotenmodell (Schicht 5) eine Liste von Partnerknoten angegeben werden kann, mit denen ein Knoten kommuniziert. Ob diese Möglichkeit benutzt wird, hängt allerdings vom ausgewählten Modell ab.

```
traffic
       [ partner <node-name> <probability>
         [...]
       ]
```

Ein Element einer solchen Liste wird angegeben durch das Wort '`partner`' gefolgt von einem Empfänger-Knotennamen und einer Wahrscheinlichkeit $0 \leq <probability> \leq 1$, mit

der eine Nachricht des Senders an diesen Empfänger gesendet wird. Die Summe dieser Wahrscheinlichkeiten muß kleiner oder gleich 1.0 sein.
Wenn in der Liste ein Knotenname angegeben wird, der mit einem * (Stern, Asterisk) endet, so wird für jeden Knoten, dessen Namen bis auf das Ende mit dem angegebenen übereinstimmt (`strncmp(3)`), ein Partner - Eintrag generiert (siehe Abschnitt A.10 [Verkehrsmatrizen], Seite 81). Das ist nützlich, wenn mittels '`COUNT`' mehrere identische Knoten mit Namen wie z.B. '`Node 42`', '`Node 43`', '`Node 44`' definiert werden.

Nach dem Einlesen aller Knotenbeschreibungen werden in den Partnerlisten die Knotennamen in Knotennummern umgesetzt.

5.4.4 Beschreibung des Netzmodells (Schicht 1)

```
[ NET
        [ <net-name>
                <parameters-for-net-modul>
          END<net-name>
        ]
  ENDnet
]
```

Die Beschreibung des für Schicht 1 zu verwendenden Netzmodells wird mit '`NET`' eingeleitet und mit '`ENDnet`' abgeschlossen. Für <*net-name*> muß der Name eines implementierten Netzmoduls eingesetzt werden. Die implementierten Netzmodule und ihre spezifischen Parameter werden ausgegeben, wenn simulan mit der Option '`-x`' aufgerufen wird (siehe Abschnitt 6 [*Bedienung von* simulan], Seite 33).

Einige Parameter sind für jedes Netzmodul erforderlich :

Mit '`bits_per_second`' wird die nominelle Übertragungsgeschwindigkeit der physikalischen Verbindung angegeben.

Mit '`min_data_len`' und '`max_data_len`' wird die minimale bzw. maximale Größe einer Nachricht angegeben, die von der Medium Access Layer übertragen werden kann.

Die Netztopologie ist implizit im ausgewählten Netzmodell enthalten. Zu den Parametern des Netzes können Angaben über Entfernungen zwischen den Knoten, über Signallaufzeiten oder über Nachbarschaftsbeziehungen gehören.

Falls für einen Parameter keine Angaben gemacht werden, wird ein Defaultwert eingesetzt. Die Defaultwerte werden ausgegben, wenn simulan mit der Option '`-p`' aufgerufen wird (siehe Abschnitt 6 [*Bedienung von* simulan], Seite 33).

5.4.5 Beschreibung der Instrumentierung

```
<probept-name>

        [ MEASURE <file-ident1> <probe-fkt>
          [ FORMAT <file-ident2> <format1> ]
          [ <STATISTIC> ]
        ]

        [..]
```

Zu den Parametern der Protokollschichtmodelle gehören die Meßpunkte (*Probepoints*. Eine an einem Meßpunkt durchzuführende Messung wird mit '`MEASURE`' definiert. <*probe-fkt*> gibt die Meßfunktion an. Dadurch wird bestimmt, was für Meßwerte gewonnen werden sollen. <*file-ident1*> ist der Name eines "Ausgabekanals" (siehe Abschnitt 5.3.1 [Definition von Ausgabekanälen], Seite 25) für die Ausgaben der Meßfunktion. Das Format dieser Ausgaben ist in die Meßfunktion einprogrammiert.

Mit '`FORMAT`' kann ein zusätzlicher Ausgabekanal angegeben werden, in den Ausgaben entsprechend dem angegebenen <*format1*> geschrieben werden.

Der Aufbau der Format-Strings ist an die Formatangaben von `printf(3)` angelehnt. Ein Format-String enthält :

- gewöhnliche Zeichen, die einfach in die Ausgabe kopiert werden,
- Konvertierungs-Angaben.
  Diese beginnen mit einem % (Prozent), danach ein Buchstabe, der angibt, *was* ausgegeben werden soll, optional gefolgt von einer Konvertierungsangabe für `printf(3)`. Den Abschluß bildet ein zweites %.
  Also : '`%`<*letter*>`[`<`printf(3)-Formatangabe`>`]`%`'
  Wenn simula n mit der Option '`-x`*n*', *n*>1, aufgerufen wird, wird für jeden Meßpunkt angegeben, welche Buchstaben für '<*letter*>' eingesetzt werden können, und was sie bedeuten.

  Einige für alle Meßpunkte verfügbare '<*letter*>' - Angaben sind :

```
 t : aktuelle Simulationszeit in ITU  (default-Format '.0f')
 T : aktuelle Simulationszeit in Sekunden  ('g')
 M : der aktuelle Meßwert  ('g')
 O : der vorhergehende Meßwert  ('g')
 u : die Zeit des vorhergehenden Meßwertes in ITU  ('.0f')
 U : die Zeit des vorhergehenden Meßwertes in Sekunden  ('g')
 n : Nummer des aktuellen Knotens  ('d')
 N : Name des aktuellen Knotens  ('s')
 c : Anzahl der simulierten Netzknoten  ('d')
 S : das ermittelte Ergebnis der statistischen Absicherung  ('g')
$x : der Wert der Laufvariablen x, als Double  ('g')
```

Für jeden Meßpunkt kann eine spezielle Formatierungsfunktion geschrieben werden, die weitere Möglichkeiten bietet.

Zu jeder Messung kann eine statistische Absicherung der Meßwerte konfiguriert werden, um die Länge der Simulationsläufe zu kontrollieren. (siehe Abschnitt 5.4.6 [Beschreibung der statistischen Absicherung], Seite 31, und Abschnitt 3.4 [Simulationslänge], Seite 12).

Durch mehrmalige Angabe von '`MEASURE`'- Definitionen können an einem Meßpunkt mehrere Messungen konfiguriert werden.

Zu Beginn jedes Simulationslaufes wird nach <*file-ident1*> eine Überschrift ausgegeben, die die Bedeutung der Meßwerte angibt.

Während des Simulationslaufes werden die Meßfunktionen aller Messungen eines Meßpunktes jedesmal dann aufgerufen, wenn an dem Meßpunkt etwas "Interessantes" passiert. Bei jedem Aufruf gibt die Meßfunktion den gemessenen Wert in einem fest einprogrammierten Format nach <*file-ident1*> aus.
Falls für die Messung eine statistische Absicherung konfiguriert ist, wird danach die Absicherungsfunktion aufgerufen.

Falls gewünscht, werden anschließend die Ausgaben entsprechend dem <*format1*> nach <*file-ident2*> geschrieben (siehe Abschnitt A.14.2 [*Die* PROBE*-Struktur*], Seite 87, sowie Abschnitt 7.9 [*Implementation von Meßspunkten*], Seite 45).

5.4.6 Beschreibung der statistischen Absicherung

Die Syntax für <*STATISTIC*> ist

```
STATISTIC [ { WAIT | STOP | NOWAIT } ] [ COMMON ] <file-ident3>
        [ FORMAT <file-ident4> <format2> ]
        <stat-modul-name>
            <parameters-for-stat-modul>
        END<stat-modul-name>
```

Mit '`STATISTIC`' wird in der Beschreibung einer Messung angegeben, daß nach jedem Aufruf der Meßfunktion geprüft werden soll, ob bereits genügend Meßwerte in dieser Messung gesammelt wurden. Das am häufigsten benutzte Kriterium dafür ist, daß die vorgegebenen Vertrauensgrenzen für die Mittelwerte der Meßwerte erreicht wurden (siehe Abschnitt 3.5 [Bestimmung von Konfidenzintervallen], Seite 13).

Der nächste Parameter gibt an, was passieren soll, wenn das benutzte Absicherungsmodul zu dem Schluß kommt, daß ausreichend viele Meßwerte gesammelt wurden :

'`WAIT`' : Die Anzahl der nicht-fertigen Messungen wird um eins erniedrigt. Erreicht diese Anzahl den Wert 0, so sind alle interessierenden Messungen fertig und der Simulationslauf wird beendet. Der Effekt ist, daß mit dem Beenden des Simulationslaufes "gewartet" wird, bis für diese Messung genügend Meßwerte gesammelt wurden.

'`NOWAIT`' : Die Vertrauensgrenzen werden ermittelt, haben aber keinen Einfluß auf die Länge des Simulationslaufes.

'`STOP`' : Der Simulationslauf wird beendet, ohne auf andere Messungen zu warten.

Als Defaultwert wird '`WAIT`' angenommen.

Wenn mehrere identische Netzknoten angelegt werden (mit dem Parameter '`COUNT`' in der Knotenbeschreibung), kann mit '`COMMON`' angegeben werden, daß die statistische Absicherung für alle diese Knoten gemeinsam durchgeführt werden soll. Dadurch läßt sich die Länge der Simulationsläufe verringern, da mehr Meßwerte für einen Meßpunkt anfallen. Das ist aber nur sinnvoll, wenn man davon ausgehen kann, daß die Meßwerte der Knoten die gleiche Verteilungsfunktion besitzen.
Wenn '`COMMON`' nicht angegeben wird, werden die Meßwerte der Knoten getrennt behandelt. Für Meßpunkte in der Schicht 1 wird '`COMMON`' stillschweigend ignoriert.

Nach dem Ende jedes Simulationslaufes werden alle konfigurierten Messungen durchgegeangen. Für jede Messung werden die durch die statistische Absicherung errechneten Werte für die Ausgabe aufbereitet (z.B. Umrechnung von ITU in Sekunden) und in einem fest einprogrammierten Format nach <*file-ident3*> ausgegeben. Falls <*format2*> in der Konfiguration angegeben ist, werden anschließend die Ausgaben entsprechend diesem Format in die für <*file-ident4*> angegebene Datei geschrieben.

Die implementierten Module für die statistische Absicherung werden in der Schablone für die Konfigurationsdatei ausgegeben, wenn s$_{j}$m$_{u}$L^{a}n mit der Option '`-x`' aufgerufen wird.

5.5 Der Abschluß der Konfiguration

```
{ AGAIN | END }
```

'`END`' gibt an, daß für jede Wertekombination der definierten Laufvariablen der Hauptteil der Konfiguration erneut gelesen und ein Simulationslauf durchgeführt werden soll. Danach wird der Programmlauf beendet.

'`AGAIN`' gibt an, daß die *gesamte* Konfiguration erneut gelesen werden soll. Das ist nützlich, falls die Konfiguration nicht von der Platte, sondern aus einer Pipe gelesen wird. Dadurch besteht die Möglichkeit, die Konfigurationen durch ein Programm erzeugen zu lassen und direkt an simuLan zu übergeben (z.B. mit dem C-Präprozessor `cpp(1)` oder mit `m4(1)`).

Kapitel 6

Bedienung von simuLAN

6.1 Aufruf

```
usage : simulan [-h] [-T <device>] [-n] [-p] [-x[<n>]]
        [ -D<modulename>[=flags] ] [ [ -f <fileid> <path> ] ... ]
        [-o <outfile>] [<conffile>]
        -h    : print long usage note
        -T    : alternate tty-device, default is /dev/tty
        -n    : read the configuration, but don't run it
        -p    : print actual configuration for each run (very verbose!)
        -x    : explain syntax of configuration file
        -D    : set debugflags for the module
        -f    : like ``FILE <fileid> <path>'' within the configuration
        -o    : use <outfile> instead of stdout for verbose outputs

        <conffile>   : name of the configuration file
                       (default is stdin)
```

Die Option '`-x`*n*' bedeutet, daß keine Simulation durchgeführt sondern eine Erläuterung der Syntax der Konfigurationsdatei ausgegeben werden soll. Wenn *n* größer als eins ist, werden zur Syntax ausführlichere Kommentare ausgegeben (Default ist 1).

Die Option '`-n`' bedeutet, daß die Konfigurationsdatei gelesen, aber kein Simulationslauf durchgeführt werden soll.

Die Option '`-p`' gibt an, daß die aus der Konfiguration gelesenen Parameter einschließlich eventuell eingesetzter Defaultwerte ausgegeben werden sollen. Die Option '`-p`' zusammen mit '`-n`' eignet sich, um die Korrektheit einer Konfigurationsdatei zu überprüfen.

Mit der Option '`-o`' kann angegeben werden, daß statt `stdout` eine andere Ausgabedatei benutzt werden soll.

Mit '`-f <`*file-ident*`> <`*filename*`>`' kann ein "Ausgabekanal" und eine zugehörige Datei definiert werden, wie mit einer '`FILE`'-Definition im Vorspann der Konfiguration.

Mit '`-D<`*modulename*`>=<`*flag*`>`' können Testausgaben einzelner Module aktiviert werden. Die Auswertung dieser Optionen muß von den jeweiligen Modulen selber durchgeführt werden (siehe Abschnitt A.16.1 [Debugflags], Seite 97).

Testausgaben während des Einlesens der Konfiguration können aktiviert werden durch '`-DRC=`*n*'. Je größer die Zahl *n* ist, desto detaillierter werden die Ausgaben.

Das letzte Argument wird als Name der Konfigurationsdatei benutzt. Wenn keine Datei angegeben ist, wird von `stdin` gelesen. Das ist insbesondere dann der Fall, wenn simuLan ganz ohne Parameter aufgerufen wird.

Fehlermeldungen und Debugausgaben werden immer nach `stderr` geschrieben.

6.2 Signale

Wenn simuLan während des Programmlaufes ein Interrupt - Signal (`SIGINT`) erhält (üblicherweise durch Eingabe von `Ctrl-C`), so wird der Simulationslauf unterbrochen und in ein Debug-Menu verzweigt. Dieses ermöglicht interaktiv das Anzeigen und Ändern der Datenstrukturen. Das Menu ist selbsterklärend, darum wird hier nicht weiter darauf eingegangen.

Auf Unix - Systemen, die `getitimer(2)` und `setitimer(2)` unterstützen, kann der Realzeit - Intervallgeber (`ITIMER_REAL`) benutzt werden, um in regelmäßigen Abständen ein Alarm-Signal (`SIGALRM`) zu erzeugen, welches die Abarbeitung der Messungen des globalen Meßpunktes '`itimer_probept`' auslöst.
Der Intervallgeber für die virtuelle Rechenzeit (`ITIMER_VIRTUAL`) wird benutzt, um bei Überschreiten der vorgebenen maximalen CPU-Zeit (Parameter '`max_cpu_time`') den Programmlauf zu beenden.

Bei langen Programmläufen können *Aufsetzpunkte* angelegt werden, von denen aus nach einem Rechnerausfall weitergerechnet werden kann. Ein solcher Aufsetzpunkt besteht aus einer ausführbaren Programmdatei, die im Datenbereich einen Speicherabzug des simuLan-Prozesses enthält. Wenn dieses Programm gestartet wird, rechnet es also von dem Zustand aus weiter, der direkt vor dem Abspeichern erreicht wurde.

Aufsetzpunkte werden angelegt, wenn simuLan ein Terminations - Signal (`SIGTERM`) oder ein Stromausfallsignal (`SIGPWR`) erhält. Dieses *Checkpointing* wurde bisher nur für Pyramid-Architekturen implementiert (Targon/35). Auf SUNs, VAXen, DECstations und Sequent Symmetry Rechnern kann stattdessen das *Condor* - System benutzt werden [BM89], welches langlaufende Prozesse lastausgleichend zwischen Rechnern verteilt, und dafür regelmäßig Aufsetzpunkte anlegt.

6.3 Fehlermeldungen

Die Fehlermeldungen von simuLan sind selbsterklärend. Die meisten Fehlermeldungen sind mit einer Nummer versehen, die bei Bedarf das Auffinden der Stelle im Programmtext erleichtern soll. Fehlermeldungen während des Einlesens der Konfigurationsdatei beginnen mit dem Namen der Datei und einer ungefähren Zeilennummer, so daß die Meldungen von der `next-error` - Funktion des GNU-Emacs interpretiert werden können [Sta88].

Kapitel 7

Implementation von Modellkomponenten

Dieses Kapitel gibt einen Überblick über die Konzepte für die Implementation von Protokollschichtmodellen. Die Protokollschichtmodelle werden in der Sprache C durch abgeschlossene Module realisiert, die über festgelegte Schnittstellen miteinander und mit dem "Grundgerüst" von simulan interagieren. Durch die Realisierung in C steht die volle Flexibilität und Performanz dieser Sprache zur Verfügung. Nachteilig ist, daß die Implementation neuer Modellkomponenten komplexer ist als bei Verwendung einer speziellen Simulationssprache.

Ein konkretes Beispiel für die Programmierung eines Protokollschichtmodells findet sich im nächsten Kapitel.

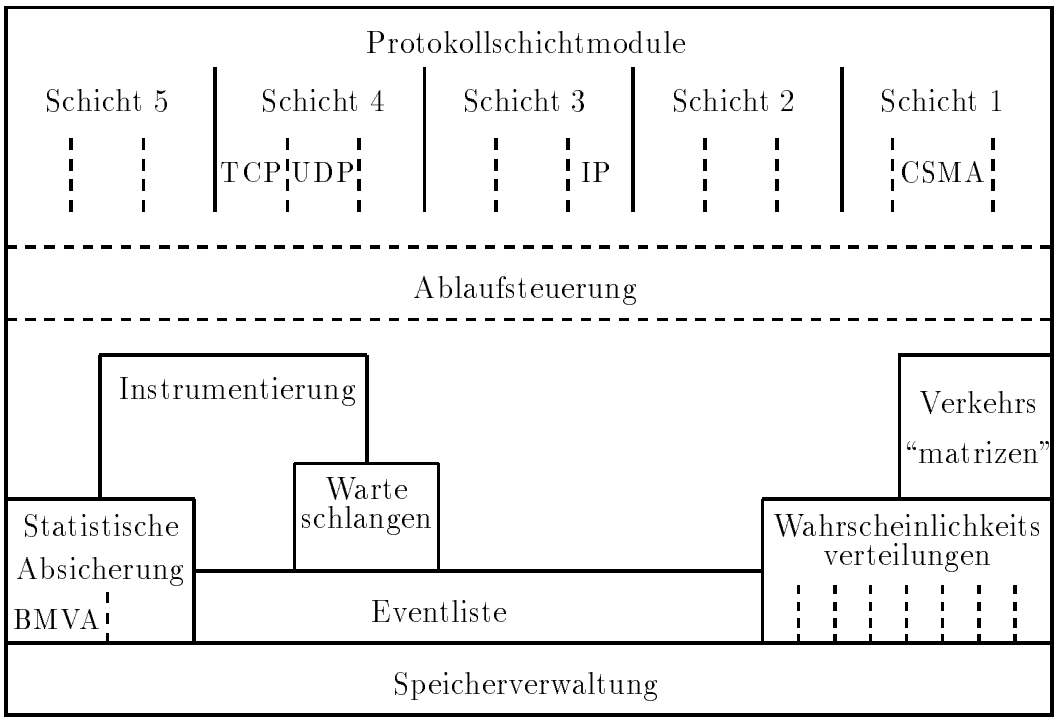


Abbildung 7.1: Hierarchie der Komponenten von simulan

7.1 Grundkomponenten für Protokollschichtmodelle

Um die Programmierung neuer Protokollschichtmodelle zu vereinfachen, steht in simuLan eine Bibliothek von Grundkomponenten zur Verfügung, die immer wieder nützlich sind. Dazu gehören Funktionen für die Speicherverwaltung, für die Verwaltung der Eventliste, für die Behandlung von Zufallsverteilungen und Verkehrsverteilungen, und für die Bearbeitung von Warteschlangen.

7.1.1 Die Eventliste

Jede Zustandsänderung im simulierten System wird durch ein Ereignis ausgelöst. Ein Ereignis wird durch eine `EVENT` - Struktur beschrieben. Diese enthält

- die Zeit zu der das Ereignis geschieht,
- einen Zeiger auf eine Funktion, die das Ereignis bearbeitet,
- einen Zeiger auf Privatinformationen, die der Verursacher des Ereignisses für die bearbeitende Funktion bereitgestellt hat,
- die Nummer des Knotens, in dem das Ereignis verursacht wurde. Bei Ereignissen, die das verbindende Netz betreffen (Schicht 1), ist das die Nummer des Senderknotens.

Die Verwaltungsfunktionen für die Eventliste können nicht wissen, welche Privatinformationen mit dem Ereignis verbunden sind. Falls simuLan mit DEBUG - Option übersetzt wurde, enthält ein `EVENT` daher auch einen Zeiger auf eine Funktion, die die Privatinformationen im Klartext ausgeben kann.

Damit die Ereignisse in zeitlicher Reihenfolge abgearbeitet werden können, werden die `EVENT`s in eine *Eventliste* einsortiert. Das Fortschreiten der simulierten Zeit geschieht, indem sie jeweils auf die Zeit des als nächstes zu bearbeitenden Ereignisses weitergesetzt wird. Dieses Ereignis wird aus der Eventliste entfernt und die angegebene Bearbeitungsfunktion aufgerufen.

Außer der Funktion zum Einfügen eines neuen `EVENT`s in die Eventliste stehen auch Funktionen zur Verfügung, um bereits einsortierte Ereginisse wieder zu entfernen bzw. als ungültig zu markieren, und um die Zeit oder die Bearbeitungsfunktion eines `EVENT`s in der Eventliste zu ändern (siehe Abschnitt A.7 [*Funktionen für die Eventliste*], Seite 76).

Für die Fehlersuche kann mit `print_event()` ein `EVENT` sowie die daran hängende Privatinformationen im Klartext ausgegeben werden.

7.1.2 Zufallsverteilungen

Damit Verteilungsfunktionen nicht fest in die Protokollschichtmodelle einprogrammiert werden müssen, werden sie durch `ZUFALL_INFO` - Strukturen beschrieben. Diese enthalten einen Zeiger auf eine Zufallsfunktion und einen Zeiger auf ein Array von Parametern für die Verteilungsfunktion. Für jede implementierte Verteilungsfunktion gibt es eine Einlesefunktion, die die Parameter der Verteilung aus der Konfigurationsdatei einliest und eine `ZUFALL_INFO` - Struktur ausfüllt.

Mit dem Makro `USE_ZUFALL()` wird eine Zufallszahl nach einer solchen Verteilung erzeugt (siehe Abschnitt A.9 [Wahrscheinlichkeitsverteilungen], Seite 79).

7.1.3 Verkehrsverteilungen

Für jeden Netzknoten kann eine Liste von Partnerknoten und zugeordneten Wahrscheinlichkeiten angegeben werden, nach der für eine Nachricht ein Empfängerknoten ermittelt wird. Die `NODE`-Struktur für die Beschreibung eines Netzknotens enthält einen Zeiger auf eine Funktion, die einen Empfängerknoten auswählt.
Ob und wie von dieser Möglichkeit Gebrauch gemacht wird, muß in die Knotenmodelle (Schicht 5) einprogrammiert werden (siehe Abschnitt 5.4.3 [Einlesen Schicht 2 - 5], Seite 28, und Abschnitt A.10 [Verkehrsmatrizen], Seite 81).

7.1.4 Warteschlangen

Ein wichtiges Element für die Modellierung von Rechensystemen sind *Warteschlangen*. In simula stehen für die Programmierung von Protokollschichtmodulen zwei Arten von Warteschlangen zur Verfügung :

- Die *Normale Warteschlange* (*NQUEUE*). Zu einer `NQEUE` gehören *n* Server, die jeweils einen Kunden bedienen können.
- Die *Ticket Queue* (*TQUEUE*) stellt *n Tickets* oder *Tokens* zur Verfügung, von denen jeder Kunde eine bestimmte Anzahl anfordert. Wenn nicht genügend Tickets vorhanden sind um die Anforderung eines Kunden zu befriedigen, wird der Kunde in die Warteschlange eingereiht.

Mit dem globalen Parameter '`max_q_length`' wird eine maximale Schlangenlänge vorgegeben, bei deren Überschreiten ein Fehler ausgelöst und der Simulationslauf beendet wird.

Als Datenstruktur zur Beschreibung der zu behandelnden Kunden hat sich der `EVENT` als günstig erwiesen.

Für jeden Warteschlangentyp stehen Funktionen zur Verfügung, um eine Warteschlange zu erzeugen, um neu ankommende Kunden bedienen zu lassen, und um wartende Kunden aus der Schlange zu entfernen. Für die Fehlersuche existieren Funktionen zum Ausgeben der Warteschlange im Klartext (siehe Abschnitt 7.8 [Debugging-Funktionen], Seite 45).

Beim Erzeugen einer Warteschlange wird angegeben, ob für diese Schlange die mittlere Wartezeit der Kunden und die mittlere Auslastung der Server bzw. die mittlere Anzahl vergebener Tokens ermittelt werden soll.

Als Bedienstrategien stehen jeweils *FIFO* und *LIFO* zur Verfügung. Wenn ein neuer Kunde an der Warteschlange ankommt, wird eine Funktion aufgerufen, die testet, ob ein freier Server bzw. genügend freie Tokens für den Kunden vorhanden sind.
Wenn ja, wird der Kunde sofort bedient und nicht in die Warteschlange eingekettet.
Anderenfalls wird geprüft, ob mit diesem Kunden die vorgegebene maximale Schlangenlänge überschritten würde. Wenn nicht, wird der Kunde vorne bzw. hinten in die Warteschlange eingekettet.
Es wird ein Returncode zurückgegeben, der anzeigt, ob der Kunde bedient wurde, ob er warten muß, oder ob die maximal zulässige Schlangenlänge überschritten würde.

Wenn ein Kunde bedient wird, muß für das Bedien-Ende ein Ereignis erzeugt werden, bei dessen Bearbeitung die reservierten Resourcen freigegeben werden. Falls noch Kunden warten, müssen diese explizit aus der Warteschlange ausgekettet werden.

Siehe Abschnitt A.8 [*Operationen auf Warteschlangen*], Seite 78 für eine detaillierte Beschreibung der Funktionen.

7.1.5 Speicherverwaltung

Am Ende jedes Simulationslaufes müssen die allokierten Datenstrukturen freigegeben werden, damit sie für den nächsten Durchlauf gemäß der angegebenen Konfiguration neu aufgebaut werden können. An vielen Stellen werden jedoch unbekannte Datenstrukturen miteinander verkettet (Zeiger auf Privatinformationen im `EVENT`). Damit nicht zu jedem solchen anonymen Zeiger ein Funktionszeiger zur Freigabe der angehängten Datenstruktur verwaltet werden muß, wurde auf `malloc(3)` eine eigene Speicherverwaltung aufgesetzt. Der mit `myalloc()` oder `myrealloc()` reservierte Speicherplatz wird nach dem Ende jedes Simulationslaufes mittels `myfree()` wieder freigegeben.

Für Datenstrukturen, die in wechselnder Folge reserviert und wieder freigegeben werden müssen (z.B. Warteschlangenelemente oder Nachrichtenbeschreibungen), können mit dem Makro `DEFUN_GET_REL_OBJECT()` Funktionen definiert werden, die eine Freispeicherliste solcher Objekte verwalten (siehe Abschnitt A.6 [Speicherverwaltung], Seite 74).

7.2 Zusammenspiel der Module

Der Benutzer wählt in der Konfigurationsdatei Protokollschichtmodelle für die Protokollschichten aus. Jedes Protokollschichtmodell wird durch ein Protokollschichtmodul in C realisiert. simuLAn wurde so konzipiert, daß beim Hinzufügen neuer Protokollschichtmodule in den bereits existierenden Programmteilen keine Änderungen vorgenommen werden müssen. Um das zu erreichen, werden die Funktionen eines Protokollschichtmoduls nicht mit Namen aufgerufen, sondern über Zeiger. Außerhalb eines Protokollschichtmoduls muß nur eine Einlesefunktion, die die Parameter für das Protokollschichtmodell aus der Konfiguration liest und die Zeiger auf die anderen Funktionen in globale Datenstrukturen einträgt, in eine Tabelle eingetragen werden.

Ein Protokollschichtmodul muß folgende Funktionen für die Interaktion mit anderen Modulen enthalten :

- Eine Einlesefunktion, die die Parameter aus der Konfiguration liest, globale Datenstrukturen ausfüllt und private Datenstrukturen anlegt. Außerdem muß sie eine Schablone für die Parameter dieses Moduls ausgeben können, wenn simuLAn mit der Option '`-x`' aufgerufen wird. Wenn simuLAn mit der Option '`-p`' aufgerufen wird, muß die Einlesefunktion die gelesenen Parameter und eventuell eingesetzte Defaultwerte ausgeben.

- Eine Initialisierungsfunktion, die nach dem Einlesen der Konfiguration und vor Beginn des Simulationslaufes aufgerufen wird. Sie dient dazu, Datenstrukturen zu initialisieren und die ersten Ereignisbeschreibungen in die Ereignisliste zu stellen. (1)

- Eine Funktion, die die Beschreibung einer zu sendenden PDU von der nächsthöheren Schicht übergeben bekommt und die Weiterleitung an die Peer-Entity im Empfänger einleitet. (2)

- Eine Funktion, die von der nächsttieferen Schicht die Beschreibung einer empfangenen PDU übergeben bekommt und deren Verarbeitung einleitet. (3)

- Eine Funktion, die von der nächsttieferen Schicht aufgerufen wird, wenn die Bearbeitung eines zuvor gegebenen Sendeauftrages abgeschlossen ist. (3)

- Eine Aufräumfunktion, die nach dem Ende des Simulationslaufes aufgerufen wird. Hier wird ggf. Speicherplatz freigegeben sowie Simulationsergebnisse ausgegeben. (1)

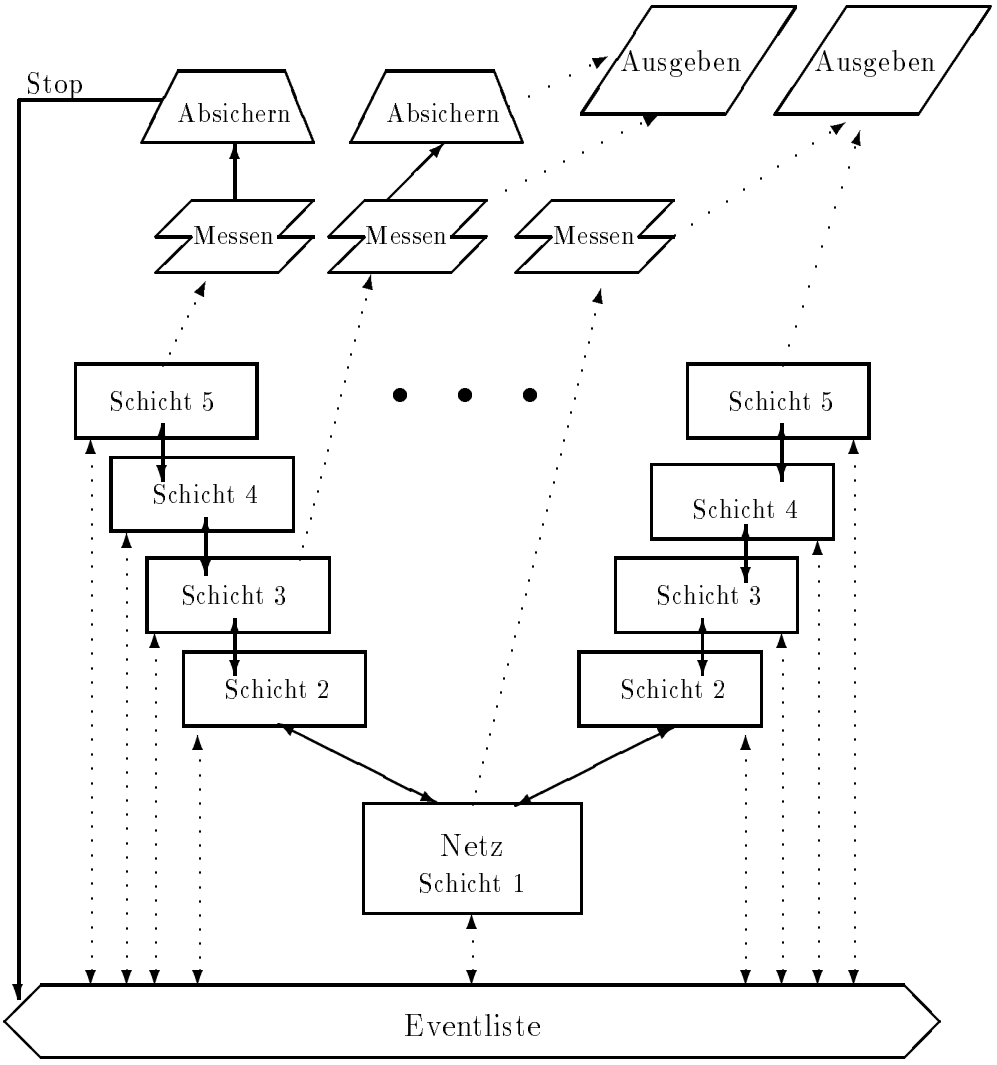


Abbildung 7.2: Zusammenspiel der Module

- Zur Fehlersuche sollte für jede Datenstruktur eines Moduls eine Funktion existieren, die diese Datenstruktur im Klartext ausdrucken kann.
- Meß- und Ausgabefunktionen für die Meßpunkte des Protokollschichtmoduls (siehe Abschnitt 7.9 [Implementation von Meßpunkten], Seite 45).

(1) Für die Schichten 2,3 und 4 sind keine Initialisierungsfunktionen vorgesehen, da diese Schichten keine Ereginisse von sich aus erzeugen, sondern nur auf Anforderung von oben oder unten.

(2) Die Protokollschichtmodule für die oberste Schicht 5 (die "Knotenmodule") enthalten keine Funktion zum Annehmen von Sendeaufträgen.

(3) Die Protokollschichtmodule für die unterste Schicht 1 (die "Netzmodule") enthalten keine Funktion zur Anzeige von empfangenen oder gesendeten PDUs von einer tieferen Schicht.

7.3 Globale Datenstrukturen

Die Aufteilung des simulierten Netzwerkes in mehrere Netzknoten, die durch ein Netz verbunden werden, spiegelt sich in den Datenstrukturen von simuLan wieder.

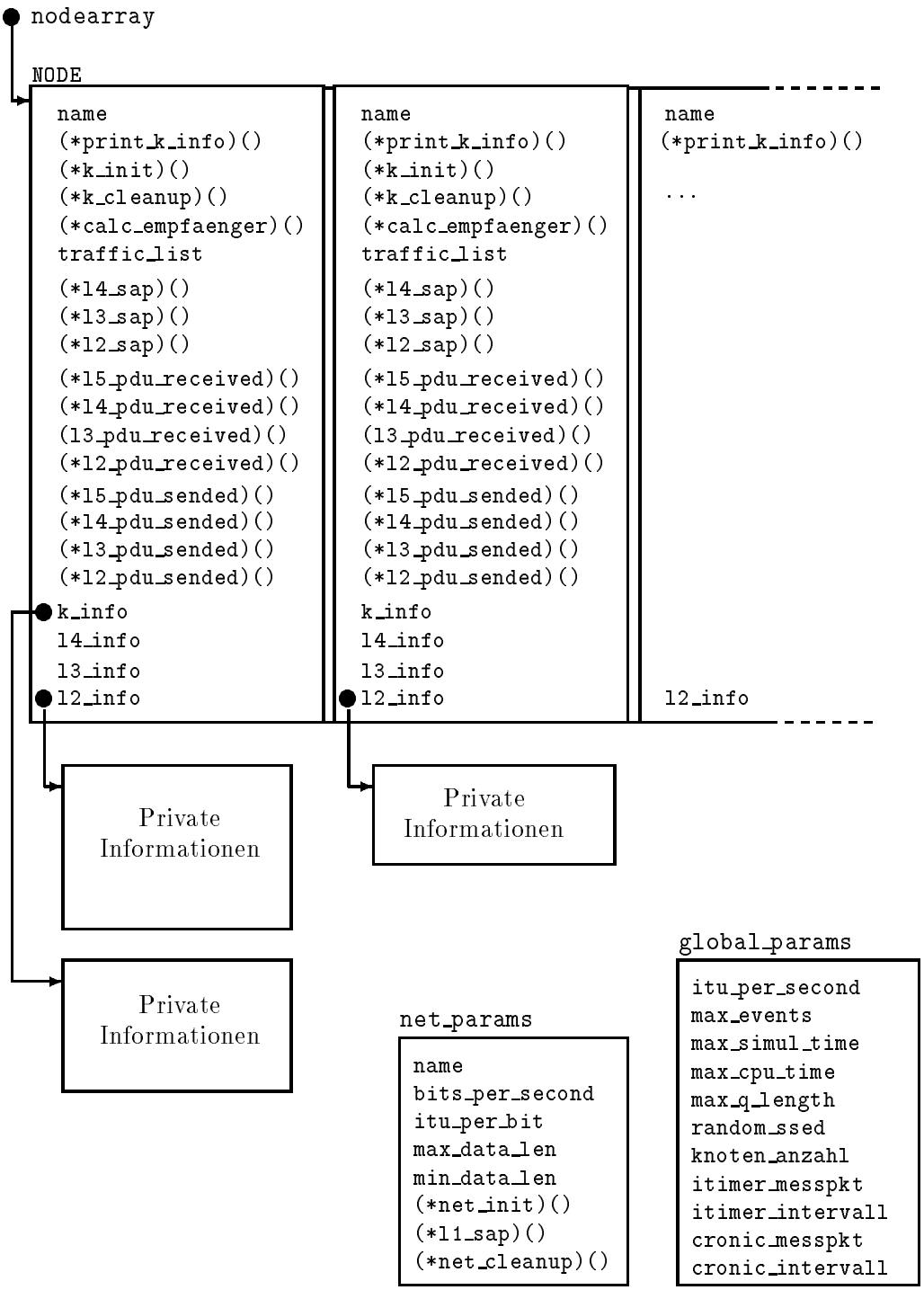


Abbildung 7.3: Globale Datenstrukturen

Während des Einlesens der Konfiguration wird ein Array von `NODE` - Strukturen angelegt, welches für jeden konfigurierten Knoten ein Element enthält. Für die unterste Protokollschicht exisitiert dagegen nur eine einzelne Struktur `net_params`.

Eine `NODE` - Struktur enthält Informationen, die für alle implementierten Protokollschichtmodelle gleich sind, und die gebraucht werden, um die für einen Knoten ausgewählten Protokollschichtmodule aufzurufen. Außerdem enthält sie Zeiger, an die private Informationen der ausgewählten Module für die Protokollschichten angehängt werden können.

Vom Netzmodul (Schicht 1) existiert nur eine Instanz, daher braucht die private Information dieses Moduls nicht über einen Zeiger an die `net_params`- Struktur gekettet werden.

7.4 Implementation von Einlesefunktionen

Für jede simuLAn - Schicht existiert eine Tabelle, in der die implementierten Protokollschichtmodelle verzeichnet sind. Ein Tabelleneintrag besteht aus

- dem Namen des Moduls, wie er in der Konfiguration angegeben wird,
- einem Zeiger auf die Einlesefunktion,
- und einer kurzen Erläuterung.

(Siehe Abschnitt A.15.7 [`xxgetfkt()`], Seite 92). Für ein neues Protokollschichtmodul muß in der entsprechenden Tabelle ein neuer Eintrag gemacht werden, darüberhinaus brauchen keine bereits vorhandenen Programmteile verändert werden.

Wenn simuLAn während des Einlesens der Konfiguration den Namen eines Protokollschichtmodells liest, wird der entsprechende Eintrag in der Tabelle für die Protokollschicht gesucht und die zugehörige Einlesefunktion aufgerufen (rekursiver Abstieg).

Die Einlesefunktionen für die Protokollschichtmodule haben folgende Aufgaben :

- Falls simuLAn mit der Option '`-x`' aufgerufen wurde, soll nur eine Schablone für die Konfiguration des Protokollschichtmodells ausgegeben werden, ohne daß Datenstrukturen aufgebaut werden.
- Sonst werden die auszufüllenden Datenstrukturen mit Defaultwerten vorbelegt und dann die aktuellen Werte aus der Konfigurationsdatei gelesen.
- Anschließend werden, soweit möglich, Plausibilitätsprüfungen durchgeführt.
- Wenn simuLAn mit der Option '`-p`' aufgerufen wurde, sollen die eingelesenen Parameter bzw. die eingesetzten Defaultwerte ausgegeben werden.

Die Angabe der Parameter für ein Protokollschichtmodell erfolgt meistens in der Form

```
[ <parameter-name> <parameter-value> ]
```

Zum Einlesen solcher Name-Wert-Paare steht die Funktion `rcreadkeywds()` zur Verfügung. Sie bekommt einen Zeiger auf eine Tabelle übergeben, in der die Parameter für das Protokollschichtmodul eingetragen sind. Jeder Eintrag besteht aus

- dem Namen des Parameters, wie er in der Konfiguration angegeben wird,
- einer Kennung für den Typ des erwarteten Parameterwertes,
- einem Zeiger auf eine Speicherstelle, in der der eingelesene Wert abgelegt werden soll, bzw. einem Zeiger auf eine aufzurufende Funktion,
- ggf. einem Zeiger auf eine Tabelle mit für diesen Parameter wählbaren Funktionsnamen,
- und einer kurzen Erläuterung.

(Siehe Abschnitt A.15.10 [`rcreadkeywds()`], Seite 93). Falls kompliziertere Parameterkonstruktionen eingelesen werden müssen, stehen auch Funktionen zum Einlesen einzelner Strings, Zahlen, Ausgabekanäle, Funktionsnamen und Meßpunktbestückungen zur Verfügung (siehe Abschnitt A.15 [Einlesen der Konfiguration], Seite 89).

7.4.1 Einlesefunktionen für Schicht 5

Die Einlesefunktionen der implementierten Protokollschichtmodule für Protokollschicht 5 werden in die Tabelle `l5models_tab` eingetragen. Sie bekommen als Parameter

- einen Zeiger auf ein Array von auszufüllenden `NODE` - Strukturen,
- die Länge dieses Arrays (aus dem Parameter '`COUNT`' in der Konfigurationsdatei),
- und die Einrücktiefe für die Ausgabe der Konfigurationsschablone bzw. der eingelesenen Parameter.

Die einfachste Möglichkeit für das Einlesen der Parameter ist, ein Array von `KEYWD_TABLE`-Strukturen anzulegen, das für jeden Parameter des Protokollschichtmodells einen Eintrag enthält. Die Parameterwerte werden in statisch angelegte Variablen eingelesen. Diese Variablen werden mit Defaultwerten belegt, dann wird die Funktion `rcreadkeywds()` aufgerufen. Wenn die eingelesenen Parameterwerte ausgedruckt werden sollen, wird das von `rcreadkeywds()` gleich mit erledigt (siehe Abschnitt A.15.10 [`rcreadkeywds()`], Seite 93).

Nachdem die eingelesenen Parameter auf Plausibilität geprüft sind, muß jedes Element des übergebenen Arrays von `NODE`s ausgefüllt werden. Dazu müssen die Zeiger auf die Funktionen zum Initialisieren und Aufräumen des Knotens, zum Ausgeben der modulspezifischen Privatinformationen und zur Benachrichtigung über empfangene und gesendete Nachrichten in die `NODE`s eingetragen werden. Falls das Knotenmodul die angebotenen Funktionen für die Verkehrsverteilungen benutzt, müssen auch die Zeiger auf die Verkehrsverteilungsfunktion und die Partnerliste ausgefüllt werden.

An den Zeiger `k_info` in den `NODE`s können private knotenbezogene Datenstrukturen des Knotenmoduls gehängt werden.

Die Funktion `copy_probepoint()` wird benutzt, um die Datenstrukturen für die Meßpunkte zu kopieren. (siehe Abschnitt A.15.9 [`copy_probepoint()`], Seite 93)

Wenn die Schablone für die Konfigurationsdatei ausgegeben werden soll, ist die globale Variable `minus_x` ungleich null. Dann kann statt `rcreadkeywds()` einfach die Funktion `rcexplainkeywds()` aufgerufen werden, die eine Erläuterung der übergebenen Parametertabelle ausgibt. Danach erfolgt der Rücksprung aus der Einlesefunktion.

7.4.2 Einlesefunktionen für die Schichten 2,3,4

Die Einlesefunktionen für die Protokollschichtmodule der Protokollschichten 2, 3 und 4 werden in die Tabellen `l2models_tab`, `l3models_tab` bzw. `l4models_tab` eingetragen.

Sie funktionieren im Wesentlichen wie die für Schicht 5. Sie müssen in dem übergebenen Array von `NODE`s die Zeiger auf die Funktionen zum Annehmen von Sendeaufträgen und für die Benachrichtigung über empfangene und gesendete Nachrichten der jeweiligen Protokollschicht ausfüllen. Für jede dieser Protokollschichten steht in den `NODE` - Strukturen ein Zeiger auf private Datenstrukturen zur Verfügung.

Initialisierungsfunktionen sind für die die Schichten 2, 3 und 4 nicht vorgesehen, da diese Schichten von sich aus keine Ereignisse erzeugen, sondern nur auf Anforderung von oben oder unten.

7.4.3 Einlesefunktionen für Schicht 1

Die Einlesefunktionen der Protokollschichtmodule für die Schicht 1 werden in die Tabelle `l1models_tab` eingetragen.

Da für die Protokollschicht 1 nur eine Instanz eines Protokollschichtmoduls existiert, bekommt die Einlesefunktion als Parameter nur eine Einrücktiefe übergeben. Sie muß in die Struktur `net_params` den Namen des Netzmodells, die nominelle Übertragungsgeschwindigkeit (in Bits pro Sekunde), die minimale und maximale Länge einer übertragbaren Nachricht, sowie Zeiger auf Funktionen für die Initialisierung und zum Annehmen von Sendeaufträgen eintragen (siehe Abschnitt A.15.15 [Einlesen der Netzparameter], Seite 97).

7.5 Initialisierungsfunktionen

Nach dem Einlesen der Konfiguration und vor Beginn des Simulationslaufes wird die Initialisierungsfunktion für das Netzmodul (Schicht 1) aufgerufen. Diese kann z.B. Signallaufzeiten zwischen den konfigurierten Knoten vorberechnen.

Anschließend wird das aufgebaute `nodearray` durchgegangen und für jeden Knoten die angegebene Initialisierungsfunktion für Schicht 5 aufgerufen. Diese dienen dazu, die ersten Ereignisse zu erzeugen.

Für die Schichten 2, 3 und 4 sind (bisher) keine Initialisierungsfunktionen vorgesehen.

7.6 Aufräumfunktionen

Nach dem Ende jedes Simulationslaufes werden die Aufräumfunktionen für die Schichten 1 und 5 der ausgewählten Module aufgerufen. Diese können Ergebnisse des Simulationslaufes ausgeben und Speicher freigeben, soweit er nicht mittels `myalloc()` oder `myrealloc()` reserviert wurde. Für die Schichten 2,3 und 4 sind (bisher) keine Aufräumfunktionen vorgesehen.

7.7 Dienstzugangs- und Anzeigefunktionen

Ein Protokollschichtmodul sendet *Protocol Data Units* (*PDU*s) an die Peer-Entities im Empfängerknoten. Dazu benutzt es die Dienste der nächsttieferen Protokollschicht. In simuLAN wird eine PDU der Schicht *N* beim Aufruf des Dienstes der Schicht *N-1* durch eine *Service Access Data Unit* (*SADU*) beschrieben, der Inhalt der PDU wird von Schicht *N-1* nicht interpretiert (transparente Übertragung).

Die `SADU` - Struktur enthält :

- die Länge der zu übertragenden Nachricht,
- die Nummer des Empfängerknotens,
- einen Zeiger auf den Inhalt der Nachricht (die Layer-*N*-PDU).

Da in simuLan jede Zustandsänderung durch ein Ereignis ausgelöst wird, hat es sich als zweckmäßig erwiesen, diese `SADU` an eine `EVENT` - Struktur zu ketten, die als Parameter an die Dienstzugangsfunktion der Schicht *N-1* übergeben wird. Dadurch wird vermieden, daß `EVENT`s in Schicht *N* freigegeben und in Schicht *N-1* sofort wieder reserviert und mit den gleichen Informationen wie vorher ausgefüllt werden müssen.

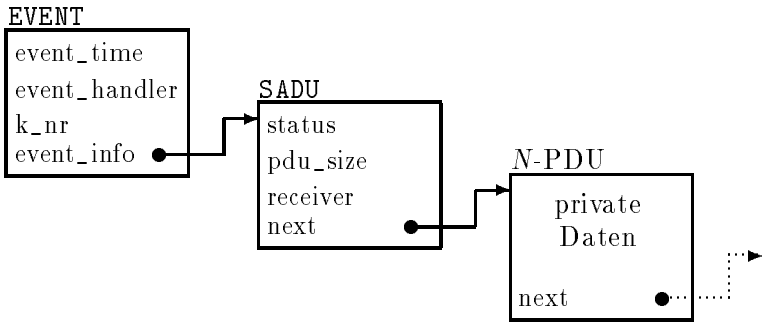


Abbildung 7.4: Verkettung der Parameter in Protokollschicht *N*

Für die Weiterleitung an Schicht *N-1* im Empfängerknoten wird die erhaltene Layer-*N*-PDU in eine PDU der Schicht *N-1* "verpackt" und die beschreibende `SADU` entsprechend ausgefüllt.

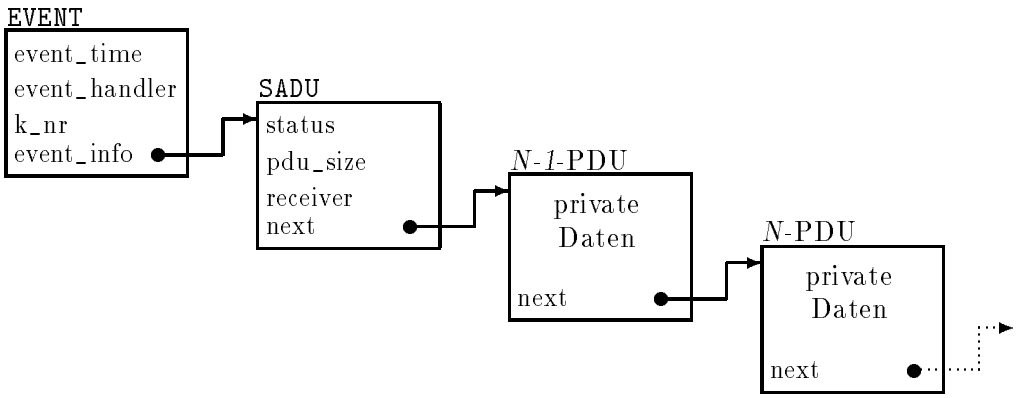


Abbildung 7.5: Verkettung der Parameter in Protokollschicht *N-1*

Nachdem ein *Frame* in Schicht 1 zum Empfängerknoten übertragen wurde, benachrichtigt das Netzmodul die Schicht 2 des Empfängers, indem es deren Anzeigefunktion für empfangene Nachrichten aufruft und ihr die übertragene Layer-2-PDU übergibt, so wie sie vom Sender an Schicht 1 übergeben wurde. Der Parameter dieser Funktion ist also wiederum eine Verkettung `EVENT`→`SADU`→Layer-2-PDU.

Wenn die Schicht *N* im Empfängerknoten eine vollständige Layer-*N+1*-PDU empfangen hat, wird sie "ausgepackt", indem die Kontrollinformation der Schicht *N* ausgekettet wird, und an die Empfangsfunktion der nächsthöheren Schicht übergeben.

Hinweis : Der Speicherplatz für die `EVENT`s, `SADU`s und die verschiedenen PDUs ist von den Funktionen des Senderknotens reserviert worden und darf auch nur dort wieder freigegeben werden. Daher müssen die Datenstrukturen vor dem Rücksprung aus den Empfangsfunktionen wieder genauso verkettet sein, wie sie übergeben wurden.

Wenn eine Empfangsfunktion sich Information aus den empfangenen PDUs zwischen zwei Aufrufen merken will, muß sie diese kopieren, statt sich nur einen Zeiger zu speichern, da die Datenstrukturen in der Zwischenzeit vom Senderknoten freigegeben oder verändert werden können.

Es werden nur vollständig und korrekt übertragene Nachrichten an die Schicht *N+1* des Empfängerknotens weitergegeben, bei Übertragungsfehlern geht die Nachricht verloren.

Nach der Benachrichtigung des Empfängers teilt das Netzmodul der Schicht 2 im Senderknoten mit, daß der Sendeauftrag ausgeführt wurde. Dazu wird deren Sendebenachrichtigungsfunktion aufgerufen und ihr wieder die Verkettung `EVENT`→`SADU`→Layer-2-PDU übergeben.

Wenn die Schicht *N* im Senderknoten feststellt, daß eine Layer-*N+1*-PDU vollständig übertragen wurde, wird sie "ausgepackt" und die Sendebenachrichtigungsfunktion der nächsthöheren Schicht aufgerufen. Die ausgekettete Layer-*N*-PDU kann freigegeben oder für den nächsten Sendeauftrag an Schicht *N-1* benutzt werden.

Siehe Abschnitt A.13 [Service Access Points], Seite 84 für eine detaillierte Beschreibung der Datenstrukuren und Schnittstellen.

7.8 Debugging-Funktionen

Wenn s$_{j}$m$_{u}$L^{a}n mit DEBUG-Option übersetzt wird, ist vorgesehen, daß jede Datenstruktur, die einen "anonymen" Zeiger auf unbekannte Objekte enthält (z.B. `EVENT`, `SADU`), dazu auch einen Zeiger auf eine Funktion enthält, die die angehängten Objekte im Klartext ausdrucken kann. Jedesmal, wenn an so einen anonymen Zeiger ein Objekt angekettet wird, muß auch die entsprechende Ausgabefunktion eingetragen werden.

Jede dieser Ausgabefunktionen muß ihrerseits die Ausgabefunktionen für die Objekte aufrufen, die an der ausgegebenen Datenstruktur hängen.

Die Ausgabefunktionen bekommen als Parameter einen Zeiger auf das auszugebende Objekt, einen `FILE` - Zeiger für die Ausgaben, eine Einrücktiefe und eine Schachtelungstiefe. Die Schachtelungstiefe wird für jedes Objekt einer Kette heruntergezählt, wenn sie null ist, wird nur das Objekt selber ausgegeben, nicht aber die daran hängenden Strukturen. Durch diesen Mechanismus läßt sich z.B. die Eventliste in beliebiger Ausführlichkeit ausdrucken.

Damit zu Funktionszeigern ein Funktionsname ausgegeben werden kann, sollten alle Funktionen, die über Zeiger aufgerufen werden, in der Tablle `event_handler_tab` eingetragen werden. Die Funktion `get_event_handler_name()` sucht zu einer Funktionsadresse den Namen aus der Tabelle und gibt ihn zurück. (Siehe Abschnitt A.16 [Debugging], Seite 97)

7.9 Implementation von Meßpunkten

Ein *Meßpunkt* besteht zunächst aus einem `PROBE*` - Zeiger in den modulspezifischen Datenstrukturen eines Protokollschichtmoduls. Falls an diesem Meßpunkt Messungen durchgeführt werden sollen, wird während des Einlesens der Konfiguration an diesen Zeiger eine Liste von `PROBE` - Strukturen gekettet. Eine `PROBE` - Struktur enthält einen Zeiger auf eine *Meßfunktion*, Platz für gemessene Werte, `FILE` - Zeiger und Formatangaben für Ausgaben, sowie ggf. Informationen für eine statistische Absicherung der gemessenen (Mittel-) Werte (zur Kontrolle der Simulationslänge).

Für jeden Meßpunkt werden eine oder mehrere Meßfunktionen programmiert. Zu jeder Meßfunktion muß auch eine Einlesefunktion und eine Ergebnis-Ausgabe-Funktion geschrieben werden.

Die Einlesefunktion wird während des Einlesens der Konfiguration aufgerufen, wenn die zugehörige Meßfunktion ausgewählt wurde. Sie trägt in die `PROBE` einen Zeiger auf die

Meßfunktion ein, einen Zeiger auf die Ergebnis-Ausgabe-Funktion, und eine Überschrift, die die Bedeutung der Meßwerte angibt. Außerdem gibt sie die Überschrift in das `FILE` aus, das für <*file-ident1*> in der Konfiguration angegeben wurde.

Falls eine statistische Absicherung der Meßwerte konfiguriert wurde, wird die Ergebnis-Ausgabe-Funktion nach dem Ende des Simulationslaufes aufgerufen. Sie dient dazu, den durch die statistische Absicherung errechneten (Mittel-) Wert aufzubereiten (z.B. Umrechnung von ITU in Sekunden), und in das `FILE stat_fd` auszugeben, das für <*file-ident3*> in der Konfiguration angegeben wurde.

Die einfachste Möglichkeit für das Einlesen von Meßpunktkonfigurationen ist folgende :

Für jeden Meßpunkt wird ein Array von `FKT_TABLE` - Strukturen erstellt, das einen Eintrag für jede Meßfunktion des Meßpunktes enthält. Als Funktionszeiger wird jeweils die Adresse der zugehörigen Einlesefunktion eingetragen.
Der letzte Eintrag der Tabelle enthält für den Funktionsnamen einen `NULL` - Zeiger. Als Funktionszeiger kann die Adresse einer Funktion für formatierte Ausgaben eingetragen werden, die speziell für den Meßpunkt angepaßt wurde (Vorbild für solche Funktionen ist `probe_printf()`, die auch als Defaultwert benutzt wird, wenn der letzte Tabelleneintrag `NULL` als Funktionszeiger enthält.)

Für jeden Meßpunkt eines Protokollschichtmoduls wird ein Eintrag in dem `KEYWD_TABLE` - Array, das die wählbaren Parameter des Protokollschichtmodells angibt, gemacht, so daß beim Einlesen der Konfiguration mittels `rcreadkeywds()` die Meßpunktkonfigurationen richtig aufgebaut werden.

Wenn in dem Protokollschichtmodul eine Zustandsänderung eintritt, durch die Meßwerte anfallen, so werden mit dem Makro `MEASURE()` die für den entsprechenden Meßpunkt definierten Messungen durchgeführt. Das Makro bekommt als Parameter einen Zeiger auf den Meßpunkt und einen Zeiger auf den `EVENT`, der die Zustandsänderung ausgelöst hat.

Die Meßfunktion der Messung ermittelt einen Meßwert und legt ihn in der `PROBE` - Struktur ab. Die Struktur enthält auch Platz, um den vorhergehenden Meßwert und die Zeit, zu der er gemessen wurde, zu speichern. Falls in der Konfiguration eine Datei für <*file-ident1*> angegeben wurde, soll die Meßfunktion den Meßwert auch in das `FILE trace_fd` ausgeben.

Siehe Abschnitt 5.4.5 [Beschreibung der Instrumentierung], Seite 29; Abschnitt A.14.1 [`probe_printf()`], Seite 86; Abschnitt A.14.2 [*Die* `PROBE` - *Struktur*], Seite 87; Abschnitt A.15.10 [`rcreadkeywds()`], Seite 93; Abschnitt A.15.7 [`xxgetfkt()`], Seite 92.

Kapitel 8

Beispiele zur Realisierung von Protokollschichtmodulen

In diesem Kapitel werden die vorgestellten Möglichkeiten, die simuLan zur Programmierung von Protokollschichtmodulen bietet, anhand konkreter Beispiele vedeutlicht.

Zunächst wird ein einfaches Modell für Protokollschicht 1 vorgestellt, danach ein komplexeres Modell für eine lasterzeugende "Anwendung". Schließlich werden noch Hinweise gegeben, wie neue Skelette für Protokollschichtmodule aus schon existierenden erzeugt werden können.

Hier wird nur auf die Realisierung von Modellen eingegangen, nicht aber auf Verifikation, Validierung oder Kalibrierung der Modelle.

8.1 Ein Netzmodul (Schicht 1) : PTP

In diesem Abschnitt wird die Realisierung des Modells eines vollvermaschten Netzwerkes als Protokollschichtmodul für Schicht 1 erläutert. In diesem (fiktiven) *Point-To-Point* - Netz (*PTP*) sind die Knoten paarweise durch Vollduplexleitungen physikalisch verbunden. Dadurch brauchen die Knoten nicht um ein Übertragungsmedium konkurrieren. Ein Knoten kann gleichzeitig Nachrichten von allen anderen Knoten empfangen, aber zu jeder Zeit nur eine Nachricht senden. Die höheren Protokollschichten müssen sicherstellen, daß kein Netzknoten eine Nachricht an Schicht 1 zum Senden übergibt, bevor die vorhergehenden Sendeaufträge vom PTP-Modul vollständig bearbeitet wurden (das leistet beispielsweise das unten erläuterte Modell '`l2simple`').

Die nachfolgend gezeigten Programmstücke sind soweit wie möglich vereinfacht, insbesondere werden Anweisungen für Debugausgaben weitgehend ausgelassen.

Der Programmtext für das Modul steht im File '`ptp.c`'. Die Deklarationen der Funktionen, die außerhalb des des PTP-Moduls namentlich bekannt sein müssen, stehen in '`ptp.h`'.

8.1.1 Datenstrukturen und Variablen des PTP-Moduls

Zunächst werden die Variablen und Datenstrukturen aus der Systembibliothek und aus der simuLan - Grundbibliothek bekannt gemacht. Mit dem Makro `P` kann das Programm sowohl mit ANSI - C als auch mit älteren Compilern übersetzt werden.

```
#include <stdio.h>
#ifdef USG              /* System 5 ist anders */
#include <string.h>
#else
#include <strings.h>
#endif

#ifndef P
#ifdef __STDC__
#define P(s) s
typedef void *VOID_P;
#else
#define P(s) ()
typedef char *VOID_P;
#endif
#endif

#include "extern.h"     /* Deklarationen für die Systembibliothek */
#include "defines.h"    /* globale Konstanten für simuLan */
#include "global.h"     /* globale Variablen für simuLan */
#include "events.h"     /* die Eventliste */
#include "rctypes.h"    /* }                                      */
#include "rcparms.h"    /* } für das Einlesen der Konfiguration */
#include "rc.h"         /* }                                      */
#include "ptp.h"
#include "simalloc.h"   /* Speicherverwaltung für simuLan */
#include "nodes.h"      /* die NODE - Struktur */
#include "zufall.h"
```

Ein zu sendendes Paket wird mit der für die physikalische Übertragung nötigen Kontrollinformation in einen *Frame* verpackt. Die Datenstruktur dafür ist der `PTP_FRAME`.

Die Komponenten `object_nr`, `print_next` und `start_time` werden zur Fehlersuche und funktionalen Analyse gebraucht, um jeden `PTP_FRAME` eindeutig identifizieren und die darin verpackte Information ausgeben zu können.
In `merk_l2_pdu_size` wird die Größe der übergebenen Nachricht für das Auspacken nach dem Übertragungsende gespeichert.
Der `next` - Zeiger dient zur Verkettung in der Freispeicherliste, und zum Anhängen der von den höheren Schichten zum Senden übergebenen privaten Datenstrukturen.

```
typedef struct S_ptp_frame
{
#ifdef DEBUG
  long object_nr;
  void (*print_next)();
  TIME_T start_time;
#endif
  int merk_l2_pdu_size;
  struct S_ptp_frame *next;
} PTP_FRAME;
```

Mit dem Makro `DEFUN_GET_REL_OBJECT()` werden die Funktionen `get_PTP_FRAME()` und `release_PTP_FRAME()` sowie der Anker der Freispeicherliste, `PTP_FRAME_freelist`, angelegt.

```
DEFUN_GET_REL_OBJECT(PTP_FRAME)
```

Zur Compilezeit kann ausgewählt werden, ob Übertragungsfehler simuliert werden sollen, um ggf. den Rechenzeitbedarf zu verringern. `Ptp_stoerzeit` enthält den Zeitpunkt der letzten aufgetretenen Übertragungsstörung.

```
#ifdef SIMUL_TRANSMIT_ERROR
static TIME_T ptp_stoerzeit;
#endif
```

Die Parameter und Simulationsergebnisse des PTP-Modells sind in der Struktur `ptp_info` zusammengefaßt.

`Overhead` gibt die Größe der Kontrollinformation (in Bits) der Frames an.
`Error_distr` enthält die Beschreibung der Verteilungsfunktion der Übertragungsstörungen. Sie wird nur beachtet, wenn die Simulation von Übertragungsstörungen eincompiliert ist.
`Probept_end_transmit` kann benutzt werden, um am Ende jeder Übertragung Meßwerte zu ermitteln.

Die `ptp_info` - Struktur wird statisch mit Defaultwerten initialisert.

```
struct
{
  double overhead;
  ZUFALL_INFO error_distr;
  PROBEPOINT probept_end_transmit;
} ptp_info

 = { 208, { NULL, NULL }, NULL };
```

Die Funktionen des Moduls werden vor ihrer Benutzung deklariert.

```
void ptp_init P((void));
void ptp_cleanup P((void));
void ptp_data_encapsulation P((EVENT *));

void ptp_end_transmit P((EVENT *));

void ptp_rc_throughput P((PROBE *));

#ifdef DEBUG
void ptp_print_frame P((PTP_FRAME *, FILE *, int depth, int indent));
#endif
```

Die folgende Tabelle enthält für jede Meßfunktion, die für den `probept_end_transmit` verfügbar ist, einen Eintrag.
Die erste Komponente jedes Eintrags ist der symbolische Name der Meßfunktion, der in der Konfiguationsdatei angegeben wird.
Die zweite Komponente zeigt auf die zur Meßfunktion gehörende Einlesefunktion.
Die dritte Komponente ist ein Kommentar, der in der Schablone für die Konfiguationsdatei ausgegeben wird.

```
FKT_TABLE ptp_probept_end_transmit_tab[] =
{
  { "ptp_throughput",  ptp_rc_throughput,  "throughput of PTP-Net" },
  { NULL, NULL, NULL }
};
```

In der folgenden Tabelle sind die für das PTP-Modell wählbaren Parameter verzeichnet (siehe Abschnitt A.15.10 [rcreadkeywds()], Seite 93).

'`Bits_per_second`', '`min_data_len`' und '`max_data_len`' sind Parameter, die für jedes Netzmodul gesetzt werden müssen, und die ggf. auch von den höheren Protokollschichten

berücksichtigt werden. Das PTP-Modell stellt sicher, daß jedes übertragene Datenpaket *vor dem Einpacken* mindestens `net_params.min_data_len` Bits lang ist, indem an kürzere Pakete Füllbits angehängt werden. Beim Auspacken werden diese Füllbits wieder entfernt. `Net_params.max_data_len` wird vom PTP-Modell nicht beachtet.

```
#define CN(x)   (VOID_P)&net_params.x
#define CE(x)   (VOID_P)&ptp_info.x

KEYWD_TABLE rcptpnettab[] =
{ { "bits_per_second",
    CN(bits_per_second),
    NULL,
    XXTYPdouble,
    Net Speed" },
  { "error_distr",
    CE(error_distr),
    distr_tab,
    XXTYPnamedhook,
    "ditribution of transmission errors" },
  { "end_transmit_probept",
    CE(probept_end_transmit),
    ptp_probept_end_transmit_tab,
    XXTYPprobe,
    "" },
  { "min_data_len",
    CN(min_data_len),
    NULL,
    XXTYPdouble,
    Minimum size of PTP-Frame in bits" },
  { "max_data_len",
    CN(max_data_len),
    NULL,
    XXTYPdouble,
    Maximum size of PTP-Frame in bits" },
  { "overhead",
    CE(overhead),
    NULL,
    XXTYPdouble,
    "Overhead for each packet" },
  { "ENDptp",   NULL,   NULL,   XXTYPend,       "end of ptp-description" },
  { NULL,       NULL,   NULL,   XXTYPundef,     NULL }
};
#undef CE
#undef CN
```

8.1.2 ptp_rc()

Die Funktion `ptp_rc()` liest die Parameter für das PTP-Modell aus der Konfiguationsdatei ein.

Dies ist die einzige Funktion des Moduls, die dem "Grundgerüst" von simuLan namentlich bekannt gegeben werden muß. Sie wird im File '`tables.c`' in die Tabelle `l1models_tab` eingetragen.

```
*** simuLan/src/simoduLe/tables.c-old  Mon Jul 29 14:12:51 1991
--- simuLan/src/simoduLe/tables.c      Tue Jul 30 02:33:50 1991
```

```
***************
*** 38,43 ****
--- 38,44 ----

  /* Funktionen aus den Netzmodellen */

+ #include "ptp.h"
  #include "csma.h"


***************
*** 161,166 ****
--- 162,169 ----

  KEYWD_TABLE l1models_tab[] =
  {
+   { PpTP",   (C)ptp_rc,      NULL,   XXTYPimmhook,
+                                      "simple Point-to-Point Net" },
    { CSMA",  (C)csma_rc,     NULL,   XXTYPimmhook,   CSMA/CD" },
    { Token", (C)rcnotimpl,   NULL,   XXTYPimmhook,   "" },
    { "ENDnet",NULL,           NULL,   XXTYPend,       NULL },
```

Wenn simuLAn mit der Option '`-x`' aufgerufen wird, soll eine Schablone für die Konfigurationsdatei ausgegeben werden. Dazu dient die Funktion `rcexplainkeywds()`.

```
void ptp_rc(indent)
int indent;
{
  if (minus_x) { rcexplainkeywds(rcptpnettab, indent); return; }
```

Anderenfalls werden zunächst die allgemeinen Netzparameter in der Struktur `net_params` mit Defaultwerten vorbelegt und die Zeiger auf die Funktionen des PTP-Moduls eingetragen.

```
  net_params.name = PpTP-Net";

  net_params.bits_per_second = 1e7;
  net_params.min_data_len = 368;        /* In Bits */
  net_params.max_data_len = 12000;      /* In Bits */

  net_params.net_init = ptp_init;
  net_params.l1_sap = ptp_data_encapsulation;
  net_params.net_cleanup = ptp_cleanup;
```

Danach werden die Parameter aus der Konfigurationsdatei gelesen. Falls simuLAn mit der Option '`-p`' aufgerufen wurde, werden dabei auch die eingelesenen Werte sowie eingesetzte Defaultwerte ausgegeben.

```
  rcreadkeywds(rcptpnettab, indent);
```

Zum Schluß werden Plausibilitätsprüfungen durchgeführt.

```
#ifdef SIMUL_TRANSMIT_ERROR
  if (!ptp_info.error_distr.zufall_fkt)
    RC_ERROR(PpTP: No error_distr", 0, 0);
#endif

#ifdef DEBUG
  if (rc_debuglevel) fputs("ptp_rc: Bye \n", stderr);
#endif
}
```

8.1.3 ptp_init()

Die Funktion `ptp_init()` wird nach dem Einlesen der Konfiguration und vor Beginn des Simulationslaufes über den Funktionszeiger `net_params.net_init` aufgerufen. Sie initialisert die Freispeicherliste für die `PTP_FRAME`s. Falls Übertragungsstörungen simuliert werden sollen, wird das erste Störungsereignis erzeugt.

```
void ptp_init()
{
  PTP_FRAME_freelist = (PTP_FRAME *)NULL;

#ifdef SIMUL_TRANSMIT_ERROR
  if (ptp_info.error_distr.zufall_fkt)
  {
    EVENT *event_p = get_EVENT();
    event_p->event_info = NULL;
#ifdef DEBUG
    event_p->print_event_info = NULL;
#endif /* DEBUG */
    event_p->event_handler = ptp_transmit_error;
    event_p->event_time = (TIME_T)USE_ZUFALL(ptp_info.error_distr);
    insert_event(event_p);
    ptp_stoerzeit = (TIME_T)0;
  }
#endif
}
```

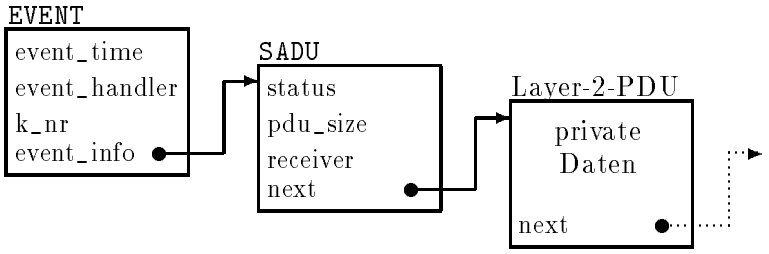
8.1.4 ptp_cleanup()

Die Funktion `ptp_cleanup()` wird nach dem Ende des Simulationslaufes über den Funktionszeiger `net_params.net_cleanup` aufgerufen. Für das PTP-Modul braucht nichts aufgeräumt zu werden.

```
void ptp_cleanup()
{
#ifdef DEBUG
  if (rc_debuglevel > 5)
  {
    fputs("ptp_cleanup(): hi & bye \n", stderr);
    fflush(stderr);
  }
#endif
}
```

8.1.5 ptp_data_encapsulation()

`Ptp_data_encapsulation()` ist die Dienstzugangsfunktion des PTP-Modells. Sie wird über den Funktionszeiger `net_params.l1_sap` aufgerufen, wenn ein Knoten ein Datenpaket übertragen möchte. Die Funktion bekommt als Parameter eine Verkettung



Der `EVENT` enthält die Zeit, zu der der Sendewunsch auftritt (der Zeitpunkt kann in der Zukunft liegen), die Nummer des Senderknotens und einen Zeiger auf eine `SADU`, die das zu übertragende Paket näher beschreibt. Wenn simuLan mit DEBUG-Option übersetzt wurde, enthält er außerdem einen Zeiger auf eine Funktion, die die angehängte `SADU` - Struktur im Klartext ausgeben kann.

Die `SADU` enthält die Größe der zu übertragenden Nachricht in Bits, die Nummer des Empfängerknotens und einen Zeiger auf private Informationen höherer Protokollschichten für diese Nachricht. Falls simuLan mit DEBUG-Option übersetzt wurde, enthält sie außerdem einen Zeiger auf eine Funktion, die die angehängte Privatinformation ausgeben kann.

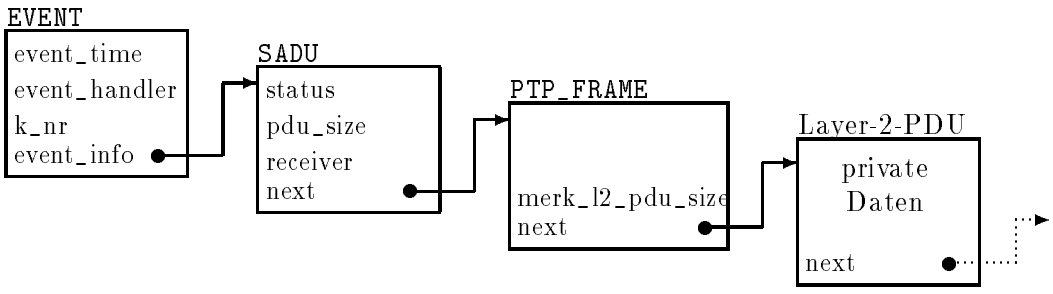
Das übergebene Paket wird in einen Frame verpackt, indem hinter der `SADU` ein `PTP_FRAME` eingekettet wird. Die übergebenen Informationen aus der `SADU` werden für das Auspacken gemerkt und dafür die Informationen über den `PTP_FRAME` eingetragen.

```
void ptp_data_encapsulation(event_p)
EVENT *event_p;
{
  PTP_FRAME *frame_p = get_PTP_FRAME();

  SADU *sadu_p = (SADU *)event_p->event_info;

  frame_p->merk_l2_pdu_size = sadu_p->pdu_size;
  frame_p->next = (PTP_FRAME *)sadu_p->next;

  sadu_p->pdu_size = MAX(sadu_p->pdu_size, net_params.min_data_len)
    + ptp_info.overhead;
  sadu_p->next = (SADU *)frame_p;
```



Dann wird für diesen Frame der Zeitpunkt des Übertragungsendes errechnet und das zugehörige Ereignis eingeplant.

```
  if (event_p->event_time < simul_time)
  {
    fputs("ptp_data_encapsulation(): event_time < simul_time \n",
          stderr);
    event_p->event_time = simul_time;
  }
  event_p->event_time += sadu_p->pdu_size * net_params.itu_per_bit;
```

```
  event_p->event_handler = ptp_end_transmit;

  insert_event(event_p);
}
```

8.1.6 ptp_end_transmit()

`Ptp_end_transmit()` ist die Ereignisbehandlung für das Ende der Übertragung eines Frames. Die Funktion wird über den Funktionszeiger `event_handler` eines `EVENT`s aufgerufen. Sie bekommt als Parameter eine Verkettung `EVENT`→`SADU`→`PTP_FRAME`→Layer-2-PDU, so wie sie in `ptp_data_encapsulation()` erzeugt wurde.

Zunächst werden am Meßpunkt `ptp_info.probept_end_transmit` die konfigurierten Messungen durchgeführt.

```
void ptp_end_transmit(event_p)
EVENT *event_p;
{
  SADU *sadu_p = (SADU *)event_p->event_info;
  PTP_FRAME *frame_p = (PTP_FRAME *)sadu_p->next;

  MEASURE(ptp_info.probept_end_transmit, (VOID_P)event_p);
```

Dann wird die transportierte Nachricht aus dem Frame ausgepackt, indem der `PTP_FRAME` ausgekettet und die Information in der `SADU` wiederhergestellt wird.

```
  sadu_p->next = (SADU *)frame_p->next;

#ifdef DEBUG
  sadu_p->print_next = frame_p->print_next;
#endif

  sadu_p->pdu_size = frame_p->merk_l2_pdu_size;
```

Nach störungsfreier Übertragung wird das ausgepackte Paket an die Empfangsfunktion des Empfängerknotens übergeben. Danach wird in jedem Fall der Sender vom Ende der Übertragung benachrichtigt (dabei werden die übergebenen Datenstrukturen `EVENT`, `SADU`, ..., freigegeben oder wiederverwendet). Zum Schluß wird der benutzte `PTP_FRAME` wieder freigegeben.

```
#ifdef SIMUL_TRANSMIT_ERROR
  if (ptp_stoerzeit
      < simul_time - sadu_p->pdu_size * net_params.itu_per_bit)
#endif
  {
    sadu_p->pdu_size = frame_p->merk_l2_pdu_size;
    (*nodearray[sadu_p->receiver].l2_pdu_received)(event_p);
  }
#ifdef SIMUL_TRANSMIT_ERROR
  else
  {
     /* Bei Übertragungsfehlern bekommt der Empfänger das Paket nicht
        zu sehen */
  }
#endif
```

```
  (*nodearray[event_p->k_nr].l2_pdu_sended)(event_p);

  release_PTP_FRAME(frame_p);
}
```

8.1.7 ptp_transmit_error()

Die Funktion `ptp_transmit_error()` speichert die aktuelle Simulationszeit als Zeit der letzten aufgetretenen Kanalstörung und erzeugt ein neues Störereignis für den Zeitpunkt der nächsten Störung. Falls Übertragungsstörungen simuliert werden sollen, wird am Ende jeder Übertragung in `ptp_end_transmit()` geprüft, ob die letzte Störung während der Übertragung des Frames passiert ist.

```
#ifdef SIMUL_TRANSMIT_ERROR
void ptp_transmit_error(event_p)
EVENT *event_p;
{
  ptp_stoerzeit = simul_time;
  event_p->event_time += (TIME_T)USE_ZUFALL(ptp_info.error_distr);
  insert_event(event_p);
}
#endif
```

In diesem Modell werden von einer Übertragungsstörung die laufenden Übertragungen auf **allen** physikalischen Verbindungen betroffen.

8.1.8 Die Instrumentierung des PTP-Modells

Als Beispiel wird die Messung des Netzdurchsatzes am Meßpunkt in `ptp_end_transmit()` gezeigt.

Der Durchsatz wird für das PTP-Netz insgesamt ermittelt. Da in diesem Modell mehrere Frames gleichzeitig auf verschiedenen physikalischen Verbindungen gesendet werden können, kann der Netzdurchsatz größer als die nominelle Übertragungsrate der Leitungen sein.
Durchsätze für einzelne Knoten können in den höheren Protokollschichten ermittelt werden.

In den Durchsatz geht die PTP-spezifische Kontrollinformation mit ein.

8.1.8.1 ptp_rc_throughput()

Wenn am Meßpunkt `ptp_info.probept_end_transmit` der Durchsatz des PTP-Netzes gemessen werden soll, wird die Funktion `ptp_rc_throughput()` aufgerufen. Sie bekommt einen Zeiger auf eine `PROBE` - Struktur übergeben, die sie initialisieren muß. Es werden die Zeiger auf die Meßfunktion `ptp_throughput()` und die Ergebnis-Ausgabe-Funktion `ptp_mean_throughput()` sowie eine Erläuterung für die Meßergebnisse eingetragen. Falls eine Ausgabedatei für die Meßspur konfiguriert ist, wird eine Überschrift für die laufenden Meßwerte ausgegeben. Die Summe der übertragenen Bits wird mit 0 vorbelegt.

```
void ptp_rc_throughput(probe)
PROBE *probe;
{
  probe->probe_fkt = ptp_throughput;
```

```
  probe->print_result = ptp_mean_throughput;

  if (probe->trace_fd)
    fputs(PpTP-net throughput: \n", probe->trace_fd);

  probe->stat_description = PpTP-Throughput";
  probe->old_val = 0.0;
}
```

8.1.8.2 ptp_throughput()

Die Funktion `ptp_throughput()` wird nach dem Ende jeder Frame-Übertragung mittels `MEASURE()` in `ptp_end_transmit()` aufgerufen. Sie bekommt als Parameter einen Zeiger auf eine `PROBE` - Struktur und einen Zeiger auf den `EVENT` mit der angehängten Beschreibung des übertragenen Frames. Die Summe der übers Netz gesendeten Bits wird in der Komponente `old_val` der `PROBE` aufaddiert.

```
void ptp_throughput(probe, event_p)
PROBE *probe;
EVENT *event_p;
{
  probe->old_val += (double)((SADU *)event_p->event_info)->pdu_size;
```

Das Verhältnis zwischen gesamter simulierter Zeit und Übertragungszeit der gesendeten Frames ergibt in diesem Modell den Durchsatz der Protokollschicht 1. Er wird für eine eventuell durchzuführende statistische Absicherung in der Komponente `akt_val` der `PROBE` abgelegt (in Bits per ITU).

```
  probe->akt_val = probe->old_val / (double)simul_time;
```

Falls für die Meßspur eine Ausgabedatei konfiguriert wurde, wird der ermittelte Durchsatz für die Ausgabe skaliert und als Paar '`<Simulationszeit> <Durchsatz>`' ausgegeben.

```
  if (probe->trace_fd)
    fprintf(probe->trace_fd, "%lg%0lf \n", (double)simul_time,
            (double)probe->akt_val * net_params.itu_per_bit);
}
```

Die Ausgaben entsprechend der in der Konfiguration angegebenen Ausgabeformate sowie der Aufruf der statistischen Absicherung werden durch das Makro `MEASURE()` ausgeführt.

8.1.8.3 ptp_mean_throughput()

Die Funktion `ptp_mean_throughput()` wird nach dem Ende des Simulationslaufes aufgerufen. Falls die Ausgabe gewünscht ist, skaliert sie den (ggf. statistisch abgesicherten) Gesamtdurchsatz und gibt ihn aus.

```
void ptp_mean_throughput(probe)
PROBE *probe;
{
  if (probe->stat_fd)
    fprintf(probe->stat_fd,
       " \nPTP-net utilisation = %4.0lf%% \n \n",
       100.0 * probe->stat_result * net_params.itu_per_bit);
}
```

8.1.9 ptp_print_frame()

Falls simuLAN mit DEBUG - Option übersetzt ist, dient `ptp_print_frame()` dazu, einen `PTP_FRAME` sowie die daran angeketteten Informationen im Klartext auszugeben.

```
#ifdef DEBUG
void ptp_print_frame(frame_p, fd, depth, indent)
PTP_FRAME *frame_p;
FILE *fd;
int depth, indent;
{
  if (fd)
    fprintf(fd, " \n%*sPTP_FRAME: start_time %le, size %d",
            indent, "",
            (double)frame_p->start_time,
            frame_p->merk_l2_pdu_size>>3);

  if (depth-- && frame_p->print_next)
    (*frame_p->print_next)(frame_p->next, fd, depth, indent+4);
}
#endif
```

8.2 Eine lasterzeugende Anwendung : MandelSpawn

> MandelSpawn is distributed in the hope that it will be useful.
> (Andreas Gustafsson, Autor von MandelSpawn)

MandelSpawn ist ein Programm zur verteilten Berechnung der bekannten Mandelbrotmenge ("Apfelmännchen"). Es besteht aus einem Clienten und mehreren Server auf verschiedenen Rechnern. Der Client, auch Master genannt, unterteilt einen rechteckigen Ausschnitt der komplexen Zahlenebene in kleine Stücke und sendet an jeden Server den Auftrag, für eines dieser Stücke die Mandelbrotmenge zu berechnen. Die Server senden die berechneten Daten zurück an den Master, der die Stücke zusammensetzt und auf einem X-Window-Display anzeigt.

Die Kommunikation zwischen dem Clienten und dem Server baut auf dem *User Datagram Protocol* (*UDP*) auf. Wenn ein Server nicht innerhalb einer bestimmten Zeit auf einen Rechenauftrag antwortet, wird er als ausgefallen betrachtet und der Rechenauftrag an einen anderen Server vergeben. Der ausgefallene Server wird jedoch nicht aufgegeben, sondern bekommt weiterhin neue Rechenaufträge. Dadurch können kurzfristig überlastete Server weiterhin ihre Rechenleistung einbringen.

Damit die Server nicht nutzlos an veralteten Aufträgen weiterrechnen, setzen sie vor Beginn der Berechnung ein Alarm - Signal für eine Auszeit. Wenn diese Zeit überschritten wird, wird der Auftrag aufgegeben und der Server neu gestartet.

Als Beispiel für eine lasterzeugende Anwendung in Protokollschicht 5 wird hier die Implementation eine Modells für den MandelSpawn - Serverdämon '`mslaved`' gezeigt. Dabei wird nur auf die Programmierung mit Hilfe der simuLAN - Grundbibliothek eingegangen, nicht aber auf die Validierung und Kalibrierung des Modells. Auch die Instrumentierung wird hier nicht mehr weiter beschrieben.

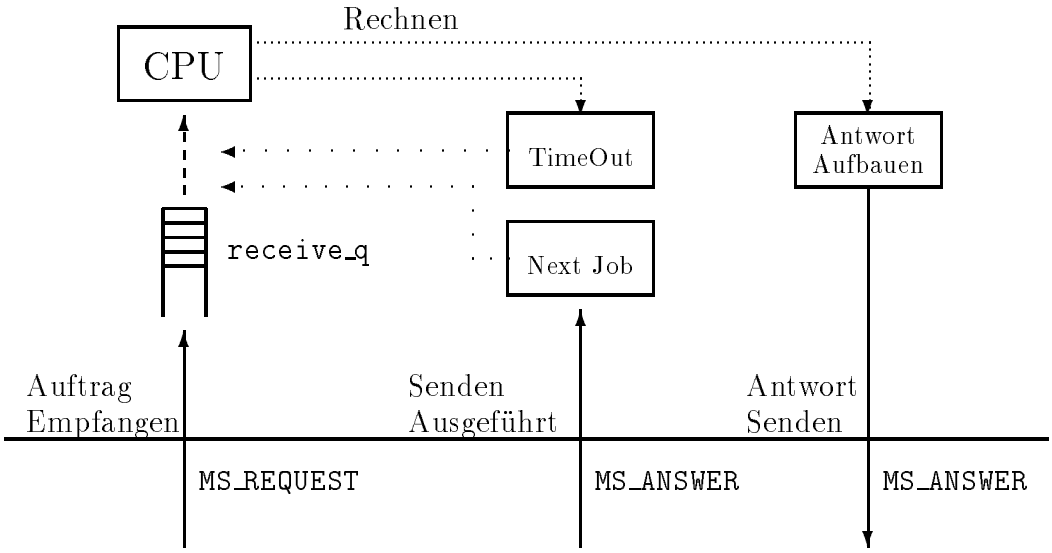


Abbildung 8.1: Ablaufdiagramm des mslaved - Modells

8.2.1 Das mslaved - Modell

Ein neu ankommender Rechenauftrag muß warten, bis alle vorhergehenden Aufträge fertig gerechnet und die Antworten zurückgesendet worden sind. Das Berechnen der Mandelbrotmenge ist sehr CPU - intensiv, stellt kaum Anforderungen an die Speicherverwaltung und keine an die Peripheriegeräte.

Daß ein Auftrag eintrift, bevor der vorhergehende abgearbeitet ist, kann nur passieren, wenn der Master den vorhergehenden Auftrag als überfällig aufgegeben hat.

Wenn die Bearbeitung eines Rechenauftrages beginnt, wird der Zeitpunkt bestimmt, zu dem der Auftrag fertig sein wird. Wenn die Rechenzeit kürzer als die maximal erlaubte Rechenzeit des Servers ist, wird ein EndOfService - Ereignis eingeplant, sonst ein TimeOut - Ereignis. Beim TimeOut wird der aktuelle Auftrag abgebrochen und der nächste aus der Warteschlange geholt. Beim EndOfService - Ereignis wird ein Antwortpaket aufgebaut und an die Dienstzugangsfunktion von Schicht 4 übergeben. Wenn von Schicht 4 die Bestätigung kommt, daß der Sendeauftrag ausgeführt wurde, wird der nächste Rechenauftrag aus der Warteschlange geholt und bearbeitet.

Dieses Modell ist nicht auf ein bestimmtes Transportprotokoll festgelegt, sondern es erwartet nur, daß Nachrichten beliebiger Größe von den tieferen Protokollschichten zum Senden angenommen werden.

8.2.2 Parameter des mslaved - Modells

Parameter dieses Modells sind

- eine Zufallsverteilung für die durchschnittliche Rechenzeit für einen Punkt der komplexen Ebene,
- die maximal erlaubte Rechenzeit des Servers für einen Auftrag,
- und die Größe der Antwortpakete, zusammengesetzt aus einem konstanten Overhead und dem Platz für die errechneten Werte.

Die Angabe der durchschnittlichen Rechenzeit pro Punkt ist eine grobe Annäherung, da die Rechenzeit für die einzelnen Punkte der Ebene sehr unterschiedlich ist, und auch vom gewählten Ausschnitt der Ebene abhängt. Aber für dieses einfache Modell reicht das aus.

8.2.3 Datenstrukturen und Variablen

Im File '`ms.h`' werden die Datenstrukturen für die Kommunikation zwischen Client und Server definiert.

Die `MS_REQUEST` - Struktur beschreibt die Rechenaufträge, die vom Master gesendet werden.
`Npixels` enthält die Anzahl der in diesem Auftrag zu bearbeitenden Punkte der komplexen Ebene.
`Job_nr` wird zur Kennzeichnung der Aufträge benutzt.
`Next` dient zur Verkettung in der Freispeicherliste.

```
typedef struct S_ms_request MS_REQUEST;
struct S_ms_request
{
#ifdef DEBUG
  long object_nr;
#endif
  long job_nr;
  int npixels;
  MS_REQUEST *next;
};
```

`MS_ANSWER` beschreibt die Antwort des Servers.
`Job_nr` wird auf die Nummer des Auftrags gesetzt.
`Master` ist die Knotennummer des Masters, der den Auftrag gesendet hat.

```
typedef struct S_ms_answer MS_ANSWER;
struct S_ms_answer
{
#ifdef DEBUG
  long object_nr;
#endif
  long job_nr;
  int npixels;
  int master;
  MS_ANSWER *next;
};
```

Der Programmtext des Moduls steht in '`mslaved.c`', die Deklarationen der exportierten Funktionen stehen in '`mslaved.h`'.

Zunächst werden die Variablen und Datenstrukturen aus der Systembibliothek und aus der simuLan - Grundbibliothek bekannt gemacht. Mit dem Makro `P` kann das Programm sowohl mit ANSI - C als auch mit älteren Compilern übersetzt werden.

```
#include <stdio.h>
#ifdef USG              /* System 5 ist anders */
#include <string.h>
#else
#include <strings.h>
#endif
```

```
#ifndef P
#ifdef __STDC__
#define P(s) s
typedef void *VOID_P;
#else
#define P(s) ()
typedef char *VOID_P;
#endif
#endif

#include "extern.h"     /* Deklarationen für die Systembibliothek */
#include "defines.h"    /* globale Konstanten für simuLan */
#include "global.h"     /* globale Variablen für simuLan */
#include "events.h"     /* die Eventliste */
#include "rctypes.h"    /* }                                      */
#include "rcparms.h"    /* } für das Einlesen der Konfiguration */
#include "rc.h"         /* }                                      */
#include "ms.h"
#include "simalloc.h"   /* Speicherverwaltung für simuLan */
#include "nodes.h"      /* die NODE - Struktur */
#include "zufall.h"
#include "queues.h"
#include "probepoint.h"
```

Für jeden als Server konfigurierten Knoten wird an den `k_info` - Zeiger der `NODE` - Struktur eine `MSLAVED_INFO` - Struktur gehängt. `Answer_overhead` und `bit_per_pixel` sind eigentlich Parameter des Masters, aber um die Abhängigkeit zwischen den Modulen möglichst gering zu halten, werden sie hier beim Server gespeichert.

```
typedef struct
{
  TIME_T timeout;
  ZUFALL_INFO itu_per_pixel;
  int answer_overhead;
  int bit_per_pixel;
  NQUEUE receive_q;
  PROBEPOINT end_of_service_probept;
} MSLAVED_INFO;
```

Mit dem Makro `DEFUN_GET_REL_OBJECT()` werden die Funktionen `get_MS_ANSWER()` und `release_MS_ANSWER()` sowie der Anker der Freispeicherliste, `MS_ANSWER_freelist`, angelegt.

```
DEFUN_GET_REL_OBJECT(MS_ANSWER)
```

Die Funktionen des Moduls werden vor ihrer Benutzung deklariert.

```
void mslaved_gwc42 P((MSLAVED_INFO *, int indent));

void mslaved_timeout P((EVENT *));
void mslaved_endofservice P((EVENT *));

void mslaved_pdu_received P((EVENT *));
void mslaved_pdu_sended P((EVENT *));

void mslaved_print_info P((MSLAVED_INFO *, FILE *, int flags, int indent));
#ifdef DEBUG
void print_ms_answer P((MS_ANSWER *, FILE *, int depth, int indent));
#endif
```

Auf die Meßfunktionen für `end_of_service_probept` wird hier nicht weiter eingegangen.

```
FKT_TABLE mslaved_e_o_s_probefkt_tab[] =
{ { NULL, NULL, NULL } };
```

Die folgende Tabelle enthält Funktionen, die in eine `MSLAVED_INFO` - Struktur die Parameter für einen bestimmten Rechnertyp eintragen.

```
FKT_TABLE mslaved_type_tab[] =
{
  { "gwc42",
    (void (*)())mslaved_gwc42,
    "Generic Workstations Company's Model 42"
  },
  { NULL, NULL, NULL }
};
```

Der `mslaved_rc_info` ist nötig, damit die folgende Tabelle statisch initialisiert werden kann.

```
MSLAVED_INFO mslaved_rc_info;
```

In der folgenden Tabelle sind die für das mslaved - Modell wählbaren Parameter verzeichnet (siehe Abschnitt A.15.10 [rcreadkeywds()], Seite 93).

```
#define M(x) (VOID_P)&mslaved_rc_info.x

KEYWD_TABLE mslaved_rc_table[] =
{
  { "Type",
    (VOID_P)&mslaved_rc_info,
    mslaved_type_tab,
    XXTYPnamedhook,
    "select Machine-Type of Server" },
  { "timeout", M(timeout), NULL, XXTYPtimet, "JOB TIMEOUT" },
  { "time_per_pixel",
    M(bit_per_pixel),
    distr_tab,
    XXTYPnamedhook,
    "distribution for CPU time per pixel" },
  { "answer_overhead",
    M(answer_overhead),
    NULL,
    XXTYPbits,
    "constant Overhead for answer packets" },
  { "byte_per_pixel",
    M(bit_per_pixel),
    NULL,
    XXTYPbits,
    "size of a pixel in answer packets" },
  { "end_of_service_probept",
    M(end_of_service_probept),
    mslaved_e_o_s_probefkt_tab,
    XXTYPprobe,
    "" },
  { "ENDmslaved",
    NULL,
```

```
    NULL,
    XXTYPend,
    "end of mslaved-description" },
  { NULL, NULL, NULL, XXTYPundef, NULL }
};
#undef M
```

8.2.4 mslaved_rc()

Die Funktion `mslaved_rc()` liest die Parameter für das mslaved - Modell aus der Konfigurationsdatei ein. Sie bekommt als Parameter einen Zeiger auf ein Array von auszufüllenden `NODE` - Strukturen, die Länge dieses Arrays, und eine Einrücktiefe für eventuelle erläuternde Ausgaben.

Dies ist die einzige Funktion des Moduls, die dem "Grundgerüst" von simuLAn namentlich bekannt gegeben werden muß. Sie wird im File '`tables.c`' in die Tabelle `l1models_tab` eingetragen.

```
*** simuLAn/src/simoduLe/tables.c-old  Tue Jul 30 11:14:20 1991
--- simuLAn/src/simoduLe/tables.c      Wed Jul 31 05:12:57 1991
***************
*** 31,36 ****
--- 31,37 ----
  #include "k0.h"
  #include "k1.h"
  #include "k1_rc.h"
+ #include "mslaved.h"

  #include "l2simple.h"

*************** FKT_TABLE l5models_tab[] =
*** 124,129 ****
--- 125,131 ----
  {
    { "k0",      k0_rc,         "Simple Statistical Model" },
    { "k1",      k1_rc,         Central Server" },
+   { "mslaved", mslaved_rc,    MandelSpawn Server Daemon" },
    { NULL,      NULL,          NULL }
  };
```

Wenn simuLAn mit der Option '`-x`' aufgerufen wird, soll eine Schablone für die Konfigurationsdatei ausgegeben werden. Dazu dient die Funktion `rcexplainkeywds()`.

```
void mslaved_rc(nodes, count, indent)
NODE *nodes;
int count, indent;
{
  MSLAVED_INFO *info_p;

  if (minus_x)
  {
    fputs(" \n \n## Parameters for mslaved - model \n", outfile);
    rcexplainkeywds(mslaved_rc_table, indent);
    return;
  }
```

Anderenfalls wird zunächst `mslaved_rc_info` mit Defaultwerten vorbelegt.

```
mslaved_rc_info.timeout = 60 * glob_params.itu_per_second;
mslaved_rc_info.itu_per_pixel.zufall_fkt = NULL;
mslaved_rc_info.itu_per_pixel.param_p = NULL;
mslaved_rc_info.answer_overhead = 20 * 8;       /* 20 Byte */
mslaved_rc_info.bit_per_pixel = 2 * 8;
mslaved_rc_info.end_of_service_probept = NULL;
```

Danach werden die Parameter aus der Konfigurationsdatei gelesen. Falls simuLan mit der Option '`-p`' aufgerufen wurde, werden dabei auch die eingelesenen Werte sowie eingesetzte Defaultwerte ausgegeben.

```
rcreadkeywds(mslaved_rc_table, indent);
```

Die eingelesenen Parameter werden auf Plausibilität geprüft.

```
if (!mslaved_rc_info.itu_per_pixel.zufall_fkt)
  RC_ERROR("MSLAVED1: No distribution for itu_per_pixel", 0, 0);
```

Schießlich werden von dem nach `mslaved_rc_info` eingelesenen Parametersatz `count` Kopien in die `nodes` eingetragen. Für das mslaved - Modell werden keine Initialisierungs- und Aufräumfunktionen gebraucht, und der Empfänger der erzeugten Nachricht ist immer der Rechner, von dem aus der Rechenauftrag gesendet wurde.

```
while (count--)
{
  nodes[count].print_k_info = mslaved_print_info;
  nodes[count].k_init = NULL;
  nodes[count].k_cleanup = NULL;
  nodes[count].calc_empfaenger = NULL;
  nodes[count].traffic_list = NULL;
  nodes[count].l5_pdu_received = mslaved_pdu_received;
  nodes[count].l5_pdu_sended = mslaved_pdu_sended;

  if (!(info_p
        = (MSLAVED_INFO *)myalloc((unsigned int)1,
                                  sizeof(MSLAVED_INFO))))
    RC_ERROR("MSLAVED2: myalloc(MSLAVED_INFO)", 0, 0);

  nodes[count].k_info = (VOID_P)info_p;

  *info_p = mslaved_rc_info;     /* struct-assignment ! */
```

Die Empfangswarteschlange wird als "Normale Warteschlange" mit einem Bediener angelegt, und es sollen laufend die durchschnittliche Schlangenlänge und Wartezeit ermittelt werden.

```
  (void)make_nqueue(&info_p->receive_q, "mslaved CPU Q", 1, 1);
```

Falls für den Meßpunkt Messungen konfiguriert wurden, muß die an dem `PROBEPOINT` hängende Liste von `PROBE` - Strukturen vervielfältigt werden.

```
  if (count>1)
    info_p->end_of_service_probept
       = copy_probepoint(mslaved_rc_info.end_of_service_probept);
}
}
```

8.2.5 mslaved_gwc42()

Die Funktion `mslaved_gwc42()` füllt eine `MSLAVED_INFO` - Struktur mit Parameterwerten für die Simulation eines Rechners vom Typ *GWC Model 42* der *Generic Workstations Company*.

```
void mslaved_gwc42(info_p, indent)
MSLAVED_INFO *info_p;
int indent;
{
  info_p->itu_per_pixel.zufall_fkt = expntl;
  info_p->itu_per_pixel.param_p
    = (double *)myalloc((unsigned)1, sizeof(double));
  if (!info_p->itu_per_pixel.param_p)
    RC_ERROR("MSLAVED3: myalloc(double): errno %d", errno, 0);
}
```

8.2.6 mslaved_pdu_received()

Die Funktion `mslaved_pdu_received()` wird über den Funktionszeiger `l5_pdu_received` der `NODE` - Struktur aufgerufen, wenn ein Rechenauftrag für den Knoten komplett von der nächsttieferen Protokollschicht empfangen wurde. Sie bekommt als Parameter eine `EVENT` - Struktur, an der eine `SADU` - Struktur hängt. An diese wiederum ist ein `MS_REQUEST` gekettet, der den Rechenauftrag beschreibt.

Aus den empfangenen Datenstrukturen müssen die relevanten Informationen kopiert und gemerkt werden, weil die Datenstrukturen vom Sender reserviert wurden und von diesem auch wieder freigegeben werden.

Der empfangene Rechenauftrag wird in die Empfangswarteschlange eingereiht. Falls noch ein vorhergehender Auftrag bearbeitet wird, muß der neu angekommene warten, sonst wird er sofort berechnet.

Dieses Empfangsverfahren weicht von der Wirklichkeit ab. In der Realität werden die Rechenaufträge (Datagramme) in der nächsttieferen Schicht in eine Warteschlange gestellt und vom mslaved - Prozeß einzeln abgenommen. Durch die in simulan vereinbarten Schnittstellen zwischen den Protokollschichtmodulenist es jedoch einfacher, eine empfangene PDU aktiv an die nächsthöhere Schicht zu übergeben, statt daß diese sie abholt. Die Funktionalität und die Kenngrößen des Modells ändern sich dadurch aber nicht.

```
void mslaved_pdu_received(event_p)
EVENT *event_p;
{
  SADU *sadu_p = (SADU *)event_p->event_info;
  MS_REQUEST *msrequest = (MS_REQUEST *)sadu_p->next;

  int k_nr = sadu_p->receiver;
  NODE *node = nodearray+k_nr;
  MSLAVED_INFO *info_p = (MSLAVED_INFO *)node->k_info;

  EVENT *e = get_EVENT();
  MS_ANSWER *msanswer = get_MS_ANSWER();

  int hilf;

  e->k_nr = sadu_p->receiver;
  e->event_info = (VOID_P)msanswer;
```

```
#ifdef DEBUG
  e->print_event_info = print_ms_answer;
#endif

  msanswer->job_nr = msrequest->job_nr;
  msanswer->npixels = msrequest->npixels;
  msanswer->master = event_p->k_nr;

  hilf = nq_push_fifo(&info_p->receive_q, e);
  if (!hilf)
  {
```

Es ist kein alter Rechenauftrag mehr vorhanden. Jetzt wird der Zeitaufwand für diesen Auftrag ermittelt. Wenn er größer als die maximal erlaubte Rechenzeit pro Auftrag dieses Servers ist, wird für den Zeitpunkt des Abbruchs ein TimeOut - Ereignis verursacht, sonst wird ein EndOfService - Ereignis eingeplant. Die Berechnung des Zeitpunktes dieses Ereignisses muß bei der Validierung und Kalibrierung des Modells wahrscheinlich noch korrigiert werden, um einen realitätsnahen Rechenzeitverbrauch zu simulieren. (Diese Berechnung wird auch in den Funktionen `mslaved_endofservice()` und `mslaved_timeout()` durchgeführt.)

```
    TIME_T end_of_service_time = simul_time
                               + USE_ZUFALL(info_p->itu_per_pixel)
                                 * msanswer->npixels;
    if (end_of_service_time > simul_time + info_p->timeout)
    {
      e->event_handler = mslaved_timeout;
      e->event_time = simul_time + info_p->timeout;
    }
    else
    {
      e->event_handler = mslaved_endofservice;
      e->event_time = end_of_service_time;
    }
    insert_event(e);
  }
  else if (-1 == hilf)
  {
```

Die maximal zulässige Schlangenlänge `glob_params.max_q_len` wird überschritten. Der Simulationslauf wird mit einer Fehlermeldung abgebrochen.

```
    END_OF_SIMULATION(MAX_Q_LEN_EXCEEDED,
                      "receive_q overflow in mslaved %d",
                      e->k_nr, 0);
    return;
  }
```

Anderenfalls muß der Rechenauftrag warten.

```
  else
  {
     /* NOP */
  }
}
```

8.2.7 mslaved_endofservice()

Die Funktion `mslaved_endofservice()` wird als Ereignisbehandlung aufgerufen, wenn ein Rechenauftrag fertig bearbeitet wurde. Sie bekommt als Parameter einen `EVENT` mit daran hängender `MS_ANSWER` - Struktur. Sie muß daraus eine Antwort für den Master zusammenstellen und an die nächsttiefere Protokollschicht zum Senden übergeben.

```
void mslaved_endofservice(event_p)
EVENT *event_p;
{
  MS_ANSWER *msanswer = (MS_ANSWER *)event_p->event_info;
  SADU *sadu_p = get_SADU();

  sadu_p->next = (SADU *)msanswer;
  sadu_p->pdu_size = msanswer->npixels;
  sadu_p->status = OK;
  sadu_p->receiver = msanswer->master;
#ifdef DEBUG
  sadu_p->print_next = print_ms_answer;
#endif

  event_p->event_info = (VOID_P)sadu_p;
#ifdef DEBUG
  event_p->print_event_info = print_sadu;
#endif

  (*nodearray[msanswer->master].l4_sap)(event_p);
}
```

8.2.8 mslaved_timeout()

Die Funktion `mslaved_timeout()` wird als Ereignisbehandlung aufgerufen, wenn ein Rechenauftrag im Server als veraltet abgebrochen wird. Sie bekommt als Parameter einen `EVENT` mit daran hängender `MS_ANSWER` - Struktur. Die Datenstrukturen werden freigegeben. Falls vorhanden, wird der nächste Auftrag aus der Warteschlange entnommen und wie in `mslaved_pdu_received()` das Bearbeitungende eingeplant.

```
void mslaved_timeout(event_p)
EVENT *event_p;
{
  MS_ANSWER *msanswer = (MS_ANSWER *)event_p->event_info;

  int k_nr = event_p->k_nr;
  NODE *node = nodearray + k_nr;
  MSLAVED_INFO *info_p = (MSLAVED_INFO *)node->k_info;
  TIME_T end_of_service_time;

  release_MS_ANSWER(msanswer);
  release_EVENT(event_p);

  if (event_p = nq_pop(&info_p->receive_q))
  {
    end_of_service_time = simul_time
                        + USE_ZUFALL(info_p->itu_per_pixel)
                          * msanswer->npixels;
    if (end_of_service_time > simul_time + info_p->timeout)
    {
```

```
      event_p->event_handler = mslaved_timeout;
      event_p->event_time = simul_time + info_p->timeout;
    }
    else
    {
      event_p->event_handler = mslaved_endofservice;
      event_p->event_time = end_of_service_time;
    }
    insert_event(event_p);
  }
}
```

8.2.9 mslaved_pdu_sended()

Die Funktion `mslaved_pdu_sended()` wird von der nächsttieferen Protokollschicht über den Funktionszeiger `l5_pdu_sended` des `NODE`s aufgerufen, um dem mslaved - Modell mitzuteilen, daß ein Sendeauftrag ausgeführt worden ist. Sie bekommt als Parameter eine `EVENT` - Struktur, an der eine `SADU` - Struktur hängt. An diese wiederum ist ein `MS_ANSWER` gekettet. Die Datenstrukturen werden freigegeben und, falls vorhanden, der nächste Auftrag aus der Warteschlange entnommen und wie in `mslaved_pdu_received()` das Bearbeitungende eingeplant.

```
void mslaved_pdu_sended(event_p)
EVENT *event_p;
{
  SADU *sadu_p = (SADU *)event_p->event_info;
  MS_ANSWER *msanswer = (MS_ANSWER *)sadu_p->next;

  int k_nr = event_p->k_nr;
  NODE *node = nodearray + k_nr;
  MSLAVED_INFO *info_p = (MSLAVED_INFO *)node->k_info;
  TIME_T end_of_service_time;

  release_MS_ANSWER(msanswer);
  release_SADU(sadu_p);
  release_EVENT(event_p);

  if (event_p = nq_pop(&info_p->receive_q))
  {
    end_of_service_time = simul_time
                        + USE_ZUFALL(info_p->itu_per_pixel)
                          * msanswer->npixels;
    if (end_of_service_time > simul_time + info_p->timeout)
    {
      event_p->event_handler = mslaved_timeout;
      event_p->event_time = simul_time + info_p->timeout;
    }
    else
    {
      event_p->event_handler = mslaved_endofservice;
      event_p->event_time = end_of_service_time;
    }
    insert_event(event_p);
  }
}
```

8.2.10 mslaved_print_info()

`Mslaved_print_info()` druckt den Inhalt einer `MSLAVED_INFO` - Struktur aus. Der Parameter `flags` gibt an, wie ausführlich die Ausgaben sein sollen.

```
void mslaved_print_info(info_p, fd, flags, indent)
MSLAVED_INFO *info_p;
FILE *fd;
int flags, indent;
{
  if (flags & (PRINT_QUEUES | PRINT_ALL))
    print_nqueue(&info_p->receive_q, fd, 0, indent);
}
```

8.2.11 print_ms_answer()

Falls simuLAn mit DEBUG - Option übersetzt ist, dient `print_ms_answer()` dazu, eine `MS_ANSWER` - Struktur im Klartext auszugeben.

```
#ifdef DEBUG
void print_ms_answer(msanswer, fd, depth, indent)
MS_ANSWER *msanswer;
FILE *fd;
int depth, indent;
{
  if (fd)
    fprintf(fd,
            " \n%*MS_ANSWER: object_nr %d job_nr %d npixels %d master %d",
            indent, "",
            msanswer->object_nr,
            msanswer->job_nr,
            msanswer->npixels,
            msanswer->master);
}
#endif
```

8.3 Protokollschichtmodule für die Schichten 2, 3, 4

Skelette für neue Protokollschichtmodule können aus existierenden Modulen einfach mit `sed(1)` erzeugt werden. Prototypen für die einzelnen Protokollschichten sind :

- Schicht 5 : Das Modell '`KO`' simuliert eine einfache Anwendung, die jeweils nach einer Denkzeit eine Nachricht erzeugt und an einen Empfänger sendet.
- Schicht 2 – 4 : Das Modell '`l2simple`' simuliert ein einfaches Protokollschichtmodell, daß dafür sorgt, daß ein Knoten keine zeitlich überlappenden Pakete an Protokollschicht 1 übergibt. Dazu wird für jeden Knoten eine Warteschlange (`NQUEUE`) erzeugt, in der Sendeaufträge warten bis alle vorhergehenden Pakete bearbeitet wurden.
- Schicht 1 : Das PTP-Modell bildet ein minimales Modell für Protokollschicht 1.

Ein Beispiel für die Benutzung des `sed(1)` (in `csh(1)` - Syntax) :

```
% sed -e 's/l2simple/newmodel/g' -e 's/L2simple/Newmodel/g'  \
> -e 's/L2SIMPLE/NEWMODEL/g' l2simple.c > newmodel.c
% sed -e 's/l2simple/newmodel/g' -e 's/L2simple/Newmodel/g'  \
> -e 's/L2SIMPLE/NEWMODEL/g' l2simple.h > newmodel.h
```

Kapitel 9

Ausblicke

simulan ist ein Werkzeug, das für die Simulation lokaler Netze hohe Flexibilität und Performanz bietet. Der Benutzer kann Netzwerkmodelle, wie bei einem Baukastensystem, aus implementierten Modellkomponenten zusammensetzen. Daß alle Komponenten in C implementiert sind, ist Vorteil und Nachteil zugleich. Einerseits können alle Möglichkeiten dieser Sprache für die Programmierung neuer "Bauelemente" ausgenutzt werden, andererseits sind die Komponenten komplexer als bei Verwendung einer speziellen Simulationssprache. Diese Nachteil wird dadurch ausgeglichen, daß simulan eine Bibliothek von Funktionen für Warteschlangen, Eventliste, Einlesen der Konfiguration, Intrumentierung, ..., zur Verfügung stellt, mit der Operationen möglich sind, die Simulationssprachen nicht anbieten (z.B. Löschen bereits eingeplanter Ereignisse).

Die Verwendung eines dynamischen Laders [HO90] könnte die Programmierung neuer Modellkomponenten weiter vereinfachen. Neu erstellte Module bräuchten dann nicht mehr in Tabellen eingetragen werden, um sie dem "Grundgerüst" von simulan bekanntzugeben, sondern würden erst bei Bedarf zur Laufzeit in den Prozeß geladen.

Dadurch, daß Ausgabeformate in der Konfigurationsdatei gewählt werden können, können für die Auswertung der Meßspuren beliebige Nachbereitungsprogramme benutzt werden (z.B. `plot(1)`, `graphics(1)` [ATT], `gnuplot(1)` [WKC+]). Es erscheint wünschenswert, eine Standardformatierung für die Ausgaben vorzugeben. Es ist jedoch noch unklar, inwieweit dies machbar ist, ohne sich auf ein bestimmtes Auswertungsprogramm festzulegen.

Interaktive Fehlersuche und funktionale Analyse durch gezieltes Verfolgen bestimmter Datenpakete und Ereignisse ist möglich, aber noch nicht ausreichend benutzerfreundlich.

simulan kann eine kommentierte Schablone für die Konfigurationsdatei ausgeben, aus der mit einem Texteditor eine Modellbeschreibung erzeugt werden kann. Damit die Modellbeschreibungen von einem Programm generiert werden können (z.B. graphischer Editor, Konfigurationsexpertensystem), ohne daß die Eingabesyntax von simulan in dem Generatorprogramm verankert werden braucht, müßte die Schablone in maschinenlesbarer Form ausgegeben werden, z.B. als Backus-Naur-Grammatik.

Für die Protokollschichten wurden bisher einige beispielhafte Modelle implementiert. Für Protokollschicht 1 existieren ein CSMA - Modell und ein einfaches Point-To-Point - Modell, für Protokollschicht 2 ein einfaches Modell, das den Zugang eines Knotens zur Schicht 1 serialisiert. Für Protokollschicht 3 wurde ein IP - Modell realisiert, für Schicht 5 verschiedene Modelle (einfacher) Anwendungen. Benötigt werden noch weitere Modelle für Standard - Protokolle, z.B. Token Ring, TCP, TC4, sowie Modelle realer Anwendungen.

Die dynamische Aggregierung könnte eine Verkürzung der Rechenzeit bei Verwendung komplexer Modellkomponenten ergeben. Die Schwierigkeit dabei ist die Prüfung, ob und wann die komplexe Komponente durch eine einfachere ersetzt werden kann.

Anhang A

simuLan Programmierhandbuch

In diesem Anhang werden die Datenstrukturen und Funktionen beschrieben, die in der Bibliothek von simuLan für die Programmierung neuer Modellkomponenten zur Verfügung stehen.

A.1 Allgemeine Datentypen

A.1.1 Simulierte Zeit: `TIME_T`

Die interne Zeiteinheit von simuLan ist *ITU* (*Indivisible Time Unit*). Sie kennt nur ganzzahlige Werte. Als Datentyp dafür wird `TIME_T` benutzt. In vielen Fällen ist ein sehr großer Wertebereich bei sehr feiner Zeitauflösung erforderlich, so daß die üblichen 32-Bit-Integers nicht ausreichen. Falls der verwendete Compiler sie zur Verfügung stellt, sollten 64-Bit-Integers benutzt werden (`long long` in GNU - C), sonst muß auf `double`s zurückgegriffen werden (üblicherweise 53 Bit Genauigkeit). (In [SD91] wird für die Simulation des Ethernet eine Zeitauflösung von 1ns gebraucht, bei 31 Bit Genauigkeit läßt sich so ein Zeitraum von 2 sec erfassen).

Die Größe der ITU wird in der Konfiguration als ITU pro Sekunde angegeben (siehe Abschnitt 5.4.2 [*Konfiguration der globalen Simulationsparameter*], Seite 27).

A.1.2 Zeiger auf unbekannte Objekte: `VOID_P`

In simuLan werden häufig Zeiger auf unbekannte Objekte benutzt. ANSI-C bietet dafür den Datentyp `void*` an, bei älteren Compilern muß jedoch `char*` benutzt werden. Deshalb wird in simuLan der Typ `VOID_P` benutzt und, abhängig vom Preprocessor-Symbol `__STDC__`, als `void*` oder als `char*` definiert.

A.2 Allgemeine Variablen

```
extern TIME_T simul_time;
extern long object_number;
```

`Simul_time` enthält immer die aktuelle simulierte Uhrzeit in ITU (siehe Abschnitt A.1.1 [`TIME_T`], Seite 71). Wenn mit der Simulation eines neuen Ereignisses begonnen wird, wird `simul_time` auf die Zeit dieses Ereignisses gesetzt.

`Object_number` kann benutzt werden, um Datenobjekte mit einer eindeutigen Kennzeichnung zu versehen. Das ist besonders für die Fehlersuche nützlich, wird aber z.B. auch gebraucht, um die Einzelteile einer zerstückelten Nachricht als zusammengehörig zu identifizieren (siehe Abschnitt A.16 [Debugging], Seite 97).

A.3 Allgemeine Makros

```
END_OF_SIMULATION(int reason, char *message, m2, m3)
```

Mit dem Makro `END_OF_SIMULATION` gibt ein Modul bekannt, daß der Simulationlauf beendet werden soll.
`Reason` gibt den Grund dafür an. Es sollte eine der folgenden Konstanten sein:

`STATISTICS_READY`, `STATISTICS_STOP`, `MAX_Q_LEN_EXCEEDED`, `IO_ERROR`, `OUT_OF_MEM`, `GENERAL_ERROR`

`Message`, `m1` und `m2` werden als zweiter, dritter und vierter Parameter an `fprintf(3)` übergeben, falls `message` ungleich `NULL` ist.

`END_OF_SIMULATION` setzt die globale Variable `events_to_go` auf *0*, um anzuzeigen, daß keine Ereignisse mehr bearbeitet werden sollen. Der alte Wert von `events_to_go` wird für die Auswertung der Abbruchursache in `saved_events_to_go` gerettet. Diese Variablen sollten in den Protokollschichtmodulen nicht direkt benutzt werden, sondern nur über das Makro.

A.4 Auswertung der Kommandozeile

```
extern char **global_argv;
extern int global_argc;

extern int minus_x;
extern int printconf;
extern int dont_run_it;

#ifdef DEBUG
extern long rc_debuglevel;
#endif /* DEBUG */
```

Die Option '`-x`*n*' bedeutet, daß keine Simulation durchgeführt sondern eine Erläuterung der Syntax der Konfigurationsdatei ausgegeben werden soll. Die globale Variable `minus_x` bekommt den Wert *n* zugewiesen (Default ist 1). Wenn *n* größer als eins ist, werden zur Syntax ausführlichere Kommentare ausgegeben.

Die Option '`-n`' bedeutet, daß die Konfigurationsdatei gelesen, aber kein Simulationslauf durchgeführt werden soll. Wenn diese Option angegeben ist, wird die globale Variable `dont_run_it` ungleich null gesetzt.

Die Option '`-p`' gibt an, daß die aus der Konfiguration gelesenen Parameter einschließlich eventuell eingesetzter Defaultwerte ausgegeben werden sollen. Wenn diese Option angegeben ist, wird die globale Variable `printconf` ungleich null gesetzt. Die Option '`-p`' zusammen mit '`-n`' eignet sich, um die Korrektheit einer Konfigurationsdatei zu überprüfen.

Ausgaben werden normalerweise in das durch die Variable `outfile` angegebene `FILE` geschrieben. Dieses ist identisch mit `stdout`, falls nicht mit der Option '`-o`' anders angegeben.

In der Kommandozeile (Option '`-f <file-ident> <filename>`' und im Vorspann der Konfigurationsdatei können Zuordnungen zwischen symbolischen File-Identifiern und Filenamen des Betriebssytems angegeben werden (siehe Abschnitt 5.3.1 [*Definition von Ausgabekanälen*], Seite 25). Von simula werden die symbolischen Filenamen '`NIL`', '`stdout`', '`stderr`' und '`outfile`' vordefiniert. Der symbolische Filename '`NIL`' gibt an, daß keine Ausgaben gemacht werden sollen, der dazugehörige `FILE` - Zeiger ist `NULL`.

Mit '`-D<modulename>=<flag>`' können Testausgaben einzelner Module aktiviert werden. Die Auswertung dieser Optionen muß von den jeweiligen Modulen selber durchgeführt werden (siehe Abschnitt A.16.1 [Debugflags], Seite 97).
Testausgaben während des Einlesens der Konfiguration können aktiviert werden durch '`-DRC=`*n*'. Je größer die Zahl *n* ist, desto detaillierter werden die Ausgaben. `Rc_debuglevel` enthält den angegebenen Zahlenwert (Maximum 9).

Das letzte Argument wird als Name der Konfigurationsdatei benutzt. Wenn keine Datei angegeben ist, wird von `stdin` gelesen. Das ist insbesondere dann der Fall, wenn simula ganz ohne Parameter aufgerufen wird.

Fehlermeldungen und Debugausgaben werden immer nach `stderr` geschrieben.

A.5 Globale Simulationsparameter

```
extern struct S_glob_params
{
  double itu_per_second;
  unsigned long max_events;
  double max_simul_time;
  double max_cpu_time;
  int max_q_length;
  int random_seed;
  int knoten_anzahl;
  PROBEPOINT itimer_probept;
  double itimer_intervall;
  PROBEPOINT cronic_probept;
  TIME_T cronic_intervall;

} glob_params;
```

`Itu_per_second` enthält die Anzahl der "Indivisible Time Units", in die eine Sekunde in der Simulation unterteilt wird. (siehe Abschnitt A.1.1 [`TIME_T`], Seite 71).

`Max_events` gibt an, wieviele Ereignisse maximal bearbeitet werden dürfen. Wenn dieser Wert überschritten wird, wird der Simulationslauf abgebrochen und die bis dahin gesammelten Ergebnisse ausgewertet.

`Max_simul_time` gibt das Ende des zu simulierenden Zeitabschnitts an. Wenn dieser Wert überschritten wird, wird der Simulationslauf abgebrochen und die bis dahin gesammelten Ergebnisse ausgewertet.

`Max_cpu_time` gibt die maximale Rechenzeit an, die für den Programmlauf verbraucht werden darf (in CPU-Sekunden). Wenn dieser Wert überschritten wird, wird der Simu-

lationslauf abgebrochen und die bis dahin gesammelten Ergebnisse ausgewertet. Die Rechenzeit wird mittels `setitimer(2)` durch den Intervallgeber für die virtuelle Rechenzeit `ITIMER_VIRTUAL` überwacht.

`Max_q_length` gibt die maximal zulässige Länge für die Warteschlangen an. Wenn dieser Wert überschritten wird, wird der Simulationslauf abgebrochen und die bis dahin gesammelten Ergebnisse ausgewertet (siehe Abschnitt A.8 [Warteschlangen], Seite 78).

`Random_seed` gibt den Startwert für den Pseudozufallsgenerator an. Wenn `random_seed` gleich `-1` ist, wird die von `time(2)` gelieferte Zeit als Startwert benutzt.

`Knoten_anzahl` enthält die Anzahl der simulierten Netzknoten.

`Itimer_probept` zeigt auf eine Liste von `PROBE` - Strukturen, mittels derer Meßwerte in regelmäßigen Abständen realer Zeit ausgegeben werden können.

`Itimer_intervall` gibt das Intervall für regelmäßig ausgelöste `SIGALRM`-Signale (in Mikrosekunden) an, mittels derer diese Ausgaben angestoßen werden (siehe `setitimer(2)`).

`Cronic_probept` bietet die Möglichkeit, Ausgaben in regelmäßigen Abständen der *simulierten* Zeit anzustoßen. Dafür wird periodisch ein spezielles Meß-Ereignis ausgelöst (*cronic event*). `Cronic_intervall` gibt das Intervall dafür in ITU an.

A.6 Speicherverwaltung

Auf `malloc(3)` wurde eine eigene Speicherverwaltung aufgesetzt, damit am Ende jedes Simulationslaufes der reservierte Speicherplatz einfach wieder freigegeben werden kann.

A.6.1 `myalloc()`, `myrealloc()`, `myfree()`

```
VOID_P myalloc(unsigned int anzahl, unsigned int groesse);
VOID_P myrealloc(MYALLOC_MAGIC **magic, unsigned int bytes);
void myfreeall(void);
```

`Myalloc()` liefert einen Zeiger auf einen Speicherbereich von mindestens `anzahl*groesse` Bytes (analog zu `calloc(3)`). `Myalloc()` reserviert Speicher mittels `calloc(3)`, die Speicherbereiche sind somit für beliebige Benutzung passend ausgerichtet. Der zurückgelieferte Zeiger darf jedoch **nicht** an `free(3)` oder `realloc(3)` übergeben werden, da `myalloc()` intern über den von ihm beschafften Speicher Buch führt.

`Myfreeall()` gibt sämtlichen von `myalloc()` oder `myrealloc()` beschafften Speicher mittels `free(3)` wieder frei. Die Funktion wird am Ende jedes Simulationsdurchlaufes in `main()` aufgerufen.

`Myrealloc()` bekommt als Parameter eine Zweifachreferenz auf eine `MYALLOC_MAGIC` - Struktur und eine Größenangabe. Die Struktur enthält einen Zeiger auf einen bereits durch `myrealloc()` beschafften Speicherbereich, der mittels `realloc(3)` auf mindestens `bytes` Bytes vergrößert wird. Als Rückgabe liefert `myrealloc()` den Zeiger auf den neuen Speicherbereich, außerdem wird an `magic` ein Zeiger auf die korrigierte `MYALLOC_MAGIC` - Struktur angehängt. Falls kein Speicher verfügbar ist, wird `NULL` zurückgegeben. Beim ersten Aufruf muß `magic` auf einen `NULL` - Zeiger zeigen, damit `myrealloc()` ganz neuen Speicherplatz beschafft. Für den Aufrufer sind Aufbau und Inhalt dieser Struktur uninteressant, er muß sich nur den Zeiger sorgfältig merken.

Beispiel :

```
USR_OBJ *user_array;
int user_array_size;
MYALLOC_MAGIC *user_array_magic;

user_array_magic = NULL;     /* Initialisierung */

while (work_to_do)
{
  user_array_size = new_nr_of_objects();
  user_array = (USER_OBJ *)myrealloc(&user_array_magic,
                                     user_array_size * sizeof(USER_OBJ));
  work_with(user_array);
}
```

A.6.2 mystrdup()

```
char *mystrdup(char *s);
```

`Mystrdup()` kopiert den übergebenen String in einen per `myalloc()` beschafften Speicherbereich und gibt einen Zeiger darauf zurück. `Mystrdup()` kann als Makro oder als Funktion implementiert sein.

A.6.3 Generische Freilisten: DEFUN_GET_REL_OBJECT()

```
DEFUN_GET_REL_OBJECT(TYP)

TYP *get_TYP(void);
void release_TYP(TYP *);
extern TYP *TYP_freelist;
```

Es kommt oft vor, daß viele gleichartige Objekte abwechselnd reserviert und wieder freigegeben werden müssen (Warteschlangenelemente, Frame - Deskriptoren).

Das Makro `DEFUN_GET_REL_OBJECT()` legt die Funktionen `get_TYP()` und `release_TYP()` an, sowie einen Anker für eine Freiliste, `TYP_freelist`.

`Get_TYP()` liefert einen Zeiger auf eine Struktur vom Typ `TYP` zurück. Falls in der Freiliste kein Objekt vorhanden ist, werden per `myalloc()` mehrere neue Objekte erzeugt und in die Freiliste eingefügt.

`Release_TYP()` hängt das übergebene Objekt in die Freiliste ein.

Das Makro erwartet, daß die Struktur `TYP` eine Komponente `TYP*next` enthält, die zur Verkettung in den Freilisten benutzt werden kann.

Falls zur Compilezeit das Präprozessorsymbol `DEBUG` definiert ist, weist `get_TYP()` an die Komponente `long object_nr` jedes zurückgelieferten Objektes eine eindeutige Nummer zu (aus der globalen Variablen `object_number`) (siehe Abschnitt A.16 [Debugging], Seite 97, und Abschnitt A.2 [`object_number`], Seite 71).

A.7 Die Eventliste

A.7.1 Die EVENT - Struktur

```
typedef struct S_event_descr
{
#ifdef DEBUG
  long object_nr;
  void (*print_event_info)(VOID_P, FILE *, int depth, int indent);
#endif /* DEBUG */

  TIME_T event_time;
  void (*event_handler)(struct S_event_descr *);
  VOID_P event_info;
  int k_nr;
  struct S_event_descr *next;
#ifdef BUSH
  struct S_event_descr *down;
#endif

} EVENT;

EVENT *get_EVENT(void);
void release_EVENT(EVENT *);
```

Ereignisse werden durch `EVENT` - Strukturen beschrieben.

`Event_time` ist der Zeitpunkt, zu dem das Ereignis passiert, das durch den `EVENT` beschrieben wird. Die globale aktuelle Simulationszeit `simul_time` wird jeweils auf die `event_time` des gerade bearbeiteten Events gesetzt.

`Event_handler` zeigt auf eine Funktion, die das Ereignis bearbeiten soll. Sie bekommt als Parameter einen Zeiger auf den `EVENT`.

`Event_info` kann auf Informationen zeigen, durch die das Ereignis für den `event_handler` näher beschrieben wird. Der `event_handler` weiß, was für ein Objekt an `event_info` hängt.

`K_nr` enthält die Nummer des Netzknotens, der den `EVENT` in die Eventliste gestellt hat. Bei `EVENT`s, die von einem Netzmodul behandelt werden, ist das die Nummer des Senders.

Falls zur Compilezeit das Präprozessorsymbol `DEBUG` definiert ist, enthält der `EVENT` außerdem eine `object_nr`, sowie einen Zeiger `print_event_info` auf eine Funktion, die die am `event_info` hängende Information ausgibt (siehe Abschnitt A.16 [Debugging], Seite 97).

`Next` dient zur Verkettung in der Freiliste (und ggf. in der Eventliste).

`Get_EVENT()` liefert einen Zeiger auf einen freien `EVENT` (siehe Abschnitt A.6.3 [*Generische Freilisten*], Seite 75).

`Release_EVENT()` stellt einen `EVENT` in die Freiliste.

A.7.2 `insert_event()`, `CHANGE_EVENT`, `CANCEL_EVENT()`

Für die Verwaltung der Eventliste existieren zwei verschiedene Implementationen :

- Als *Implicit Heap* [Sed88].
- Als *Bottom Up Skew Heap* (*BUSH*) nach [ST83]. Diese Implementation hat den Vorteil, daß das Einfügen eines neuen Elementes und das Entfernen des kleinsten Elementes aus der Eventliste sehr schnell gehen. Es ist jedoch nicht möglich, ein beliebiges Element aus dem BUSH zu entfernen oder bei einem bereits im BUSH befindlichen Element den Schlüsselwert zu ändern und es dann innerhalb des BUSHes zu verschieben. Der ursprüngliche Algorithmus wurde erweitert, so daß gleichzeitige Ereignisse korrekt behandelt werden.

Abhängig vom Präprozessorsymbol `BUSH` wird zur Compilezeit die BUSH- oder die *Implicit Heap* - Implementation ausgewählt. *Beide* Implementationen stellen folgende Operationen zur Verfügung :

```
void insert_event(EVENT *);
CHANGE_EVENT_TIME(EVENT *, TIME_T);
DELAY_EVENT_TIME(EVENT *, TIME_T);
CHANGE_EVENT_TIME_AND_HANDLER(EVENT *, TIME_T, void (*)(EVENT *));
CANCEL_EVENT(EVENT *);
```

`Insert_event()` sortiert den `EVENT`, auf den der übergebene Zeiger zeigt, gemäß der darin enthaltenen `event_time` in die Eventliste ein.

Das Makro `CHANGE_EVENT_TIME()` bekommt einen Zeiger auf einen bereits in der Eventliste stehenden `EVENT`, dessen `event_time` es auf die übergebene Zeit ändert und die Eventliste neu sortiert.
Achtung: Abhängig von der Implementierung der Eventliste kann ggf. der übergebene Zeiger selber verändert werden.

`DELAY_EVENT_TIME()` arbeitet genauso, erwartet aber, daß die neue Eventzeit *später* ist als die bisherige `event_time` des `EVENT`s. Dadurch ergeben sich ggf. Optimierungen.
Achtung: Abhängig von der Implementierung der Eventliste kann ggf. der übergebene Zeiger selber verändert werden.

`CHANGE_EVENT_TIME_AND_HANDLER()` setzt `event_time` und `event_handler` eines bereits in der Eventliste stehenden `EVENT`s und sortiert die Eventliste neu. Das Makro ist geeignet, um Eventdescriptoren zu "recyceln", statt den `EVENT` zu canceln und einen neuen in die Liste zu stellen.
Achtung: Abhängig von der Implementierung der Eventliste kann ggf. der übergebene Zeiger selber verändert werden.

`CANCEL_EVENT()` entfernt den angegebenen `EVENT` aus der Eventliste. Die am `event_info` hängende Information muß vom Aufrufer **vorher** gerettet oder freigegeben werden, da der `EVENT` anschließend für den Aufrufer nicht mehr verfügbar ist. Je nach Implementierung der Eventliste kann `CANCEL_EVENT()` ein Makro oder eine Funktion sein.

A.7.3 `get_next_event()`

```
EVENT *get_next_event(void);
```

`Get_next_event()` entfernt den `EVENT` mit der kleinsten `event_time` aus der Eventliste und gibt einen Zeiger darauf zurück. Die Funktion wird nur in der Hauptschleife in `control()` aufgerufen. Dort wird die globale aktuelle `simul_time` auf die `event_time` des `EVENT`s gesetzt und dann der `event_handler` aufgerufen (siehe Abschnitt A.2 [`simul_time`], Seite 71).

A.7.4 print_event()

```
void print_event(EVENT *, FILE *, int depth, int indent);
```

`Print_event()` gibt eine Beschreibung eines `EVENT`s in das angegebene `FILE` aus.

Falls `depth` gleich 0 ist, werden nur die Daten aus dem `EVENT` selber ausgegeben, für `depth>0` wird die in `print_event_info` angegebene Funktion aufgerufen. `Indent` gibt die Einrückung der Zeilenanfänge an (siehe Abschnitt A.16 [Debugging], Seite 97).

A.8 Warteschlangen

Es stehen zwei Arten von Warteschlangen zur Verfügung :

NQUEUE Die *Normale Warteschlange* besitzt *n* Server, die jeweils einen Kunden bedienen können.

TQUEUE Die *Ticket Queue* stellt *n Tickets* oder *Tokens* zur Verfügung, von denen jeder Kunde eine bestimmte Anzahl anfordert.

Als Bedienstrategien stehen jeweils FIFO und LIFO zur Verfügung. Der globale Parameter `glob_params.max_q_length` gibt eine maximale Schlangenlänge an, bei deren Überschreiten ein Fehler ausgelöst wird.

Als Datenstruktur für die Beschreibung der zu behandelnden Kunden hat sich der `EVENT` als günstig erwiesen.

A.8.1 make_nqeue(), nq_push, nq_pop(), print_nqueue()

```
NQUEUE *make_nqueue(NQUEUE *, char *name, int server_anz, int stat_wanted);
int nq_push_lifo(NQUEUE *, EVENT *);
int nq_push_fifo(NQUEUE *, EVENT *);
EVENT *nq_pop(NQUEUE *);
print_nqueue(NQUEUE *, FILE *, int depth, int indent);
```

`Make_nqueue()` initialisiert eine "normale Warteschlange". Falls für die `NQUEUE` ein `NULL` - Zeiger übergeben wird, wird der Speicherplatz mittels `myalloc()` beschafft, sonst wird der Speicherplatz benutzt, auf den der übergebene Zeiger zeigt. Wenn `stat_wanted` ungleich 0 ist, wird Buch geführt über die Zahl der bedienten Kunden, Wartezeiten, Schlangenlängen, Auslastung der Server.
Ein Zeiger auf die initialisierte `NQUEUE` wird zurückgegeben.

`Nq_push_lifo()` prüft, ob die angegebene `NQUEUE` mindestens einen freien Server hat.
Wenn ja, wird `0` zurückgegeben, und der Aufrufer muß den `EVENT` für das entsprechende "end-of-service"-Ereignis in die Eventliste stellen.
Wenn nein, wird der `EVENT` vorne (LIFO) in die Warteschlange eingehängt und `1` zurückgegeben. Falls die Schlangenlänge dadurch `glob_params.max_q_length` überschreiten würde, wird `-1` zurückgegeben und die Schlange nicht verändert.

`Nq_push_fifo()` arbeitet genauso, fügt aber *hinten* (FIFO) in die Warteschlange ein.

`Nq_pop()` hängt das vorderste Element aus der angegebenen Warteschlange aus und liefert einen Zeiger darauf zurück. Falls die Schlange leer ist, wird `NULL` zurückgegeben.

`Print_nqueue()` druckt Informationen über die angegebene Warteschlange aus. Falls `depth` gleich 0 ist wird nur die Verwaltungsinformation / Statistik ausgegeben, für `depth>0` werden auch die wartenden `EVENT`s ausgegeben (siehe Abschnitt A.7.4 [`print_event()`], Seite 78).

A.8.2 `make_tqeue()`, `tq_push`, `tq_pop()`, `print_tqueue`

```
TQUEUE *make_tqueue(TQUEUE *, char *name, int server_anz, int stat_wanted);
int tq_push_lifo(TQUEUE *, EVENT *);
int tq_push_fifo(TQUEUE *, EVENT *);
EVENT *tq_pop(TQUEUE *);
print_nqueue(NQUEUE *, FILE *, int depth, int indent);
```

Diese Funktionen arbeiten wie die für die "normalen Warteschlange". Ein `EVENT` wird in die Schlange eingehängt, wenn nicht genügend Tickets vorhanden sind, um seine Anforderung zu erfüllen.

Wenn ein Kunde seine Tickets freigibt, wird `tq_pop()` mit der Anzahl der freiwerdenden Tickets aufgerufen. Anschließend muß `tq_pop()` solange aufgerufen werden, bis kein wartender Kunde mehr befriedigt werden kann :

```
TQUEUE *q;
EVENT *e;

e = tq_pop(q, nr_of_tickets_of_old_customer);
work_with(e);
while (e = tq_pop(q, 0))
{
    work_with(e);
}
```

A.9 Wahrscheinlichkeitsverteilungen

Aufbauend auf den von der Systembibliothek zur Verfügung gestellten `random` - Funktionen bietet simuLan einige Verteilungsfunktionen an. Welche Systemroutine zur Bestimmung der Zufallszahlen benutzt wird, geben die Makros `RANDOM()` und `SRANDOM()` in '`zufall.h`' an.

A.9.1 `ZUFALL_INFO`, `USE_ZUFALL()`

```
typedef struct S_zufall_info
{
  double (*zufall_fkt)(double *);
  double *param_p;
} ZUFALL_INFO;

(double)USE_ZUFALL(ZUFALL_INFO)
```

In simuLan können Verteilungsfunktionen durch `ZUFALL_INFO` - Strukturen beschrieben werden.

Da jede Wahrscheinlichkeitsverteilung andere Parameter bekommen muß, wird an `param_p` ein Array mit den zu `zufall_fkt` gehörenden Parametern gehängt, das von der jeweiligen Verteilungsfunktion passend interpretiert wird. Dadurch braucht die Wahrscheinlichkeitsverteilung erst zur Laufzeit aus der Konfiguration festgelegt werden.

Das Makro `USE_ZUFALL()` bildet eine C-Expression, die aus der in dem `ZUFALL_INFO` angegebenen Wahrscheinlichkeitsverteilung einen Double-Wert berechnet.

A.9.2 Als Funktion implementierte Wahrscheinlichkeitsverteilungen

```
double uniform(double *)
double konstant(double *)
double bimodal(double *)
double expntl(double *)
double normal(double *)
double hyperexpntl(double *)
double erlang(double *)
```

Diese Funktionen sollten immer nur über das Makro `USE_ZUFALL()` aufgerufen werden. Sie bekommen einen Zeiger auf ein Array von Double-Werten übergeben. Die Elemente des Arrays werden jeweils in der angegebenen Weise als Parameter der Verteilungsfunktion interpretiert.

`Uniform(upperbound)` liefert eine Zufallszahl im Intervall `[0, upperbound]`.

`Konstant(x)` liefert `x` als konstanten Wert.

`Bimodal(prob_low, mean_low, var_low, mean_high, var_high)` ist die Überlagerung zweier Normalverteilungen. Negative Werte werden auf 0.0 abgebildet.

`Expntl(lambda)` erhält als Parameter den Erwartungswert.

`Normal(mean, variance)` liefert eine normalverteilte Zufallszahl.

`Hyperexpntl(nr_stages, weight1, expect1, weight2, expect2, ...)` liefert eine hyperexponentialverteilte Zufallszahl.

`Erlang(k, lambda)` realisiert die Erlang-k-Verteilung.

(Eine ausführlichere Beschreibung der Verteilungen und ihrer Berechnung findet sich z.B. in [Fis73].)

A.9.3 `UNIFORM()`, `INT_UNIFORM()`, `EXPNTL()`, `INT_EXPNTL()`

```
(double) UNIFORM
(int) INT_UNIFORM(int low, int high)
(TIME_T) EXPNTL(double lambda)
(double) INT_EXPNTL(double lambda)
```

Dies sind einige Makros, die dort verwendet werden können, wo die Verteilungsfunktion von vorneherein festgelegt ist (z.B. Backoff-Algorithmus nach IEEE 802.3).

`UNIFORM` gibt einen Double-Wert im Bereich `[0..1[` zurück.

`INT_UNIFORM()` liefert eine ganze Zahl im Bereich `[low...high]`.

`EXPNTL()` liefert einen exponentialverteilten `TIME_T`.

`INT_EXPNTL()` liefert einen *abgerundeten* exponentialverteilten Double-Wert.

A.10 Verkehrsmatrizen

```
int uniform_empf(int sender);

typedef struct S_traffic_list
{
  int empfaenger_nr;
  char *empfaenger_name;
  float prob;
  struct S_traffic_list *next;
} TRAFFIC;

int listen_empf(int sender);

int self_empf(int sender);
```

In s$_{i}$m$_{u}$L^{a}n sind zwei Methoden vordefiniert, mit denen für eine zu sendende Nachricht ein Empfängerknoten ausgewählt werden kann :

- `Uniform_empf()` wählt so lange aus dem Intervall `[0...glob_params.knoten_anzahl]` gleichverteilt eine Knotennummer aus, bis ein vom Sender verschiedener Empfängerknoten gefunden wird.
- Für eine detailliertere Modellierung kann zu den Knoten eine Liste von Partnerknoten und zugehörigen Wahrscheinlichkeiten angegeben werden (siehe Abschnitt 3.2.3 [*Verkehrsmatrizen*], Seite 11, und Abschnitt A.15.12 [`traffic_rc()`], Seite 95).

`Self_empf()` gibt die Nummer des Senders als Empfängernummer zurück. Das wird gebraucht, um bei der Simulation eines einzelnen Knotens Endlosschleifen in `uniform_empf()` zu vermeiden.

A.11 Knotenbeschreibungen

Es wird ein Array angelegt, welches für jeden simulierten Knoten die Informationen für die Kommunikation zwischen den Modulen sowie Zeiger auf die privaten, knotenbezogenen Informationen der für den Knoten gewählten Protokollschichtmodelle enthält. Die Knotennummer ist der Index in diesem Array.

A.11.1 Die NODE - Struktur

```
typedef struct S_nodehead
{
  char *name;

  void (*print_k_info)(VOID_P k_info, FILE *fd, int prflag, int indent);

/* prflag gibt an, wie ausführlich die Ausgaben sein sollen */
#define PRINT_ALL       1
#define QUEUES          2

  void (*k_init)(int k_nr);
  void (*k_cleanup)(int k_nr);
  int (*calc_empfaenger)();
  TRAFFIC *traffic_list;

  void (*l4_sap)();
  void (*l3_sap)();
  void (*l2_sap)();

  void (*l5_pdu_received)();
  void (*l4_pdu_received)();
  void (*l3_pdu_received)();
  void (*l2_pdu_received)();

  void (*l5_pdu_sended)();
  void (*l4_pdu_sended)();
  void (*l3_pdu_sended)();
  void (*l2_pdu_sended)();

  VOID_P k_info;
  VOID_P l4_info;
  VOID_P l3_info;
  VOID_P l2_info;

} NODE;

extern NODE *nodearray;
```

`Name` zeigt auf einen String, der den Knotennamen enthält. In der Konfigurationsdatei können die Knotennamen ein "*%d*" enthalten, welches dann mittels `sprintf(3)` durch die Knotennummer ersetzt wird.

`Print_k_info` zeigt auf eine Funktion, die die privaten Informationen, auf die der Zeiger `k_info` zeigt, im Klartext ausdruckt. `K_info` ist der Zeiger auf die auszugebende Datenstruktur, `fd` zeigt auf das `FILE`, in welches ausgegeben werden soll, `prflag` gibt an, wie ausführlich die Ausgaben sein sollen, `indent` gibt die gewünschte Einrückung der Zeilenanfänge an.

`K_init` zeigt auf eine Funktion, die die Simulation der Schicht 5 dieses Knotens initialisert. Sie wird vor Simulationsbeginn aus `main()` aufgerufen.

`K_cleanup` zeigt dementsprechend auf eine Funktion, die nach Simulationsende aufräumt und Ergebnisse ausgibt.

`Calc_empfaenger` zeigt auf eine Funktion, mittels derer die Empfänger für die von dem Knoten erzeugten Nachrichten ermittelt werden (siehe Abschnitt A.10 [Verkehrsmatrizen], Seite 81).

`Traffic_list` zeigt ggf. auf die zugehörige Empfängerliste (siehe Abschnitt A.10 [Verkehrsmatrizen], Seite 81).

`L`*n*`_sap` bildet den *Service Access Point* für Netzschicht *n*. Das ist ein Zeiger auf eine Funktion, die aufgerufen wird, wenn die Schicht *n+1* eine Nachricht senden möchte. (Der *SAP* für Schicht 1 ist `net_params.l1_sap`.)

`L`*n*`_pdu_received` zeigt auf eine Funktion, die von Schicht *n-1* aufgerufen wird, um an Schicht *n* eine empfangene Nachricht zu übergeben.

`L`*n*`_pdu_sended` zeigt auf eine Funktion, die von Schicht *n-1* aufgerufen wird, um Schicht *n* mitzuteilen, daß ein von Schicht *n* an Schicht *n-1* gegebener Sendeauftrag ausgeführt wurde.

`L`*n*`_info` ist ein Zeiger, an den das für Schicht *n* ausgewählte Modul knotenbezogene private Informationen hängen kann.

A.11.2 `lookupnodename()`

```
int lookupnodename(int index, char *nodename);
```

`Lookupnodename()` sucht zum gegebenen Knotennamen den Index im `nodearray`, beginnend beim übergebenen `index`. Es ist vorgesehen, daß die Knotennamen in der Konfigurationsdatei die Zeichenfolge "*%d*" (Prozent d) enthalten können. Dafür wird dann eine fortlaufende Nummer eingesetzt (siehe sprintf(3)). Daher wird in `lookupnodename()` ein simples Pattern-Matching unterstützt :

Falls der übergebene Name mit einem `*` (Asterisk, Stern) endet, gilt der erste Knoten (in der Reihenfolge der Knotendefinitionen), dessen Index größer oder gleich dem übergebenen `index` ist, und bei dem der Namensanfang übereinstimmt, als Treffer.

Es erfolgt keine Fehlerprüfung. Falls kein passender Knoten gefunden wird, wird die Gesamtzahl der simulierten Knoten (`glob_params.knoten_anzahl`) zurückgegeben, was nicht als gültige Knotennummer auftreten kann.

A.12 Netzbeschreibung

```
extern struct S_net_params
{
  char *name;
  double bits_per_second;
  double itu_per_bit;
  int max_data_len;
  int min_data_len;
```

```
    void (*net_init)(void);
    void (*l1_sap)(EVENT *);
    void (*net_cleanup)(void);

  } net_params;
```

In simuLan werden alle Knoten durch *ein* Netz verbunden. Daher existiert nur eine Netzbeschreibung. Die globale Variable `net_params` enthält Informationen, die für alle implementierten Netzmodelle vorhanden und öffentlich bekannt sein müssen.

`Name` zeigt auf den Namen des modellierten Netzes.

`Bits_per_second` gibt die nominelle Übertragungsgeschwindigkeit an, so wie sie aus der Konfiguration gelesen wurde.

`Itu_per_bit` wird in rcnet() aus `itu_per_second/ net_params.bits_per_second` berechnet.

`Max_data_len` und `Min_data_len` geben die maximale bzw. minimale Nachrichtengröße in Bits an, die vom Netz übertragen werden kann.

`Net_init` zeigt auf eine Funktion, die vor Beginn der Simulation aufgerufen wird um das Modul zu initialisieren.

`L1_sap` stellt den Service Access Point dar. Es zeigt auf eine Funktion, die von der übergeordneten Schicht aufgerufen wird, um einen Sendeauftrag zu übergeben (siehe Abschnitt A.13 [*Schnittstellen*], Seite 84).

`Net_cleanup` zeigt auf eine Funktion, die nach Ende des Simulationslaufes aufgerufen wird. Sie kann ggf. Meßergebnisse ausgeben und Aufräumarbeiten ausführen.

A.13 Schnittstellen zwischen den Protokollschichtmodulen

In simuLan ist die Simulation von bis zu 5 Protokollschichten vorgesehen. Dabei simuliert die oberste Schicht (Nr. 5) die lasterzeugende "Anwendung", Schicht 1 simuliert den Zugang zum Übertragungsmedium sowie die physikalische Übertragung, entsprechend der MAC-Layer nach [IEE88a]

A.13.1 Die `SADU` - Struktur

```
typedef struct S_sadu
{
#ifdef DEBUG
  long object_nr;
  void (*print_next)();
#endif /* DEBUG */
  int status;
  int pdu_size;
  int receiver;
  struct S_sadu *next;
} SADU;

/* Werte für SADU.pdu_status */
```

```
#define OK                  0
#define SEND_FAILED         1

#ifdef DEBUG
#define ERROR_IN_TRANSMIT   3
#define REC_NOT_READY       4
#endif /* DEBUG */

SADU *get_SADU P((void));
void release_SADU P((SADU *));
void print_sadu(SADU *, FILE *, int depth, int indent);
```

`SADU` ist die Datenstruktur, mit der die an der Simulation der Datenübertragung beteiligten Funktionen die Informationen übergeben bekommen, die sie zur Ausführung ihrer Aufgaben benötigen (siehe Abschnitt 7.7 [Dienstzugangsfunktionen], Seite 43).

`Pdu_status` kann benutzt werden, um z.B. Informationen über Übertragungsstörungen weiterzugeben.

`Pdu_size` gibt die Größe der zu übertragenden Nachricht an.

`Receiver` enthält die Knotennummer des Empfängers.

`Next` wird zum einen für die Verkettung in der Freiliste benutzt, zum anderen kann daran private Information gehängt werden, die eine Schicht im Senderknoten ihrem Peer im Empfängerknoten zukommen lassen möchte.

`Print_sadu()` druckt die in einer `SADU` - Struktur enthaltene Information aus (siehe Abschnitt A.7.4 [`print_event()`], Seite 78, und Abschnitt A.16 [Debugging], Seite 97).

A.13.2 Der Service Access Point `l`*n*`_sap`

Der Zeiger `l`*n*`_sap` im `NODE` zeigt auf eine Funktion, mit der die Schicht *n+1* im Sender die Schicht *n* beauftragt, eine Nachricht zu übertragen. Die Funktion bekommt als Parameter einen `EVENT` mit angehängter `SADU`. Der `EVENT` enthält :
`event_time` : Die Zeit, zu der der Sendewunsch auftritt.
`k_nr` : Die Knotennummer des Senders.
`event_info` : Einen Zeiger auf eine `SADU`.

Die `SADU` enthält bereits :
`pdu_size` : Die (simulierte) Größe der Nachricht.
`receiver` : Die Knotennummer des Empfängers.
`next` : Ein Zeiger auf private Informationen der Schicht *n+1*. (Der "Inhalt" der zu übertragenden Nachricht.)

Der SAP für die unterste Schicht (das "Netzmodul") ist in `net_params.l1_sap` verzeichnet.

A.13.3 Nachrichten empfangen: `L`*n*`_pdu_received`

Der Zeiger `L`*n*`_pdu_received` im `NODE` zeigt auf eine Funktion, mit der die Schicht *n* im Empfänger die Schicht *n+1* benachrichtigt, daß eine Nachricht für sie vollständig empfangen wurde. Die Funktion bekommt als Parameter einen Zeiger auf einen `EVENT` mit angehängter `SADU`, so wie er vom Sender an `l`*n*`_sap` übergeben wurde.

Hinweis: Die Datenstrukturen sind vom Sender erzeugt worden und dürfen nur vom Sender freigegeben werden. Sie müssen daher beim Rücksprung aus dieser Funktion wieder genauso aussehen wie sie übergeben wurden. Falls der Empfänger sich Informationen aus den empfangenen Daten merken will, muß er eine Kopie anlegen (wie im wirklichen Leben).

Es dürfen nur Nachrichten weitergegeben werden, die ungestört und vollständig übertragen wurden.

A.13.4 Sendeauftrag abschließen: `Ln_pdu_sended`

Der Zeiger `Ln_pdu_sended` im `NODE` zeigt auf eine Funktion, mit der die Schicht *n* im Sender die Schicht *n+1* benachrichtigt, daß ein gegebener Sendeauftrag ausgeführt wurde. Die Funktion bekommt als Parameter einen Zeiger auf einen `EVENT` mit angehängter `SADU`, so wie er an Schicht *n* übergeben wurde.

Die übergebenen Datenstrukturen müssen in dieser Funktion ggf. freigegeben bzw. neu verwendet werden. Die Funktion wird immer aufgerufen, egal, ob die Übertragung gestört war oder nicht. Der Sender erfährt normalerweise nicht, ob der Empfänger die Nachricht korrekt erhalten hat.

Hinweis : Der Empfänger kann feststellen, ob der Sender das gleiche Protokoll benutzt, indem er den Zeiger auf die SAP-Funktion bzw. die `l5_pdu_received` - Funktion des Senderknotens betrachtet.

A.14 Instrumentierung

Ein Meßpunkt besteht aus einer Liste von `PROBE` - Strukturen, welche Angaben über die Meß- und Ausgabefunktionen sowie eine eventuelle statistische Absicherung enthalten.

A.14.1 Ausgabeformatierung: `probe_printf()`

```
void probe_printf(PROBE *, EVENT *, FILE *, char *format);
```

`Probe_printf()` eignet sich, um Meßwerte in einem Format auszugeben, welches nicht fest im Programm verdrahtet ist, sondern in der Konfigurationsdatei angegeben werden kann. `Format` zeigt auf einen String, der angibt, wie diese Ausgaben formatiert werden sollen. Der Aufbau dieses Strings ist an die Formatangaben von `printf(3)` angelehnt.

Ein Format-String enthält zwei Arten von Objekten :

- gewöhnliche Zeichen, die einfach in die Ausgabe kopiert werden,
- Konvertierungs-Angaben, eingeschlossen in zwei % (Prozent).
  '`%<letter>[<printf(3)-Formatangabe>]%`'

```
Für '<letter>' werden definiert:
  t : aktuelle simul_time in ITU    (default-Format '.0f')
  T : aktuelle simul_time in Sekunden  ('g')
  M : der aktuelle Meßwert akt_val der PROBE  ('g')
  O : der old_val der PROBE  ('g')
  u : old_time in ITU  ('.0f')
```

```
        U : old_time in Sekunden  ('g')
        n : Nummer des aktuellen Knotens (event_p->k_nr),
            soweit für den Meßpunkt anwendbar  ('d')
        N : Name des aktuellen Knotens  ('s')
        c : Anzahl der simulierten Netzknoten  ('d')
        S : Das ermittelte Ergebnis der statistischen Absicherung
            (stat_result)  ('g')
       $x : der Wert der Laufvariablen x, als Double  ('g')
```

Für jeden Meßpunkt kann nach dem Vorbild von `probe_printf()` jeweils eine erweiterte Funktion geschrieben werden, welche weitere Möglichkeiten anbietet.

A.14.2 Die PROBE - Struktur

```
typedef struct S_probepoint *PROBEPOINT;
typedef struct S_probepoint PROBE;

struct S_probe
{
  PROBE *next;
  PROBE *probeptlist;

  void (*probe_fkt)(PROBE *, EVENT *);
  FILE *trace_fd;

  double akt_val;
  double old_val;
  TIME_T old_time;

  char *format;
  void (*format_printf)(PROBE *, EVENT *, FILE *, char *format);
  FILE *format_fd;


  void (*print_result)();

  char *result_format;
  FILE *result_format_fd;


  double stat_result;

  void (*stat_fkt)(PROBE *);
  char stat_wait_flag;

#define STAT_WAIT 'y'
#define STAT_STOP 's'
#define STAT_NOWAIT ' \0'

  void (*print_stat_result)(PROBE *);
  char *stat_description;
  FILE *stat_fd;
```

```
    VOID_P stat_vars;
    int stat_vars_size;
    int stat_vars_common;
};

MEASURE(PROBE *M, VOID_P Pt)
```

An einer Meßstelle können gleichzeitig mehrere Messungen durchgeführt werden. Dafür wird an einen Zeiger `PROBE*` eine Liste von `PROBE` - Strukturen gehängt, die über den `next` - Zeiger verkettet sind.

`Probeptlist` dient zum Aufbau einer Liste aller angelegten `PROBE` - Strukturen, die am Ende der Simulation durchgegangen wird, um die Endergebnisse der Messungen auszugeben.

`Probe_fkt` zeigt auf eine Funktion, die einen Meßwert berechnet. Sie legt den Meßwert in `akt_val` ab und kann ihn ggf. in das File `trace_fd` schreiben. Diese Funktion wird mittels `MEASURE()` jedesmal aufgerufen, wenn an der bestimmten Stelle im Modell etwas passiert. (z.B. Ende der Übertragung eines Frames über das Medium).

`Old_val` kann benutzt werden, um den Wert der vorhergehenden Messung zu speichern, `old_time` kann den Zeitpunkt der vorhergehenden Messung aufnehmen (soweit das für die Meßfunktion erforderlich ist).

`Stat_fkt` zeigt auf eine Funktion zur statistischen Absicherung der Meßwerte (Varianzen, Vertrauensintervall), für die Kontrolle der Simulationslänge.

`Stat_wait_flag` gibt an, was passieren soll, wenn das Absicherungsmodul zu dem Ergebnis gekommen ist, daß genug Meßwerte ermittelt wurden (siehe Abschnitt A.14.3 [*Schnittstelle zur statistischen Absicherung*], Seite 89).

`Stat_result` enthält den von der statistischen Absicherung errechneten (Mittel-) Wert.

`Print_result` zeigt auf eine Funktion, die am Ende der Simulation den vom Absicherungsmodul errechneten Mittelwert in `stat_result` ggf. aufbereitet (z.B. Umrechnung von ITU in Sekunden) und ihn ins File `stat_fd` ausgibt, falls `stat_fd` ungleich `NULL` ist.

Zusätzlich kann das errechnete Ergebnis entsprechend dem `result_format` in das File `result_format_fd` ausgegeben werden. Dafür wird wieder die in `format_printf` verzeichnete Funktion benutzt.

`Print_stat_result` zeigt auf eine Funktion, die am Ende der Simulation aufgerufen wird, um die von der `stat_fkt` ermittelten Varianzen und Konfidenzintervalle in das File `stat_fd` zu schreiben. `Stat_description` zeigt auf einen String, der die Bedeutung der von der Statistik ermittelten Werte erklärt.

`Stat_vars` zeigt auf private Informationen des Statistikmoduls für diese Messung, die Größe dieser Datenstruktur (in Bytes) steht in `stat_vars_size`.

Wenn mehrere gleichartige Knoten definiert werden, gibt `stat_vars_common` an, ob sie alle für diese Messung eine gemeinsame statistische Absicherung teilen sollen (zur Verkürzung der Programmlaufzeiten), oder ob für jeden Knoten die Absicherung getrennt erfolgen soll (siehe Abschnitt A.15.14 [Einlesen der Knotenparameter], Seite 96, und Abschnitt 5.4.6 [Beschreibung der statistischen Absicherung], Seite 31).

Das Makro `MEASURE()` durchläuft die Liste der `PROBE`s eines `PROBEPOINT`s und ruft jeweils die `probe_fkt`, `stat_fkt` sowie die `format_printf` - Funktion zur Gewinnung und Ausgabe des aktuellen Meßwertes auf.

A.14.3 Die Schnittstelle zur statistischen Absicherung

Die Meßfunktion der `PROBE` legt ihre laufend gemessenen Werte in `akt_val` ab. Anschließend wird die Absicherungsfunktion aufgerufen, die prüft, ob bereits genügend Meßwerte ermittelt wurden. Den berechneten Mittelwert der Meßwerte muß sie in `stat_result` ablegen. Er muß vor dem Ausgeben eventuell aufbereitet werden.

Wenn die Absicherungsfunktion zu dem Schluß gekommen ist, daß genügend Werte gemessen wurden, um die Anforderungen an Varianz bzw. Konfidenzintervall zu genügen, dann bestimmt der Wert des `stat_wait_flag`s, was zu tun ist :

- `WAIT` : Die globale Variable `stat_not_ready` wird dekrementiert. Erreicht sie den Wert null, so bedeutet das, daß alle Statistikinstanzen, für die der Wert `WAIT` angegeben war, zu Ende gekommen sind.
  Mit dem Makroaufruf `END_OF_SIMULATION(STATISTICS_READY, 0, 0, 0)` muß der Simulationslauf beendet werden.
  Das Simulationsmodul muß ggf. dafür sorgen, daß es nicht weiterhin aufgerufen wird, indem es die `stat_fkt` auf `NULL` setzt.
- `NOWAIT` : Es muß ggf. die `stat_fkt` auf `NULL` gesetzt werden, sonst nichts.
- `STOP` : Es muß `END_OF_SIMULATION(STATISTICS_STOP, 0, 0, 0)` aufgerufen werden, um den Simulationslauf zu beenden.

Nach dem Ende des Simulationslaufes wird zuerst die in `print_stat_result` verzeichnete Funktion aufgerufen, die den von der statistischen Absicherung errechneten (Mittel-) Wert `stat_result` ggf. aufbereitet (skaliert) und in das File `stat_fd` ausgibt. (Zu jeder Meßfunktion muß eine solche Ergebnis-Ausgabe-Funktion erstellt werden, siehe Abschnitt 7.9 [Implementation von Meßpunkten], Seite 45.)

Falls `result_format_fd` ungleich `NULL` ist, wird danach die `format_printf` - Funktion aufgerufen, um den in `stat_result` errechneten Wert gemäß dem `result_format` in das File `result_format_fd` auszugeben.

Anschließend wird die in `print_stat_result` verzeichnete Funktion aufgerufen, um die errechneten Varianzen und Konfidenzintervalle in das File `stat_fd` auszugeben.

A.15 Einlesen der Konfiguration

Die Konfigurationsdatei wird mittels einer Art rekursiven Abstiegs eingelesen. Für jedes Modul muß eine Funktion geschrieben werden, die die zugehörigen Parameter einliest. In eine Tabelle muß dann ein Schlüsselwort eingetragen werden, bei dessen Auftreten die Einlesefunktion des Moduls aufgerufen wird.

A.15.1 Fehlerbehandlung:`RC_WARNING`,`RC_SYNTAX_ERROR`,`RC_ERROR`, `RC_FATAL`

```
(void) RC_WARNING(char *message, arg2, arg3)
(void) RC_SYNTAX_ERROR(char *message, arg2, arg3)
(void) RC_ERROR(char *message, arg2, arg3)
(void) RC_FATAL(char *message, arg2, arg3)
```

Diese Makros geben den Namen des Konfigurationsfiles und die aktuelle Zeilennummer im gleichen Format aus wie `grep(1)` mit der Option '`-n`', so daß sie vom GNU-Emacs verarbeitet werden können. Die Argumente werden als zweiter, dritter und vierter Parameter an `fprintf(3)` übergeben und auf `stderr` ausgedruckt.

`RC_WARNING()` gibt nur die Meldung aus.

`RC_SYNTAX_ERROR()` und `RC_ERROR()` geben die Meldung aus und brechen das Einlesen der Konfiguration ab. Die beiden Makros unterscheiden sich nur durch den Text '`Syntax`' gegenüber '`Error`'.

`RC_FATAL()` gibt die Meldung aus und ruft dann `abort(3)` auf.

A.15.2 Worte Einlesen : `xxgetword()`, `xxungetword()`

```
#define XXWORDLEN 512
extern char xxword[XXWORDLEN];
extern int xxlinenr;

void xxgetword(void);
void xxungetword(void);
```

`Xxword` enthält das zuletzt aus der Konfiguration gelesene Wort

`Xxlinenr` zählt die aktuelle Zeile in der Konfiguration.

`Xxgetword()` liest das nächste Wort ein und legt es in der globalen Variablen `xxword` ab (siehe Abschnitt 5.2 [Zerlegung der Konfigurationsdatei in Worte], Seite 24).

`Xxungetword()` stellt das zuletzt gelesene Wort in die Eingabe zurück. `Xxungetword()` kann als Makro definiert sein.

A.15.3 Worte ausgeben: `xxprintword()`

```
void xxprintword(FILE *, char *word);
```

`Xxprintword()` druckt ein Wort so aus, daß es von `xxgetword()` wieder eingelesen werden könnte:
Nichtdruckbare Zeichen werden als Oktalcode dargestellt, Sonderzeichen (Doublequote, Backslash, Backspace, Linefeed, ...) werden in spezieller Notation ausgegeben (siehe Abschnitt 5.2 [Zerlegung der Konfigurationsdatei in Worte], Seite 24). Das ganze Wort wird in Doublequotes eingeschlossen.

A.15.4 Ausgabeumlenkung: `xxgetfd()`

```
FILE *xxgetfd(void)
```

`Xxgetfd()` liest das nächste Wort aus der Konfiguration und sucht es in der Liste der definierten symbolischen File-Identifier (siehe Abschnitt A.4 [Auswertung der Kommandozeile], Seite 72). Ein Zeiger auf das zugehörige `FILE` wird zurückgegeben. Falls das Wort nicht als File-Identifier definiert ist, wird `NULL` zurückgegeben.

(Die Liste der definierten File-Identifier wird mittels `rcfiles()` erstellt.)

A.15.5 Zahlenangaben einlesen:xxgetint(),xxgetulong(), xxgetfloat(), xxgetdouble()

```
int xxgetint(void);
unsigned long xxgetulong(void);
float xxgetfloat(void);
double xxgetdouble(void);
double xxgetforvalue(/*char*/ int name);
```

simuLan bietet die Möglichkeit, Simulationsparameter zu variieren und für jede Variation einen Simulationslauf durchzuführen (siehe Abschnitt 5.3.2 [Definition von Laufvariablen], Seite 25).

`Xxgetint()` liest ein Wort aus der Konfiguration (siehe Abschnitt 5.2 [*Worte*], Seite 24). Wenn das Wort mit einem `$` (Dollar) beginnt, wird der Rest des Wortes als Name einer Laufvariablen interpretiert und der zugehörige Wert aus der Liste der Laufvariablen herausgesucht. Anderenfalls wird das Wort als Zahlenangabe aufgefaßt. Der ermittelte Wert wird zurückgegeben. Falls die Umwandlung in eine Zahl, bzw. die Suche in der Liste der Laufvariablen fehlschlägt, so wird eine Meldung ausgegeben (siehe Abschnitt A.15.1 [`RC_WARNING()`], Seite 89).

`Xxgetulong()`, `xxgetfloat()` und `xxgetdouble()` arbeiten entsprechend.

`Xxgetforvalue()` bekommt als Parameter den Namen einer Laufvariablen und gibt deren augenblicklichen Wert als Double zurück.

Hinweis: Alle numerischen Variablen werden mittels `sscanf(3)` zunächst als Double eingelesen und dann in den verlangten Zahlentyp umgewandelt.

Hinweis: Eine Laufvariable mit Namen x vom Type t_1 ist verschieden von einer Variablen x vom Typ t_2.

Bug: Von den Namen der Laufvariablen ist bisher nur der erste Buchstabe signifikant.

A.15.6 Einheitenangaben: xxgetbits(), xxgettime_t(), xxgetmeter()

```
int xxgetbits(void);
TIME_T xxgettime_t(void);
double xxgetmeter(void);
double xxgetdouble_with_unit(void);
```

`Xxgetbits()` liest ein Wort aus der Konfiguration und versucht, es als ganze Zahl zu interpretieren (siehe Abschnitt A.15.5 [`xxgetint()`], Seite 91). Das darauffolgende Wort kann eine Einheitenangabe sein. Falls keine Einheitenangabe folgt, wird '`Bit`' angenommen (Faktor 1) und das Wort in die Eingabe zurückgestellt (siehe Abschnitt A.15.2 [`xxgetword()`], Seite 90). Der eingelesene Zahlenwert wird in Bits umgerechnet und zurückgegeben.
Folgende Einheitenabgaben werden verstanden: '`MB`', '`MByte`', '`Mb`', '`Mbit`', '`KB`', '`KByte`', '`Kb`', '`Kbit`', '`B`', '`Byte`', '`byte`', '`b`', '`Bit`', '`bit`'.

`Xxgettime_t()` liest einen Double-Wert und versucht, das darauffolgende Wort als Zeiteinheit zu interpretieren. Falls keine Einheitenangabe folgt, wird das Wort in die Eingabe zurückgestellt und '`sec`' angenommen (Faktor 1.0).
Folgende Einheitenangaben werden erkannt: '`min`', '`s`', '`sec`', '`ms`', '`us`', '`ns`', '`itu`'.
Der eingelesene Zahlenwert wird zunächst mit dem Einheiten-Faktor multipliziert, dann

mittels `glob_params.itu_per_second` in ITU umgerechnet, in einen `TIME_T` gewandelt und zurückgegeben. (Falls die Einheit '`itu`' angegeben ist, erfolgt keine Umrechnung.)

`Xxgetmeter()` liest einen Double-Wert und versucht, das darauffolgende Wort als Längeneinheit zu interpretieren. Falls keine Einheitenangabe folgt, wird das Wort in die Eingabe zurückgestellt und '`meter`' angenommen (Faktor 1.0).
Folgende Einheitenangaben werden erkannt: '`km`', '`m`', '`meter`', '`dm`', '`cm`', '`mm`', '`um`', '`nm`'.
Der eingelesene Zahlenwert wird mit dem Einheiten-Faktor multipliziert und zurückgegeben.

`Xxgetdouble_with_unit()` liest einen Double-Wert ein und versucht, das darauffolgende Wort als Informationsmenge oder Zeiteinheit zu interpretieren. Falls keine Einheitenangabe folgt, wird das Wort in die Eingabe zurückgestellt und der eingelesene Wert nicht skaliert. Zeitangaben werden mittels `glob_params.itu_per_second` in ITU umgerechnet. Diese Funktion ist besonders für das Einlesen von Verteilungsparametern nützlich siehe Abschnitt A.15.11 [Einlesen von Verteilungsfunktionen], Seite 95).

A.15.7 Funktionen auswählen: `xxgetfkt()`

```
typedef struct S_rcfkttab
{ char *name;
  void (*fkt)();
  char *explain;
} FKT_TABLE;

void (*xxgetfkt(FKT_TABLE *tab))();
void xxexplainfkt(FKT_TABLE *table, int indent);
```

Die `FKT_TABLE` - Struktur ordnet einer Funktion einen *symbolischen Namen* zu. `Name` ist der symbolische Name der Funktion, so wie er in der Konfigurationsdatei angegeben werden kann, `fkt` enthält die Adresse der Funktion, `explain` zeigt auf einen erläuternden String.

`Xxgetfkt()` bekommt als Parameter ein Array von solchen `FKT_TABLE` - Strukturen. Die Funktion liest das nächste Wort aus der Konfigurationsdatei und sucht in der übergebenen Tabelle nach einem Eintrag, dessen `name` mit dem gelesenen Wort übereinstimmt. Der Zeiger auf die zugehörige Funktion wird zurückgegeben (Der Aufrufer muß wissen, welche Parameter die gelieferte Funktion bekommen muß, und welchen Datentyp sie liefert). Falls die Tabelle keinen Eintrag für das eingelesene Wort enthält, wird ein Syntaxfehler gemeldet und ein `NULL` - Zeiger zurückgegeben. Falls die globale Variable `printconf` gesetzt ist (siehe Abschnitt A.4 [Auswertung der Kommandozeile], Seite 72), werden das eingelesene Wort und ein Zeilenvorschub ausgegeben.

Die Funktion `xxexplainfkt()` druckt die Funktionsnamen und Erläuterungen zu der übergebenen `FKT_TABLE` aus. `Indent` gibt die gewünschte Einrückung der Zeilenanfänge an.

Die Tabelle muß mit einem Eintrag enden, der in der Komponente `name` einen `NULL` - Zeiger enthält.

A.15.8 Instrumentierung einlesen: `rcprobepoint()`

```
void rcprobepoint(PROBE **probept, FKT_TABLE *table, int indent);
```

`Rcprobepoint()` liest die Parameter für einen Meßpunkt aus der Konfigurationsdatei ein und erzeugt daraus eine Liste von `PROBE` - Strukturen (siehe Abschnitt A.6.1 [`myalloc()`], Seite 74; Abschnitt A.14 [Instrumentierung], Seite 86), die an den Zeiger gehängt wird, auf den der übergebene Doppelzeiger zeigt.

Falls die globale Variable `printconf` gesetzt ist (siehe Abschnitt A.4 [Auswertung der Kommandozeile], Seite 72), so werden die gelesenen Daten einschließlich evtl. eingesetzter Default-Werte ausgedruckt. `Indent` gibt dafür die gewünschte Einrückung der Zeilenanfänge an.

`Table` zeigt auf ein Array von `FKT_TABLE` - Strukturen, welches die für diesen Meßpunkt definierten Meßfunktionen angibt. Für jede Meßfunktion muß eine zugehörige Einlesefunktion geschrieben werden, auf die die Komponente `fkt` des Tabelleneintrags zeigt. Wenn in der Konfiguration eine Meßfunktion ausgewählt wird, so wird der entsprechende Eintrag in der Tabelle gesucht und die zugehörige Einlesefunktion aufgerufen. Diese Einlesefunktion bekommt als Parameter einen Zeiger auf eine `PROBE` - Struktur. Sie muß darin die Komponenten `probe_fkt`, `print_result` und `stat_description` ausfüllen (siehe Abschnitt A.14.2 [*Die* `PROBE` *- Struktur*], Seite 87). Falls `trace_fd` ungleich `NULL` ist, sollte eine Überschrift für die laufenden Meßwerte ausgegeben werden.

Wenn für einen Meßpunkt nach dem Vorbild von `probe_printf()` eine eigene Funktion zur Ausgabeformatierung geschrieben wurde, so muß der letzte Eintrag der Tabelle (der mit dem `NULL` - Zeiger, siehe Abschnitt A.15.4 [`xxgetfd()`], Seite 90), in der Komponente `fkt` einen Zeiger auf diese Funktion enthalten. Dadurch wird die meßpunktspezifische Funktion als `format_printf` - Funktion in die `PROBE` eingetragen. Default ist `probe_printf()`.
Der `explain` - Zeiger des letzten Tabelleneintrages sollte auf einen String zeigen, der die meßpunktspezifische Ausgabefunktion erläutert.

A.15.9 `copy_probepoint()`

```
PROBEPOINT copy_probepoint(PROBEPOINT probept);
```

`copy_probepoint()` erzeugt von einer verketteten Liste von `PROBE` - Strukturen eine Kopie und gibt einen Zeiger auf den Anfang der Liste zurück. Dabei wird Speicher mittels `myalloc()` reserviert.

Diese Funktion wird beim Einlesen der Knotenbeschreibungen gebraucht, wenn mehrere identische Knoten konfiguriert werden, um die beim Einlesen angelegten `PROBE`en zu vervielfältigen.

A.15.10 `rcreadkeywds()`

```
typedef struct S_rctable
{
  char *name;
  VOID_P value;
  FKT_TABLE *tab;
  char type;
  char *explain;
} KEYWD_TABLE;

void rcreadkeywds(KEYWD_TABLE *table_p, int indent);
```

Die `KEYWD_TABLE` - Struktur ordnet einem in der Konfigurationsdatei erlaubten Schlüsselwort eine Bedeutung zu.

`Name` zeigt auf das Schlüsselwort, so wie es in der Konfiguration erwartet wird.
`Type` gibt die Bedeutung des Schlüsselwortes an.
Die Verwendung von `value` hängt vom `type` ab, im Allgemeinen zeigt `value` auf die Speicherstelle, an der ein eingelesener Wert abgelegt werden soll (siehe unten).
`Explain` zeigt auf einen String, der als Kommentar in der Erläuterung der Tabelle ausgegeben wird, falls die globale Variable `minus_x` ungleich null ist.

Der letzte Eintrag des Arrays muß in der Komponente `name` einen `NULL` - Zeiger enthalten.

`Rcreadkeywds()` bekommt einen Zeiger auf ein Array von `KEYWD_TABLE` - Strukturen übergeben.

`Rcreadkeywds()` liest solange Schlüsselworte und dazugehörige Parameter aus der Konfigurationsdatei, bis ein Schlüsselwort vom Typ `XXTYPend` gefunden wird. Die anderen Worte aus der Tabelle dürfen in beliebiger Reihenfolge und beliebig oft auftreten (auch wenn das nicht immer sinnvoll ist).

Es wird jeweils ein Wort eingelesen, der zugehörige Eintrag in dem übergebenen Array von `KEYWD_TABLE` - Strukturen gesucht und die dem angegebenen `type` entsprechende Aktion ausgeführt :

- XXTYPint : eine ganze Zahl wird eingelesen (siehe Abschnitt A.15.5 [`xxgetint()`], Seite 91) und in der Speicherstelle, auf die der in der Komponenten `value` des Arrayeintrags angegebene Zeiger zeigt, als `int` abgelegt.
- XXTYPbits : eine Nachrichtenlänge wird eingelesen und in `*value` als `int` abgelegt (siehe Abschnitt A.15.6 [`xxgetbits()`], Seite 91).
- XXTYPulong : eine ganze Zahl wird eingelesen und in `*value` als `unsigned long` abgelegt (siehe Abschnitt A.15.5 [`xxgetulong()`], Seite 91).
- XXTYPfloat : ein Float-Wert wird eingelesen und in `*value` als `float` abgelegt (siehe Abschnitt A.15.5 [`xxgetfloat()`], Seite 91).
- XXTYPdouble : ein Double-Wert wird eingelesen und in `*value` als `double` abgelegt (siehe Abschnitt A.15.5 [`xxgetdouble()`], Seite 91).
- XXTYPtimet : eine Zeitangabe wird eingelesen und in `*value` als `TIME_T` abgelegt (siehe Abschnitt A.15.6 [`xxgettime_t()`], Seite 91).
- XXTYPstring : ein Wort wird eingelesen (siehe Abschnitt A.15.2 [`xxgetword()`], Seite 90) und in `*value` als `char[]` abgelegt.
- XXTYPvoidfkt, XXTYPintfkt, XXTYPdoublefkt, XXTYPtimetfkt : Die Komponente `tab` des Arrayeintrags zeigt auf ein Array von `FKT_TAB` - Strukturen. Dieses Array wird an `xxgetfd()` (siehe Abschnitt A.15.4 [`xxgetfd()`], Seite 90) übergeben, um einen Funktions-Identifier aus der Konfiguration zu lesen. Der zurückgelieferte Zeiger wird in `*value` abgelegt.
- XXTYPimmhook : `value` zeigt auf eine Funktion, welche direkt aufgerufen wird. Sie bekommt als Parameter einen Integer, der die Einrückung der Zeilenanfänge für eventuelle Ausgaben angibt.
- XXTYPnamedhook : Die Komponente `tab` des Arrayeintrags zeigt auf ein Array von `FKT_TAB` - Strukturen. Dieses Array wird an `xxgetfkt()` (siehe Abschnitt A.15.7 [`xxgetfkt()`], Seite 92) übergeben, um einen Funktions-Identifier aus der Konfiguration zu lesen. Die zurückgelieferte Funktion wird aufgerufen, falls `xxgetfkt()` nicht `NULL` liefert. Sie bekommt als Parameter den in `value` angebenen Wert sowie einen Integer, welcher die Einrückung der Zeilenanfänge für eventuelle Ausgaben angibt.

- XXTYPprobe : `value` zeigt auf einen Zeiger, an den eine Liste von `PROBE` - Strukturen gehängt wird, `tab` zeigt auf eine Tabelle mit den für diesen Meßpunkt definierten Meßfunktionen (siehe Abschnitt A.15.8 [`rcprobepoint()`], Seite 92).
- XXTYPend : Das Einlesen von Schlüsselworten gemäß der übergebenen Tabelle wird beendet. Falls die globale Variable `printconf` gesetzt ist, wird die Tabelle durchgegangen und die Defaultwerte der nicht in der Konfiguration angegebenen Schlüsselworte ausgedruckt.
- XXTYPundef : Schlüsselwort mit undefiniertem Typ - sollte nicht auftreten.

Wenn `printconf` gesetzt ist, werden die eingelesenen Daten ausgegeben (siehe Abschnitt A.4 [Auswertung der Kommandozeile], Seite 72). Der Parameter `indent` gibt dafür die Einrückung der Zeilenanfänge an. Nach den gelesenen Daten werden eventuell eingesetzte Default-Werte ausgedruckt.

Wenn `minus_x` ungleich `NULL` ist, so werden keine Daten eingelesen, sondern eine Beschreibung der in der übergebenen Tabelle definierten Eingabesyntax ausgegeben.

Von den aufgerufenen Funktionen wird erwartet, daß sie ebenfalls die Variablen `printconf` und `minus_x` beachten und ggf. die von ihnen gelesenen Werte ausgeben (lassen) bzw. eine Erläuterung der Eingabesyntax drucken.

A.15.11 Einlesen von Verteilungsfunktionen

```
extern FKT_TABLE distr_tab[];
ZUFALL_INFO *rc_get_zufall_info_vektor(int n, int indent);
```

Das Array `distr_tab` enthält einen Eintrag für jede wählbare Verteilungsfunktion. Zu jeder Verteilungsfunktion muß eine Funktion für das Einlesen der zugehörigen Parameter geschrieben werden. Ein Zeiger auf diese Einlesefunktion sollte in `distr_tab` abgelegt werden (siehe Abschnitt A.15.7 [`xxgetfkt()`], Seite 92).

Die Funktion `rc_get_zufall_info_vektor()` erzeugt ein Array von `ZUFALL_INFO` - Strukturen und liest für jedes Element eine Verteilungsfunktion und deren Parameter aus der Konfiguration. Der Parameter `n` gibt die Länge des anzulegenden Arrays an. `Indent` gibt die gewünschte Einrückung der Zeilenanfänge an, falls Erläuterungen ausgedruckt werden sollen (siehe Abschnitt A.4 [Auswertung der Kommandozeile], Seite 72).

A.15.12 Einlesen von Verkehrsverteilungen : `traffic_rc()`

```
extern TRAFFIC *traffic_rc_list;
void traffic_rc(int indent);
```

`Traffic_rc()` liest für einen Netzknoten eine Liste von Partnerknoten und zugehörigen Wahrscheinlichkeiten ein. Ein Listenelement besteht aus dem Wort '`partner`', gefolgt von einem Knotennamen und einer Zahl im Bereich [0...1]. Das Einlesen wird beendet, wenn das Wort '`partner`' nicht gefunden wird. Aus den Angaben wird eine verkettete Liste von `TRAFFIC` - Strukturen erzeugt, die an `traffic_rc_list` gehängt wird. Der Aufrufer muß anschließend diese Liste an den `traffic_list` - Zeiger in der `NODE` - Struktur des konfigurierten Knotens hängen.

Diese Liste enthält statische Information, so daß sie von gleichartigen Knoten, die mit '`COUNT <n>`' angelegt wurden, geteilt werden kann. (Es braucht also nur der Zeiger auf die Liste kopiert werden.)

Nachdem alle Knotenbeschreibungen eingelesen worden sind, werden in den angelegten `TRAFFIC` - Listen automatisch die Knotennamen in Knotennummern umgesetzt.

Die Funktion kann in einer `KEYWD_TABLE` - Tabelle für ein Layer-5-Modell wie folgt benutzt werden :

```
{ Traffic", traffic_rc, NULL, XXTYPimmhook, Traffic-relations" }
```

A.15.13 Einlesen der globalen Simulationsparameter

```
extern KEYWD_TABLE rcparmtab[];
extern FKT_TABLE statmoduls_tab[];
extern FKT_TABLE itimer_probefkt_tab[];
extern FKT_TABLE cronic_probefkt_tab[];
```

Das Array `rcparmtab` enthält Einträge für die globalen Simulationsparameter in der Struktur `glob_params`. Außerdem werden dort die Schlüsselworte '`NODES`' und '`NET`' für das Einlesen der Knoten- und Netzparameter definiert.

`Statmoduls_Tab` enthält einen Eintrag für jedes verfügbare Modul zur statistischen Absicherung und Kontrolle der Simulationslänge.

`Itimer_probefkt_tab` enthält Einträge für die Meßfunktionen, die für die durch den Intervalltimer angestoßenen Ausgaben definiert sind.

`Cronic_probefkt_tab` enthält Einträge für die Meßfunktionen, die für die Ausgaben definiert sind, welche durch den regelmäßig stattfindenden *Cronical Event* angestoßen werden.

A.15.14 Einlesen der Knotenparameter

```
extern FKT_TABLE l5models_tab[];
extern FKT_TABLE l4models_tab[];
extern FKT_TABLE l3models_tab[];
extern FKT_TABLE l2models_tab[];
extern NODE *nodearray;
void rcnodes(int indent);
```

`Rcnodes()` legt das globale Array `nodearray` an, welches die Informationen über die zu simulierenden Netzknoten enthält.

Das Array `l5models_tab` enthält für jedes implementierte Knoten- (d.h. Layer-5-) Modell einen Eintrag (siehe Abschnitt A.15.4 [`xxgetfd()`], Seite 90) :

`Name` zeigt auf den Namen des Modells, so wie er in der Konfiguration verwendet wird.
`Fkt` zeigt auf eine Funktion, die die für das Modell spezifischen Parameter einliest und im `nodearray` ablegt.
`Explain` zeigt auf einen String, der als Kommentar in der Syntax-Erläuterung ausgegeben wird, falls die globale Variable `minus_x` ungleich null ist.

Die modellspezifische Einlesefunktion bekommt die Parameter `NODE*newnodes`, `int count` und `int indent`.

`Newnodes` zeigt auf ein Array von `NODE` - Strukturen,
`count` gibt die Länge dieses Arrays an. Dadurch werden jeweils mehrere Knoten mit gleichem Parametersatz definiert.
`Indent` gibt die Einrückung der Zeilenanfänge für eventuelle Erläuterungen an.

Die Einlesefunktion muß in den `NODE`s die Komponenten `print_k_info`, `k_init`, `k_cleanup`, `calc_empfaenger` (und `traffic_list`), `l5_pdu_sended` und `l5_pdu_received` ausfüllen. An den Zeiger `k_info` können Datenstrukturen mit privaten Informationen gehängt werden (siehe Abschnitt A.11.1 [*Die* `NODE` - *Struktur*], Seite 82).

Analog dazu enthält das Array `l4models_tab` einen Eintrag für jedes wählbare Layer-4-Modell.

Die Layer-4-Einlesefunktion bekommt die gleichen Parameter wie oben, sie muß in den `NODE`s die Komponenten `l4_sap`, `l4_pdu_sended` und `l4_pdu_received` ausfüllen. An den Zeiger `l4_info` können privaten Informationen gehängt werden.

Das Array `l3models_tab` enthält einen Eintrag für jedes wählbare Layer-3-Modell, das Array `l2models_tab` enthält einen Eintrag für jedes wählbare Layer-2-Modell. Die Schnittstellen zur Einlesefunktion sind wie oben.

A.15.15 Einlesen der Netzparameter

```
extern KEYWD_TABLE l1models_tab[];
void rcnet(void);
```

Das Array `l1models_tab` enthält für jedes implementierte Netz- (d.h Layer-1-) Modell einen Eintrag (siehe Abschnitt A.15.10 [`rcreadkeywds()`], Seite 93) :

Die Komponente `Name` zeigt auf den Namen des Modells, so wie er in der Konfiguration verwendet wird.
`Value` zeigt auf eine Funktion, die die für das Modell spezifischen Parameter einliest.
`Tab` enthält einen `NULL` - Zeiger.
`Type` ist `XXTYPimmhook`.
`Explain` zeigt auf einen String, der als Kommentar in der Syntax-Erläuterung ausgegeben wird, falls die globale Variable `minus_x` ungleich null ist.

Die modellspezifische Einlesefunktion bekommt als Parameter einen Integer, welcher die gewünschte Einrückung der Zeilenanfänge angibt, falls Syntaxerläuterungen oder gelesene Werte ausgegeben werden sollen (wenn `minus_x` oder `printconf` ungleich null sind).

Die Einlesefunktion muß in der globalen Struktur `net_params` die Komponenten `name`, `bits_per_second`, `min_data_len`, `max_data_len`, `net_init` und `l1_sap` ausfüllen (siehe Abschnitt A.12 [Netzbeschreibung], Seite 83).

A.16 Debugging

A.16.1 Debugflags

```
char *rc_get_debug_level(char *modulename, long *level);
```

Debugausgaben und Konsistenzprüfungen sind sehr laufzeitintensiv. Daher kann zur Compilezeit zwischen einer Debuggingversion und einer Production Version von simuLan gewählt werden, indem das Präprozessorsymbol `DEBUG` definiert wird oder nicht. In der Debuggingversion kann auf der Kommandozeile die Fehlersuche in den einzelnen Modulen durch die Angabe von '`-D<modulename>=<flag>`' aktiviert werden. (Die Bedeutung der Debugflags muß im Sourcecode nachgesehen werden.) (Siehe Abschnitt A.4 [Auswertung der Kommandozeile], Seite 72).

`Rc_get_debug_level()` sucht in der Kommandozeile nach '`-D<modulename>=<int>`' und stellt den int-Wert in `*level` zur Verfügung (`long`, weil wir gerne 32 Bits benutzen wollen). Die Bedeutung des Wertes bleibt dem Modul selber überlassen. Es wird ein Zeiger auf den <*int*>-String zurückgegeben, dadurch können die Module ggf. auch nicht-numerische Argumente übergeben bekommen.

A.16.2 Datenstrukturen und Funktionen

```
extern FKT_TABLE event_handler_tab[];
char *get_event_handler_name(void (*f)());
```

In der Debugversion nimmt das Makro `DEFUN_GET_REL_OBJECT()` an, daß die Datenstrukturen eine Komponente `object_nr` enthalten (siehe Abschnitt A.6.3 [*Generische Freilisten*], Seite 75). Alle von den `get_TYP()` - Funktionen gelieferten Objekte bekommen in der Komponente `object_nr` eine eindeutige Kennzeichnung zugewiesen. `Object_nr` sollte die erste Komponente der Datenstrukturen sein. Dadurch lassen sich Programmierfehler leichter finden, bei denen Zeiger auf unbekannte Objekte (`VOID_P`) falsch interpretiert werden.

Zum Debuggen sollte für jede Datenstruktur eine Funktion geschrieben werden, welche die in der Datenstruktur enthaltenen Informationen im Klartext ausgibt. Falls die Struktur einen Zeiger auf unbekannte Privatinformationen enthält, so sollte in der Debugversion auch ein Zeiger auf eine Funktion enthalten sein, die die angehängte Privatinformation ausgibt. Jedesmal, wenn diese Privatinformation geändert wird, muß auch die entsprechende Funktion eingetragen werden. Als Beispiel für eine solche Funktion soll `print_event()` dienen:

```
void print_event(EVENT *event_p, FILE *fd, int depth, int indent)
{
#ifdef DEBUG
 if (fd)
  fprintf(fd, "%*sEVENT: %15.7le | %20s | %4hu | objnr %ld |",
       indent, ,
  (double)event_p->event_time,
  (char *)get_event_handler_name(event_p->event_handler),
  event_p->k_nr,
  event_p->object_nr);

 if (depth-- && event_p->print_event_info)
  (*event_p->print_event_info)(event_p->event_info, fd, depth, indent+2);

#else

 if (fd)
  fprintf(fd, "%*sEVENT: %15.7le | %20s | %4hu |",
```

```
        indent, ,
    (double)event_p->event_time,
    (char *)get_event_handler_name(event_p->event_handler),
    event_p->k_nr);
#endif /* DEBUG */

} /* end print_event() */
```

Mit der Funktion `get_event_handler_name()` können die Namen der `event_handler` - Funktionen im Klartext ausgedruckt werden. `Get_event_handler_name()` sucht zu einem übergebenen Funktionszeiger den Namen aus der Tabelle `event_handler_tab`, die für jede interessierende Funktion einen Eintrag enthalten muß.

A.16.3 Interaktive Fehlersuche

Wenn simuLAn während des Programmlaufes ein Interrupt - Signal (`SIGINT`) erhält (üblicherweise durch Eingabe von `Ctrl-C`), so wird der Simulationslauf unterbrochen und in ein Debug-Menu verzweigt. Dieses ermöglicht interaktiv das Anzeigen und Ändern der Datenstrukturen. Das Menu ist selbtserklärend, darum wird hier nicht weiter darauf eingegangen.

Diese Funktion ist noch ausbaufähig, insbesondere das interaktive (adaptive) Ändern von Simulationsparametern kann interessante Möglichkeiten bieten.

Anhang B

Übersicht über Paketaufbau – Protokolloverheads

Ethernet	Bit	Byte	Sum	
Preamble	56	7	7	
SFD	8	1	8	
Dest.	48	6	14	
Source	48	6	20	
Type	16	2	22	
min. Data	368	46	68	
max. Data	12000	1500		1522
CRC	32	4	72	1526

→**Overhead** = 208 Bit = 26 Byte.

IP	Bit	Byte	Sum
Vers	4	1/2	1/2
Len	4	1/2	1
Serv.typ	8	1	2
Tot.lgth.	16	2	4
Ident	16	2	6
Flags	3		
Frag.Offset	13		8
Time	8	1	9
Proto	8	1	10
Hd.Chksum	16	2	12
Src. IPadr	32	4	16
Dst. IPadr	32	4	20
Options	?	?	?
Data	?	?	?

Arp	Bit	Byte	Sum
Hardware	16	2	2
Protokoll	16	2	4
HLEN	8	1	5
PLEN	8	1	6
Operation	16	2	8
Sender HA	48	6	14
Sender IA	32	4	18
Target HA	48	6	24
Target IA	32	4	28

TCP	Bit	Byte	Sum
Src.Port	16	2	2
Dst.Port	16	2	4
Seq.Nr.	32	4	8
Ack.Nr.	32	4	12
Offset	4		
Reserved	4		13
Code	8	1	14
Window	16	2	16
Chksum	16	2	18
Urgent	16	2	20
Options	?	?	?
Data	?	?	?

UDP	Bit	Byte	Sum
Src.Port	16	2	2
Dst.Port	16	2	4
Length	16	2	6
Chksum	16	2	8
Data	?	?	?

ICMP	Bit	Byte	Sum
Type	8	1	1
Code(0)	8	1	2
Chksum	16	2	4
Ident.	16	2	6
Seq. Nr.	16	2	7

Anhang C

Programmquellen

Die beiliegende Diskette enthält den kompletten Quellcode für simuLan. Dieser umfaßt die Grundbibliothek, die bisher implementierten Protokollschichtmodule sowie die gesamte Dokumentation.

Die Diskette ist auf 720 Kbyte formatiert und mit `tar(1)` beschrieben. Das `tar`-Archiv enthält ein mit `freeze(1)` kompaktiertes `tar`-Archiv des simuLan-Dateibaumes, und die Quellen für das `freeze`-Programm, so wie sie vom Author Kent Landfield in den USENET-News verbreitet wurden (`freeze` erzielt eine deutlich bessere Kompaktierung als das übliche `compress(1)`, ist aber noch nicht so weit verbreitet).

Der Inhalt der Diskette kann mit `tar(1)` eingelesen werden. Dann können die simuLan-Quellen mit `melt(1)` oder `fcat(1)` (den Gegenstücken zu `freeze`) "aufgetaut" und wiederum mit `tar` ausgepackt werden.

```
% tar xvf /dev/rfd0 simuLan.tar.F
% fcat simuLan.tar.F | tar xvof -
```

Im simuLan-Dateibaum liegt ein '`README`', welches eine Anleitung zum Übersetzen und Installieren des Programms und der Dokumentation enthält.

Literaturverzeichnis

Nicht alle hier aufgeführten Veröffentlichungen hatten Einfluß auf die Entwicklung von simuLan. Wichtig sind die Quellen, auf die im Text verwiesen wird, sowie die mit Anmerkungen versehenen. Die anderen Hinweise sind aufgeführt, um einen Überblick über die Literatur zu diesem Thema zu geben.

[AB88] Ian F. Akyildiz und Hunter Bolch. Mean Value Analysis Approximation for Multiple Server Queueing Networks. *Performance Evaluation*, 8:77–91, 1988.

[AC88] C.C. Ang und E.J.P. County. An experiment in modelling communication protocols using NPNs. *Aust. Comput. J. (Australia)*, 20(2):73–8, Mai 1988.

[Ada83] N.R. Adam. Achieving a confidence interval for parameters estimated by simulation. *Management Science*, 29(7):856–866, Juli 1983.

Vergleich dreier Varianten des BMVA-Verfahrens. Die vorgeschlagene Methode B wurde in simuLan implementiert.

[Agr85] S.C. Agrawal. *Metamodelling. A Study of Approximations in Queueing Models.* MIT, 1985.

[Aha87] M. Ahamad. An Efficient Algorithm for Name Resolution in Computer Networks. *Computer Networks and ISDN Systems*, 13:301–311, 1987.

[Akh88] S. Akhtar. Performance evaluation of CSMA/CD networks using Markovian approach. *GLOBECOM '88. IEEE Global Telecommunications Conference and Exhibition - Communications for the Information Age*, 2:1108–12, 1988. Cat. Nr. 88CH2535-3.

[AKK+87] P.D. Amer, R.N. Kumar, R. Kao, T. Phillips, und L.N. Cassel. Local Area Broadcast Network Measurement: Traffic Characterization. *Proceedings of COMCON Spring 87,San Francisco*, Seite 64–70, Februar 1987.

[AL79] G.T. Almes und E.D. Lazowska. The Behavior of Ethernet-like Computer Communications Networks . *Proceedings 7th Symposium on Operating Systems Principles*, Seite 66–81, Dezember 1979.

[ATT] Unix system graphics guide. Available from AT&T Technologies, Inc.

[Azm88] M. Azmoodeh. *Abstract Data Types and Algorithms.* Macmillian Education LTD, 1988.

[BBC77] R.M. Brown, J.C. Browne, und K.M. Chandy. Memory Management and Response Time. *Communications of the ACM*, 20(3):153–165, März 1977.

[BC83] R. Berry und K.M. Chandy. Performance models of token ring local area networks. *Proceedings 1983 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, Seite 266–274, August 1983.

[BC88a] S.L. Beuerman und E.J. Coyle. The Delay Characteristics of CSMA/CD Networks. *IEEE Trans. Commun.*, 36(5):553–563, 1988.

[BC88b] Andreé B. Bondi und Yie-Min Chuang. A new mva-based approximation for closed queueing networks with a preemptive priority server. *Performance Evaluation*, 8:195–221, 1988.

[BDMH87] M. Basu, G. Dutta, T.W. Morgan, und D.P. Heyman. A comparison study of the Queueing Network Analyzer with a CSMA/CD simulation. *Sixth Annual International Phoenix Conference on Computers and Communications. 1987 Conference Proceedings*, Seite 25–27, 1987. Cat. No. 87TH0179-2.

[Ber89] B.R. Bertan. Simulation of MAC layer queueing and priority strategies of CEBus. *IEEE Trans. on Consumer Electronics*, 35:557–563, August 1989.

[BM87] O.J. Boxma und B.W. Meister. Waiting-Time Approximations in Multi-Queue Systems with Cyclic Service. *Performance Evaluation*, 7:59–70, 1987.

[BM89] A. Bricker und M.J.Litzkow. Condor technical summary. Bericht, University of Wisconsin, 1989.

[BMI87] S. Bhatia, H.T. Mouftah, und M. Ilyas. Hybrid modeling techniques for complex computer networks. *IEEE International Conference on Communications '87: Communications - Sound to Light. Proceedings.*, 3:1311–15, 1987. Cat. No. 87CH2424-0.

[BMK] D.R. Boggs, J.C. Mogul, und C.A. Kent. Errata for "Measured capacity of an Ethernet: myths and reality".

[BMK88] D.R. Boggs, J.C. Mogul, und C.A. Kent. Measured capacity of an Ethernet: myths and reality. In *Proc. SIGCOMM '88 Symposium on Communications Architectures and Protocols*, Band 18, Seite 222–34, August 1988.

Based on measurements of an actual implementation, the authors show that for a wide class of applications, Ethernet is capable of carrying it's nominal bandwitdth of useful traffic, and allocates bandwidth fairly.
Hauptsächliche Vorlage für die Verifikation des Ethernetmodells.

[Bor88] Petra Borowka. Hierarchische Modellierung eines SUN-Clusters mit dem Modellierungs- und Analysetool HIT. Diplomarbeit, Uni Dortmund, Informatik IV, Februar 1988.

[Bra88] P. Bratley. *Algorithmics: Theory and practice.* 1988.

[Bro87] R.G. Brookshire. Rating the IBM compatibles. *BYTE*, 12(12):193–6,198, 1987.

[BS87] H. Beilner und F.-J. Stewing. Concepts and techniques of the performance modelling tool, HIT. *Discrete Event Simulation and Simulation and Operations Research. Proceedings of the European Simulation Mulitconference*, Seite 38–43, 1987.

[Bul82] W.G. Bulgren. *Discrete System Simulation*, Kapitel 5,6,7, Seite 119ff. Publisher Hall, 1982.

Simulation, Experimentation and Verification. Generating Pseudo-Random Numbers.

[Bux87] W. Bux. Modeling token ring networks – a survey. *Messung, Modellierung und Bewertung von Rechensystemen: 4. GI/ITG Fachtagung, Erlangen, West Germany, 29. Sept. - 1.Okt 1987*, Seite 192–221, 1987.

[Bux89] W. Bux. Modeling token ring networks – a survey. *Informatik - Fachberichte 205. Kommunikation in verteilten Systemen. Grundlagen, Anwendungen, Betrieb.*, Seite 192–221, 1989.

[Buz76] J.P. Buzen. Fundamental Operational Laws of Computer System Performance. *Acta Informatica*, 7(2):167–182, 1976.

[BW75] J.W. Boyse und D.R. Warn. A Straight forward Model for Computer Performance Prediction. *Computing Surveys*, 7(2), 1975.

[CB89] P. Callahan und B. Bradley. New Token Ring versus Ethernet: Counterpoint. *Data Communications*, 18(1):127–34, Januar 1989.

[CL83] E.J. Coyle und B. Liu. A matrix representation of CSMA/CD networks. *IEEE International Conference on Communications: Integrating Communication for World Progress*, 2:745–8, Juni 1983.

[Com88] Douglas Comer. *Internetworking with TCP/IP*. Prentice Hall, 1988.

Principles, Protocols, and Architecture. Überblick über die wichtigsten RFCs. Sehr gute Einführung in die Internet-Architektur.

[Con88] A.E. Conway. Performance modeling of multi-layered OSI communication architectures. *Submitted to a Conference / ICC 89*, 1988.

Warteschlangenmodelle für Protokollschichten.

[DG88] W. Dobosiewicz und P. Gburzyński. Ethernet with segmented carrier. In *Proceedings of IEEE Computer Networking Symposium*, Seite 72–78, Washington, DC, April 1988.

[DG90] W. Dobosiewicz und P. Gburzyński. Performance of Piggyback Ethernet. In *IEEE IPCCC'90*, Seite 516–522, Scottsdale, AZ, März 1990.

[DGR90] W. Dobosiewicz, P. Gburzyński, und P. Rudnicki. An Ethernet-like CSMA/CD protocol for high speed bus LANs. In *IEEE INFOCOM'90*, Seite 238–245, 1990.

[dM89] G. de Montmollin. The trend is going towards distributed PC networks. *Sysdata (Schweiz)*, 20(3):67,69–70, März 1989.

[DMM88] S. Dravida, M.J. Master, und C.H. Morton. A Method to Analyze Performance of Digital Connections. *IEEE Trans. Commun.*, 36(3):298–305, 1988.

[dSeSM87] Edmundo de Souza e Silva und Richard R. Muntz. Approximate Solutions for a Class of Non-Product Form Queueing Network Models. *Performance Evaluation*, 7:221–242, 1987.

[Dus88] Klaudia Maria Dussa. Messen, Auswerten und Vergleichen von Meßwerten verteilter Anwendungen. Diplomarbeit, Uni Erlangen-Nürnberg, Informatik IV, 1988.

Vergleich von Munix-RPC und TCP.

[ea79] J. Moura et al. Evaluation of collision control algorithms in Ethernets. *Proceedings of the 6th Data Communications Symposium*, 1979. Cat. Nr. 79CH1405-0.

[EDF87] P. Ein-Dor und J. Feldmesser. Attributes of the performance of central processing units: a relative performance prediction model. *Communications of the ACM*, 30(4):308–317, April 1987.

[ET88] C. Evequoz und C. Tropper. On Reassembly Delay in Packet Switching Networks. *Computer Networks and ISDN Systems*, 15:1–25, 1988.

[Far88] B.N. Farah. Local area networks - is there a standard ? *J. Comput. Inf. Syst.*, 28(4):20–3, 1988.

[Fer78] Domenico Ferrari. *Computer Systems Performance Evaluation.* Prentice Hall, 1978.

Model Calibration and Validation.

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

[Fis78] G.S. Fishman. *Principles of Discrete Event Simulation.* John Wiley & Sons, 1978. (FH 8495).

Autoregressive Method.

[FKLA87] V.S. Frost, M.S. Kashefipour, W. Laure, und R. Ananthanpilla. A combined LAP-D/CSMA-CD simulation model for the study of local area networks. *IEEE International Conference on Communications '87: Communications - Sound to Light. Proceedings.*, 3:1305–10, 1987. Cat. No. 87CH2424-0.

[FM77] W.R. Franata und K. Maly. An Efficient Data Structure for the Simulation Event Set. *Communications of the ACM*, 20(8):596–602, August 1977.

[Fra87] Klaus Franke. Zum Verhalten lokaler Netze vom CSMA/CD-Typ bei großem Verkehrsaufkommen. *Wissenschaftliche Zeitschrift der TU Karl-Marx-Stadt*, 29(4):498–503, 1987.

[FWLS88] V.S. Frost, Jr. W.W. LaRue, und K.S. Shanmugan. Efficient techniques for the simulation of computer communications networks. *IEEE Journal on selected Areas in Communications*, 6(1):130–45, Januar 1988.

[GKS+85] K. Görgen, H. Koch, G. Schulze, B. Struif, und K. Truöl. *Grundlagen der Kommunikationstechnologie: ISO-Architektur offener Kommunikationssysteme.* Springer, Berlin, Heidelberg, 1985.

[Gla89] B. Glass. The token ring. *BYTE*, 14(1):363–8,370,372,374,376, Januar 1989.

[GM80] P. Gburzyński und J. Maitan. An object-oriented configurable simulator for low-level communication protocols. Technical report tr 90-8, University of Alberta, Edmonton, Alberta, Canada, März 1980.

[Gon85] T.A. Gonsalves. Performance Characteristics of 2 Ethernets: an Experimental Study. *Proceedings of the ACM SIGMETRICS Conf. on Modeling and Measurement*, Seite 78–86, 1985.

[GR87] P. Gburzyński und P. Rudnicki. A better-than-Token protocol with bounded packet delay time for ethernet-type LAN's. In *Symposium on the Simulation of Computer Networks*, Seite 110–117, Colorado Springs, Co., August 1987. IEEE.

[GR88a] P. Gburzyński und P. Rudnicki. A Simulation Model of a CSMA/CD Communication Channel. *Submitted to INFOCOM '89*, Seite 1–14, Juli 1988.

[GR88b] P. Gburzyński und P. Rudnicki. Modeling low-level communication protocols: A modular approach. In *Proceedings of the 4th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Seite 89–108, Palma de Mallorca, Spain, 1988. Universitat de les Illes Balears.

[GR89a] P. Gburzyński und P. Rudnicki. LANSF—a modular system for modelling low-level communication protocols. In R. Puigjaner und D. Potier, Herausgeber, *Modeling Techniques and Tools for Computer Performance Evaluation*, Seite 77–93. Plenum Publishing Company, 1989.

[GR89b] P. Gburzyński und P. Rudnicki. *The LANSF Protocol Modeling Environment, version 2.0.* The University of Alberta, Department of Computing Science, TR 89-19, Edmonton, 1989.

[GR89c] P. Gburzyński und P. Rudnicki. A note on the performance of ENET II. *IEEE Journal on Selected Areas in Communications*, 7:424–427, April 1989.

[GR89d] P. Gburzyński und P. Rudnicki. On formal modelling of communication channels. In *IEEE INFOCOM'89*, Seite 143–151, 1989.

[GR89e] P. Gburzyński und P. Rudnicki. A virtual token protocol for bus networks: Correctness and performance. *INFOR*, 27:183–205, 1989.

[GR90a] P. Gburzyński und P. Rudnicki. Modelling of reactive systems in SMURPH. In *Proceedings of the European Simulation Multiconference*, Seite 661–667, Erlangen-Nuremberg, W-Germany, Juni 1990.

[GR90b] P. Gburzyński und P. Rudnicki. On executable specifications, validation, and testing of MAC-level protocols. In E. Brinksma, G. Scollo, und Vissers C. A., Herausgeber, *Proceedings of the 9th IFIP WG 6.1 Int. Symposium on Protocol Specification, Testing, and Verification, June 1989, Enschede, The Netherlands*, Seite 261–273. North-Holland, 1990.

[GR91a] P. Gburzyński und P. Rudnicki. LANSF: a protocol modelling environment and its implementation. *Software Practice and Experience*, 21(1):51–76, Januar 1991.

[GR91b] P. Gburzyński und P. Rudnicki. Object-oriented simulation is SMURPH: A case study of dqdb protocol. In *Proceedings of 1991 Western Multi Conference on Object-Oriented Simulation*, Seite 12–21, Anaheim, California, Februar 1991.

[GT88] T.A. Gonsalves und F.A. Tobagi. On the performance effects of station locations and access protocol parameters in Ethernet networks. *IEEE Trans. Commun.*, 36(4):441–9, April 1988.

[Gus87] R. Gusella. The Analysis of Diskless Workstation Traffic on an Ethernet. Bericht UCB/CSD 87/379, Computer Science Division, Uni. California, Berkeley, 1987.

[GW89] O. Gihr und M. Weixler. Messung der Datenverkehrsprofile in lokalen Netzen. *Informatik - Fachberichte 205. Kommunikation in verteilten Systemen. Grundlagen, Anwendungen, Betrieb.*, Seite 861–877, 1989.

[Hau86] B.M. Hauzeur. A model for naming, Addressing, and Routing. *ACM Transactions on Office Information Systems*, 4(4):293–311, Oktober 1986.

[HC87] S.E. Hauser und M.A. Crocker. A performance study of an Ethernet-based image transmission protocol. *Proceedings of the 12th Conference on Local Computer Networks*, Seite 86–94, 1987. Cat. No. 87CH2445-5.

[Hen77] James O. Henriksen. An Improved Events List Algorithm. *Proceedings of the 1977 Winter Simulation Conference*, Seite 547–557, 1977.

An Adaptive version of an indexed list algorithm.

[Hen83] James O. Henriksen. Event List Management - A Tutorial. *Proceedings of the 1983 Winter Simulation Conference*, Seite 543–551, 1983.

[Hic88] P.R. Hicks. Throughput and delay performance measurement of packet switched data networks. *Aust. Telecommun. Res. (Australia)*, 22(2):23–37, 1988.

[HKW88] M.B. Hidalgo, W.S. Kelly, und T.N. Washburn. Verification / validation of performance simulation using ANSI X3.102 measurements. *IEEE Netw.*, 2(6):15–16,23–28, November 1988.

[HL82] H.D. Huges und L. Li. Simulation model of an Ethernet. *Computer Performance (GB)*, 3(4):210–17, Dezember 1982.

[HL83] H.D. Huges und L. Li. A performance study of ethernet. *Proceedings of the CMG XIII International Conference, San Diego, CA, USA, 14-17 Dec. 1982*, Seite 79–85, 1983.

Simple discrete event simulation of Ethernet. Different network loads and backoff schemes.

[HL87] Ching-Tarng Hsieh und Simon S. Lam. Two Classes of Performance Bounds for Closed Queueing Networks. *Performance Evaluation*, 7:3–30, 1987.

[HL88] E. Heck und R. Lange. An aggregated model of CSMA/CD as an example for runtime reduction in LAN-simulation. *Proceedings: Workshop on the Future Trends of Distributed Computing Systems in the 1990s*, Seite 51–6, 1988. Cat. No. 88TH0228-7.

[HM82] B. Hawe und M. Marathe. Performance of a simulated Ethernet programming enviroment. *Electro/82 Conference Record*, Seite 21–4/1–8, Mai 1982.

[HO90] W.Wilson Ho und Ronald A. Olsson. An approach to genuine dynamic linking. Bericht, University of California, Davis, 1990.

Keywords : Dynamic Linking, Incremental Program Development, Program Customization, Debugging and Testing.

[IEE85] Local Area Networks: Token-Passing Bus, Std 802.4. Bericht, ANSI/IEEE, 1985.

[IEE88a] Local Area Networks: Introduction, Std 802.1. Bericht, ANSI/IEEE, 1988.

[IEE88b] Local Area Networks: Logical Link Control, Std 802.2. Bericht, ANSI/IEEE, 1988.

[IEE88c] Local Area Networks: CSMA/CD, Std 802.3. Bericht, ANSI/IEEE, 1988.

Der Ethernet-Standard.

[IEE89] Local Area Networks: Token Ring Access Method and physical Layer Specifications, Std 802.5. Bericht, ANSI/IEEE, 1989.

[IM88] M. Ilyas und H.T. Mouftah. Simulation tools for computer communication networks. *GLOBECOM '88. IEEE Global Telecommunications Conference and Exhibition - Communications for the Information Age*, 3:1702–6, 1988.

[ISO88] ISO7498: Basic Reference Model for Open Systems Interconnections (CCITT Recommendation X.200). Bericht, ISO/CCITT, 1988.

[ISO89] Status of OSI (and related) Standards 6/1/89. *Computer Communication Reviews*, 19(3), Juli 1989.

[JM88] J. Jackman und D.J. Medeiros. A graphical methodology for simulating communication networks. *IEEE Trans. Commun.*, 36(4):459–64, April 1988.

[Jon86] D.W. Jones. An Empirical Comparison of Priority-Queue and Event-Set Implementations. *Communications of the ACM*, 29(4):300–311, April 1986.

Vergleich von Linearer Liste, Implicit Heap, Leftist Tree, Two-List Event-Set, Henriksen's Event Set, Binomial Queue, Pagode, Skew Heap, Spaly Tree, Pairing Heap. Es gibt keinen universell optimalen Event-Set Algorithmus; Bottom Up Skew Heaps sind unter den meisten Voraussetzungen gut einsetzbar.

[KB88] R. Kuruppillai und N. Bengston. Performance analysis in local area networks of interconnected token rings. *Computer Communications*, 11(2):59–64, April 1988.

[Kel86] K.H. Kellermayr. *Lokale Computernetzwerke - LAN.* Springer, Wien, New York, 1986.

[Khe88] Naim A. Kheir. *Systems Modelling and Computer Simulation.* Marcel Dekker Inc, 1988.

[Kin89] B. Kindel. How fast is fast ? *BYTE*, 14(2):251–4, Februar 1989.

Measuring a computer's performance is not as simple as users would like.

[Kle76] Kleinrock. *Queuing Systems*, Band I + II. Wiley, 1976.

Ausführliche Darstellung der Warteschlangentheorie.

[Kle87] J.P.C. Kleijnen. *Statistical Tools for Simulation Practioners.* Marcel Dekker Inc, 1987.

[Knu73] D.E. Knuth. *The Art of Computer Programming*, Band 3, Sorting and Searching. Addison-Wesley, 1973.

[Kob78] H. Kobayashi. *Modeling and analysis. An introduction to system performance evaluation methodology.* Addison-Wesley, 1978. Aus der Lehrbuch-Sammlung.

[KP86] R.M. Keller und P. Panangaden. Semantics of digital networks containing intermediate modules. *Distributed Computing*, 1:235–245, 1986.

[Lan83] H. Langendörfer. *Rechnerverbund*, Band 5 der Reihe *Informatik-Skripten.* Technische Universität Braunschweig, 1983.

[Lan86] H. Langendörfer. *Messen, Modellieren, Simulation*, Band 16 der Reihe *Informatik-Skripten.* Technische Universität Braunschweig, 1986.

[Lan88] Rainer Lange. Ein Werkzeug zur LAN-Modellierung. Diplomarbeit, Uni Dortmund, Informatik IV, Februar 1988.

Verschiedene Betriebssystem- und Benutzermodelle, Aggregierung von Zugriffsverfahren und Übertragungsmedien (in HIT).

[Lav83] Stephen S. Lavenberg, Herausgeber. *Computer Performance Modeling Handbook.* Notes and Reports in Computer Sience and Applied Mathematics. Academic Press, 1983.

[LB87] C.E. LaBarre und P.J. Brusil. Architecture for evaluating LAN traffic and performance. *Proceedings of the 12th Conference on Local Computer Networks*, Seite 109–16, 1987. Cat. No. 87CH2445-5.

[LC79] A.M. Law und J.S. Carson. A Sequential Procedure for Determining the Length of a Steady-State Simulation. *Operations Research*, 27(5):1011–1025, 1979.

[LG87] U. Lübbe und O. Gihr. Messung von Verkehrsprofilen in Lokalen Netzen. *DATACOM*, unknown volume(11-12):88–94, 1987.

[LH88] M.S. Leuner und J.L Hammond. A monitoring system for an Ethernet installation. *Proceedings of the Twentieth Southeastern Symposium on System Theory*, Seite 160–4, 1988. Cat. No. 88CH2553-6.

[Lit85] D. Litzba. *Auswertung von Simulationsdaten mittels autoregressiver Modelle*, Band 203. Uni Dortmund, 1985.

[LK82] A.M. Law und W.D. Kelton. *Simulation Modelling and Analysis*. McGraw - Hill, 1982.

[LMB88] J.B. Lewoc, L.Z. Misiura, und L. Blach. Performance evalution for computer networks. *Adv. Model. Simul.*, 14(2):37–43, 1988.

[LO89] P.A.W. Lewis und E.J. Orav. *Simulation Methodology for Statisticans, Operations Analysts, and Engeneers (Volume 1)*, Band 1 der Reihe *Statistic/Probability*. Wadsworth & Brooks/Cole, 1989. (FN 9724 (1)).

[LS88] P. Liggesmeyer und L. Sturm. Augmented petri nets - a specification tool for complex applications. *Angewandte Informatik*, 30(6):252–8, Juni 1988.

[LSS89a] D. Litzba, M. Sczittnick, und F.-J. Stewing. *Eine Online-Update-Auswertungskomponente auf Basis autoregressiver Modelle*. Uni Dortmund, Informatik IV, April 1989.

Auswertung von Ausgaben von BORIS und HIT.

[LSS89b] D. Litzba, M. Sczittnick, und F.-J. Stewing. Yet another simulation output analysis algorithm: the autoregressive, online-update evaluation technique of the modelling tool, HIT. *(to be) presented at the 3rd European Simulation Conference, Edinburgh*, September 1989.

Simulation Run Length Control.

[LZ86] E.D. Lazowska und J. Zahorjan. File Access Performance of Diskless Workstations. *ACM Transactions on Computer Systems*, 4(3):238–268, August 1986.

[LZGS84] E.D. Lazowska, J. Zahorjan, G.S. Graham, und K.C. Sevcik. *Quantitative System Performance. Computer Systems Analysis Using Queueing Network Models*. Prentice Hall, Inc., 1984.

[Mac87] M.H. MacDougall. *Simulating Computer Systems. Techniques and Tools*. Computer Systems Series. MIT, 1987.

[MC90] B. Müller-Clostermann. Modelle von Rechensystemen - Zuverlässigkeit, Leistung, Funktion. Skript einer Vorlesung an der TU Braunschweig, 1990.

Zuverlässigkeitsnetze, Markov-Ketten, Petri-Netze.

[MCF87] M.A. Marsan, G. Chiola, und A. Fumagalli. Timed Petri net model for the accurate performance analysis of CSMA/CD bus LANs. *Computer Communications (UK)*, 10(6):304–12, Dezember 1987.

[Mee86] F. Meelamkavil. *Computer Simulation and Modelling*. John Wiley & Sons, 1986.

[Mel87] A.S. Melamed. Performance of Unix-based network file systems. *IEEE Micro (USA)*, 7(1):25–38, Februar 1987.

[MFW79] J.A.B. Moura, J.A. Field, und J.W. Wong. Evaluation of Collision Control Algorithms in Ethernets. *Proceedings of the 6th Data Communications Symposium*, 1979. CH1405-0/79/0000-0082.

[MHO89] J.W. Münch, L. Hofman, und R. Ostermann. Durchsatzmessungen auf einem Breitbandnetz. *Eingereicht zur Veröffentlichung für die Proceedings des 8. GI-Fachgesprächs über Rechenzentren am 02. / 03. März 1989 in Düsseldorf*, 1989.

[ML84] M. Melman und M. Livny. Models of the Ethernet protocol using DISS language. *4th Jerusalem Conference on Information Technology (JCIT)*, Seite 89–98, Mai 1984. Cat. No. 84CH2022-2.

[Mol85] Michael K. Molloy. Collision Resolution on the CSMA/CD Bus. *Computer Networks and ISDN Systems*, 85(?)(9):209–214, 1985.

[MS81] W.M. McCormack und R.G Sargent. Analysis of Future Event Set Algorithms for Discrete Event Simulation. *Communications of the ACM*, 24(12):801–812, Dezember 1981.

[MT88] M. Murata und H. Takagi. Two-layer modeling for local area networks. *IEEE Trans. Commun.*, 36(9):1022–34, September 1988.

[OHdNB87] S. Ong, E.C. Hu, A. Martin de Nicolas, und R.A. Baluch. A performance evaluation of microprocessors. *1987 SID International Symposium. Digest of Technical Papers*, Seite 408–11, 1987.

[OJH84] P.J.P. O'Reilly und Jr. J.L Hammond. An efficient simulation technique for performance studies of CSMA/CD local networks. *IEEE J. Sel. Areas Commun. (USA)*, SAC-2(1):238–49, Januar 1984.

[OlWmK88] M.T. Ozsu, Kok lung Wong, und Tse men Koon. System modeling and analysis using Petri nets. *Syst. Anal.-Model.-Simul. (East Germany)*, 5(1):3–25, 1988.

[O'R87] P. O'Reilly. Burst and Fast Packet Switching: Performance Comparisons. *Computer Networks and ISDN Systems*, 13:21–32, 1987.

[Par85] P. Parizek. Experience and results in simulation modeling of computer networks. *Systems Analysis and Simulation 1985. Proceedings of the 2nd International Symposium*, 1:441–4, August 1985.

[Pet91] Stefan Petri. s$_{i}$m$_{u}$L^{a}n – Ein Werkzeug zur Simulation lokaler Netze. Diplomarbeit, TU Braunschweig, 1991.

[Pid89] Michael Pidd. *Computer Modelling for Discrete Simulation.* John Wiley & Sons, 1989.

[PM90] S.R. Pakkam und C.N. Manikopoulos. Performance evaluation of the Consumer Electronic Bus. *IEEE Transactions on Computer Electronics*, 1990. (In press.).

[PP88] A. Pattavina und G. Pacifici. Performance Evaluation of Ring Networks Supporting a Packet Voice Service. *Computer Networks and ISDN Systems*, 15:31–45, 1988.

[Pre88] L. Press. Benchmarks for LAN performance evaluation. *Communications of the ACM*, 31(8):1014–17, August 1988.

[PS87] K. Prasad und A. Singhal. Simulation of ethernet performance based on single server and single queue model. *Proceedings of the 12th Conference on Local Computer Networks*, Seite 74–85, 1987. Cat. No. 87CH2445-5.

[PsT88] Gee-Swee Poo und Tiow seng Tan. Performance comparison of PC LAN file servers - an experimental study. *Computer Communications*, 11(2):71–9, April 1988.

[Ram87] K. Ramamritham. Channel Characteristics in Local-Area Hard Real-Time Systems. *Computer Networks and ISDN Systems*, 13:3–13, 1987.

[Ram88] G. Ramamurthy. An analytical model for UNIX systems. *AT&T Technical Journal*, Seite 86–99, September 1988.

[RFC81] Transmission control protocol. Bericht, for Defense Advanced Research Project Agency by Information Sciences Institute, University of Southern California, September 1981.

[RS88] K.M. Rege und B. Sengupta. Response Time Distribution in a Multiterminal Computer with Terminal Traffic. *Performance Evaluation*, 8:41–50, 1988.

[SA85] B.W. Stuck und E. Arthurs. *A computer and communications network analysis primer*. Prentice Hall Software Series. Prentice Hall, Inc., 1985.

Nette Einführung in die analytische Modellierung einfacher Systeme, einschließlich einfacher Modelle von LANs - aber mehr auch nicht.

[Sar87] R.G. Sargent. An overview of verification and validation of simulation models. *1987 Winter Simulation Conference Proceedings*, Seite 33–9, 1987.

[SC81] C.H. Sauer und K.M. Chandy. *Computer Systems Performance Modelling*. Prentice Hall, 1981.

[Sch82a] B. Schmeiser. Batch size effects in the analysis of simulation output. *Operations Research*, 30(3):556–568, 1982.

[Sch82b] B. Schmidt. Computer science and the general theory of models - an introduction. *Angewandte Informatik*, 24(1):35–42, Januar 1982.

[Sch82c] B. Schmidt. Die Bestimmung von Konfidenzintervallen in der Simulation stochastischer zeitdiskreter Systeme. *Elektronische Rechenanlagen*, 24(3):118–124, 1982.

Vergleich verschiedener Methoden: Independent Replication, Batch Means, Regenerative, Spectrum, Autoregression.

[Sch85] B. Schwarmer. Application of Computer aided tools in system analysis using modelling techniques. *Angewandte Informatik*, 27(6):225–31, Juni 1985.

Über die Tools COPE und USPE.

[SD91] Ralf Schmidt-Dannert. $s_{i}m_{u}L^{a}n$: Modellierung und Simulation lokaler Netzewerke. Diplomarbeit, TU Braunschweig, 1991.

[Sed88] Robert Sedgewick. *Algorithms*, Kapitel 11 - Priority Queues, Seite 145–162. Addison-Wesley Series in Computer Sience. Addison-Wesley Publishing Company, Inc., second edition Auflage, 1988.

Das Kapitel "Priority Queues" diente als Vorlage für die Implicit Heap - Implementation der Eventliste.

[SH80] J.F. Shoch und J.A. Hupp. Measured Performace of an Ethernet Local Network. *Communications of the ACM*, 23(12):711–721, Dezember 1980.

[Sha88] J.G. Shanthikumar. On some queueing models of time division multiple access communication system. *Performance Evaluation*, 8(4):289–304, August 1988.

[SM83] C.H. Sauer und E.A. MacNair. *Simulation of Computer Communication Systems*. Prentice Hall, 1983.

[Spa88] S. Spanier. Design and implementation of a LAN monitoring tool. *Computer Communications*, 11(2):85–9, April 1988.

[Sri88] M.M. Srinivasan. An approximation for mean waiting times in cyclic server systems with nonexhaustive service. *Performance Evaluation*, 9(1):17–33, November 1988.

[SSSB86] T. Simon, O. Spaniol, und J. Suppan-Borowka. Performance measurements of different protocol layers in a LAN enviroment. *EFOC/LAN 86 Proceedings.*, Seite 198–204, 1986.

[ST83] Daniel Dominic Sleator und Robert Endre Tarjan. Self-Adjusting Binary Trees. *Proceedings of the ACM SIGCAT Symposium on Theory of Coumputing*, Seite 235–245, 1983.

Die Vorlage für die BUSH-Implementation der Eventliste.

[ST86] Daniel Dominic Sleator und Robert Endre Tarjan. Self-Adjusting Heaps. *SIAM Journal on Computing*, 15(1):52–69, Februar 1986.

[Sta88] Richard M. Stallman. GNU Emacs Manual, 1988.

[Sta91] Richard M. Stallman. Using and Porting GNU CC, 1991.

[Str89] N. Strole. Inside Token Ring Version II, according to Big Blue. *Data Communication*, 18(1):117–25, Januar 1989.

[Swe78] H.D. Swetman. Hybrid Simulation Models of Computer Systems. *Communications of the ACM*, 21(9):718–723, September 1978.

[Tan87] A.S. Tanenbaum. *Operating Systems*. Prentice Hall, 1987.

[Tan89] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 2. Auflage, 1989.

[TB84] A. Thomasian und P. Bay. Analytic solution of an integrated performance model of a computer communication network with window flow control. *Computer Communication Reviews*, 14(2):225–32, 1984.

[TGR88] P. Tran-Gia und T. Raith. Performance analysis of finite capacity polling systems with nonexhaustive service. *Performance Evaluation*, 9(1):1–16, November 1988.

[TH79] T.A. Tobagi und V.B. Hunt. A performance analysis of carrier sense multiple access with collision detection. *Proceedings of the Loacal Area Communication Network Symposium*, 1979.

[TM87] H. Takagi und M. Murata. Output processes of slotted persistent CSMA and CSMA/CD systems. *Computer Network ISDN Systems*, 13(4-5):333–46, 1987.

[TTH88] T. Takine, Y. Takahashi, und T. Hasegawa. An approximate analysis of a buffered CSMA/CD. *IEEE Trans. Commun.*, 36(8):932–41, August 1988.

[Ulf86] H.E. Ulfers. PACKSIM – a personal computer implementation of a packet switched network simulation model. *Proceedings of the 1986 Winter Simulation Conference*, Seite 769–775, 1986.

[Vid88] V.P. Vidomenko. Criterial evaluation of network topologies. *Autom. Control Comput. Science*, 22(4):27–33, 1988.

[Wil77] N.C. Wilhelm. A General Model for the Performance of Disk Systems. *Journal of the Association for Computing Machinery*, 24(1):14–31, Januar 1977.

[WKC+] Th. Williams, C. Kelley, J. Campbell, D. Kotz, und R. Lang. GNUPLOT – an interactive plotting program.

[WMF80] J.W. Wong, J.A.B. Moura, und J.A. Field. Hierarchical modelling of local area computer networks. *NTC '80. IEEE 1980 National Telecommunications Conference, Houston*, Seite 37.1/1–7, 1980.

[WS88] A.C. Weaver und C.F. Summers. The IEEE token bus - A performance bound on GM MAP. *IEEE Trans. Ind. Electron.*, 35(1):13–17, Februar 1988.

Begriffsdefinitionen

Ein *Rechnernetz* besteht aus einzelnen *Netzknoten*, die durch ein *Netz* verbunden sind.

Ein Netzknoten wird durch eine *Knotenbeschreibung* beschrieben, das Netz durch eine *Netzbeschreibung*.

Ein Netzknoten umfaßt mehrere (nämlich vier) *Protokollschichten* (Nr. 2 - 5), das Netz besteht aus einer Protokollschicht (Nr. 1).

Ein *Protokollschichtmodell* modelliert eine Protokollschicht, es wird durch ein *Protokollschichtmodul* realisiert (im C–Programm).

Die Protokollschichtmodelle für Protokollschicht Nr. 1 werden auch *Netzmodell* genannt, das Protokollschichtmodul dazu *Netzmodul*.

Die Protokollschichtmodelle für Protokollschicht Nr. 5 werden auch als *Knotenmodell* bezeichnet, die Protokollschichtmodule dazu als *Knotenmodule* (diese Schicht modelliert die Vorgänge "im Inneren des Knotens").

Das Rechnernetz insgesamt wird durch eine *Modellbeschreibung* beschrieben.

Die *Konfiguration* einer Simulation enthält neben der Modellbeschreibung auch Angaben über die Simulationsumgebung (Zuordnung von Ausgabekanälen, Variation von Parametern der Modellbeschreibung, Beschränkung von Resourcen).

Ein *Programmlauf* beginnt mit dem Aufruf von s$_{\text{j}}$m$_{\text{u}}$l^{a}n, und endet wenn s$_{\text{j}}$m$_{\text{u}}$l^{a}n terminiert.

Während eines Programmlaufes werden kein, ein oder mehrere *Simulationsläufe* durchgeführt, abhängig u.a. davon, was in der Konfiguration über die Variation von Parametern angegeben wird.

Die Peer-to-Peer- Kommunikation erfolgt in Protokollschicht 1 mittels *Frame*s, in Protokollschicht 2 mittels *Paket*en, in Protokollschicht 3 mittels *Datagramm*en, in Protokollschicht 5 mittels *Nachricht*en.

Frame, Paket, Datagramm und Nachricht werden allgemein als *Protocol Data Unit* (*PDU*) bezeichnet.

Die Notation `popen(3)` bezieht sich auf die Funktion `popen()`, wie sie in Sektion 3 des Unix Reference Manuals beschrieben ist. (Sektion 1 : Benutzerkommandos, Sektion 2 : Systemaufrufe, Sektion 3 : Standardfunktionen)

Gesamtindex

A

Absicherungs-Funktion . . . 13
Absicherungs-Funktionen, Tabelle der . . . 96
Allgemeine Datentypen . . . 71
Allgemeine Makros . . . 72
Allgemeine Variablen . . . 71
Analytische Modelle . . . 9
Application Layer . . . 5
Aufsetzpunkte . . . 34
Ausblicke . . . 70
Ausdrucken von Datenstrukturen . . . 98
Ausgabeformatierung . . . 29, 86
Ausgabeumlenkung . . . 25, 90
Auswertung der Kommandozeile . . . 72
Autoregression . . . 14

B

Batch Mean Value Analysis . . . 14
Batches . . . 14
Begriffsdefinitionen . . . 115
`bimodal()` . . . 80
bit (als Einheit) . . . 91
`bits_per_second` . . . 83
BMVA . . . 14
BMVA-Algorithmus . . . 15
Bottom Up Skew Heap . . . 77
Bus . . . 7
BUSH . . . 76, 77

C

`calc_empfaenger` . . . 82
`CANCEL_EVENT()` . . . 76
Carrier Sense Multiple Access . . . 7
CEP . . . 4
`CHANGE_EVENT_TIME()` . . . 76
`CHANGE_EVENT_TIME_AND_HANDLER()` . . . 76
Checkpointing . . . 34
Compilerflags . . . 97
Condor . . . 34
Confidence Level . . . 13
Connection Endpoint . . . 4
`control()` . . . 77
`copy_probepoint()` . . . 93
`cronic_probefkt_tab` . . . 96
CSMA/CD . . . 7

D

darstellbare Topologien . . . 19
Data Link Layer . . . 4
Datagramm . . . 115
Datenstrukturen und Funktionen (Debug) . . . 98
Datenstrukturen, Ausdrucken von . . . 98
DEBUG-Option . . . 97
Debugflags . . . 97
Debugging . . . 97
Debugoptionen in Kommandozeile . . . 97
`DEFUN_GET_REL_OBJECT()` . . . 75
`DELAY_EVENT_TIME()` . . . 76
Dienstprimitive . . . 3
Dienstprotokolle . . . 3
Dienstzugangspunkte . . . 3
Discrete Event Simulation . . . 10
`dont_run_it` . . . 72
dynamische Aggregierung . . . 18

E

Einheiten in der Konfiguration . . . 91
Einleitung . . . 1
Einlesen der globalen Simulationsparameter 96
Einlesen der Knotenparameter . . . 96
Einlesen der Konfiguration . . . 89
Einlesen der Netzparameter . . . 97
Einlesen von Verkehrsverteilungen . . . 95
Einlesen von Verteilungsfunktionen . . . 95
`END_OF_SIMULATION()` . . . 72, 89
Ereignis, nächstes . . . 77
ereignisorientierte Simulation . . . 10
Ereignisorientierte Simulation . . . 10
Ereignisse . . . 10
`erlang()` . . . 80
Ethernet . . . 7
`EVENT` . . . 76
`event_handler_tab` . . . 98
Eventliste . . . 36, 76
`expntl()` . . . 80
`EXPNTL()` . . . 80

F

FCFS . . . 78
Fehlermeldungen . . . 34
FIFO . . . 37, 78
File-Identifier . . . 90
Fixed Time Increment . . . 11
`FKT_TABLE` . . . 92
Frame . . . 4, 44, 48, 115
freeze . . . 102
Freispeicherlisten . . . 75
Funktionenindex . . . 123
Funktionsnamen, Tabelle der (Debugging) . 98

G

GENERAL_ERROR . . . 72

Generische Freilisten 75
Gesamtindex 116
`get_TYP` 75
`get_EVENT()` 76
`get_event_handler_name()` 98
`get_MS_ANSWER()` 60
`get_next_event()` 77
`get_PTP_FRAME()` 48
`get_SADU()` 84
`glob_params` 73
Globale Simulationsparameter 73
Globale Simulationsparameter, Tabelle der . 96

H

Heap 76
Hybrid-Modelle 9
`hyperexpntl()` 80

I

Implement. Wahrscheinlichkeitsverteilungen 80
Implicit Heap 77
Index 116
Indivisible Time Unit 11, 71
`insert_event()` 76
Instrumentierung 86
Instrumentierung einlesen 92
`INT_EXPNTL()` 80
`INT_UNIFORM()` 80
Interaktive Fehlersuche 99
Interrupt-Signal 99
IO_ERROR 72
`itimer_probefkt_tab` 96
ITU 11, 71
`itu_per_bit` 83
`itu_per_second` 73

K

`k_cleanup` 82
`k_init` 82
`KEYWD_TABLE` 93
`knoten_anzahl` 73
Knotenbeschreibung 82, 115
Knotenmodell 20, 115
Knotenmodelle, Tabelle der 96
Knotenmodule 115
Knotenname 28
Kommandozeile, Auswertung 72
Kommandozeile, Debugoptionen 97
Kommentare in der Konfiguration 24
Konfidenzintervall 13
Konfiguration 115
Konfiguration, '`FORMAT`' 29
Konfiguration, Ausgabeformate 29
Konfiguration, Ausgabeumlenkung 25
Konfiguration, Einheiten 91
Konfiguration, Instrumentierung 29
Konfiguration, Kommentare 24
Konfiguration, Laufvariablen 25
Konfiguration, Sonderzeichen 24
Konfiguration, Vorspann 24
Konfigurationsdatei 22
Konfigurationsdatei, Aufbau 22
`konstant()` 80

L

Längeneinheiten 91
l*n*_pdu_received 85
l*n*_pdu_sended 86
l*n*_sap 85
`l1_sap` 83
`l1models_tab` 97
`l2models_tab` 96
`l3models_tab` 96
`l4models_tab` 96
`l5models_tab` 96
LAN 7
Laufvariablen 25
Layer1-Modelle, Tabelle der 97
Layer2-Modelle, Tabelle der 96
Layer3-Modelle, Tabelle der 96
Layer4-Modelle, Tabelle der 96
Layer5-Modelle, Tabelle der 96
lexikalische Analyse 90
LIFO 37, 78
`listen_empf()` 81
Literaturverzeichnis 103
LLC 7
Local Area Networks 7
Logical Link Layer 7
Lokale Netze 7
`lookupnodename()` 83

M

MAC 7
`make_nqeue()` 78
`make_tqeue()` 79
MandelSpawn 57
`max_cpu_time` 73
`max_data_len` 83
`max_events` 73
MAX_Q_LEN_EXCEEDED 72
`max_q_length` 73
`max_simul_time` 73
Meßdaten 86
Mefl@dq sfunktion 12, 45
Mefl@dq spunkt 12, 45
`MEASURE()` 87
Medium Access Layer 7
`menu()` 99
meter (als Einheit) 91
Metropolitan Area Networks 7
`min_data_len` 83
`minus_x` 72
Modellbeschreibung 115
Modellieren mit simuLAN 19
Monte Carlo Methode 11
`MS_ANSWER` 59
`MS_ANSWER_freelist` 60
`MS_REQUEST` 59
mslaved 57

mslaved_endofservice() . . . 66
mslaved_gwc42() . . . 64
MSLAVED_INFO . . . 60
mslaved_pdu_received() . . . 64
mslaved_pdu_sended() . . . 67
mslaved_print_info() . . . 68
mslaved_rc() . . . 62
mslaved_timeout() . . . 66
myalloc() . . . 74
MYALLOC_MAGIC . . . 74
myfree() . . . 74
myrealloc() . . . 74
mystrdup() . . . 75

N

N-Entities . . . 3
N-Instanzen . . . 3
N-Protokoll . . . 3
Nachricht . . . 115
Nachrichten empfangen . . . 85
Nachrichtenlängen . . . 91
Name - Wert - Paare . . . 26
net_cleanup . . . 83
net_init . . . 83
net_params . . . 83
Network Layer . . . 4
Netz . . . 115
Netzbeschreibung . . . 83, 115
Netzknoten . . . 115
Netzmodell . . . 115
Netzmodelle, Tabelle der . . . 97
Netzmodul . . . 115
Netzparameter . . . 83
Netzparameter, Einlesen der . . . 97
NODE . . . 82
nodearray . . . 82, 96
normal() . . . 80
Normale Warteschlange . . . **37**, 78
nq_pop() . . . 78
nq_push_fifo() . . . 78
nq_push_lifo() . . . 78
NQUEUE . . . **37**
NQUEUE . . . 78

O

object_number . . . 71
OUT_OF_MEM . . . 72

P

Paket . . . 115
Paketaufbau . . . 100
Partnerinstanz . . . 3
PDU . . . 3, 43, 115
Peer-Entity . . . 3
Physical Layer . . . 4
Point-To-Point - Netz . . . 47
Presentation Layer . . . 5
print_event() . . . 78, 98
print_ms_answer() . . . 68
print_nqueue() . . . 78
print_sadu() . . . 84
print_tqueue() . . . 79
printconf . . . 72
Protokollschichten in simuLan . . . 19
Protokollschichten, Schnittstellen . . . 84
Protokolldateneinheiten . . . 3
Protokollschichtmodell . . . 20, 115
Protokollschichtmodul . . . 115
Protokollschichten . . . 115
PROBE . . . 87
PROBE einlesen . . . 92
probe_printf() . . . 86
PROBEPOINT kopieren . . . 93
Probepoints . . . 30
Programmers Reference . . . 71
Programmierhandbuch . . . 71
Programmlauf . . . 115
Programmquellen . . . 102
Protocol Data Unit . . . 3, 43, 115
PTP - Netz . . . 47
ptp_cleanup() . . . 52
ptp_data_encapsulation() . . . 52, 54
PTP_FRAME . . . 48
PTP_FRAME_freelist . . . 48
ptp_info . . . 49
ptp_init() . . . 52
ptp_mean_throughput() . . . 56
ptp_print_frame() . . . 57
ptp_probept_end_transmit_tab . . . 49
ptp_rc() . . . 50
ptp_rc_throughput() . . . 55
ptp_stoerzeit . . . 48
ptp_throughput() . . . 56
ptp_transmit_error() . . . 55

Q

Quellen . . . 102

R

RANDOM . . . 79
Randomseed . . . 79
rc() . . . 89, 96
rc_debug_level . . . 72
RC_ERROR() . . . 89
RC_FATAL() . . . 89
rc_get_debug_level() . . . 72, 97
RC_SYNTAX_ERROR() . . . 89
RC_WARNING() . . . 89
rcexplainkeywds() . . . 93
rcfiles() . . . 90
rcnet() . . . 97
rcnodes() . . . 96
rcparmtab . . . 96
rcprintdefaults() . . . 93
rcprobepoint() . . . 92
rcptpnettab . . . 50
rcreadkeywds() . . . 93
Rechnernetz . . . 115
Regeneration . . . 14
release_TYP . . . 75

`release_EVENT()` . . . 76
`release_MS_ANSWER()` . . . 60
`release_PTP_FRAME` . . . 48
`release_SADU()` . . . 84
Replikation . . . 13

S

`SADU` . . . 43, 84
SAP . . . 3, 85
Schichtenprotokoll . . . 3
Schlüsselworte . . . 93
Schnittstellen zw. Protokollschichtmodulen . 84
seconds (als Einheit) . . . 91
sed . . . 68
Sendeauftrag abschließen . . . 86
Service Access Data Unit . . . 43
Service Access Point . . . 3, 83, 84
Service Primitives . . . 3
Session Layer . . . 4
Signale . . . 34
`simul_time` . . . 71
Simulationsergebnis . . . 86
Simulationslauf . . . 115
Simulative Methoden . . . 9
Simulierte Zeit . . . 71
Sonderzeichen in der Konfiguration . . . 24
Speicherverwaltung . . . 74
Spektrum-Methode . . . 14
`SRANDOM` . . . 79
`stat_not_ready` . . . 89
`stat_wait_flag` . . . 89
STATISTICS_READY . . . 72
STATISTICS_STOP . . . 72
Statistische Absicherung . . . 89
Statistische Absicherung, Tabelle der Funkt. 96
`statmoduls_tab` . . . 96
symbolischen Namen . . . 92
Syntax Error . . . 89
Syntaxtabelle . . . 93

T

Tabelle der Absicherungs-Funktionen . . . 96
Tabelle der Funktionsnamen (Debugging) . 98
Tabelle der Globale Parameter . . . 96
Tabelle der Knotenmodelle . . . 96
Tabelle der Layer1-Modelle . . . 97
Tabelle der Layer2-Modelle . . . 96
Tabelle der Layer3-Modelle . . . 96
Tabelle der Layer4-Modelle . . . 96
Tabelle der Layer5-Modelle . . . 96
Tabelle der Netzmodelle . . . 97
Ticket Queue . . . **37**, 78
Tickets . . . 37, 78
`TIME_T` . . . 71
Token-Bus . . . 8
Token-Ring . . . 8
Tokens . . . 37, 78
Topologien, darstellbare . . . 19
`tq_pop()` . . . 79
`tq_push_fifo()` . . . 79
`tq_push_lifo()` . . . 79
TQUEUE . . . **37**
`TQUEUE` . . . 78
Trace Driven . . . 11
`TRAFFIC` . . . 81
`traffic_rc()` . . . 95
Trafficliste . . . 81
Transport Layer . . . 4
`TYP_freelist` . . . 75
Typenindex . . . 121

U

UDP . . . 57
`UNIFORM` . . . 80
`uniform()` . . . 80
`uniform_empf()` . . . 81
Unterbrechung durch den Benutzer . . . 99
`usage()` . . . 72
`USE_ZUFALL()` . . . 79
User Datagram Protocol . . . 57

V

Variablenindex . . . 122
Verbindungsendpunkt . . . 4
Verkehrsmatrix . . . 11, 81
Verkehrsverteilungen einlesen . . . 95
Verteilungsfunktionen, Einlesen von . . . 95
Verteilungsfunktionen, implementierte . . . 80
`VOID_P` . . . 71
Vorspann der Konfiguration . . . 24

W

Wahrscheinlichkeitsverteilungen . . . 79
Wahrscheinlichkeitsverteilungen, implement. 80
Warteschlangen . . . 37, 78
Warteschlangennetz . . . 9
Wort . . . 24

X

`xxexplainfkt()` . . . 92
`xxgetbits()` . . . 91
`xxgetdouble()` . . . 91
`xxgetdouble_with_unit()` . . . 91
`xxgetfd()` . . . 90
`xxgetfkt()` . . . 92
`xxgetfloat()` . . . 91
`xxgetforvalue()` . . . 91
`xxgetint()` . . . 91
`xxgetmeter()` . . . 91
`xxgettime_t()` . . . 91
`xxgetulong()` . . . 91
`xxgetword()` . . . 90
`xxlinenr` . . . 90
`xxprintword()` . . . 90
`xxungetword()` . . . 90
`xxword` . . . 90
`XXWORDLEN` . . . 90

Z

Zeit, kleinste Einheit . . . 71

Zeit, Weitersetzen 77
Zeiteinheiten 91
`ZUFALL_INFO` 79
Zwischenraum 24

Typenindex

E

EVENT 76

F

FKT_TABLE 92

K

KEYWD_TABLE 93

M

MS_ANSWER 59
MS_REQUEST 59
MSLAVED_INFO 60
MYALLOC_MAGIC 74

N

NODE 82
NQUEUE 78

P

PROBE 87
PTP_FRAME 48

S

SADU 43, 84

T

TIME_T 71
TQUEUE 78
TRAFFIC 81

V

VOID_P 71

Z

ZUFALL_INFO 79

Variablenindex

B

bits_per_second 83

C

cronic_probefkt_tab 96

D

dont_run_it 72

E

event_handler_tab 98

G

glob_params 73

I

itimer_probefkt_tab 96
itu_per_bit 83
itu_per_second 73

K

knoten_anzahl 73

L

l1models_tab 97
l2models_tab 96
l3models_tab 96
l4models_tab 96
l5models_tab 96

M

max_cpu_time 73
max_data_len 83
max_events 73
max_q_length 73
max_simul_time 73
min_data_len 83
minus_x 72
MS_ANSWER_freelist 60

N

net_params 83
nodearray 82, 96

O

object_number 71

P

printconf 72
PTP_FRAME_freelist 48
ptp_info 49
ptp_probept_end_transmit_tab 49
ptp_stoerzeit 48

R

rc_debug_level 72
rcparmtab 96
rcptpnettab 50

S

simul_time 71
stat_not_ready 89
stat_wait_flag 89
statmoduls_tab 96

T

TYP_freelist 75

X

xxlinenr 90
xxword 90
XXWORDLEN 90

Funktionenindex

B

bimodal() 80

C

calc_empfaenger 82
CANCEL_EVENT() 76
CHANGE_EVENT_TIME() 76
CHANGE_EVENT_TIME_AND_HANDLER() . . . 76
control() 77
copy_probepoint() 93

D

DEFUN_GET_REL_OBJECT() 75
DELAY_EVENT_TIME() 76

E

END_OF_SIMULATION() 72, 89
erlang() 80
expntl() 80
EXPNTL() 80

G

get_TYP 75
get_EVENT() 76
get_event_handler_name() 98
get_MS_ANSWER() 60
get_next_event() 77
get_PTP_FRAME() 48
get_SADU() 84

H

hyperexpntl() 80

I

insert_event() 76
INT_EXPNTL() 80
INT_UNIFORM() 80

K

k_cleanup 82
k_init 82
konstant() 80

L

l1_sap 83
listen_empf() 81
lookupnodename() 83

M

make_nqeue() 78
make_tqeue() 79
MEASURE() 87
menu() 99
mslaved_endofservice() 66
mslaved_gwc42() 64
mslaved_pdu_received() 64
mslaved_pdu_sended() 67
mslaved_print_info() 68
mslaved_rc() 62
mslaved_timeout() 66
myalloc() 74
myfree() 74
myrealloc() 74
mystrdup() 75

N

net_cleanup 83
net_init 83
normal() 80
nq_pop() 78
nq_push_fifo() 78
nq_push_lifo() 78

P

print_event() 78, 98
print_ms_answer() 68
print_nqueue() 78
print_sadu() 84
print_tqueue() 79
probe_printf() 86
ptp_cleanup() 52
ptp_data_encapsulation() 52, 54
ptp_init() 52
ptp_mean_throughput() 56
ptp_print_frame() 57
ptp_rc() 50
ptp_rc_throughput() 55
ptp_throughput() 56
ptp_transmit_error() 55

R

RANDOM 79
rc() 89, 96
RC_ERROR() 89
RC_FATAL() 89
rc_get_debug_level() 72, 97
RC_SYNTAX_ERROR() 89
RC_WARNING() 89
rcexplainkeywds() 93
rcfiles() 90
rcnet() 97
rcnodes() 96

rcprintdefaults() 93
rcprobepoint() 92
rcreadkeywds() 93
release_TYP 75
release_EVENT() 76
release_MS_ANSWER() 60
release_PTP_FRAME 48
release_SADU() 84

S

SRANDOM 79

T

tq_pop() 79
tq_push_fifo() 79
tq_push_lifo() 79
traffic_rc() 95

U

UNIFORM 80
uniform() 80
uniform_empf() 81
usage() 72
USE_ZUFALL() 79

X

xxexplainfkt() 92
xxgetbits() 91
xxgetdouble() 91
xxgetdouble_with_unit() 91
xxgetfd() 90
xxgetfkt() 92
xxgetfloat() 91
xxgetforvalue() 91
xxgetint() 91
xxgetmeter() 91
xxgettime_t() 91
xxgetulong() 91
xxgetword() 90
xxprintword() 90
xxungetword() 90