KfK 4727
Mai 1990

# Betriebssystem-integrierte experimentelle Leistungsbewertung verteilter Echtzeitsysteme

P. Kohlhepp
Institut für Datenverarbeitung in der Technik

Kernforschungszentrum Karlsruhe

KERNFORSCHUNGSZENTRUM KARLSRUHE
Institut für Datenverarbeitung in der Technik

KfK 4727

# BETRIEBSSYSTEM - INTEGRIERTE EXPERIMENTELLE LEISTUNGSBEWERTUNG VERTEILTER ECHTZEITSYSTEME

Peter Kohlhepp

von der Fakultät für Informatik der Universität (TH) Karlsruhe
genehmigte Dissertation

KERNFORSCHUNGSZENTRUM KARLSRUHE GMBH, KARLSRUHE

Kernforschungszentrum Karlsruhe GmbH
Postfach 3640, 7500 Karlsruhe 1

ISSN 0303-4003

**Kurzfassung**

In dieser Arbeit wird eine neue Methode und die zugehörige Systemarchitektur zur experimentellen Leistungsbewertung verteilter Echtzeitsysteme entwickelt. Sie beruht auf einer virtuellen Zeitführung, die direkt als Erweiterung der Ablaufsemantik der verteilten Zielmaschinen für die zu entwickelnden Echtzeitsysteme integriert wird, bestehend aus Echtzeit-BS-Kern-Schnittstellen, Zielsprachen, und der unterliegenden Rechner-, E/A-Geräte- und Rechnernetz-Hardware. Solche Echtzeitanwendungen können ganz oder in Teilen als reale, d.h. in ihrem realen Zeitverhalten bewertete Testobjekte mit beliebigen prozeßorientierten Simulationsmodellen kooperieren (Echtteilesimulation in Nicht-Echtzeit-Umgebung). Das Leistungsverhalten eines realen Testobjekts in simulierter Experimentumgebung ist übertragbar auf eine im Kommunikationsverhalten identische Echtzeitumgebung. Einschränkende Annahmen über die konkreten Kommunikations- und Synchronisationsmechanismen, Sprachen, Rechnerbetriebsmittel und Zuteilungsregeln der Zielmaschinen werden dabei nicht benötigt. Die Simulation ist allgemeingültig, d.h. die Modelle umfassen die Zeigler'schen discrete event-Netze als Spezialfall.

Die Hauptanwendung dieses Konzeptes liegt im hierarchischen Entwurf und der schrittweisen Verfeinerung von Prozeßführungssystemen (PFS) und ihrer Leistungsbewertung anhand der globalen Regelgrößen eines simulierten technischen Prozesses, wobei die Implementierung, Wartung und Anpassung im betrieblichen Einsatz nahtlos mit abgedeckt werden. Neue Lösungsansätze eröffnen sich auch für das Problem, verteilte Echtzeitsysteme interferenzarm und interaktiv zu testen.

Das Konzept der Echtteilesimulation in Nicht-Echtzeit-Umgebung wird zuerst systemtheoretisch definiert, dann auf Rechensysteme und deren BS- und Anwender-SW als reale Testobjekte angewandt. Ein Monoprozessor-Echtzeit-BS-Kern wird zu einem hybriden BS-Kern für Echtzeitausführung und prozeßorientierte Simulation erweitert. Schwerpunkte sind die Erweiterung des Prozeßkonzeptes (R-, S- und W-Prozesse), der abstrakte Datentyp der Virtuellen Uhr, die Prozessorzuteilungsregeln, die Organisation der passiven Betriebsmittel und die E/A-Gerätesteuerung. Ferner werden neue Verfahren zur zeitlichen Synchronisation mehrerer solcher über Rechnernetz gekoppelter Monoprozessorsysteme entwickelt, die sowohl zeitdiskrete Simulationsmodelle als auch reale Testobjekte enthalten können. Die Synchronisationsalgorithmen sind transparent für die höheren BS-Schichten und die Anwendungsprogrammierung. Auf die Architektur verteilter Modellierungs- und Zielsysteme und auf spezielle Modellkomponenten wird in Beispielen eingegangen.

# INTEGRATED PERFORMANCE SIMULATION OF DISTRIBUTED EMBEDDED SYSTEMS


## Abstract

This thesis presents a new methodology and system architecture for the experimental performance evaluation of distributed real time applications.
A general set of rules is introduced extending the operational semantics from real time to virtual time for any **given** target machine, consisting of real time nucleus, network, and peripheral interfaces, implementation languages and processor hardware.
Real time applications including their target machines are thus executable simulation models by themselves, obviating the need for separate simulation languages/systems. The basic capabilities of conventional simulation languages -e.g. the simulation of Zeigler's discrete event networks - are preserved. Moreover, individual model components are progressiveley replaced by components operating in real time, such that their functional and timing behaviour in virtual-time environment is always transferrable to the corresponding real-time environment.

The main application is the hierarchical refinement and performance evaluation of process control systems in closed-loop operation with simulated plants. All stages from functional simulation to conformance testing, maintenance and updating on the target system are covered. New techniques for interactive, yet interference-free debugging of distributed hard real time systems are rendered possible.

Real time components in a virtual time environment are first defined system-theoretically. The first step to implement them in distributed computing systems is to extend the local operating systems to become both real time machines and process oriented simulators. The refinement of the process concept (including 'host' processes, 'simulated' and 'real' target processes), the virtual clock object, the dispatching mechanisms, the organization of passive resources, and I/O device control are highlighted.
For connecting several nodes via network, new algorithms for distributed event driven simulation, both tightly and loosely coupled ones, have been developed. They work correctly for arbitrary discrete event networks, are transparent to the simulation programmer, and permit the integration of real time components.
The architecture of distributed simulation and target systems is also discussed, and some particular model components are defined.

**Inhaltsverzeichnis**

Seite

# 1. Einleitung

## 1.1 Charakterisierung von Echtzeitsystemen

Die vorliegende Arbeit stellt einen neuen methodischen Ansatz und eine Systemarchitektur zur experimentellen, entwicklungsbegleitenden Leistungsbewertung verteilter Prozeßführungssysteme (PFSe) vor, also Rechensystemen zum Überwachen, Steuern und Regeln technischer Prozesse (TPe). Das dramatisch verbesserte Preis-Leistungs-Verhältnisses der Mikrorechner-Hardware, und die Fortschritte der Sensor- und Kommunikationstechnik /FAE 84/ haben in den letzten Jahren zu immer anspruchsvolleren Prozeßlenkungsaufgaben geführt, in denen eine Vielzahl von Entscheidungsebenen - von der direkten Steuerungsebene bis zur Betriebsoptimierung, -planung und Prozeßdiagnose - integriert sind und die heute in der Regel dezentral, hierarchisch und/oder redundant aufgebaut sind. Während noch vor 10-15 Jahren Prozeßrechnerprogrammierung durch die Speicher- und Laufzeitoptimierung überschaubarer, prozeßnaher Regelalgorithmen dominiert wurde, haben in zunehmendem Maße Gesichtspunkte wie die Beherrschung der SW-Komplexität, Programmkorrektheit, Ökonomie der SW-Entwicklung, Flexibilität, Portabilität, Ausfalltoleranz etc. an Bedeutung gewonnen. Die Programmiertechniken und SW-Strukturen von PFSen haben sich denen allgemeiner verteilter DV-Systeme (z.B. Informationssysteme, Bürokommunikation) immer mehr angenähert. Die dynamischen und zeitlichen Abläufe in PFSen wurden dadurch schwerer durchschaubar, ohne daß die echtzeitspezifischen Anforderungen durch die HW-Fortschritte entschärft worden sind.

Nach DIN 44330 /DIN 78/ bedeutet Echtzeitbetrieb

"der Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, daß die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können, je nach Anwendungsfall, nach einer zufälligen Verteilung oder zu vorbestimmten Zeitpunkten anfallen."

Die Wirkung eines PFS ist danach nicht durch eine Zuordnung von Eingabedaten zu Resultaten (z: E->R), sondern durch eine Zuordnung zeitlicher Verläufe solcher Größen, das Ein-/Ausgangsverhalten, charakterisiert:

z: (f: T->E) -> (g: T->R)

wobei T entsprechend der ständigen Betriebsbereitschaft ein beliebiges Zeitintervall ist.

Oft übersehen wird allerdings die Tatsache, daß die Verarbeitungs-Zeitspannen meist nicht a priori "vorgegeben" sind, sondern ihre Bedeutung erst durch die übergeordnete Aufgabe erfahren, die Führungsgrößen oder Zustandsgrößen einer technischen Anlage über gewisse Zeiträume innerhalb erlaubter Grenzen zu halten oder zu optimieren. In diese globale Zielfunktion gehen sowohl die funktionellen Eigenschaften des PFS als auch seine zeitliche Dynamik untrennbar ein. Dies unterscheidet PFSe von anderen verteilten DV-Systemen. Dort hat die theoretische Informatik mit Erfolg versucht, die Spezifikation und den Nachweis funktioneller Eigenschaften

unabhängig von Zeit und Geschwindigkeit zu gestalten, und die zeitlichen Eigenschaften (performance) separat zu betrachten und ggf. zu optimieren. Hierzu einige kurze **Beispiele:**

Ein typisches Beispiel zeitunabhängiger Semantik ist die Serialisierbarkeit nebenläufiger Transaktionen in Datenbanken: Der Endzustand, den eine solche Menge von Transaktionen auf der Datenbank hinterläßt, ist derselbe wie bei einer strikt seriellen Abarbeitung, und unabhängig von der relativen Geschwindigkeit und zeitlichen Verzahnung der einzelnen Transaktionen. Eine verteilt ausgeführte zeitdiskrete Simulation sollte dieselben Zeitreihen von Leistungsgrößen errechnen wie eine sequentielle Simulation, unabhängig von der relativen Geschwindigkeit der Ereignissimulation auf verschiedenen Rechnern.

Eine Kommunikations-Transportschicht mit Sende- und Empfangs-Protokollinstanz und einem i.a. unzuverlässigen Übertragungskanal, leitet jede Folge von Nachrichten vollständig, in derselben Reihenfolge (FIFO) und ohne Duplikationen vom Sender zum Empfänger weiter, unabhängig von deren Geschwindigkeiten und der physikalischen Übertragungsdauer.

Die Güte einer digitalen Regelung, die auf einen zeitkontinuierlichen technischen Prozeß ständig einwirkende Störgrößen ausregelt, reagiert dagegen i.a. empfindlich auf Diskrepanzen der tatsächlichen Abtastzeiten der Meßwerte und Totzeiten der Stellwerte gegenüber den im Regelungs-Entwurf angenommenen Zeitkonstanten /SCK 84/,/WEI 85/ (dem Zustand der Datenbank im ersten Beispiel entspricht hier der Zustand der Regelstrecke).

Zwischen einem technischen Prozeß und einem DV-System herrscht eine losere Kommunikation, als zwischen (Rechen-)Prozessen innerhalb eines DV-Systems. Anstelle der **kausalen** Sicherheit ("Ereignis b kann erst stattfinden, wenn Ereignis a stattgefunden hat") tritt die **temporale**, nur unter gewissen zeitlichen Annahmen gültige Sicherheit /SMI 87/. Blockieren eines Prozesses oder Restaurieren eines früheren Prozeßzustands, welche z.B. die Serialisierbarkeit in verteilten Systemen erst möglich machen, sind auf technische Prozesse i.a. nicht anwendbar. Das PFS muß vielmehr mit dem technischen Prozeß Schritt halten. Um Schritt halten zu können, wurden "losere" Formen der Kooperation zwischen den Subsystemen innerhalb des PFS entwickelt, mit dem Ergebnis, daß die Zeitabhängigkeit nicht nur an der Schnittstelle zum technischen Prozeß besteht, sondern in das PFS selbst hineingetragen wird.

**Beispiele**

Eine "Zustands-Mailbox" /KLM 82/ ist ein Kommunikationskanal, von dem immer die zuletzt gesendete ("aktuellste") Information empfangen wird. Empfangen (Lesen) und Senden (Überschreiben) sind nichtblockierend. Die Verar-beitung veralteter Information wird vermieden, anders als beim FIFO-Übertragungskanal. Die Folge der gelesenen Werte ist nur mehr eine Teilfolge (mit Wiederholungen) der senderseitig übergebenen Wertefolge; **welche** Teilfolge, ist ohne Kenntnis der relativen Geschwindigkeiten der Sender- und Empfängerprozesse nicht feststellbar.

Oft wird die Gültigkeit von Prozeßinformationen auch **explizit** durch eine Zeitdauer begrenzt, und die Information danach vernichtet. Hiervon können funktionelle Entscheidungen, z.B. Stellaktionen, abhängen. In /MLE 84/ wird z.B. mit Hilfe der Gültigkeitszeit die Konsistenz (redundanter) Eingabedaten in Realzeit-Datenbasen entschieden (wo z.B. eine binäre Schalterstellung von mehreren Prozessen unabhängig abgetastet wird und einige Prozesse die letzte Änderung gerade noch mitbekommen, andere aber nicht).

/MAA 84/ und /KOM 85/ behandeln die Kommunikation redundanter, diversitärer Komponenten mit Mehrheitsvotierung, welche periodisch nach jedem Rechenschritt, aber wegen der diversitären Realisierung nicht zeitsynchron, ihre Zwischenergebnisse austauschen. Die Ankunft des ersten Ergebnisses eröffnet bei einer Komponente eine Datenaustauschphase, deren Dauer

durch eine Zeitüberwachung T begrenzt wird. Sind mit dem Ablauf von T die Ergebnisse einer Mehrheit der Komponenten eingetroffen, wird angenommen, daß die übrigen Komponenten fehlerhaft arbeiten. Liegen die Ergebnisse nur von einer Minderheit vor, wird davon ausgegangen, daß eine fehlerhafte Komponente die Austauschphase verfrüht eröffnet hat. Wiederum sind die funktionellen Eigenschaften (hier die Fähigkeit, Fehler zu diagnostizieren und zu tolerieren) ohne Kenntnis der Zeit T in Relation zu den zulässigen Laufzeitdiskrepanzen der Komponenten und der Nachrichtenübertragung nicht zu beurteilen.

### 1.2 Aufgabenstellung für die Leistungsbewertung

Die wechselseitige Abhängigkeit funktioneller und zeitlicher Eigenschaften von PFSen hat einschneidende Konsequenzen für deren hierarchischen Entwurf. Die Korrektheit einer Systemimplementierung relativ zu einem Entwurf muß durch den Vergleich des Ein-/Ausgabeverhaltens beider Systeme beurteilt werden, also Vergleich von Funktionen mit zeitlichen Verläufen als Argumenten, statt Eingabedaten. Das zeitliche Verhalten der Implementierung eines untergeordneten Subsystems kann auf die funktionellen Eigenschaften des Gesamtsystems zurückwirken. Ein umfassender Ansatz, der das Verifikations- bzw. Syntheseproblem unter dieser Fragestellung analytisch zu lösen gestattet (d.h. die "Korrektheit" einer Implementierung zu beweisen, bzw. korrekte Implementierungen herzuleiten gestattet), ist noch nicht in Sicht; er müßte Systemtheorie bzw. Regelungstheorie, Programmverifikationstechniken für verteilte Systeme, Graphentheorie und Warteschlangentheorie integrieren.

Daher müssen in der Praxis entwicklungsbegleitend **experimentelle** Hilfsmittel zur Leistungsbewertung eingesetzt werden, auf die sich diese Arbeit konzentriert. Mehrere **Entwurfsvarianten** des zu entwickelnden Systems werden unter vorgegebenen Anfangswerten und Eingangsverläufen simuliert oder als reales Testobjekt ausgeführt und anhand der interessierenden **Leistungsgroeßen** beurteilt.
Unsere wichtigsten Leistungsgrößen sind nach 1.1 die Führungsgrößen des technischen Prozesses (TP). Ihre rechnerische Bestimmung setzt voraus, daß seine Zustands-, Führungs-, Meß-, Stell- und Störgrößen durch dynamische Modelle im Zeitbereich (DGL-Modelle oder diskrete Zustandsmodelle) mathematisch beschreibbar sind. Der Nutzen experimenteller Leistungsbewertung hängt letztlich von der Qualität und Aussagekraft der ermittelten Leistungsdaten ab. Hierbei ist der Abbildungsprozeß zwischen einem realem System bzw. gedanklichem Entwurfskonzept, einem abstrakten (Simulations)Modell und einem ablauffähigem Experiment zu beachten, der zu Informationsverlusten, und damit zu Diskrepanzen zwischen den realen bzw. zu erwartenden und den experimentell gewonnenen Leistungsgrößen führen kann (Abb. 1.1).

(1) **Modellierungsfehler**
z.B. Modellierung struktureller Systemeigenschaften zu wenig detailliert; schwer schätzbare Systemparameter; Fehler in der Festlegung der Experimentbedingungen (Anfangswerte, Ein-/Ausgangs- und Zustandsgrößen), unzulängliche Wahl der Leistungsgrößen oder ihrer Aufbereitung; fehlerhafter Informationstransfer zwischen Systementwickler und Modellierer; Inkonsistenz zwischen gedanklichem Konzept und der Semantik einer Modellierungssprache

(2) **Verfahrensfehler**
z.B. einschränkende Bedingungen des Experimentiersystems an das zu bewertende Testobjekt/Modell, deren der Anwender sich nicht bewußt ist; Einschränkungen der meßbaren Leistungsgrößen; numerische Verfahrensfehler; zeitliche Diskretisierungsfehler; Verfälschung der Meßergebnisse des Testobjektes durch das Experimentiersystem (Interferenz).

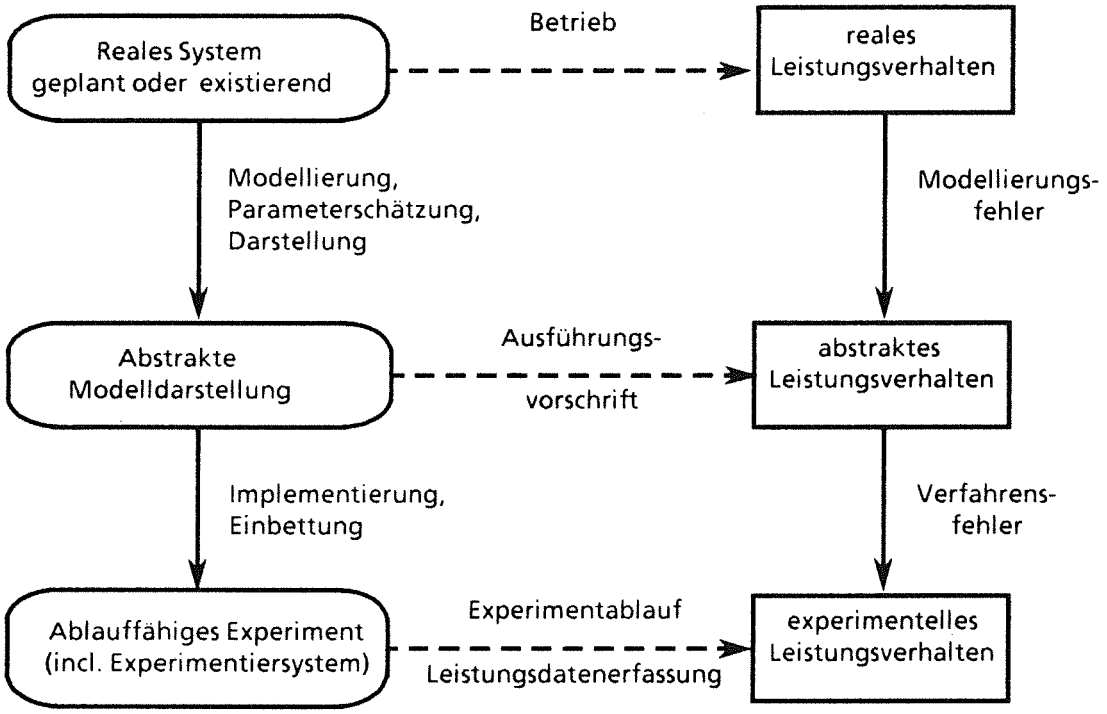



**Abb. 1.1:** Abbildungsschritte und Informationsverluste bei der experimentellen Leistungsbewertung

Folgende Anforderungen sind an leistungsfähige Hilfsmittel zur experimentellen Leistungsbewertung von PFSen zu stellen:

(a) Frühzeitig einsetzende Leistungsvorhersage der Führungskonzepte unter Einbeziehung des technischen Prozesses, aber zunächst ohne Berücksichtigung ihrer DV-technischen Realisierung

(b) Schritthalten mit allen folgenden Entwicklungsphasen des PFS, einschließlich Implementierung, Test und Anpassung im Betrieb; dabei Bewertung anhand einheitlicher und anwendungsorientierter Leistungsgrößen ("nahtlose Übergänge")

(c) Einheit des PFS als reales Entwicklungsobjekt und als Modell/Testobjekt für die Leistungsbewertung in allen Phasen

(d) Effizienz der Leistungsbewertung zur Beurteilung von Alternativen im Entwurfsstadium (a)

(e) Geringe Fehler bei der Leistungsbewertung im Implementierungs- und Teststadium (b)

(f) Unabhängigkeit der Verfahren zur Leistungsbewertung von Eigenschaften der zu entwickelnden Anwendung (Anwendungsneutralität)

(g) gute Unterstützung des Anwendungs-Entwicklers bei der Experimentdefinition, -steuerung, -auswertung und -diagnose.

Da ein großer Teil der Fehler eines SW-Produktes i.d.R. auf die Anforderungsspezifikation oder den Entwurf zurückgeht, kann ein frühzeitig ausführbarer und bewertbarer PFS-Entwurf (a) wirkungsvoll zur Entdeckung solcher Fehler beitragen, vorausgesetzt der Entwicklungs- und Experimentieraufwand liegt deutlich unter dem einer vollen Implementierung (d). Dennoch kommt es letztlich auf das absolute Leistungsverhalten des realisierten PFS am konkreten TP an, und nicht nur auf einen qualitativen Vergleich mehrerer Entwurfsalternativen. Ein Großteil der gesamten Entwicklungskosten entfällt üblicherweise auf den Test oder die Wartung und Anpassung existierender SW-Systeme. Daraus folgt die Bedeutung von (b) und (e). Nach Punkt (c) sollen Modellierungsfehler nur auf der Seite des TP, aber nicht des PFS auftreten können. Die Leistungsbewertung soll mit allen Entwurfs- und Implementierungsentscheidungen des PFS und ihrer zunehmend detaillierteren leistungsrelevanten Information schritthalten, d.h. weder vorauseilen noch nachhinken. Zugleich wird so der Modellierungsaufwand für das PFS minimiert. Die Anwendungsneutralität (f) gewährleistet zusammen mit (c), daß die Leistungsbewertung durch den Anwendungsentwickler, ohne Mithilfe von Modellierungsspezialisten und ohne sich einschränkender Bedingungen der Testumgebung bewußt sein zu müssen, benutzt werden kann.

### 1.3 Ziele, Hauptideen und Gliederung der Arbeit

Als grundlegend für die vorliegende Arbeit lassen sich folgende, noch näher zu begründende Thesen formulieren.

(T1)
Die Bewertung verteilter DV-Systeme (PFS) im Entwurfsstadium, und in jedem Fall die Nachbildung ihrer Umgebung (TP), erfordern eine **virtuelle** Zeitführung, die von den Modellen selbst kontrolliert wird und unabhängig von der externen Echtzeit beim Experimentablauf ist. Rein echtzeitorientierte Verfahren wie Meßmonitore oder Echtzeitsimulationen sind für diesen Zweck ungeeignet.

(T2)
Traditionell wird eine virtuelle Zeitführung immer als Teil einer Entwurfs- oder Simulationssprache und ihrer zugehörigen Entwicklungs- und Laufzeitumgebung (**Simulationsmaschine**) angeboten. Der Modellierer, der diese Leistung in Anspruch nehmen will, übernimmt damit zugleich alle anderen Sprachkonstrukte, syntaktischen und semantischen Eigenschaften als Paketlösung, sobald er sich für eine bestimmte Sprache entschieden hat.

(T3)

Mag diese "Abgeschlossenheit in einer Sprache" den Vorteil haben, unterschiedliche Aspekte und Entwicklungsstadien eines Zielsystems in nur wenigen Sprachelementen einheitlich darstellen zu können, so stellt sie andererseits **das** zentrale Hindernis dar, um die Einheit der Darstellung von Zielsystem und Modell konsequent durchzuhalten (1.2 (c)). Die Verschiedenheit der Simulationsmaschine von der verteilten **Zielmaschine** mit ihren i.a. heterogenen Sprachen, Betriebssystem-Diensten und HW-Betriebsmitteln, auf der schließlich das PFS ablaufen wird, führt zu einer Kluft zwischen der eigentlichen Systementwicklung und der Systemmodellierung/Leistungsbewertung.

Diese Kluft wird durch folgende Hauptideen geschlossen:

(T4)

Die Dienstleistungen der möglichen Zielmaschinen werden als **Teilmenge** der Modellierungssprachen zur Verfügung gestellt - können also direkt benutzt werden - indem die virtuelle Zeitführung generell als **funktionelle Erweiterung** dieser **Zielmaschinen** eingeführt wird.

(T5)

Vollständig implementierte Subsysteme des PFS können auf ihrer Zielmaschine als meßbare Komponenten, sog. **reale Testobjekte**, entsprechend ihrem realen Betriebsmittel- und Zeitverbrauch bewertet werden und dabei im Experiment mit einer simulierten, d.h. virtuellen Zeitführung unterliegenden Umgebung kooperieren. Als reales Testobjekt wird die leistungsrelevante Information eines PFS optimal ausgenutzt.

Unter Zielmaschine als tiefster Abstraktionsebene des Entwicklungsobjektes wird die Ebene der Echtzeit-BS-Kerne verstanden, die - neben den sequentiellen Ziel-Programmiersprachen - die Schnittstellen zum Zugriff auf die Rechner-Betriebsmittel (Prozessoren, Arbeitsspeicher, physikalische E/A-Geräte, Kommunikationsmedien etc) umfasse und kooperierende Prozesse auf verschiedene Arten unterstütze. Das Zeitkonzept (Echtzeit oder virtuelle Zeit oder eine Mischform) wird damit als ein Attribut des Ablaufmodells kooperierender Prozesse definiert, weitgehend orthogonal zur funktionellen Architektur der Anwendung und den sie tragenden Sprachen. Entscheidend für den Erfolg dieses **zielorientierten** oder **integrierten** Simulationsansatzes (T4),(T5) ist letztlich, ob es gelingt, die folgenden zusätzlichen Anforderungen zu erfüllen:

(T6) **Interferenzfreiheit**

Ein reales Testobjekt kooperiert im Experiment über eine gemeinsame Schnittstelle mit einer simulierten Umgebung, die eine reale, in Echtzeit operierende Umgebung des späteren Einsatzes darstellt. Die am realen Testobjekt beobachtbaren Leistungsgrößen in simulierter Umgebung müssen übertragbar sein auf dessen Einsatz in der zugehörigen, schnittstellengleichen Echtzeitumgebung. Sie dürfen nicht davon abhängen, **wie** die Simulationsmodelle implementiert sind.

Diese Eigenschaft heißt Interferenzfreiheit. Nur unter dieser Prämisse ist die Integration realer Testobjekte sinnvoll.

(T7) **Erweiterungseigenschaft**

Flexible Simulationsumgebungen können im Prinzip "beliebige" Entwurfsalternativen von Systemen **modellieren**. Neu ist nun, daß zusätzlich **reale** Zielsysteme entwickelt und bewertet werden. Konsequenterweise sollten dem Entwickler die Freiheitsgrade, um ein gutes (effizientes, fehlertolerantes etc.) Zielsystem unter gewissen Randbedingungen zu realisieren, in der Experimentierumgebung erhalten bleiben. Z.B. könnte der Entwickler die Zielmaschine wechseln wollen. Die virtuelle Zeitführung (T4) sollte also als Erweiterungsvorschrift **gegebener** Dienste der Zielmaschinen verstanden werden, und (T4),(T5),(T6) möglichst unabhängig von speziellen Voraussetzungen über eine konkrete Zielmaschine erfüllt werden.

(T8) **Allgemeingültige Simulation**

Existierende Simulationssprachen stellen Schnittstellendienste zur Verfügung, mit denen (in einem noch zu präzisierenden Sinn) "beliebige" zeitdiskrete Modelle spezifiziert und korrekt simuliert werden können, und verbergen dabei dem Modellierer die unterliegende Ablaufmaschine, z.B. Großrechner-BS. Damit werden Anwendungsunabhängigkeit und Allgemeingültigkeit ((f) in 1.2) der Simulationsverfahren und eine gewisse Modellportabilität erreicht.

Wenn nun Simulationsfunktionen allein auf die Dienste bestimmter Echtzeit-BS-Kerne zugeschnitten sind, ist zu erwarten, daß deren Verschiedenartigkeit voll auf die Modellierung durchschlägt. Komplementär zu (T7) sollten daher die Dienste einheitlich so erweitert werden, daß auf allen Zielmaschinen eine allgemeingültige Simulation als Mindeststandard erreicht wird. Damit bleiben die Fähigkeiten konventioneller Simulationsmaschinen im integrierten Simulationsansatz erhalten. Sonstige, für das Simulationskonzept nicht entscheidende funktionelle Unterschiede der Zielmaschinen (z.B heterogene Filestruktur) lassen sich auf höheren BS-Schichten ausgleichen.

Als wesentliche **Einschränkungen** bzw. Abgrenzungen der Arbeit sind zu nennen:

- Die Zielsysteme sind Echtzeit**anwendungen**, d.h. die unterste mögliche Abstraktionsebene für Entwurf/Implementierung sei die BS-Kern-Schnittstelle. Falls der BS-Kern oder die Prozeßrechner-HW selbst zu entwickeln und simulativ zu bewerten sind, müßte die virtuelle Zeitführung auf einer entsprechend tiefer liegenden Basismaschine aufsetzen, um die Vorteile des neuen Ansatzes weiter nutzen zu können. Die vorgeschlagene Realisierung orientiert sich an einer konventionellen Mehrrechnerarchitektur als Zielmaschine: mehrere Mono-(evtl. auch Multi-)prozessorsysteme, jeder mit einem multitasking-Echtzeit-BS-Kern ausgestattet, sind über ein Rechnernetz lose gekoppelt. Rechnerarchitekturen, die bereits auf der BS-Kern- oder HW-Ebene grundlegend andere Organisationsprinzipien zur Unterstützung kooperierender Prozesse anwenden, wie z.B. Datenflußrechner /RED 84/, OCCAM Transputer-

Netze /MAS 85/ oder verteilte LISP- oder PROLOG-Maschinen im Bereich der Künstlichen Intelligenz, würden technisch andere Lösungen erfordern, wollte man den integrierten Simulationsansatz auf sie übertragen.

- Als reale Testobjekte kommen in der Praxis ausschließlich **Rechensysteme** in Frage, jedoch nicht der zu steuernde technische Prozeß. Letzterer ist stets durch Simulationsmodelle vertreten. Die korrekte Integration technischer Anlagen als reale Testobjekte in eine Nicht-Echtzeit-Umgebung ist nur unter sehr einschränkenden Bedingungen möglich.
  Da der Test unvollständiger und fehlerhafter Steuerungs-SW an einem realen technischen Prozeß ein hohes Risiko und i.a. geringe Flexibilität des Experimentierens beinhaltet, wird der praktische Nutzen der Methode durch diesen Verzicht nur wenig beeinträchtigt.

Die Arbeit ist wie folgt gegliedert.
Kapitel 2 untersucht Methoden zur Leistungsbewertung auf ihre Eignung für Echtzeitsysteme und dient zur Erhärtung einiger der o.g. Thesen, z.B. was die echtzeitorientierten Methoden (T1) betrifft. Vor allem werden die Unterschiede zwischen dem konventionellen (T2),(T3) und dem "zielorientierten" Simulationsansatz (T4) schärfer herausgearbeitet.
In Kapitel 3 wird nach einer Charakterisierung der Zielsysteme (PFSe, technische Prozesse, Leistungsgrößen) und ihrer Entwicklungsstadien und Experimente das Simulationskonzept (T4)-(T8) formal definiert. Daraus wird die Grobarchitektur eines verteilten zielorientierten Simulations-Testbetts und die Anforderungen an seine wichtigsten Subsysteme abgeleitet. Neben der Simulation werden weitere Anwendungen dieser Architektur aufgezeigt.
Der Durchführungsteil (Kap. 4 und 5) befaßt sich mit dem Aufbau einer verteilten Laufzeitumgebung zur integrierten Simulation und Echtzeitausführung. In Kapitel 4 wird ein detailliertes Konzept zur funktionellen Erweiterung eines Monoprozessor-Echtzeit-BS-Kerns zu einem BS-Kern mit virtueller Zeit beschrieben. Herausgestellt wird insbesondere die Kongruenz von BS-Architekturen zur Echtzeitsteuerung und zur prozeßorientierten Simulation, die entscheidend für die Ökonomie des Erweiterungskonzeptes ist.
In Kap. 5 werden n über ein Rechnernetz gekoppelte Monoprozessorsysteme so synchronisiert, daß eine korrekte, allgemeingültige verteilte Simulation erreicht wird. Aufbauend auf den Erkenntnissen und Verfahren zur verteilten discrete-event-Simulation werden neue Synchronisationsverfahren entwickelt, die speziell auf die Integration realer Testobjekte abgestimmt sind.
In Kapitel 6 wird an Beispielen auf die Architektur verteilter Modellierungs- und Zielsysteme eingegangen, die auf dem Systemkern von Kap. 4 und 5 aufsetzen. Ferner werden Komponenten zur Modellierung kontinuierlich-diskreter technischer Prozesse und virtuelle Mehrrechnerumgebungen vorgestellt.
Kap. 7 faßt die Ergebnisse zusammen und zeigt Ansatzpunkte für weitere Forschung und Entwicklung auf dem Gebiet der integrierten Simulation auf.

## 2. Methoden zur Leistungsbewertung von Echtzeitsystemen

Methoden zur Leistungsbewertung von Rechensystemen oder technischen Systemen dienen dazu, interessierende Leistungskenngrößen eines zu entwickelnden oder existierenden Systems in Abhängigkeit von Entwurfsvariablen oder sonstigen charakteristischen Einflußgrößen zu bestimmen. Eine erste Einteilung der für Echtzeitsysteme relevanten Methoden führt auf **analytische Methoden**, bei denen ein direkter, durch eine mathematische Formel beschriebener Zusammenhang zwischen Einflußgrößen und Leistungsgrößen besteht, und **experimentelle Methoden**, bei denen die Leistungsgrößen für jede feste Wertekombination der Einflußgrößen durch Beobachtung des Systems über einem Zeitraum gewonnen werden. Wenn die Leistungsgrößen sich analytisch - exakt oder numerisch - berechnen lassen, ist eine solche Lösung i.a. den experimentellen Verfahren an Effizienz überlegen. Daher gehen wir zuerst auf die analytischen Verfahren ein.

### 2.1 Analytische Methoden

Die neben den Führungsgrößen wichtigsten Leistungsgrößen eines PFS sind die Antwortzeiten, d.h. die Zeitspannen zwischen dem Auftreten bzw. Erfassen von Prozeßsignalen und der Ausgabe daraus abgeleiteter Stellsignale (vgl. 3.1.3.2). Die durch das System fließenden Prozeßinformationen können als Aufträge gesehen werden, welche um die zu ihrer Bearbeitung benötigten Betriebsmittel (CPU, ASP, A/D-und D/A-Wandler, Kommunikationskanäle etc.) an **Bedienstationen** konkurrieren. Prinzipiell ist ein PFS also, wie jedes andere DV-System, als Warteschlangennetz modellierbar und analysierbar /KLE 75/,/WAL 78/. Dazu müssen folgende Einflußgrößen bekannt sein

- Bedienzeit-Verteilungsfunktion der Aufträge an den Stationen; bei offenen WS-Netzen zusätzlich: Ankunftszeit-Verteilung externer Aufträge
- Bedienstrategien (z.B. FIFO, statische Priorität, shortest-job-next, präemptiv/nichtpräemptiv)
- Beschreibung des Auftragsflusses: Übergangs-Wahrscheinlichkeit eines Auftrags von einer Bedienstation i zu einer Nachfolgestation j.

Der zeitliche Verlauf der Anzahl von Aufträgen an den Bedienstationen bildet einen homogenen, diskreten Markov-Prozeß. Aus den Zustandsübergangsraten lassen sich die stationären Zustandswahrscheinlichkeiten berechnen, und hieraus

- die Wartezeiten bzw. Systemzeiten der Aufträge an einzelnen Stationen oder im Gesamtsystem (insbesondere die o.g. Antwortzeiten)
- die Auslastungen der Stationen
- den Systemdurchsatz an Aufträgen.

Effiziente rekursive und iterative Berechnungsschemata existieren u.a. für die sog. BCMP-Netzklasse /BCM 75/ /HLA 84/, bei denen sich die Zustands-WS des Gesamtsystems als

Produktform der Zustands-WSen einzelner Stationen ergibt. Wesentliche Annahmen und Einschränkungen dieser Netzklasse sind jedoch für Echtzeitsysteme nicht erfüllt:

**(A) Einflußgrößen, Modellspezifikation**

- Als Bedienstrategien sind nur FIFO, LIFO, oder processor sharing zugelassen und für FIFO-Stationen sind exponentialverteilte Ankunfts-/Bedienzeitverteilungen vorgeschrieben.

  In PFSen gilt typischerweise präemptive Bedienung nach statischen/dynamischen Prioritäten, mit beliebigen, i.a. vom Arbeitszustand des technischen Prozesses abhängigen Ankunfts-/Bedienzeitverteilungen. Die Annahmen über diese Verteilungen sind außerdem für ein existierendes System schwer zu validieren /HLA 84/; und bei einem erst zu entwickelnden System ist es ziemlich willkürlich, solche Vorgaben zu machen, da sie aus Sicht des PFS-Entwicklers keine problembezogenen **Entwurfsvariablen** darstellen.
- Aufträge belegen Bedienstationen nacheinander, und nicht gleichzeitig.

  In Wirklichkeit ist die Simultanbelegung von Betriebsmitteln durch Aufträge keine Seltenheit (z.B. CPU und Arbeitsspeicher, E/A-Gerät und DMA-Kanal).
- Der Auftragsfluß ist durch eine konstante Auftragspopulation mit stationären Verzweigungswahrscheinlichkeiten gegeben.

  In Wirklichkeit werden Aufträge dynamisch erzeugt, vernichtet, in Unteraufträge aufgespalten und wieder zusammengeführt. In dem konkreten algorithmischen Entwurf des PFS ist sehr viel mehr Information über die Kontroll- und Datenflüsse enthalten, als von den statistischen Verfahren ausgenutzt werden kann.
- Jeder wartende Auftrag wird auch bedient.

  Bereits aus den Beispielen in Kap.1 (Zustands-Mailbox, Gültigkeitsintervall) geht hervor, daß Echtzeitsysteme sich Aufträgen und Daten entledigen, bevor ein Rückstau veralteter Information entsteht.

**(B) Leistungsgrößen**

- Sobald mehr als eine einzige Bedienstation vom Typ (M|M|K) /KLE 75/ vorhanden ist, liefert die Analyse i.d.R. nur **Mittelwerte** der Antwortzeiten. Gefragt ist in Echtzeitsystemen dagegen immer die vollständige **Verteilungsfunktion** ("mit welcher Häufigkeit treten Antwortzeiten > a auf?").
- Vom einfachsten Fall eines reinen Geburten-/Sterbeprozesses abgesehen sind nur stationäre Aussagen möglich; in Echtzeitsystemen sind aber auch Einschwingvorgänge wichtig (z.B. bei Störungen im technischen Prozeß oder Komponentenausfällen im PFS).
- Nur systemnahe Leistungsgrößen, z.B. Auslastungen der HW-BM, lassen sich auf einfache Weise bestimmen; erwünscht sind oft auch anwendungsnähere Leistungsaussagen (vgl. 3.1.3.3).

Viele dieser Einschränkungen spielen nicht erst bei einer **sehr** detaillierten Modellbildung (Implementierungs- und Testphase) eine Rolle. Die Verteilungsfunktion der Antwortzeiten, also worst-case-Verhalten, ist bereits für die Kapazitätsplanung wichtig.
Für Netze, die nicht alle Einschränkungen erfüllen, existieren zum Teil Näherungsverfahren, deren wichtigste auf dem Dekompositionsprinzip beruhen. Sie zielen vor allem auf Anwendungen mit simultaner BM-Benutzung bzw. Betriebsmittelhierarchien. Z.B. für jeden konstanten "multiprogramming"-Grad, der mit dem verfügbaren Arbeitsspeicher realisiert werden kann, ergibt die WS-Analyse des Bediensystems für die CPU eine mittlere Durchsatzrate, die als zustandsabhängige Bedienrate zur Lösung in das übergeordnete Arbeitsspeicher-Warteschlangenmodell eingesetzt wird /HLA 84/. Auch iterative Verfahren auf der Basis dieses Ansatzes wurden veröffentlicht /JAL 82//FRN 82/. Auch die Näherungsverfahren liefern nur Mittelwerte. Ferner können i.a. keine genauen Schranken für ihre Verfahrensfehler gegenüber der exakten Markov-Analyse angegeben werden. Zur Kontrolle muß also immer auch eine Simulation unter denselben Modellannahmen und demselben Detaillierungsgrad durchgeführt werden. Der Effizienzvorteil der analytischen Methoden geht dann aber verloren.

Ein Hauptproblem der Warteschlangenanalyse ist ihre starke Abhängigkeit von speziellen mathematischen/statistischen Eigenschaften der Modelle. D.h. bei Verfeinerungen der Modellstruktur oder Änderungen ihrer Parameter - und seien diese aus Anwendersicht noch so unscheinbar - müssen oft grundsätzlich andere Lösungsverfahren ausgewählt/entworfen werden. Dieser, der eigentlichen Entwicklungsaufgabe nicht direkt zugutekommende Aufwand wiegt sicherlich schwerer als der Rechenzeitaufwand zur Experimentauswertung.

Zu den analytischen Methoden zur Leistungsbewertung von Echtzeitsystemen zählen neben den Markov-Modellen auch graphentheoretische Ansätze /COR 85//MAH 84//SAJ 87//HAR 87//ZAS 86/. Aus der Darstellung der Datenflüsse eines Rechensystems - als zeitattributierter Präzedenzgraph oder Petrinetz - werden obere Schranken (statt Mittelwerten) für die Programmdurchlaufzeiten längs der kritischen Pfade abgeleitet. Obwohl diese Methoden dem Echtzeitcharakter eher gerecht zu werden scheinen als Warteschlangennetze, erweisen sich ihre Modellannahmen als eher noch restriktiver und unrealistischer. Typischerweise müssen alle möglichen Zugriffskonflikte auf gemeinsame Betriebsmittel, die Anzahl der Prozesse, oft sogar die Anzahl von Schleifendurchläufen in Algorithmen statisch bekannt sein. Die Komplexität der Verfahren wächst typischerweise exponentiell mit dem Umfang der Anwendung (z.B. der Zahl der Datenobjekte und Instanzen).
Warteschlangentheorie wurde in /CLE 84//THB 83/ kombiniert mit Präzedenzrelationen; die typischen Einschränkungen (z.B. nur Mittelwertanalyse) sind die gleichen wie bei der reinen Warteschlangenanalyse. Mit der Analyse von Antwortzeiten in Echtzeitsystemen, sowie dem verwandten Syntheseproblem, gegebene Antwortzeit-Restriktionen top-down zu verfeinern /COR 85/, befassen sich eine Reihe von Arbeiten. Überraschend ist, wie wenige sich dagegen mit dem wichtigeren, aber analytisch noch schwierigeren Problem befassen, wie sich die Antwortzeiten auf

die Regelgrößen eines technischen Prozesses auswirken, oder wie sie aus den Regelgrößen abgeleitet werden können. /KRS 83/ definiert eine Kostenfunktion eines Reglers in Abhängigkeit von dessen Totzeiten, die sich aus einem integralen Gütefunktional der Regelgrößen des Gesamtsystems herleitet. Bemerkenswert ist, daß die gesamte Verteilungsfunktion der Totzeit, und nicht nur deren Mittelwerte oder Maxima, eine Rolle spielt.
Schnieder und Krafft /SCK 84/ versuchen, den Einfluß variabler Totzeiten eines einschleifigen Regelkreises auf dessen Regelgrößen zu quantifizieren. Aus dem Stellsignal eines "perfekten" und eines funktionell identischen, aber in seinen Abtast- und Totzeiten schwankenden Reglers wird ein Differenz-Stellsignal gebildet, das in die Dynamik-Gleichung des Gesamtsystems eingesetzt werden kann. Um die Methode anwenden zu können, müssen die Schwankungen der Abtast- und Totzeiten statistisch unabhängig vom Betriebszustand des technischen Prozesses sein, eine wegen der Datenabhängigkeit der Verarbeitungszeiten problematische Annahme.

Festzuhalten bleibt: trotz großer Fortschritte der analytischen Methoden stellt ihre starke Anwendungsabhängigkeit noch immer ein Hindernis dar, sie auf PFSe von realistischer Größe und Komplexität einsetzen zu können, ohne Expertenwissen vom Entwickler verlangen zu müssen.

## 2.2 Experimentelle Methoden

### 2.2.1 Systemtheoretische Begriffsbildung

#### 2.2.1.1 Systemspezifikation

Im folgenden werden einige Grundbegriffe zur experimentellen Beobachtung und Simulation von Systemen anhand des von B.P. Zeigler entwickelten Formalismus /ZEI 76//ZEI 84a/ eingeführt. Mit seiner Hilfe läßt sich präzisieren, wann eine Simulation "allgemeingültig" ist (vgl. (T8) in 1.3), oder bestimmte Arten zeitlichen Verhaltens charakterisieren.
Dieser Formalismus stellt zugleich ein Rahmenwerk zur Modellbeschreibung bereit. Diese systemtheoretische Darstellung ist jedoch nicht zu verwechseln mit den konkreten - i.a. für jede Anwendung verschiedenen - Modellierungs- und Programmiersprachen des Anwendungsentwicklers.

Den Kern des Zeigler'schen Formalismus bildet eine Hierarchie von Systemspezifikationen, welche mit dem sichtbaren Ein-/Ausgangsverhalten von Systemen beginnt und zu einer immer detaillierteren Beschreibung der internen Systemstruktur fortschreitet. Beziehungen zwischen Systemen gleicher Beschreibungsebene, z.B. Simulation eines Systems durch ein anderes, werden durch Abbildungen (Morphismen) ausgedrückt. Schließlich werden Bedingungen angegeben, unter denen Systembeschreibungen und Morphismen einer Ebene auf entsprechende Objekte und Morphismen der nächsttieferen bzw. höheren Ebene transformiert werden können. I.f. werden nur einige der für die Arbeit wichtigsten Definitionen wiedergegeben.

Grundlegend für den Experimentbegriff ist die Verarbeitung von Eingabeinformation aus einem Wertebereich X zur Ausgabeinformation aus einem Wertebereich Y, beide über demselben Zeitintervall eines Zeitbereiches T betrachtet. Durch die Wahl der Wertebereiche X und Y wird zunächst der Zweck des Experimentierens festgelegt ('experimental frame' /ZEI 76/, Kap. 11); dabei wird die Menge der möglichen Eingabezeitfunktionen

$(X,T) := \{\omega \mid \omega : [t0,t1) \rightarrow X, t0,t1 \in T\}$,

auf eine Teilmenge $\Omega \subseteq (X,T)$ von zulässigen, interessierenden Eingabehistorien eingeschränkt. Ein **Experiment** durchführen heißt, ein System in einen definierten Startzustand zu bringen, mit einem $\omega \in \Omega$ zu beaufschlagen, seine Ausgabehistorie y über demselben Zeitintervall zu beobachten und ggf. weiter auszuwerten. Ein solches experimentell beobachtbares Systemverhalten wird beschrieben durch eine Ein-/Ausgabe-Funktionszuordnung /ZEI 76/.

**Def. 2.1 (I/O-Funktionszuordnung, IOFO)**

Eine I/O-Funktionszuordnung (IOFO) ist ein Tupel $IOFO := (T,X,\Omega,Y,F)$ mit

- T: Zeitbereich ($T = \mathbf{R}^+ \cup \{\omega\}$ oder $T = \mathbf{N} \cup \{\infty\}$)
- X: Eingabemenge
- $\Omega$C (X,T) Menge von Eingabehistorien(-segmenten)
- Y: Ausgabemenge
- F: $\{f \mid f:\Omega \rightarrow (Y,T)$ und $dom(\omega) = dom(f(\omega) \forall \omega \in \Omega\}$ Menge der Ein-/Ausgabefunktionen der IOFO (dom ($\omega$) Definitions-Zeitintervall von $\omega$)

Die Menge F spielt die Rolle von "Startzuständen" oder "Voreinstellungen". Zu gegebenem $f \in F$ wird aus jeder Eingabehistorie $\omega$ eine eindeutige Ausgabefunktion $f(\omega)$ produziert.

Der erste Schritt zu einer strukturierten Systembeschreibung liegt in der Einführung eines internen Systemzustands.

**Def. 2.2 (I/O-System)**

Ein I/O-System ist ein Tupel $IOS := (T,X,\Omega,S,Y,\delta,\lambda)$, mit
T,X,$\Omega$,Y wie in Def. 2.1,

- S: Zustandsraum
- $\delta$: $S \times \Omega \rightarrow S$ Zustandsübergangsfunktion, die die "Automateneigenschaft" $\delta(q,\omega \bullet \omega') = \delta(\delta(q,\omega),\omega')$ f. alle $\omega,\omega' \in \Omega$, $s \in S$ besitzt, wobei $\omega \bullet \omega'$ das aus zwei Eingabesegmenten $\omega$, $\omega'$ mit angrenzenden Definitionsintervallen zusammengesetzte Eingabesegment ist
- $\lambda$: $S \rightarrow Y$ Ausgabefunktion.

Trivialerweise entspricht jedem I/O-System auch eine I/O- Funktionszuordnung $(T,X,\Omega,Y,F_S)$ mit

$F_S = \{f_s: \Omega \rightarrow (Y,T) \mid s \in S$ und $f_s(\omega)(t) := \lambda(\delta(s,\omega|_{t>})), \omega \in \Omega, t \in dom(\omega)\}$

(sukzessive Anwendung der Übergangsfunktion $\delta$ auf alle linken Teilstücke $\omega|_{t>}$ von $\omega$ bis t). Umgekehrt ist ein $f \in F$ einer IOFO nur dann durch ein I/O-System realisierbar, wenn f **kausal**, d.h. die Ausgabe von S bis zu jedem Zeitpunkt t durch die vergangene oder gegenwärtige Eingabe eindeutig bestimmt ist:

$f(\omega|_{t>}) = f(\omega)|_{t>}$ f. alle $t \in dom(\omega)$.

Im nächsten Schritt werden die Eingabe- und Ausgabesegmente eingeschränkt auf stückweise konstante Zeitfunktionen (die Stellen der Änderungen werden als Ereignisse bezeichnet). Auch der interne Zustand unterliegt zeitdiskreten Übergängen, wobei zwischen **externen**, durch Eingabeereignisse bewirkten, und **autonomen** Übergängen nach Ablaufen einer zustandsabhängigen Zeitspanne unterschieden wird.

**Def. 2.3 (discrete-event-System, DEVS)**

Ein discrete-event-System ist ein Tupel DEVS : = (T,X,S,Y,δ__ex,ta,δ__∅,λ) mit
T,X,S,Y ; wie in Def. 2.2.

δ__ex: $S \times T \times X \rightarrow S$ Externe Zustandsübergangsfunktion
δ__ex(s,e,x) neuer Zustand, wenn externe Eingabe x eintrifft, nachdem e Zeiteinheiten im Zustand s verstrichen sind

ta: $S \rightarrow T$ Timeout-Funktion ta(s): Zeitspanne bis zum nächsten autonomen Zustandsübergang bei Ausbleiben externer Eingabe (ta(s) = 0, ta(s) = ∞ möglich)

δ__∅: $S \rightarrow S$ Autonome Zustandsübergangsfunktion
δ__∅(s) neuer Zustand, nachdem ta(s) Zeiteinheiten im Zustand s ohne Eingabe verstrichen sind

λ: $S \rightarrow Y$ Ausgabefunktion.

Einem DEVS, das wohldefiniert ist (vgl. /ZEI 76/, Kap.9), läßt sich ein äquivalentes I/O-System nach Def. 2.2 zuordnen, dessen Übergangsfunktion δ aus der iterativen Anwendung von δ__∅ bzw. δ__ex auf den aktuellen Zustand bzw. das nächste Eingabeereignis in einer Historie ω besteht und das eine Ausgabe λ(s) zu den diskreten Zeitpunkten autonomer Zustandsübergänge δ__∅(s) erzeugt.

Die Bedeutung des DEVS-Formalismus liegt darin, daß für alle in den gängigen Simulationssprachen formulierten zeitdiskreten Simulationsmodelle auch äquivalente DEVS-Modelle existieren. Jeder Modellierungsansatz, der seinerseits die Klasse der DEVS-Modelle umfaßt, kann den Anspruch auf Allgemeingültigkeit (T8) erheben.

Auf der obersten Ebene des Zeigler'schen Ansatzes wird die Kopplung mehrerer gleichartiger Systeme DEVSe, betrachtet.

**Def. 2.4 (discrete-event-Netz, DEVN)**

Ein discrete-event-Netz ist ein Tupel DEVN : = $(\mathbf{K},(DEVS_i)_{i \in \mathbf{K}}, (infl_i)_{i \in \mathbf{K}}, SELECT)$ mit

**K**: Indexmenge der DEVS-Komponenten $DEVS_i$

$DEVS_i$: Komponente i

$\mathbf{infl_i \subseteq K}$ Menge der von $DEVS_i$ beeinflußten Nachbarkomponenten ($j \in infl_i \Leftrightarrow DEVS_i$ sendet an $DEVS_j$ Ausgabe)

SELECT: $2^{\mathbf{K}} \rightarrow \mathbf{K}$ Funktion, die eine Rangfolge der Elemente beliebiger Teilmengen von DEVS-Komponenten mit zeitgleichen Zustandsübergängen spezifiziert.

Folgende Randbedingungen müssen erfüllt sein:

- $SELECT(M) \in M \; \forall \; M \subseteq K$
- $i \notin infl_i \; \forall \; i \in K$
- $Y_i = \{(j,x_j) | j \in infl_i, \; x_j \in X_j\}$, d.h. die Ausgabe der Komponente $DEVS_i$ an eine Nachbarkomponente entspricht dem Typ der von ihr erwarteten Eingabe.

Mit Hilfe dieses Konzeptes lassen sich komplexe DEVS-Systeme hierarchisch in Teilkomponenten zerlegen bzw. aus solchen Teilkomponenten zusammensetzen (Modularisierungskonzept). Entscheidend hierfür ist die **Kompositionseigenschaft**: jedes Netz DEVN von DEVS-Komponenten ist selbst wieder ein DEVS. Der globale Zustand dieses DEVN ist das kartesische Produkt der Komponenten-Zustandsräume, seine globale Zustandsübergangsfunktion besteht in der Iteration folgender Schritte (eine formale Definition wird in /ZEI 84/, Kap. 8, angegeben):

- Auswahl einer Komponente $DEVS_i$ aus der Menge $IMMINENT := \{s_1,...,s_n\}$ der Komponenten mit kleinster verbleibender Zeit $ta(s_i)-e_i$ durch die SELECT-Funktion
- autonomer, lokaler Zustandsübergang $\delta\_\_\omega_i$ und Ausgabe $\lambda_i(s_i)$
- externe lokale Zustandsübergänge $\delta\_\_ex_j(s_j,e_j,\lambda_i(s_i).j)$ aller betroffenen Nachbarkomponenten.

### 2.2.1.2 Simulation, Echtzeitsimulation

Von Simulation spricht man, wenn die Experimente statt am Originalsystem S an einem zweiten System S' durchgeführt werden. Der interessierende Anfangszustand und die Eingabe für S werden in eine entsprechende Situation für S' übersetzt, die resultierende Ausgabe unter S' ermittelt und die Ergebnisse in das System S zurücktransferiert.

Für den einfachsten Fall der I/O-Funktionszuordnungen erhält man

**Def. 2.5 (Simulationszuordnung, I/O-Morphismus)**

Seien $S = (T,X,\Omega,Y,F)$ und $S' = (T',X',\Omega',Y',F')$ I/O-Funktionszuordnungen. Ein Paar von Abbildungen

$(g,k)$, $g: \Omega \rightarrow \Omega'$, $k:(X',T') \rightarrow (X,T)$

heißt **I/O-Morphismus (Simulationszuordnung) von S' auf S**
("S wird durch S' simuliert") $\Leftrightarrow$ für alle $f \in F$ existiert ein $f' \in F'$ mit $f = k \circ f' \circ g$.

Der wichtigste Spezialfall ist der, daß die Entsprechung der Historien f, f' für jeden Zeitpunkt einzeln mittels Abbildungen $\varnothing: T' \rightarrow T$, $\psi: T \rightarrow T'$ der Zeitbereiche ($\varnothing$ und $\psi$ monoton wachsend, $\varnothing \circ \psi = id(T')$ und Abbildungen $w\_\_x: X \rightarrow X'$, $w\_\_y: Y' \rightarrow Y$ der Wertebereiche nachvollziehbar wird, so daß

$g(\omega) = w\_\_x \circ \omega \circ \varnothing$, $k(y') = w\_\_y \circ y' \circ \psi$ für alle $\omega \in \Omega$, $y' \in (Y',T')$

gilt (schritthaltende Simulationszuordnung).

Die Abb. $\varnothing: T \rightarrow T'$ bildet z.B. einen kontinuierlichen Zeitbereich $\mathbb{R}$ auf den diskreten Wertebereich $\mathbb{N}$ einer Simulationsuhr ab und $w\_\_x$ physikalische Größen in ihre Programm- oder Maschinendarstellung. Schon aufgrund dieser Diskretisierung existiert eine exakte Simulation nach 2.5 in

der Praxis selten. Mit Hilfe einer Norm || auf Zeitfunktionen kann man die Simulationsgenauigkeit für gegebenen Anfangszustand f und Eingabehistorie ω definieren als
$\delta(\omega,f) := \| f(\omega) - k(f'(g(\omega))) \|$.

Def. 2.5 setzt zeitliche Verläufe (Zeitfunktionen) unterschiedlicher Systeme in Beziehung. Dadurch kann sowohl die Relation zwischen einem realen System und einem abstrakten Modell, als auch zwischen abstraktem Modell und lauffähigem Simulationssystem beschrieben werden. Die Zeit ist dabei in jedem Fall eine abstrakte Größe (unabhängige Variable der Funktionen). Wenn man nun ein Simulationsmodell S' selbst in seinem Ablauf beobachtet, kann man zusätzlich fordern, daß seine Ausgabe in ihrer realen zeitlichen Dynamik der des Originalsystems S entspricht. Ist y(t) differenzierbar, so bedeutet dies

d/dt y(t) = d/dte y'(te)

(te sei die externe Echtzeit bei der Beobachtung des Ablaufes von S').

Man spricht dann von Echtzeitsimulation oder Echtzeitexperiment. Zunächst also ordnet man jedem interessierenden Ereignis e im Ablauf von S' den tatsächlichen Zeitpunkt te(e) seines Vorkommens zu. Die Menge E der Ereignisse sei unterteilt in

E__x: Eingabe-Ereignisse (Anwendung von δ__ex bei einem DEVS)

E__y: Ausgabe-Ereignisse (Anwendung von λ)

E__s: interne Ereignisse (Anwendung von δ__ω)

Der einem Ereignis $e \in E\_\_x$ [E__y] entsprechende Ein-/Ausgabewert sei w__x(e) [w__y(e)]. Eine Folge e0,e1,...,en von Eingabe-[Ausgabe-]Ereignissen heißt (mit einem Faktor d skalierte) **Echtzeit-Codierung** einer zeitdiskreten Historie z:[T,T']->X[Y], wenn für alle Stützstellen $(t_i,z_i)$ $(T \le t_i \le t_{i+1} M \le T')$ in z gilt

$z_i$ = w__x ($e_i$) [w__y ($e_i$)] und

$t_i = d^*te(e_i) + d_0$ mit geeigneten $d_0, d \in \mathbf{R}$.

d>1 bedeutet "schneller als Echtzeit" (Zeitraffer)

d<1 "langsamer als Echtzeit" (Zeitlupe).

In einem ausführbaren Simulationsmodell S' eines IOFO S gehört zu jeder Folge von Eingabeereignissen $e\_\_x_1,..,e\_\_x_n \in F(E\_\_x)$ eine i.a. je nach Anfangszustand von S' unterschiedliche Folge von Ausgabeereignissen $e\_\_y_1,..,e\_\_y_m \in \mathbf{F}(E\_\_y)$.

**Def. 2.6 (Echtzeitsimulation)**

Eine I/O-Funktionszuordnung S=(T,X,Ω,Y,F) wird in Echtzeit durch ein ausführbares Simulationsmodell S' simuliert, wenn für jedes f∈F und jede Folge von Eingabeereignissen e__x(1),..,e__x(n) ∈**F**(E__x), welche Echtzeitcodierung einer Historie ω∈Ω ist, eine zugehörige Folge von Ausgabeereignissen e__y(1),..,e__y(m) ∈**F**(E__y) erzeugt wird, welche Echtzeitcodierung von f(ω) ist.

Mit anderen Worten: wenn S' von seinen Nachbarn die Eingabe von S echtzeitgetreu erhält, übergibt S' auch die Ausgabe von S echtzeitgetreu an seine Nachbarn.

Die praktische Bedeutung von Def. 2.6 liegt im Bereich sog. Echtteile-Simulationen. In einem Simulationsmodell eines gekoppelten Systems $K_1,..,K_n$ werden einige der Komponenten $K_i$ als reales Testobjekt übernommen und ihre Umgebung, d.h. die übrigen Komponenten $K_j$, in Echtzeit simuliert (**Echtteilesimulation in Echtzeitumgebung, echtzeitorientiertes Experimentieren**).

**Beispiele**

(1) Das bekannteste Beispiel sind Trainingssimulatoren, insbesondere Flug- und Fahrzeugsimulatoren /FOR 83//LIR 86/. Die wesentliche "reale Komponente" ist hier ein **Mensch** (Operateur), der am Simulator die Bedienung einer technischen Anlage erlernt. Hier ergibt sich die Notwendigkeit einer Simulation in Echtzeit zwingend: nicht nur die Wirkung der Bedienereingriffe auf das technische System als solche ist korrekt wiederzugeben, sondern auch ihre reale zeitliche Dynamik ist den sensorischen und motorischen Fähigkeiten des Menschen entsprechend erlebbar zu machen. Das Prinzip der Trainingssimulatoren wurde nicht nur auf Luft- und Raumfahrzeuge, sondern auch auf verfahrenstechnische Prozesse angewandt /KEL 85/.

(2) Falls bei der Entwicklung und dem Leistungstest technischer Anlagen Teile des **technischen Prozesses** als reale Komponenten integriert werden (z.B. Elektronik, Sensorik, Antriebe) spricht man von 'hardware-in-the-loop'-Simulationen /NGD 86/.

(3) Von besonderem Interesse für uns ist schließlich der Fall, daß ein **digitales PFS** die reale Komponente darstellt. Ihre Leistungsmessung mit Hilfe von HW- und SW-Monitoren oder sog. Testbettsystemen /FBW 82/ stellt einen Sonderfall dar: die "simulierte Umgebung" besteht dann nur aus Meßeinrichtungen zur Zeit- und Ereigniserfassung und ggf. -verarbeitung, sowie Lastgeneratoren oder einfachen Prozeßsignal-Generatoren, die auf die Prozeßperipherie einwirken. Interessanter und aussagekräftiger wird es, wenn die Rückkoppelung der Stell- zu den Meßgrößen durch ein originalgetreues dynamisches Modell des technischen Prozesses geschlossen wird. Hierzu kann eine Analog- oder Hybridrechnerkonfiguration, oder aber - flexibler, kostengünstiger, und mit höherer numerischer Genauigkeit - ein separates Mikrorechnersystem eingesetzt werden. Dieser Weg wurde z.B. in /KOH 81//PIM 83//PSS 85/ vorgeschlagen und experimentell untersucht.

### 2.2.2 Echtzeitorientierte Leistungsbewertung

#### 2.2.2.1 Echtzeitsimulation des technischen Prozesses

Die wesentlichen Erfahrungen aus /KOH 81//KOH 82/ bei der Echtzeitsimulation kontinuierlicher technischer Prozesse als Testumgebung für digitale PFSe werden in knapper Form wiedergegeben, da sie wesentlich die These (T1) in Kap. 1.3 (Unzulänglichkeit der rein echtzeitorientierten Methoden) erhärten. Dabei werden folgende, für viele Echtzeit-Simulationssysteme zutreffende Annahmen gemacht:

(1) Der TP sei durch ein gewöhnliches, nichtlineares DGS beschreibbar:

$\mathbf{x}(t) = F(\mathbf{x}(t), \mathbf{u}(t))$ $\quad \mathbf{x} \in \mathbf{R}^n$ Zustandsvektor

$\mathbf{y}(t) = G(\mathbf{x}(t))$ $\quad \mathbf{u} \in \mathbf{R}^m$ Stellgrößenvektor

$\quad \mathbf{y} \in \mathbf{R}^l$ Meßgrößenvektor

(2) Das Anfangswertproblem (1) mit $\mathbf{x}(t_0) = x_0$ wird iterativ durch ein Einschrittverfahren mit äquidistanter Schrittweite h gelöst:

$\mathbf{x}[k] := F_h(\mathbf{x}[k-1], \mathbf{u}[k-1])$ $\quad\quad$ $\mathbf{x}[k] := \mathbf{x}(kh)$

$\mathbf{y}[k] := G(\mathbf{x}[k])$ $\quad\quad$ $\mathbf{u}[k] := \mathbf{u}(kh)$

Falls unterschiedliche Schrittweiten für unterschiedliche Teilprozesse oder dynamische Schrittweitensteuerung angewandt werden, so sei o.E. h die kleinste überhaupt vorkommende Schrittweite.

(3) Das Verfahren (2) wird im Mittel alle h Zeiteinheiten in Echtzeit aktiviert. Zu den Zeitpunkten $t_{e,k}$ werde die gültige Stellgrößenbelegung des PFS als $\mathbf{u}[k]$ übernommen; zu den Zeitpunkten $t_{a,k}$ werde ein neuer Prozeßzustand $\mathbf{x}[k]$ bzw. Meßgrößen $\mathbf{y}[k]$ zur Verfügung gestellt, wobei i.a.

$kh \le t_{e,k} < t_{a,k} \le (k+1)h, \quad k = 0,1,...$

Die Schnittstelle zwischen TP und PFS sei asynchron, d.h. das PFS kann zu beliebigen (zufälligen) Zeitpunkten Stellwerte ausgeben oder Meßwerte anfordern; dennoch stellen die $t_{e,k}$ bzw. $t_{a,k}$ das kleinste Zeitraster dar, innerhalb dessen Stellwertänderungen der Umgebung bzw. Zustandsgrößenänderungen durch die Umgebung wahrgenommen werden.

Die durch diese Simulation verursachten Verfahrensfehler setzen sich aus drei Anteilen zusammen:

(a) **Eingangsdiskretisierungsfehler**

Abbildung einer beliebigen Treppenfunktion $\mathbf{u}:[t0,t1] \rightarrow U$ auf eine Treppenfunktion $\mathbf{u}'$ mit äquidistanten Stützstellen $(kh, \mathbf{u}(t_{e,k}))$

(b) **Numerischer Verfahrensfehler**

lokaler Diskretisierungsfehler des Einschrittverfahrens beim Übergang

$\mathbf{x}'[k] := F_h(\mathbf{x}'[k-1], \mathbf{u}'[k-1])$

(c) **Ausgangsdiskretisierungsfehler**

Abbildung der durch die äquidistanten Stützstellen $(kh, \mathbf{x}'[k])$ laufenden kontinuierlichen Lösung auf eine Treppenfunktion $x'':[t0,t1] \rightarrow X$ mit den Stützstellen $(t_{a,k}, \mathbf{x}'[k])$.

Die Auswirkungen der Modellierungsfehler des TP als offenes System auf die Zustandstrajektorien eines geschlossenen Wirkungskreises unter ein und demselben PFS sind analytisch schwer abschätzbar. In /KOH 82/ wurde unter Vernachlässigung des numerischen Verfahrensfehlers in (b) eine Rückwärtsinterpretation des globalen Fehlers angegeben: Der geschlossene Wirkungskreis aus dem in Echtzeit simulierten zeitdiskreten Prozeßmodell S' und dem **realen** PFS F kann auch als Wirkungskreis mit dem **realen** kontinuierlichen TP S, aber einer anderen Prozeßführung F" interpretiert werden, deren zeitliches Verhalten (Abtastzeiten, Totzeiten) um Beträge der Größenordnung h gegenüber F verfälscht ist. Dies ist deshalb interessant, weil mit Hilfe der Echtzeitsimulation gerade die zeitliche Dynamik des PFS in seiner Zielumgebung realitätsgetreu erfaßt werden sollte. Wie klein sollte also h gewählt werden, um diesem Ziel möglichst nahe zu kommen, und was ist realistischerweise möglich?

In /KOH 82/ wurde dazu eine einfache Abschätzung (Unschärfebeziehung) unter der Annahme einer zyklischen Aktivierung des PFS angegeben. Mit

Z__R Zeitliche Periode des PFS, z.B. kleinstes gemeinsames Vielfaches der Abtastzeiten aller Regelkreise,
$\simeq$ c__R/v__R mit c__R: Komplexität des PFS (Operationen innerhalb einer Periode Z__R)
v__R: Durchsatz des PFS-Rechensystems (Operationen pro Zeiteinheit)

Z__M Zeitliche Periode des Prozeßmodells (Integrationsschritt (2)),
$\simeq$ c__M/v__M mit c__M: Komplexität (**NB**: **unabhängig** von h!)
v__M: Durchsatz des Modellrechnersystems

d Ablaufgeschwindigkeit der Simulation gegenüber Echtzeit (2.2.1.2)

K Verhältinis Z__R/h der Zeitkonstante Z__R des PFS zur Modellschrittweite h

gilt

$$(2.1) \quad \frac{K}{d} \leq \frac{c\_R}{v\_R} * \frac{v\_M}{c\_M} = \text{const für eine gegebene Konfiguration}$$

Bei der Interpretation dieser Formel ist nun zu beachten

- d ist ein Maß für die zeitliche Wiedergabegenauigkeit ("Realitätsnähe") des PFS; nur für d = 1 wird die Echtzeit-Dynamik des PFS unverfälscht reproduziert
- K mißt das Auflösungsvermögen des Prozeßmodells. Wenn Schwankungen in den Verarbeitungszeiten des PFS vom Betrag $\varepsilon$ in den Regelgrößenverläufen noch erkennbar sein sollen, muß für die Abtastzeit $h \leq \varepsilon/2$ (Shannon'sches Abtasttheorem), nach Ergebnissen von Wells /WEL 83/ sogar $h \leq \varepsilon/8$ gelten, um in der Praxis eine gute Auflösung zu erreichen. Nun sind die Schwankungen $\varepsilon$ ihrerseits im Mittel viel kleiner (um einen Faktor $a >> 1$) als die Periode Z__R, sonst läge ein Entwurfsfehler im PFS vor, d.h. erforderlich ist
  $K = Z\_\_R/h \geq 8Z\_\_R/\varepsilon \geq 8a.$
- die Komplexität c__M des Prozeßmodells gleicht in der Praxis der der Prozeßführung c__R oder übertrifft diese, wenn das Prozeßmodell die dem Prozeßführungskonzept unterliegenden Annahmen und Vereinfachungen wirksam überprüfen soll. Mit einem Faktor
  $g := c\_\_M/c\_\_R \geq 1$
  folgt dann aus (2.1)

$$(2.2) \quad \frac{v\_M}{v\_M} \geq 8*g*a >> 1$$

Dies bedeutet: unabhängig vom allgemeinen HW-Leistungsniveau müßte das Rechensystem zur Prozeßsimulation immer ein Vielfaches leistungsfähiger sein als das zur Prozeßführung, um aussagefähige Ergebnisse zu garantieren.

Dabei stellt der hier geschilderte Fall einer gleichmäßigen, periodischen Belastung noch einen eher gutartigen Fall dar. Falls bei der Prozeßmodellierung asynchrone Ereignisse zu erkennen

und zeitgerecht auszugeben sind wie in /GOB 84/, ergeben sich noch härtere Probleme für die Echtzeitsimulation.

**Beispiel**

Bei der Simulation kontinuierlicher Prozesse mit zeitdiskreten Ereignissen (vgl. /HEL 80/,/SCH 84/, und Kap. 6.2.1) wird vor jedem Integrationsschritt geprüft, ob bei unveränderten externen Einflußgrößen im nächsten Integrationsschritt ein solches Ereignis ('crossing', z.B. Grenzwertüberschreitung) stattfindet; falls ja, ist der Zeitpunkt t' mit einer Auflösung h_c << h innerhalb eines Intervalls [t, t+h] zu lokalisieren (durch Bisektion). Bei t' ist z.B. ein Alarm an das PFS auszugeben.
Übertragen auf den **Echtzeitfall** bedeutet das: die Prüfung beginnt frühestens zum Zeitpunkt t, da ihr Ergebnis von der **aktuellen** Stellgrößenbelegung abhängt, und muß bis t+h_c beendet sein, weil schlimmstenfalls hier bereits ein Ereignis stattfindet. Dies bedeutet gegenüber einem normalen Integrationsschritt eine Erhöhung der Rechnerbelastung um einen Faktor h/h_c*log(h/h_c). Falls h/h_c=100, müßte eine Kapazitätsreserve von ca 700 gegenüber einer rein kontinuierlichen Simulation vorhanden sein, um die Echtzeitanforderungen zu **garantieren** (auch wenn diese Kapazität nur selten real benötigt wird).

In der Praxis wird der Benutzer sich natürlich mit der verfügbaren Rechenleistung abfinden; ggf. wird die Abtastrate reduziert. Wenn Verletzungen der Echtzeitbedingungen nur sporadisch vorkommen, werden sie gar nicht wahrgenommen. Und hier liegt das Kernproblem echtzeitorientierter Leistungsbewertung: der Benutzer hat kaum eine Chance, in den Gesamtergebnissen den eigentlich interessierenden Anteil des Testobjektes (Dynamik des technischen Prozesses, funktionelle und zeitliche Eigenschaften des PFS) von den verfälschenden Einflüssen der Implementierung der Prozeßmodelle (genannt **Interferenz**) zu trennen. Die Ergebnisse hängen in zufälliger Weise von der Experimentmaschine (Modellrechnersystem) ab; die Erstellung wiederverwendbarer, portabler Simulations-SW wird dadurch unmöglich gemacht (vgl. hierzu bereits /GEA 77/).

Der Einsatz von Echtzeitsimulationen sollte daher auf den o.g. Bereich (1) der Trainingssimulatoren beschränkt bleiben, wo er wegen der direkten Rückkoppelung mit einem menschlichen Bediener wirklich notwendig ist- und gerade dort sind ihre Zeitbedingungen auch leichter zu erfüllen, weil der Mensch im Vergleich zu digitalen PFSen ein vergleichsweise "träges" dynamisches System ist.

### 2.2.2.2 Leistungsvorhersage durch echtzeitorientierte Experimentumgebungen

Auch für die Leistungsvorhersage des Entwicklungsobjektes (PFS) im Entwurfsstadium bieten echtzeitorientierte Experimentumgebungen, die über zusätzliche Meßeinrichtungen verfügen, keine optimale Unterstützung - obwohl ein solcher Einsatz prinzipiell möglich ist. Dies gilt sowohl für die Demonstration der Funktionsfähigkeit ('prototyping'), als auch vor allem für das zeitliche Verhalten.
Die Hauptprobleme liegen hier in der Bewertung eines unvollständigen Testobjektes. Beim Testen großer, verteilter SW-Systeme in ihrer Zielumgebung werden zwar fehlende Module oder Benutzer oft durch 'dummy'-Module ersetzt, welche Echtzeit-Verzögerungen (DELAY's)

enthalten und den zeitlichen Ablauf des Gesamtsystems im groben nachzubilden gestatten. Insgesamt erweisen sich solche Konstrukte zur zeitlichen Nachbildung/Vorhersage in Entwicklung befindlicher Systeme aber als unflexibel, schwerfällig, ungenau und ineffizient.
Unflexibel u.a. deshalb, weil ein DELAY semantisch nicht unterscheidet, ob die entsprechende Zeitdauer eine Zwischenankunftszeit zweier Ereignisse, oder eine Verbrauchszeit, z.B. CPU-Zeit eines noch fehlenden Algorithmus, repräsentiert.
Ein Hauptproblem der Echtzeitbewertung von Prototypen ist, daß mit jeder **Funktion** im System automatisch das Zeitverhalten ihrer vorläufigen **Implementierung** bewertet wird, auch wenn es noch keine Aussagekraft für die Einsatzversion besitzt, sondern nur die Effizienz des Codegenerators einer Prototyp-Sprache mißt. Benötigt wird eine **Entkoppelung** von funktionellem und zeitlichem Verhalten, so daß das Zeitverhalten als **Entwurfsparameter** flexibel und gezielt variiert werden kann ("was wäre, wenn die Funktion f in einer gedachten Zielumgebung eine gegebene zeitliche Charakteristik hätte").
Ferner ist der Echtzeit-Test einer viele Leerzeiten enthaltenden Anwendung ineffizient; ein Test in Zeitraffer würde andererseits das Zeitverhalten der bereits realisierten Teile verfälschen.

Diese Nachteile sind nur durch eine virtuelle Zeitführung zu vermeiden.

### 2.3 Experimentelle Leistungsbewertung im hierarchischen Entwurf

Während bei den echtzeitorientierten Verfahren zur Leistungsbewertung die Genauigkeit und Verfahrensfehler den "Knackpunkt" bilden, steht bei Simulationsmethoden in virtueller Zeit der Prozeß der hierarchischen Detaillierung des zu entwickelnden Zielsystems (PFS) und seiner **Darstellung** im Vordergrund. Konkret geht es um die Thesen (T2),(T3) in 1.3, daß die Einheit der Darstellung des Zielsystems aus Entwicklersicht und als Testobjekt bzw. Simulationsmodell für die Leistungsbewertung mit herkömmlichen Modellierungswerkzeugen verhindert wird. Die mit zunehmender Detaillierung wachsende Kluft zwischen Systementwicklung und -modellierung führt zu Informationsverlusten, Inkonsistenz und erhöhtem Modellierungsaufwand.
Betrachtet wird eine Folge schrittweise detaillierter Entwürfe $E_0,E_1,..,E_n$, in der jedes $E_{i+1}$ aus $E_i$ durch neue Entwurfsentscheidungen*) aus einer Menge möglicher Alternativen $E_{i+1,1},..,E_{i+1,k}$ entsteht, die in ihren Auswirkungen vergleichend bewertet werden sollen. $E_n$ sei ein "reales System", wenn in $E_n$ keine Entscheidungen mehr zu treffen sind; jedes $E_i$ kann als vergröbertes Modell von $E_{i+1}$ angesehen werden.
In Abschnitt 3.2 werden Beispiele gegeben, welche Entwurfsentscheidungen in welchen Entwicklungsstadien eines PFS typischerweise zu treffen und wie sie zu bewerten sind, doch spielt dies für die folgende allgemeine Diskussion keine Rolle. Ebenso kann die Tatsache ignoriert werden, daß der Entwurfsprozeß nicht strikt top-down verläuft, sondern daß u.U. Entwurfsentscheidungen revidiert werden müssen.

*) z.B. Festlegung der SW-Struktur (Datenobjekte,Operationen), Verfahrensauswahl, Betriebsmittel-Zuordnung, Verfahrensparameter, Tuning-Paramter etc.

### 2.3.1 Architektur hierarchischer Modellierungssysteme

Da PFSe keine beliebigen sozio-ökonomisch-technischen Systeme sind, sondern verteilte Rechensysteme, die i.a. gewissen Architekturkonzepten und Konstruktionsrichtlinien folgen, wird die Diskussion unter folgender vereinfachender **Annahme** geführt: die **Detaillierungs hierarchie** $E_0,...,E_n$ korrespondiert mit einer Hierarchie **abstrakter Maschinen** (z.B. Regelungsebenen, Kommunikationsebenen). Jedem Detaillierungsgrad $E_i$ entspricht also eine (Teil)Hierarchie $(M_0,...,M_i)$ bereits entworfener oder realisierter abstrakter Maschinen. Jede Maschine $M_i$ sei wiederum als Paar $(S_i, Z_i)$ gegeben, wobei

- $S_i$ eine **Sprache** $S_i$ zur Darstellung von $M_i$ mit vorgegebenen Konzepten (Datenobjekte und Operationen, Modularisierungskonzepte, Kontrollstrukturen, semantische Regeln für den Zugriff auf oder die Kooperation von Objekten, vordefinierte Standardfunktionen etc.);
- $Z_i$ die konkreten Objekte, Attribute und Dienstleistungen, die die Maschine $M_i$ zur Verfügung stellt und die **in** der Sprache $S_i$ dargestellt sind, also ihre Konzepte benutzen.

Die Beziehung $M_i \rightarrow M_{i+1}$ beschreibt, daß $M_i$ mit Hilfe von Dienstleistungen der nächsttieferen Maschine $M_{i+1}$ realisiert wird. Das Ein-/Ausgangsverhalten von $M_i$ ergibt sich aus dem der benutzten Maschine $M_{i+1}$, der vorgegebenen Semantik der Sprache $S_i$ sowie der eigentlichen Implementierung von $Z_i$ mit Hilfe von $M_{i+1}$ und $S_i$.
Die Begriffe "Sprache" und "Maschine" korrespondieren eng: die Konstrukte der Sprache S werden als Dienstleistungen durch eine geeignete Basismaschine $M_S$ erbracht, auf die sich höhere Maschinen (S, Z) stützen; umgekehrt entspricht der Maschine (S,Z) eine um die konkreten Objekte Z der Anwendung erweiterte Sprache S.

Die vereinfachende Gleichsetzung von hierarchischer Detaillierung bzw. Dekomposition und Hierarchie abstrakter Maschinen (als Rechner-Architekturprinzip) wird in verschiedenen Arbeiten kritisiert /KLU 85/,/SAN 77/. Solange man jedoch Maschinenhierarchie nicht nur als Aufrufhierarchie von Funktionen oder Prozeduren versteht, bleibt die obige Darstellung hinreichend allgemein. Der schichtenweise Aufbau von Modulhierarchien in Modula2 oder package-Hierarchien in ADA, kann damit ebenso beschrieben werden wie die hierarchische Dekomposition eines PrT-Netzes in Teilnetze. Die vorgegebenen Regeln der Sprache betreffen im ersten Fall u.a. Schachtelung, Gültigkeitsbereiche, Benutzungsrechte und dynamischen Beziehungen zwischen aufrufenden und aufgerufenen Moduln (synchrone /asynchrone Operationen, gegenseitiger Auschluß oder Wiedereintrittsfähigkeit). Im zweiten Fall bestehen sie aus syntaktischen Regeln zur Konstruktion von Netzen und Teilnetzen, und dynamischen Regeln, wie ein Teilnetz innerhalb eines Gesamtnetzes aktiviert wird (Schaltregel, nebenläufiges Schalten, Atomarität des Schaltens, Konfliktauflösung, ggf. mit zeitlicher Semantik des Schaltens).

Zur entwurfsbegleitenden Leistungsvorhersage muß das Zielsystem in einer Modellierungssprache W als Entwurf $(W,Z_0),...,(W,Z_i)$ entsprechend seinem Detaillierungsgrad i dargestellt, und unter der zugehörigen **Simulationsmaschine** $M_W$ ausgeführt werden. Zwei Haupteigenschaften zeichnen $M_W$ speziell als Simulationsmaschine gegenüber einer "konventionellen" Laufzeitmaschine (Betriebssystem) aus:

(1) Virtuelle Zeitführung, die zur expliziten Nachbildung des zeitlichen Ablaufs des Zielsystems dient.

(2) Vordefinierte Funktionen für spezielle Lösungsverfahren, z.B. numerische Integration oder Zufallszahlengenerierung, sowie Schnittstellen zur Erfassung, statistischen Auswertung und Darstellung von Leistungsgrößen.

Die interne, u.U. selbst hierarchische Realisierung von $M_W$ ist für das folgende belanglos.

**Beispiele** für W, $M_W$

(a) Standard-Simulationssprachen wie SIMULA /ROH 73/, SIMSCRIPT /KVM 75/, GASP /RIC 82/,

(b) blockorientierte Simulationssprachen auf der Basis von Warteschlangennetzen (GPSS /SCH 84/, QGERT, SLAM /PRP 79//ORY 85/ SIMAN /PEF 84/)

(c) Modellierungssprachen auf der Basis von zeitattributierten Petri-Netzen (z.B. Funktionsnetze, PrT-Netze)

(d) Kombinierte Sprachen, die zur Modellspezifikation "im Großen" (d.h. zur Spezifikation der Nebenläufigkeiten im System und der Schnittstellen zwischen nebenläufigen Objekten) Petrinetze o.ä. einsetzen, zur detaillierten algorithmischen Spezifikation dagegen sequentielle Programmiersprachen. Beispiele sind etwa
- BORIS /DEM 84/, eine auf Kanal-Instanz-Netzen mit spezieller Schaltregel basierende, blockorientierte Simulationssprache, deren sequentielle (Instanzen)Algorithmen in PASCAL formuliert sind.
- DAEMON /LES 86/, eine Modellierungs- und Entwurfssprache zur Leistungs- und Verfügbarkeitsuntersuchung fehlertoleranter, verteilter Netzwerkanwendungen, die auf erweiterten Funktionsnetzen basiert und als Modellierungssprache für die Instanzalgorithmen sequentielles SIMULA vorsieht.

(e) Modellierungssysteme, die Spracherweiterungen einer der Simulationssprachen aus (a) bis (c) sind und vordefinierte Modellkomponenten spezieller Anwendungsbereiche enthalten. Besonders gut eignet sich hierfür SIMULA mit seinen Klassenhierarchien. OASIS /UNB 82/ und DEMOS sind SIMULA-basierende Modellierungs- und Entwurfssprachen für verteilte Rechensysteme, die vielfältige, parametrisierbare Submodelle für HW- und SW-Betriebsmittel bereitstellen.

(f) Höhere Programmiersprachen (u.a. ADA, PASCAL, C), die um einfache Dienste zur zeitdiskreten Simulation erweitert wurden, z.B. zum Einplanen und Aktivieren von Ereignissen in einer Zeitliste.

(g) Sprachen bzw. Umgebungen, deren Schwerpunkt mehr auf der SW-Spezifikation liegt, die aber Spezifikationen direkt als Simulationsmodell ausführen oder wenigstens ausführbare Simulationsmodelle rechnergestützt generieren können, z.B.
- PAISLEY /ZAV 82//ZAS 86/, eine Spezifikationssprache für Echtzeitsysteme, welche kooperierende Prozesse mit einer funktionalen Programmiersprache zur Spezifikation der sequentiellen Prozeßalgorithmen verbindet.

- SARA /EFR 86/, ein interaktives System für Entwurf, Modellierung und Leistungsbewertung verteilter Systeme, das ebenfalls eine graphische, aus Petrinetzen abgeleitete Entwurfssprache (Graph Model of Behaviour) zur Spezifikation der Systemtopologie und der Kontroll- und Datenflüsse vorsieht.
- SREM /ALF 85/, eine Entwicklungsumgebung für die Anforderungsspezifikation und den Grobentwurf von Echtzeitsystemen. Die oberste Ebene bildet die Definitionssprache RSL (requirement statement language) für die statische Systemstruktur (Definition der Verarbeitungsinstanzen, ihrer Ein-/Ausgabedaten, Nachrichtentypen, zeitliche Anforderungen etc.). Die dynamischen Aspekte werden durch eine datenflußorientierte graphische Sprache (R-Netze) beschrieben. Aus diesen Beschreibungen können Simulationsmodelle automatisch generiert und ausgewertet werden.

(h) Modell-Generierungssysteme, mit denen ausführbare Modelle aus einem engen Anwendungsbereich, etwa Fertigungsstraßen oder Signalübertragungstechnik, automatisch mit Hilfe von Konfigurationswerkzeugen, vordefinierten Modellbausteinen und einer grafischen Modellbeschreibung erzeugt werden /SMM 86//SWR 87/.

Der Zielsystem-Entwurf $(W,Z_0),...,(W,Z_i)$ und $M_W$, i.f. auch **Wirtsmaschine** genannt, allein genügen i.a. noch nicht für die Leistungsbewertung, sondern folgende Komponenten kommen hinzu (vgl. Abb. 2.1):

(A) ein ebenfalls in W formuliertes **Black-Box-Modell Z** aller bereits benutzten, aber bisher weder implementierten noch durch die Sprache W selbst bereitgestellten Dienstleistungen des Zielsystems, repräsentiert z.B. durch ihren Bedarf an virtuellen Betriebsmitteln (abstrakter Zeitbedarf, CPU-Bedarf, Speicher-Bedarf, Umfang zu transportierender Nachrichten etc.)

(B) **Umgebungsmodelle**, im Fall von PFSen Modelle des technischen Prozesses; ferner Komponenten zur Leistungsgrößenmessung, deren Detaillierungsgrad mit dem des Zielsystems korrespondiert ($L_i$ in Abb. 2.1.a).

Nach Anforderung (c) in Kapitel 1.2 soll die Modellierungssprache W zugleich **die** Entwurfssprache des Zielsystems sein, muß sich also für Entwurf und Implementierung verteilter Softwaresysteme eignen und insbesondere deren hierarchische Verfeinerung (top-down-Entwurf) unterstützen; ungeeignet hierfür wären aus diesem Grund viele der blockorientierten Simulationssprachen unter (b).

### 2.3.2 Wirtsorientierte Leistungsbewertung

Dem Systementwickler/Modellierer stehen zu Beginn seines Projektes zwar i.a. verschiedene Modellierungssprachen W zur Verfügung; hat er sich aber für eine entschieden, ist ihr Repertoire an Dienstleistungen und semantischen Regeln ihm **vorgegeben**. Genau dies trifft aber auch auf das Zielsystem (PFS) selbst zu: es wird letztlich mit Hilfe **existierender** Schnittstellen und Dienste realisiert (Hersteller-Echtzeit-BS-Kerne, Rechnernetz-Kommunikationsdienste, Zielrechner-Sprachen, Zielrechner-Hardware, Prozeßperipherie-Schnittstellen, aber u.U. auch komplexere, wiederverwendete SW-Subsysteme), wobei durchaus mehrere Alternativen solcher **Zielmaschinen** zur Auswahl stehen können. Normalerweise sind nun Wirts- und Zielmaschine **semantisch verschieden**, d.h. durch unterschiedliche Dienste, Sprachen und Ablaufsemantik

gekennzeichnet.
Hier liegt der Kern des Problems, daß sich die Gleichheit der Modellierungs- und der Entwicklungssprachen im wirtsorientierten Ansatz nicht bis zum lauffähigen Zielsystem durchhalten läßt und daß Zielsystem-Entwicklung und -Leistungsbewertung schließlich auseinanderklaffen.

Ausgehend von $(W,Z_0),...,(W,Z_i)$ beginne die Implementierung des Zielsystems, dargestellt als Hierarchie $(S_j,Z_{i+1}),..(S_j,Z_j),...,(S,Z)$ mit einer gegebenen Zielmaschine $M_Z:=(S,Z)$ als unterster Schicht (Abb. 2.1b,c linke Seite). Dem unter $M_W$ simulierten Ein-/Ausgangsverhalten von $(W,Z_0),..(W,Z_i)$ steht das reale Ein-/Ausgangsverhalten gegenüber, das sich auf die Implementierungsentscheidungen $(Z_{i+1},...,Z_j)$, Sprache $S_j$ und die Zielmaschineneigenschaften von $M_Z$ gründet. Das Problem ist: wie läßt sich hinreichende Übereinstimmung beider Systemverhalten - in Bezug auf die Führungsziele des PFS - prüfen bzw. konstruktiv sicherstellen? Traditionell werden zwei Wege beschritten.

#### 2.3.2.1 Stufen-Ansatz

Stufen-Ansatz bedeutet: man setzt in den verschiedenen Entwicklungsstadien (Entwurf, Implementierung, Test) nacheinander unterschiedliche Werkzeuge für die Leistungsbewertung ein.
Die Modellierung und Bewertung auf der Wirtsmaschine wird mit dem Entwurfsstadium $(W,Z_0),...,(W,Z_i)$ beendet. Nachdem die Implementierung des PFS auf der Zielmaschine $M_Z$ lauffähig geworden ist, werden dort **echtzeitorientierte** Verfahren, z.B. Meßsysteme, eingesetzt. Folgende Konsequenzen ergeben sich:

- Um überhaupt anwendungsorientierte, **vergleichbare** Leistungsgrößen zu erhalten, sind die unter $M_W$ benutzten Umgebungsmodelle $(W,L_i)$ des technischen Prozesses in der Zielumgebung $M_Z$ neu zu implementieren/installieren und **in Echtzeit zu simulieren**. Dies erfordert einen erheblichen zusätzlichen Aufwand, insbesondere falls die Zielmaschine keine gute Unterstützung für die Echtzeitsimulation bietet.
- Angesichts der Interferenzprobleme (2.2.2.1) ist der Aussagewert der ermittelten Leistungsgrößen auf der Zielmaschine zweifelhaft.
- Falls Diskrepanzen der Leistungsgrößen zwischen Simulations- und Zielumgebung dem Zielsystem zugeschrieben werden, sind dann Eigenschaften der Zielmaschine oder Implementierungsentscheidungen des Entwicklers verantwortlich, und ggf. welche (Schichten/Subsysteme/Moduln)? Eine gezielte **Variation** zeitlicher Einflußgrößen zur Sensitivitätsanalyse - z.B. Rechenzeiten von Algorithmen, Kapazität von HW-Komponenten - ist in einer Echtzeitumgebung nach 2.2.2.2 kaum möglich. Alle Einflüsse der Implementierung und der Zielmaschine tragen nur noch summarisch zum Gesamtverhalten bei.

Mit diesem Ansatz ist schon die Symptomerfassung beeinträchtigt und erst recht die Diagnose und die gezielte Verbesserung des Systemverhaltens. Das Problem verschärft sich, falls der Ent-

wurf $(W,Z_0),...,(W,Z_i)$ selbst revidiert wird und mehrere Iterationen zwischen Modellierungs- und Zielumgebung nötig werden.
Übrigens spielt es keine Rolle, ob die Implementierung des Zielsystems manuell durch den Entwickler oder - in Zukunft zunhemend auch in verteilten Systemen - durch automatische Transformation eines Compilers oder Programm-Generierungssystems erfolgt. Ein automatisiertes System wäre denselben Unsicherheiten in der Bewertung seiner Entscheidungen v.a. hinsichtlich Echtzeitverhalten, unterworfen.

#### 2.3.2.2 Parallel-Ansatz

Die Leistungsbewertung wird auf der Wirtsmaschine $M_W$ parallel zur Implementierung des Zielsystems weitergeführt, indem die Schichten $..,(S_i, Z_j)$ und letztlich die Zielmaschine $M_Z$ selbst als zeitdiskrete Modelle $...,(W,Z_j^W)$ in der Sprache W nachgebildet werden (Abb. 2.1b). Dies erlaubt die Leistungstests in der komfortablen Modellierungsumgebung und in virtueller Zeit fortzusetzen, erfordert aber letztlich verdoppelten Aufwand für Entwicklung, Versionsführung, gegenseitige Konsistenzprüfung von Modell und Zielsystem, der nur der Bewertung, nicht aber der eigentlichen Entwicklung des Zielsystems zugute kommt.

Um hinreichend präzise Leistungsdaten zu erhalten, ist es i.a. notwendig, sogar die **interne** Schichtenhierarchie $...,(S_n,Z_n)$ der Zielmaschine $M_Z$ in der Sprache $M_W$ nachzubilden (**Modellierung unterhalb der Implementierungs-Schnittstelle**, Abb. 2.1c). Während die internen Mechanismen von $M_Z$ für den **Entwickler** Implementierungsgeheimnis bleiben und in der Regel gar nicht zugänglich sind, wird ihre Kenntnis vom **Modellierer** benötigt.

**Beispiel**
Ein typisches Beispiel ist die Nachbildung von Hersteller-BS-Kernen zur Echtzeitsteuerung. Diese setzen vielfältige CPU-Zuteilungsstrategien ein (Zeitscheiben, statische/ dynamische Prioritäten, situationsabhängig präemptive/nichtpräemptive Zuteilung, Kombinationen mehrerer Strategien), die einen großen Einfluß auf die Zielgrößen der Anwendung haben können, aber selten exakt und vollständig im Systemhandbuch dokumentiert sind.
Ähnlich verhält es sich mit den internen Strategien zur Arbeitsspeicherverwaltung, die durch eine Vielzahl leistungsrelevanter Kriterien gekennzeichnet sind (/WET 84/, Kap.3):
- Auswahl wartender Prozesse bei der Zuteilung (nach Priorität oder FIFO, nach Angebot und Nachfrage, durch Reservierung, Eigenbelegung oder Fremdbelegung)
- Organisation des ASP-Vorrats und Auswahl freier ASP-Stücke
- Wiedereingliederung freigewordener ASP-Stücke in den Vorrat.

Alle in Abb. 2.1b und 2.1c dargestellten Modelle müssen, da sie **Nachbildungen** des Zielsystems sind, **kalibriert** und **validiert** werden, sobald Implementierungsergebnisse auf der Zielmaschine durch Messungen bewertbar sind. Die Validierung des Modells der Zielmaschine muß dabei aufgrund des Ein-/Ausgangsverhaltens auf der Systemdienst-Schnittstelle erfolgen, z.B. mit Hilfe von Regressionsmethoden /FER 78/; ein Vergleich der internen Modell- und Zielsystem-Strukturen würde das Implementierungsgeheimnis brechen. Die Validierung des Zeitverhaltens einer verteilten Zielmaschine, z.B. der Antwortzeiten des Kommunikationssystems, setzt ein statistisch genau bekanntes Benutzerverhalten (Lastprofil) voraus. In jedem Fall werden aber

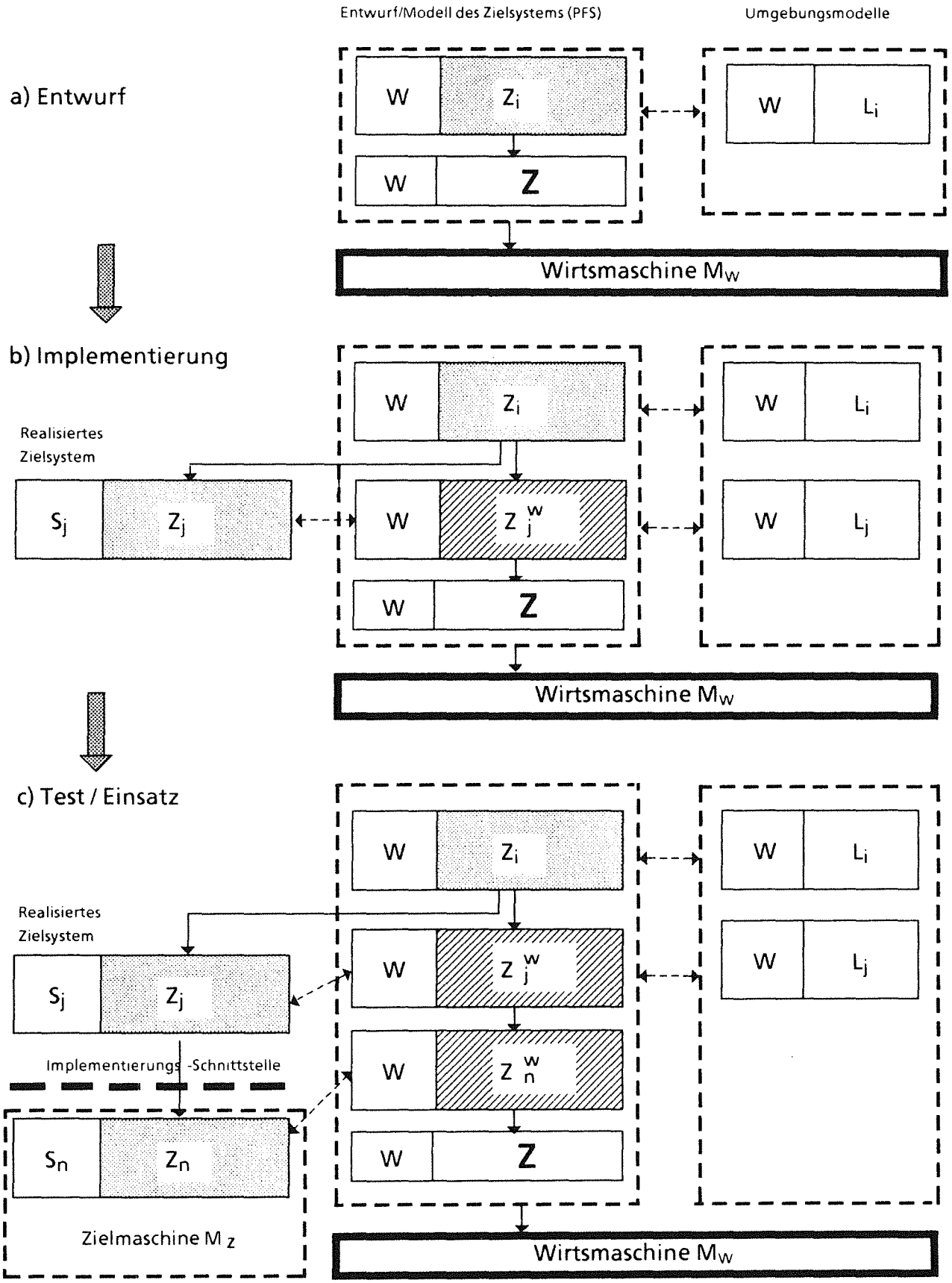



**Abb.** 2.1 Wirtsorientierte Simulation (Parallel-Ansatz)

weiterhin die kalibrierten Modelle des PFS an dem simulierten technischen Prozeß getestet - der volle "Informationsgehalt" des realen PFS und seiner Zielmaschine wird trotzdem nicht ausgenutzt.

### 2.3.3 Zielorientierte Leistungsbewertung (Integrierte Simulation)

Wie in 1.3, wird im folgenden angenommen:

(i) Die Zielmaschinen unterstützen neben der Zuteilung der physikalischen Rechner-Betriebsmittel zumindest die Verwaltung von Rechenprozessen und deren Kommunikation und Synchronisation;

(ii) Die zu entwickelnden Zielsysteme sind Echtzeit**anwendungen**; ihre Implementierungs-Schnittstelle liege **auf** oder **oberhalb** der Zielmaschine; d.h. BS-Kern oder Hardware werden für die Anwendung nicht selbst entwickelt.

Der zielorientierte Simulationsansatz basiert auf folgender **Grundidee**: Statt mit Hilfe einer gegebenen Simulationsmaschine $M_W$ beliebige Zielmaschinen $M_Z$ **nachzubilden**, und auf dieser nachgebildeten Zielmaschine laufende Echtzeitanwendungen zu bewerten, werden **alle Zielmaschinen** $M_Z$ von vornherein **funktionell** so **erweitert** ($M_Z \rightarrow M'_Z$), daß sie zugleich Simulationsmaschinen und damit zur Leistungsvorhersage tauglich sind (Abb. 2.2).
Wesentlich hinzu kommt gegenüber $M_Z$ der Teilbereich virtuelle Zeitführung ($M_V$ in Abb. 2.2, vgl. Punkt (1) in 2.3.1). Virtuelle Zeit als Erweiterung kommunizierender Prozesse (Annahme (i)) führt naheliegenderweise, allerdings nicht zwingend, auf einen prozeßorientierten Simulationsansatz /FRA 77/.

Wichtig an diesem Ansatz sind drei Aspekte.

(A) Wenn eine allgemeingültige Simulation vorgesehen wird ((T8) in 1.3) - und bereits sehr wenige neue Dienste genügen hierfür - so können komfortable Modellierungssysteme mit vergleichbarem Aufwand wie im wirtsorientierten Ansatz aufgebaut werden (Abb. 2.2a). Solche, für $M_W$ geschriebene Modellierungsanwendungen benutzen $M'_Z$ ausschließlich als Simulationsmaschine. Die zielmaschinenspezifischen Eigenschaften ($M_Z$) sind für die Modelle irrelevant, etwa so wie für Simulationsmodelle in SIMULA die Eigenschaften des unterliegenden (Großrechner-)Betriebssystems.

(B) Modellkomponenten, die speziell PFS-Entwürfe, also Softwarekomponenten, repräsentieren können bis zur Implementierungs-Schnittstelle ihrer zukünftigen Zielmaschine $M_Z$ verfeinert und unter $M'_Z$ simulativ bewertet werden, wobei der **volle** Dienstumfang, den $M_Z$ für Realzeitanwendungen zur Verfügung stellt, auch als **Modellierungssprache** benutzt wird (Abb. 2.2b). Genau dies wird durch die Erweiterung von $M_Z$ zu $M'_Z$ erreicht.

(C) Wenn $M'_Z$ auch Zielmaschine für den Realzeiteinsatz bleibt, können vollständig implementierte PFSe unter (B) in ihrem Echtzeitverhalten und realen Betriebsmittel-Verbrauch

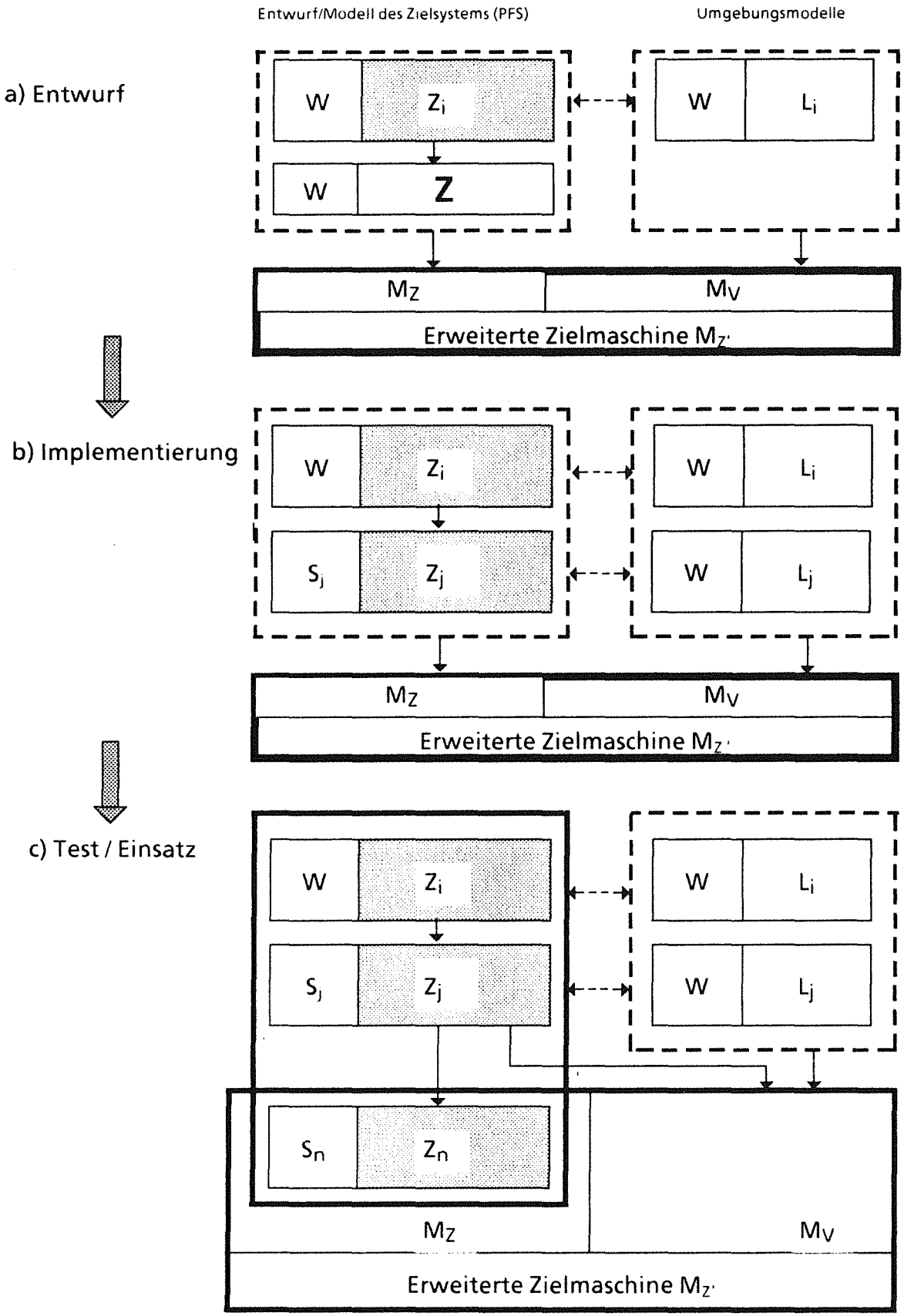



**Abb.** 2.2 Zielorientierte Simulation

bewertet werden. Das Modell **Z** zur Repräsentation nicht implementierter Dienste und ihres BM-Verbrauchs entfällt (Abb. 2.2c).

Der zielorientierte Simulationsansatz bietet gegenüber dem Stufen- oder Parallelansatz folgende Vorteile:

- Eine Nachbildung des Zielsystems (PFS), insbesondere seiner Implementierung, erübrigt sich, da jede vom Entwickler gewählte Entwurfs- oder Implementierungssprache wegen (B) zugleich eine zulässige Simulationssprache ist.

  Die globalen Zielgrößen des technischen Prozesses bilden einen einheitlichen Maßstab zur Beurteilung aller Entwicklungsstadien und -alternativen; die Umgebungsmodelle sind nur einmal zu implementieren.
- HW- und SW- Modelle werden innerhalb ihrer simulierten Umgebung durch reale Testobjekte **ersetzt,** und nicht nur durch verfeinerte bzw. kalibrierte Modelle; alle übrigen Einflußfaktoren im Wirkungskreis bleiben unverändert.

  Die Reihenfolge der Ersetzungsschritte ist dabei im Prinzip beliebig.
- Da nun explizit kontrolliert werden kann, welche Teile des Wirkungskreises entsprechend ihrer realen zeitlichen Dynamik zu den Ergebnissen beitragen und welche als simulierte Umgebung einer virtuellen Zeit unterliegen, wird das Interferenzproblem aus 2.2.2.1 entschärft. Allerdings muß dazu konstruktiv gewährleistet werden, daß die Echtzeitdynamik des realen Testobjektes selbst von der simulierten auf die Echtzeitumgebung übertragbar ist ((T6) in 1.3).

  Damit eröffnen sich dann weitere neue Anwendungen der integrierten Simulation: interferenzarme debugging-Methoden für verteilte Echtzeitsysteme (vgl. 3.6.3).

Der Unterschied läßt sich auch so formulieren: die meisten Modellierungssprachen bieten einen "semantikfreien" Ansatz. Wofür ein Simulations-Prozeß steht - für einen technischen Prozeß, eine Bedienstation, einen menschlichen Benutzer, Hardware-Baustein oder SW-Prozeß -, ist weder für den Modellierer noch für das Modellierungssystem von Belang. Der zielorientierte Simulationsansatz enthält zwar diese Sichtweise als Möglichkeit **auch**, erlaubt aber zusätzlich die Tatsache auszunutzen, daß gewisse Modelle DV-Systeme repräsentieren und ihre Simulationsmaschine zugleich Zielmaschine sein kann.

## 2.4 Diskussion verwandter Systemansätze

Nach der Diskussion der methodischen Aspekte sollen noch einige konkrete Systeme zur experimentellen Leistungsbewertung diskutiert werden.

Die folgenden, in den USA entwickelten Ansätze sind unter dem Oberbegriff **Testbettsystem** /FWB 82/ bekannt. Darunter wird üblicherweise verstanden (eine präzise und einheitliche Definitione dieses Begriffes fehlt leider):

- Die Zielsysteme bzw. Testobjekte sind vollständige DV-Anwendungen, und nicht nur Subsysteme wie Betriebssysteme, E/A-Geräte oder Kommunikationsprotokolle. Ihre

**Leistungsbewertung** anhand klar definierter Leistungsgrößen ist ein Hauptziel der Modellierung, und nicht nur die Erprobung funktioneller Eigenschaften (prototyping).

- Gegenstand der Bewertung ist (auch) die Realisierungsphase, und nicht nur Entwurf/ Spezifikation der Testobjekte.
- Das Experimentiersystem ist durch generalisierte Schnittstellen oder Dienste, etwa für die Leistungsgrößenmessung und -auswertung, prinzipiell für mehr als eine einzige Anwendung geeignet.

Die Ansätze werden anhand des folgenden Kriterienkatalogs verglichen (vgl. Tab. 2.1). Darin sind die in Kap. 1.3 genannten zentralen Kriterien der allgemeingültigen Simulation, Erweiterungseigenschaft und Interferenzfreiheit mit enthalten.

**Kriterium 1: Entwurfsunterstützung**

Hier geht es darum, ob das **Entwurfsstadium** der Testobjekte gezielt unterstützt wird (z.B. durch Entwurfs- oder prototyping-Sprachen, durch höhere Kommunikationsdienste, durch die Möglichkeit, unvollständige Programme zu testen) und ob für die **Modellierung** besondere Hilfen zur Verfügung stehen, z.B. Modellierungssysteme.

**Kriterium 2: Allgemeingültigkeit der Simulation** - vgl. (T8) in 1.3

**Kriterium 3: Implementierungsnahe Leistungsvorhersage**

Als sehr wesentliches Kriterium in der Diskussion von 2.2.2.2, 2.3 hat sich herausgestellt, ob eine Leistungsvorhersage mit virtueller Zeit nahtlos bis zum Implementierungs- und Teststadium auf der Zielmaschine durchzuhalten ist und solche Modelle dort mit realen Testobjekten kooperieren/um Rechnerbetriebsmittel konkurrieren können.

**Kriterium 4: Erweiterungseigenschaft** - vgl. (T7) in 1.3

**Kriterium 5: Verteilte Experimente**

Dies bedeutet, daß nicht nur das Testobjekt im Einsatz ein verteiltes DV-System ist, sondern auch seine Leistungsbewertung (Simulation, Messung) selbst verteilt abläuft.

**Kriterium 6: Interferenzarmut**- vgl. (T6) in 1.3

**Kriterium 7: HW-Abhängigkeit**

Testbettsysteme, die dedizierte HW-Komponenten für die Leistungsmessung erfordern, bzw. zumindest entscheidend davon profitieren, seien durch das Merkmal 'H' bzw. 'HV' gekennzeichnet; solche, die Standardkomponenten für die Rechner- und Kommunikationss-HW verwenden, durch ein 'S'. In HW-Testbettsystemen kann der Entwickler bei schlechtem Leistungsverhalten seiner Anwendung nicht ohne weiteres die Zielmaschine wechseln, wenn er die Dienste des Testbetts zur Leistungsbewertung weiterhin nutzen möchte. Aus Anwendersicht hat dieses Kriterium ähnliche Bedeutung wie die Erweiterungseigenschaft.

In den nächsten beiden Kriterien werden wichtige Architekturmerkmale des Testbetts berücksichtigt.

**Kriterium 8: Zielmaschinen-Integration**

Dieses Kriterium trifft zu, wenn die Laufzeitmaschine (Rechner-, Prozeßperiherie und Kommunikations-HW, BS-Kern) für das Testobjekt zugleich Laufzeitmaschine für dessen Simulation und Leistungsvorhersage ist.

**Kriterium 9: Zeitführungskonzept**

Hier geht es um die Zeit, auf die sich die zeitabhängigen Leistungsgrößen beziehen. Folgende Merkmale sind möglich:

- E Echtzeit oder skalierte Echtzeit (Meß-, Echtzeitsimulationssystem)
- S Simulationszeit (ausschließlich durch das Modell gesteuert)
- SE Simulationszeit, die aber z.Teil durch Zeitmessungen während des Simulationsexperiments beeinflußt wird
- H hybride Zeit, d.h. Simulationszeit, die in gewissen Zeitintervallen echtzeitsynchron ist (vgl. 3.4.3.3).

Bei Modellen, die aus autonomen, nebenläufigen kooperierenden Komponenten bestehen, ist noch der Grad ihrer **zeitlichen Autonomie** wichtig (vgl. Kap. 5):

- Z zentrale, einheitliche Uhr für alle Komponenten
- VE eigene Simulationsuhr für jede Komponente, Uhren synchonisiert
- VL eigene Simulationsuhr für jede Komponente , Uhren asynchron

Bei echtzeitorientierten Systemen sind nur die Merkmale Z oder VE möglich.

Schließlich wird noch kurz auf die Anwendungsaspekte eingegangen.

**Kriterium 10: Anwendungsschwerpunkt (Zielsysteme)**

Als Beispiele sind Informationssysteme, Bürokommunikation, allgemeine Netzwerkanwendungen, Prozeßführungssysteme u.a. zu nennen.

**Kriterium 11: Entwurfs-/Implementierungssprachen**

Hiermit sind die Entwicklungssprachen für die zu entwickelnden Anwendungen gemeint, nicht die des Testbettsystems selbst.

**Diskussion der Ansätze**

**PPML**

Die Entwurfsmethode von J.W. Sanguinetti /SAN 77//SAN 79/ ist deshalb von Bedeutung, weil sie die erste und bis heute einzige Arbeit darstellt, in der ein zielorientierter Simulationsansatz nach 2.3.3 verfolgt wurde. Auf der Basis einer gemeinsamen Basismaschine für Echtzeitausführung und Simulation soll ein Zielsystem-Modell schrittweise zu einem realen Zielsystem verfeinert werden ('the simulator is to become the system being simulated'). Kernstück ist die auf

kooperierenden sequentiellen Prozessen basierende Sprache PPML*), die nur wenige Konstrukte enthält:

- asynchrones Senden (SEND) und blockierendes Empfangen (RECEIVE);
- sequentielle Kontrollstrukturen: 2 bedingte Anweisungen, wobei die Bedingung entweder vom Inhalt einer Nachricht (UNLESS) oder von einer booleschen Zufallsgröße (IF) abhängig ist, Sprunganweisung (GOTO)
- eine sequentielle Anweisung, die ein zu entwickelndes Codestück, das als Ergebnis eine Nachricht msg liefern soll, durch einen abstrakten Zeitbedarf von T Zeiteinheiten repräsentiert (SET <msg> SIM T).

Die "Simulationsanweisungen" IF und SET werden später durch Code einer sequentiellen Zielsprache ersetzt.
Aufgabe der PPML-Laufzeitmaschine ist es, die "echten" Anweisungen SEND, RECEIVE, UNLESS, GOTO sowie alle Zielsprachenanweisungen auszuführen und die "Ersatzanweisungen" (IF, SET) zu simulieren. PPML führt für jeden Prozeß eine eigene Simulationsuhr, die bei allen Anweisungen außer SET echtzeit-synchron ist und bei Ausführung eines SETmsgSIMT um den Betrag T inkrementiert wird. Die PPML-Prozesse können auf einer Einprozessor- oder simulierten Mehrprozessor-Anlage nach einem recht einfachen Verfahren (vgl. 3.6.1) ausgeführt werden.
Dieses Verfahren funktioniert aber nur, wenn jeder PPML-Prozeß ununterbrochen den Prozessor belegt, bis er sich selbst durch ein RECEIVE blockiert (Koroutinenmodell); ein Prozeß kann also nicht unterbrochen werden. Jeder PPML-Prozeß repräsentiert einen Software-Prozeß für einen PPML-Prozessor. Die Möglichkeit, eine externe Umgebung in virtueller Zeit zu simulieren, fehlt.

An die Entwicklung **einsatzfähiger**, effizienter Zielsysteme in der kargen, heute eher altertümlich wirkenden Sprache PPML ist allerdings auch kaum zu denken. Die Vorteile des zielorientierten Ansatzes gegenüber dem wirtsorientierten, insbesondere Echtzeitbewertung, gehen aber verloren, sobald der Entwickler zu einer **anderen** Basismaschine bzw. -sprache Z' mit anderen Formen der Basiskommunikation und -synchronisation oder Prozeßzuteilung als PPML übergehen will. PPML stellt zwar **eine** Maschine für Ausführung und Simulation zur Verfügung, aber keine **Vorschrift**, um beliebige Zielmaschinen so zu erweitern, daß sie alle dieselben Schnittstellen zur Simulation (IF, SET) besitzen. Man vergleiche die Diskussion in 3.6.1, auf welche Zielmaschinen-Eigenschaften es hier insbesondere ankommt.

**S.L. Ward**

In der Arbeit /WAR 79/ wird ein Hersteller-BS-Kern (RSX-11M) neben der Echtzeitsteuerung zur Simulation in virtueller Zeit eingesetzt. Dabei entstehen zwei algorithmisch fast identische, sich

*) Die PPML-Sprache wurde ursprünglich konzipiert, um das Kommunikationsverhalten von Prozessen **analysieren** zu können ('message transmission expressions'), was sich jedoch nur für sehr kleine Beispiele als praktikabel erwies.

nur in der Zeitführung unterscheidende BS-Versionen. Da keine Ausführung von simulierten und Echtzeitanwendungen im selben Experiment vorgesehen wird, bleibt /WAR 79/ in den technischen Möglichkeiten hinter /SAN 77/ zurück.

**JADE**

Im Rahmen eines umfangreichen, 1983 begonnenen und noch laufenden Forschungsvorhabens an der University of Calgary wird eine Entwicklungs- und Testumgebung für verteilte Anwendungssysteme geschaffen /WBC 83//UBC 84//JLS 87/. Schwerpunkte sind die Spezifikation, Prototyping, Implementierung, und Simulation, sowie die Entwicklung leistungsfähiger Werkzeug-Schnittstellen und Benutzeroberflächen. Als zentrales Bindeglied dient ein Kommunikationsprotokoll (JIPC), über das alle verteilten Anwendungen unter JADE (Simulationsmodelle, Zielsysteme, Hilfssysteme) in den Sprachen C, ADA, SIMULA oder PROLOG kommunizieren. In JADE gibt es 3 Klassen von Laufzeitsystemen:

a) die realen Zielmaschinen (z.B. UNIX), auf denen JIPC und die Anwenderprozesse implementiert sind
b) eine Laufzeitumgebung zur verteilten Simulation nach dem Time Warp Algorithmus /JES 83/, /CLU 85/.
c) einen zentralen Simulations-Controller (JEMS) /UBC 84/ /LOU 85/, der eine eingeschränkte Kooperation 'simulierter' und 'realer' Anwendungen im selben Experiment erlaubt. Hierzu wird eine spezielle Version von JIPC benutzt, welche netzweit alle zeitgestempelten JIPC-Botschaften über den zentralen Controller leitet. Bei realen Anwendungen werden die Zeitstempel durch Messung der Zeitspannen zwischen JIPC-Aufrufen auf ihren Zielrechnern gewonnen. Die zeitlich chronologische Ausführungsreihenfolge wird durch den Controller ähnlich wie in /SAN 79/ erreicht.

In /HAU 88/ wird ein Simulations-Testbett und eine zugehörige Anwendung (DSPT, 'distributed simulation and prototyping testbed') auf der Grundlage von JIPC entworfen, wobei diese Laufzeitsysteme in verschiedenen Entwicklungsstadien genutzt werden sollen.
Festzuhalten bleibt

- die grundlegenden Restriktionen sind i.w. die gleichen wie in /SAN 77/ (Koroutinenmodell).
- die Zuteilungsentscheidungen, welcher Anwenderprozeß als nächster rechnet, werden aus den lokalen Zielmaschinen heraus in den zentralen Simulations-Controller verlagert. Die Wechselwirkungen zwischen lokaler Prozeßkommunikation, lokaler CPU-, ASP- und E/A-Zuteilung mehrerer Prozesse derselben Zielmaschine im realen Einsatz werden dadurch zerstört. Das gemessene Zeitverhalten der realen Testobjekte ist also von geringem Aussagewert.
- architektonisch handelt es sich nicht um eine (zielmaschinen)integrierte, sondern eine **aufgesetzte** Simulation. Die Integration der realen Testobjekte funktioniert auch nur sehr

indirekt: aus Gründen der Schnittstellenkonsistenz müssen z.B. die realen Anwendungen innerhalb von JEMS nochmals durch "Surrogat-Prozesse" repliziert werden.

**Single Step Simulation**

In /KOH 81/ und /KOH 82/ wurde versucht, speziell das in 2.2.2.1 diskutierte Interferenzproblem zwischen einem realen PFS (zentraler Prozeßrechner) und einem in Echtzeit auf einem separaten Modellrechner simulierten technischen Prozeß zu lösen. Es wurde eine hybride Zeitführung vorgeschlagen, die in die Sende- und Empfangsroutinen (Unterbrechungsroutinen) für den Prozeßdatenaustausch integriert ist: bei jeder Aktivierung des Prozeßmodells durch eine Meßwertanforderung oder Stellwertausgabe werden die Echtzeituhren auf beiden Rechnern angehalten und die Ausführung des realen Testobjekts suspendiert, bis die i.a. rechenzeitaufwendigen Prozeßmodellrechnungen beendet sind. Hierdurch bleibt der Zeitbedarf der Modellrechnungen sowohl aus Sicht der Experimentauswertung, als als auch des PFS selbst, transparent.

**DST**

Eine sehr ähnliche Zeitführung verwendet auch das am Honeywell Computer Sciences Center entwickelte Distributed System Testbed (DST) /FBW 82//BES 82/ zum Leistungstest von Rechnernetz-Protokollen, und sein Vorläufer CHIMPNET /KFJ 79/. Der simulierten Umgebung entsprechen hier Programme zur Leistungsgrößenerfassung und Lastgeneratoren (Instrumentierungs-Software).
Bemerkenswert ist, daß der BS-Kern der Experimentrechner unterschiedliche Prozeßklassen für reales Testobjekt und Instrumentierungs-SW, sowie deren gegenseitige Beeinflussung durch Ereignisvariable vorsieht. Ein hierarchischer Entwurf, in dem Simulationsmodelle schrittweise zu realen Testobjekten verfeinert werden, wird nicht verfolgt. Bei DST handelt es sich um eine **dedizierte** Testbettumgebung, die i.w. **nur** interferenzarme Leistungsmessung unterstützt, was u.a. an folgenden Punkten ersichtlich ist:

- DST ist ein Hardware-Testbett. Es setzt eine dedizierte HW-Konfiguration mit einem zentralen Instrumentierungsrechner voraus, der zentral die Echtzeituhren aller anderen Rechner steuert, und einen Instrumentierungsbus mit spezieller Schnittstellen-Logik.
- Bei dem BS-Kern der DST-Experimentrechner handelt es sich um eine Speziallösung, nicht um eine allgemeine Vorschrift zur Erweiterung gegebener Echtzeit-BS-Kerne. Hierzu müßten z.B. zusätzlich die Interferenzprobleme infolge der Konkurrenz um endlichen Arbeitsspeicher oder um E/A-Geräte, wie bei virtuellem Speicher, untersucht werden.
- Die Systemdienste unterstützen keine allgemeine discrete-event-Simulation. U.a. fehlen eine zeitüberwachte Botschaftenkommunikation und eine von der Echtzeituhr unabhängige Simulationsuhr.

**DESIGN**

So heißt eine Entwicklungsumgebung für Netzwerkanwendungen /MUE 86//MUE 84//MUE 88/, die auf der verteilten Programmiersprache DC aufbaut. DC ist eine Erweiterung von C um ein hierarchisches Prozeßkonzept, sog. Funktionseinheiten, und sieht u.a. Dienste für deren Verwaltung und Kommunikation über ports vor. Mit Hilfe einer universellen Werkzeug-Schnittstelle kann sowohl eine reale DC-Anwendung schrittweise entworfen, getestet und installiert werden, als auch ein Simulationsmodell derselben weitgehend automatisch erstellt und verteilt ausgeführt werden. Die wesentlichen Unterschiede in der Vorgehensweise zur vorliegenden Arbeit sind:

- die Festlegung auf **eine** Sprache (DC) als Basismaschine (deren u.U. komplexe Implementierungen auf unterschiedlichen Zielmaschinen nicht in die Leistungsbewertung eingehen)
- die Verwendung getrennter Laufzeitumgebungen, in denen entweder nur Echtzeitausführung, Messung bzw. Echtzeitsimulation ("1:1"-Simulation MUE 84/), oder nur verteilte Simulation möglich sind.

**ADL/ADS/AOS**

Dieses Projekt /SMD 83//ELL 83/ zählt zu den wenigen Testbettsystemen, die gezielt das **Entwurfsstadium** von **Realzeitanwendungen** unterstützen. Die funktionellen und algorithmischen Aspekte werden durch eine Spezifikationssprache (BPL, /HOO 84/) beschrieben, die statische Komponentenstruktur (Mengengerüst) und ihre Allokation auf mögliche HW-Konfigurationen durch eine graphische Konfigurationssprache (ADL). Mit Hilfe eines Entwicklungssystems (ADS) kann eine virtuelle Rechnerumgebung (AOS) für die Anwendung erzeugt und der Prototyp (Anwendung + AOS) auf einem Rechnernetz von 8 gekoppelten VAX11-780 getestet werden. Als sehr kritisch ist das Fehlen eines virtuellen Zeitkonzeptes anzusehen. Das ganze System wird in **Echtzeit** unter VMS als Wirts-BS bewertet; damit ist weder eine **Vorhersage** des Zeitverhaltens unter AOS, noch eine aussagekräftige Messung des tatsächlichen Zeitverhaltens (unter VAX oder einer anderen Zielmaschine) möglich.

**White** und **Paulk** /WHP 85/ untersuchen den Einsatz von time-sharing-Systemen, z.B. Großrechnernetzen, für Realzeit-Experimente, um deren komfortable Entwicklungsumgebung nutzen zu können. Neben einem Algorithmus zur Uhrensynchronisation werden in /WHP 85/ verschiedene Maßnahmen gefordert, um die aus der BM-Zuteilung der Wirtsrechner-BSe resultierenden Störeinflüsse (Seitentauschalgorithmen, E/A-Verkehr, Zeitscheibenzuteilung) auszuschalten, und andererseits zur Experimentsteuerung benötigte, aber nicht vorhandene Realzeit-BS-Funktionen in die Anwendung zu verlagern. Dies widerspricht der integrierten Vorgehensweise diametral (vgl. 3.6.1).

Hauptanwendung des von **Knight** und **Gregory** /KNG 84/ beschriebenen Testbettsystems ist nicht die Leistungsanalyse, sondern der Test von Fehlererholungsstrategien in verteilten Systemen (Zielsprache ADA) unter kontrollierten Ausfallbedingungen. Es können beliebige virtuelle Rechnerkonfigurationen und Verbindungstopologien, Komponentenausfälle und -wiederanläufe simuliert werden. Erwähnenswert ist eine verteilte, lose gekoppelte Zeitführung, die aber auf die spezielle Anwendung zugeschnitten (nicht allgemeingültig) ist. Bei den übrigen untersuchten, hier aber nicht näher diskutierten Testbettansätzen handelt es sich entweder um reine Echtzeitsysteme /NUT 83//BRL 82//KAN 84//HAH 87//KLM 89/ bzw. echtzeit-skalierte Zeitführung (ISSOS /SKR 85/), oder um rein virtuelle Zeitführung in zentralen Laufzeitumgebungen /CAV 83//FOJ 85/; teils wird über das Zeitkonzept überhaupt nichts ausgesagt /RAL 87/.

**Zusammenfassung**

Brauchbare (komplementäre) Ansätze zum Aufbau einer integrierten Simulation finden sich eher in den älteren Arbeiten /FBW 82/, /KOH 82/, bzw. /SAN 79/, /LOU 85/. In den neueren Testbettansätzen ist zwar die Tendenz zu mächtigeren (Entwurfs- und Konfigurations)Sprachen, Entwicklungssystemen und grafischen Benutzeroberflächen erkennbar, aber in den Laufzeitsystem-Architekturen ist eher ein Rückschritt zu reinen Simulations- oder reinen Echtzeitsystemen zu beobachten.
Es fällt auf, daß die genannten Arbeiten weder untereinander noch in späteren Arbeiten allzu große Resonanz gefunden haben. Wichtige Gründe hierfür scheinen zu sein:

- Die Simulationsfunktionen sind zu stark spezialisiert und enthalten nicht die Fähigkeiten bekannter zeitdiskreter Simulationssprachen als Teilmenge. Damit ist die Leistungsvorhersage von DV-Systemen im Entwurfsstadium und die Simulation ihrer Umgebung sehr erschwert.
- Die Vorteile der Integration realer Testobjekte gegenüber einer reinen Simulation oder einer Messung entfallen, weil aufgrund starker Einschränkungen keine realistischen Zielsysteme innerhalb des Ansatzes entwickelt werden können, oder die Experimentergebnisse nicht auf den Einsatz übertragbar sind.
- Leistungsfähige Verfahren zur verteilten Simulation, die die integrierte Simulation erst auf verteilte Zielsysteme anwendbar machen, sind erst nach 1983 entstanden.

**Tab. 2.1** Vergleichstabelle einiger Testbettansätze ("-" : Kriterium unzutreffend, z.B. weil kein reales Testobjekt oder keine simulierte Umgebung vorhanden; "?" : keine Angaben in der Literatur)

| Kriterium | PPML | JADE | DST | Single Step | Ward | DESIGN | ADL/ADS | White-Paulk | Knight-Gregory |
|---|---|---|---|---|---|---|---|---|---|
| 1. Entwurfsunterstützung | | | | | | | | | |
| - Zielsystem | J | J | N | N | N | J | J | ? | N |
| - Modellierung | N | J | N | J | N | J | J | ? | J |
| 2. Allgemeingültigkeit Simulation | N | J | N | N | N | N | - | - | N |
| 3. Implementierungsnahe Leistungsvorhersage | J | N | N | N | J | J | N | ? | N |
| 4. Erweiterungs-eigenschaft | N | N | N | N | N | N | - | - | - |
| 5. Verteilte Experimente | N | J | J | N | N | J | J | J | J |
| 6. Interferenzarmut | - | N | (J) | (J) | - | - | N! | N! | - |
| 7. HW-Abhängigkeit | S | S | H | HV | S | S | S | S | S |
| 8. Zielmaschinen-Integration | J | N | J | N | J | N | N | - | - |
| 9. Zeitführung | | | | | | | | | |
| - Echtzeit-Bezug | H | SE | H | H | S od.E | S od. E | E | E | S |
| - zeitliche Autonomie | VL | VL | Z | VE | Z | VL | VE | VE | VL |
| 10. Anwendungs-schwerpunkt | BSe, Informa-tionssyst. | Netzwerk-Anwendungen (E-mail), Transaktions-Systeme, grafische Anwendungen | LAN-Transport-Protokolle (z.B. ALOHA) | zeitkritische digitale Regelungen | PFSe | Netzwerk-Anwendungen | Echtzeit-Anwendungen Netzwerk-BS-Ftnen | Echtzeit-Anwendungen | Fehlertolerante verteilte Anwen-dungen |
| 11. Entwurfs-/Implemen-tierungs-Sprache(n) | PPML | C, SIMULA, PROLOG; ADA | ? | PASCAL, PDP11-Ass. | FORTRAN Makro-Ass. | DC | BPL + ADL, PASCAL, VMS-Dienste | ? | ADA |
| 12. HW-Konfig. | ? | VAX, SUN XEROX 1108 | M6809 | PDP11-34 | PDP11 | VAX11-780 | VAX11-780 (P11-C-Bus) | VAX11-780 | VAX11-780 IBM-PC |

## 3. Gesamtkonzept einer Experimentumgebung zur integrierten Leistungsbewertung

In diesem Kapitel wird das zielorientierte Simulationskonzept und seine verteilte Laufzeitumgebung dafür entworfen.
Zuerst klären wir einige häufig benutzte Begriffe.

**Testobjekt** bezeichnet das Gesamtsystem, dessen funktionelles und zeitliches Verhalten experimentell untersucht/geprüft werden soll. Bei Echtzeitsystemen als wichtigster Anwendung ist dies der geschlossene Wirkungskreis aus Prozeßführungssystem (PFS) und technischem Prozeß (TP). Beide werden in 3.1 mit ihren Leistungsgrößen grob charakterisiert. Ein anderes Beispiel wären Teilnehmersysteme (z.B. Buchungssysteme) zusammen mit einem Lastmodell ihrer künftigen Benutzer.

**Zielsystem** ist der Teil des Testobjektes, der **entwickelt**, also in seinem Verhalten explizit beeinflußt wird (PFS), der Rest, seine **Umwelt**, sei vorgegeben (TP). In 3.2 werden die wichtigsten Entwicklungsstadien für PFSe als Zielsysteme und die zugehörigen Experimentziele skizziert.

**Reales Testobjekt** ist der Teil des Testobjektes, der in einem konkreten Experiment nach seinem Echtzeitverhalten bewertet wird, seine **simulierte Umgebung** der Rest, dessen Zeitverhalten durch eine virtuelle (künstliche) Zeit nachgebildet wird. Diese (von der nach Zielsystem/Umwelt i.a. verschiedene) Unterteilung ist typisch für die integrierte Simulation. Die notwendigen Konzepte hierfür werden in 3.3 entwickelt.

Als **Test-** (oder **Experiment-**)**Umgebung** werden alle HW- und SW-Komponenten bezeichnet, die nicht zum Testobjekt gehören, aber zu dessen experimenteller Untersuchung notwendig sind, also u.a. die Laufzeitumgebung (Betriebssystem) für die Simulation, die Komponenten zur Erfassung, Auswertung und Darstellung der Leistungsgrößen und das übergeordnete Entwicklungs- und Leitsystem (3.4, 3.5).

### 3.1 Charakterisierung des Testobjektes

Der geschlossene Wirkungskreis besteht aus folgenden Komponenten

| | |
|---|---|
| TP_1,...,TP_k | Komponenten des technischen Prozesses (einschließlich Stellgliedern, Leistungselektronik, Analog-Regelkreisen, Abtast- und Haltegliedern etc.) |
| INT_1,...,INT_l | Prozeßperipherie-Komponenten (Analog-Eingabe AI, Analog-Ausgabe AO, Digital-Eingabe DI, Digital-Ausgabe DO) |
| PFS_1,...,PFS_m | Leittechnische Komponenten (Erfassung, Zustandsschätzung, Steuerung, Regelung, ggf.Bediener-Schnittstelle). |

Alle Komponenten können als I/O-Funktionszuordnungen bzw. I/O-Systeme nach Def. (2.1),(2.2) angesehen werden, deren Verknüpfung und zeitliche Interaktionen in Abb. (3.1) charakterisiert werden.

**Klassen von Zeitfunktionen**

| | |
|---|---|
| $(X,T)^d$ | differenzierbare Funktionen (kontinuierlicher Zeitbereich T) |
| $(X,T)^c$ | sonstige zeitkontinuierliche Funktionen |
| $(X,T)^{sc}$ | stückweise konstante Funktionen (Treppenfunktionen) |
| $(X,T)^e$ | Ereignisfunktionen ($x(t) <> \sigma$ nur an endlich vielen Stellen) |

**Ein-/Ausgabe-Schnittstellen (Zeitfunktionen):**

| | |
|---|---|
| $\xi_i \in (XS,T)^c$ | $F = \mathbf{R}^s$ Störgrößen (Umwelteinflüsse) auf Teilproz. i |
| $u_i \in (U,T)^c$ | $U = \mathbf{R}^m$ Stellgrößen für Teilproz. i (Strom-/Spannungseing.) |
| $z_i \in (Z,T)^d$ | $Z = \mathbf{R}^r$ Regelgrößen (Führungsgrößen) für Teilproz. i |
| $x_i \in (X,T)^d$ | $X = \mathbf{R}^n$ interne Zustandsgrößen für Teilproz. i |
| $x_{ij} \in (X,T)^d$ | Kopplungsgrößen von TP_i an TP_j |
| $y_i \in (Y,T)^d$ | $Y = \mathbf{R}^p$ direkt meßbare Größen von Teilproz. i |
| $uu_i \in (UU,T)^{sc}$ | zeit- und wertdiskrete Stellgrößen von PFS_i ($UU = [-2^n,2^n]$ (AO) oder $UU = \{0,1\}$ (DO)) |
| $yr_i \in (AM,T)^e$ | Meßwertanforderungen für INT_i (AI,DI) (AM Menge der analogen und binären Meßstellen) |
| $yy_i \in (YY,T)^{sc}$ | zeit- und wertdiskrete Meßgrößen von INT_i ($YY = [-2^n,2^n]$ (AI) oder $YY = \{0,1\}$ (DI)) |
| $ya_i \in (A,T)^e$ | asynchrone Prozeßereignisse, Alarme (A Menge von Alarm- bzw. Unterbrechungseingängen) |
| $k_{ij} \in (K,T)^{sc}$ | interner Informationsaustausch PFS_i <-> PFS_j |
| $s_i \in (S,T)^{sc}$ | externe Sollvorgaben an PFS (Bedienereingaben) |
| $b_i \in (B,T)^{sc}$ | Bedieneranzeigen (K, S, B beliebig strukturierte diskrete Wertebereiche) |

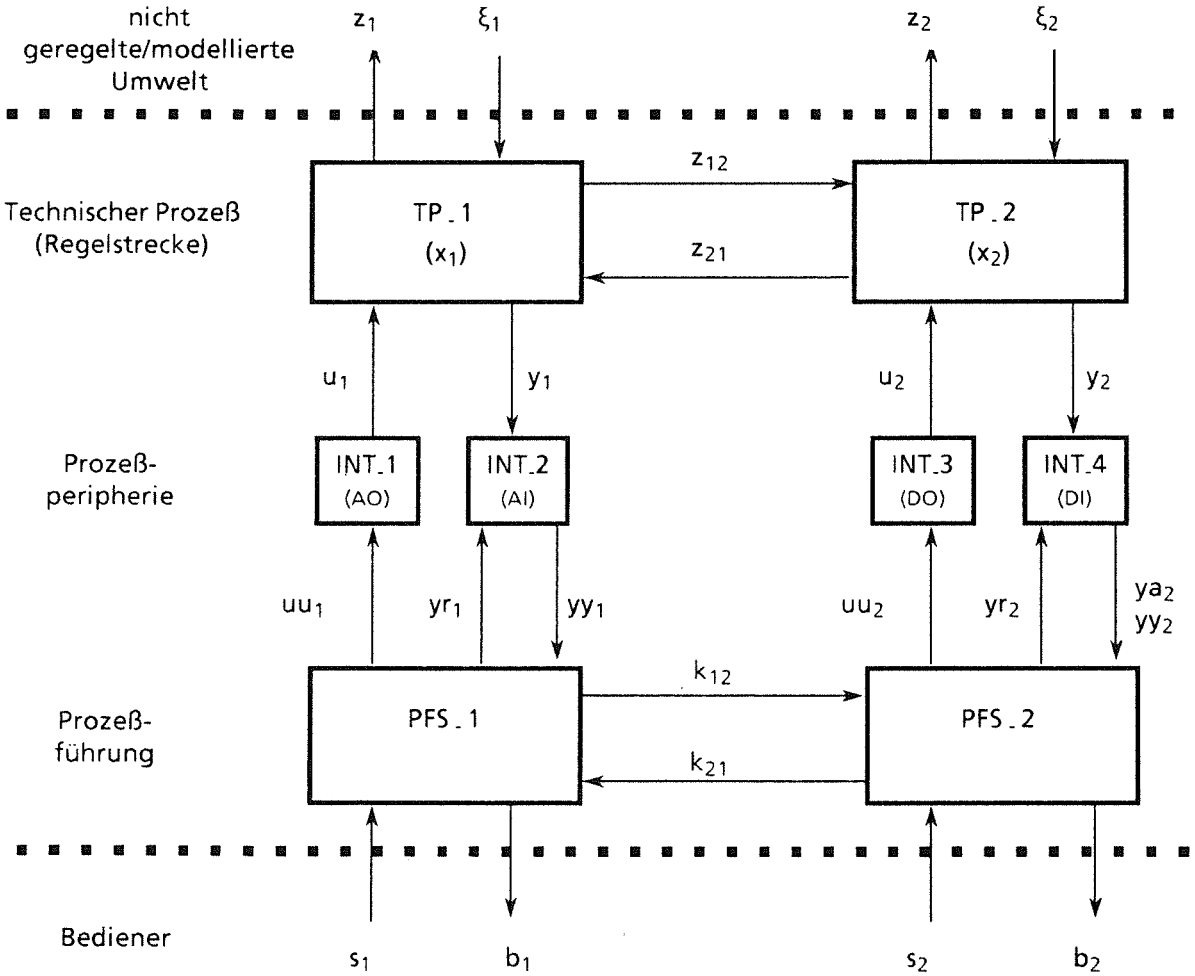



**Abb. 3.1** Komponenten eines geschlossenen Wirkungskreises und ihre Verknüpfung

### 3.1.1 Technischer Prozeß

Aus Sicht des PFS stellt sich der technische Prozeß im Echtzeiteinsatz und in den Experimenten durch die Prozeßperipherie-Schnittstelle (Signale uu, yr, yy, ya in Abb 3.1) einheitlich dar. Während im Einsatz diesen Signalen analoge physikalische Größen entsprechen (Spannungs- bzw. Strompegel am Prozeßpanel), entstammen sie in den Experimenten zeitdiskreten Simulationsmodellen, die als **Prozeßelemente** (PE) bezeichnet und in Abschnitt 6.2.1 genauer beschrieben werden. I.f. werden nur ihre wichtigsten Eigenschaften zusammengefaßt.

**(1) Zerlegung in Teilprozesse**

Die Modellzerlegung in PEe entspricht der Zerlegung des realen TP (Abb. 3.1) in Teilprozesse nach geographischen und physikalischen Kriterien. Ein PE besitzt dieselben Nachbarkomponenten und Ein-/Ausgangsgrößen wie der zugehörige Teilprozeß in der Realität. Mehrere PE'e können allerdings konzeptionell zu einem PE höherer Ordnung zusammengefaßt werden (hierarchische Vergröberung/Verfeinerung).

**(2) Modell-Dynamik**

Die interne Verknüpfung der Ein-, Ausgangs- und Zustandsgrößen eines PE kann auf zwei Arten erfolgen.

**A) Kontinuierliches PE**

Der Zustandsverlauf x(t) des modellierten Teilprozesses ergibt sich als Lösung eines Systems gewöhnlicher, i.a. nichtlinearer Differentialgleichungen erster Ordnung. Beispiele sind etwa ein Elektromotor mit (Ankerwinkel, Winkelgeschwindigkeit, -beschleunigung) als Zustandsgrößen, ein Armgelenk eines Roboters oder die Bewegungsgleichungen eines Flugzeugs. Die Parameter des DGS und die Komponenten des Zustands können sich sprungförmig ändern ("diskrete Ereignisse"), d.h. der Verlauf x(t) ist i.a. nur stückweise stetig differenzierbar.

**B) Diskretes PE**

Hierbei ändern sich die Zustandsgrößen des modellierten Teilprozesses grundsätzlich nur zu diskreten Zeitpunkten. Beispiele sind etwa eine Werkzeugmaschine, ein Förderband oder sonstige Transporteinrichtung oder ein Regallager.

Natürlich kann ein und derselbe technische Prozeß je nach dem gewünschten Detaillierungsgrad sowohl durch ein kontinuierliches als auch durch ein diskretes Prozeßelement modelliert werden.

**(3) Ein-/Ausgabe-Zeitfunktionen**

Kontinuierliche Signalverläufe zwischen unterschiedlichen Teilprozessen (vgl. Abb. 3.1) werden ausnahmslos durch zeitdiskrete Verläufe (Treppenfunktionen) der entsprechenden PE'e angenähert. Der Stützstellenabstand ergibt sich allerdings dynamisch, nach der zeitlichen Dynamik der signifikanten Änderungen, und ist i.a. nicht äquidistant. Jedes PE ist selbst dafür

verantwortlich, signifikante Änderungen seiner Ausgangsgrößen den betroffenen Nachbar-PE'en explizit anzuzeigen (vgl. 6.2.1).

### 3.1.2 Prozeßführungs-System

#### (1) Hardware-Konfiguration

Die betrachteten Zielmaschinen für den Einsatz der PFSe sind lose über ein Rechnernetz gekoppelte Mehrrechnersysteme (Abb. 3.2). Als Kommunikationsstrukturen kommen Ringstrukturen (token ring), Punkt-zu-Punkt-Rechnerkopplungen, sternförmige Verkopplungen oder hierarchisch angeordnete, den Führungs- und Regelungsebenen entsprechende Bussysteme /FAE 84/ in Frage.
Die Knotenrechner seien Monoprozessorsysteme, bestehend aus Prozessor (meist 16 oder 32 bit Wortbreite), lokalem RAM-Speicher, EPROM, und Peripherieanschlüssen, insbesondere Netzzugangs-Controller. Daneben verfügen die direkt am TP angeschlossenen Vorort-Rechner über die Prozeß- bzw. Wandlerperipherie, die übrigen Rechner über Datenperipherie.
Notwendig ist ferner ein Entwicklungsrechner, von dem die PFS-Programme geladen (downloading) und bedient werden und der selbst Teil des PFS sein kann, z.B. ein Betriebsführungsrechner.
Bei den i.f. Kapiteln zu beschreibenden Lösungsansätzen wird von folgenden **Annahmen** ausgegangen:

- Die Interrechnerkommunikation erfolge durch Nachrichtenaustausch über das Rechnernetz, nicht über gemeinsame Speicherbereiche.
- Jeder Rechner verfüge exklusiv über eine lokale, programmierbare und lesbare Echtzeituhr für die zeitliche Ablaufsteuerung.
- Elementare Schutzmechanismen (HW-Speicherschutz, privilegierter Systemmodus) der Rechner werden vorausgesetzt.

#### (2) Funktionale und hierarchische Gliederung der Software

Bei den meisten verteilten PFSen findet man eine hierarchische Schichtenarchitektur mit folgender groben Funktionszuordnung (Abb. 3.3):

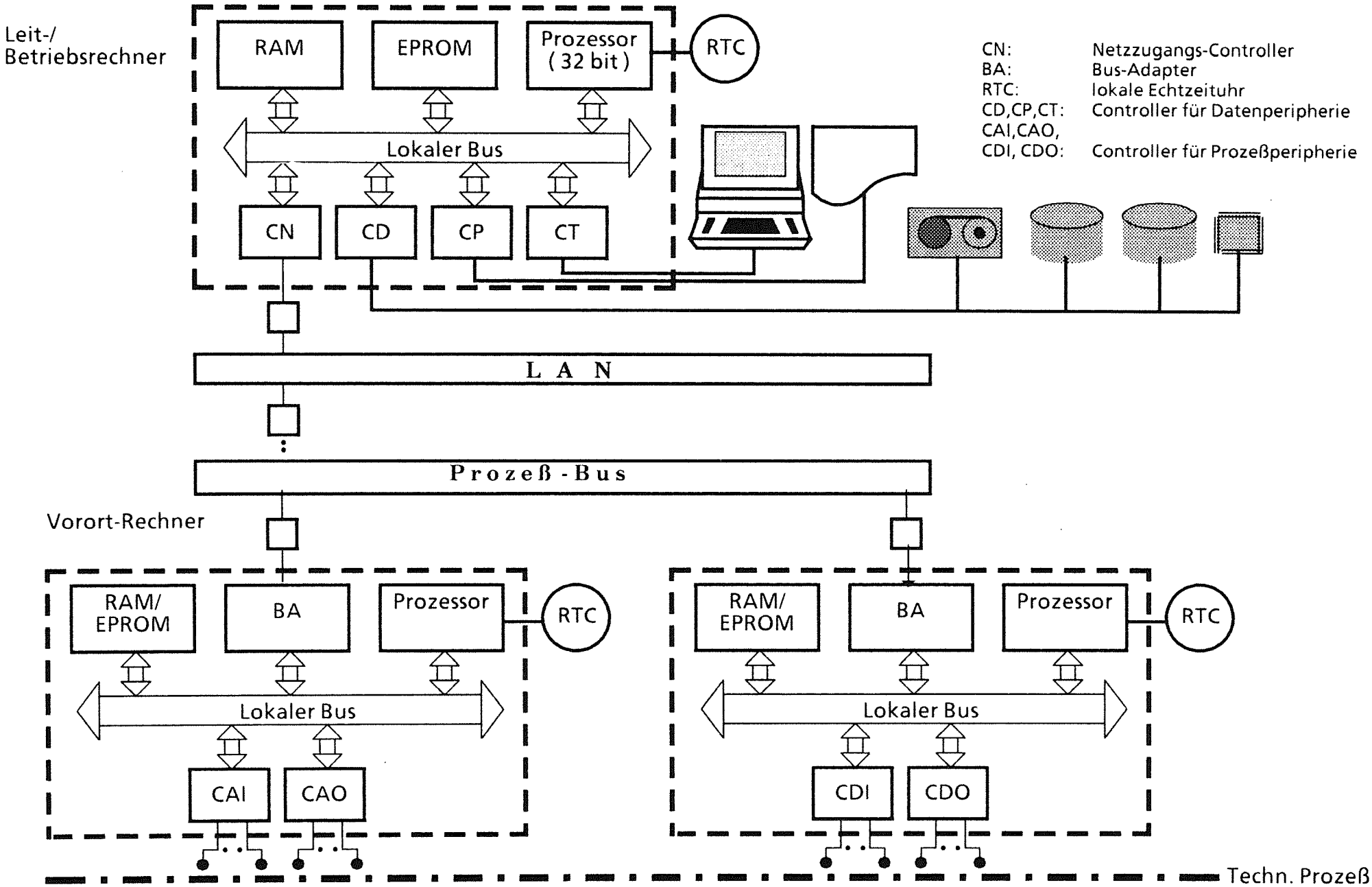



**Abb.** 3.2: Typische Hardware-Konfiguration eines verteilten PFS

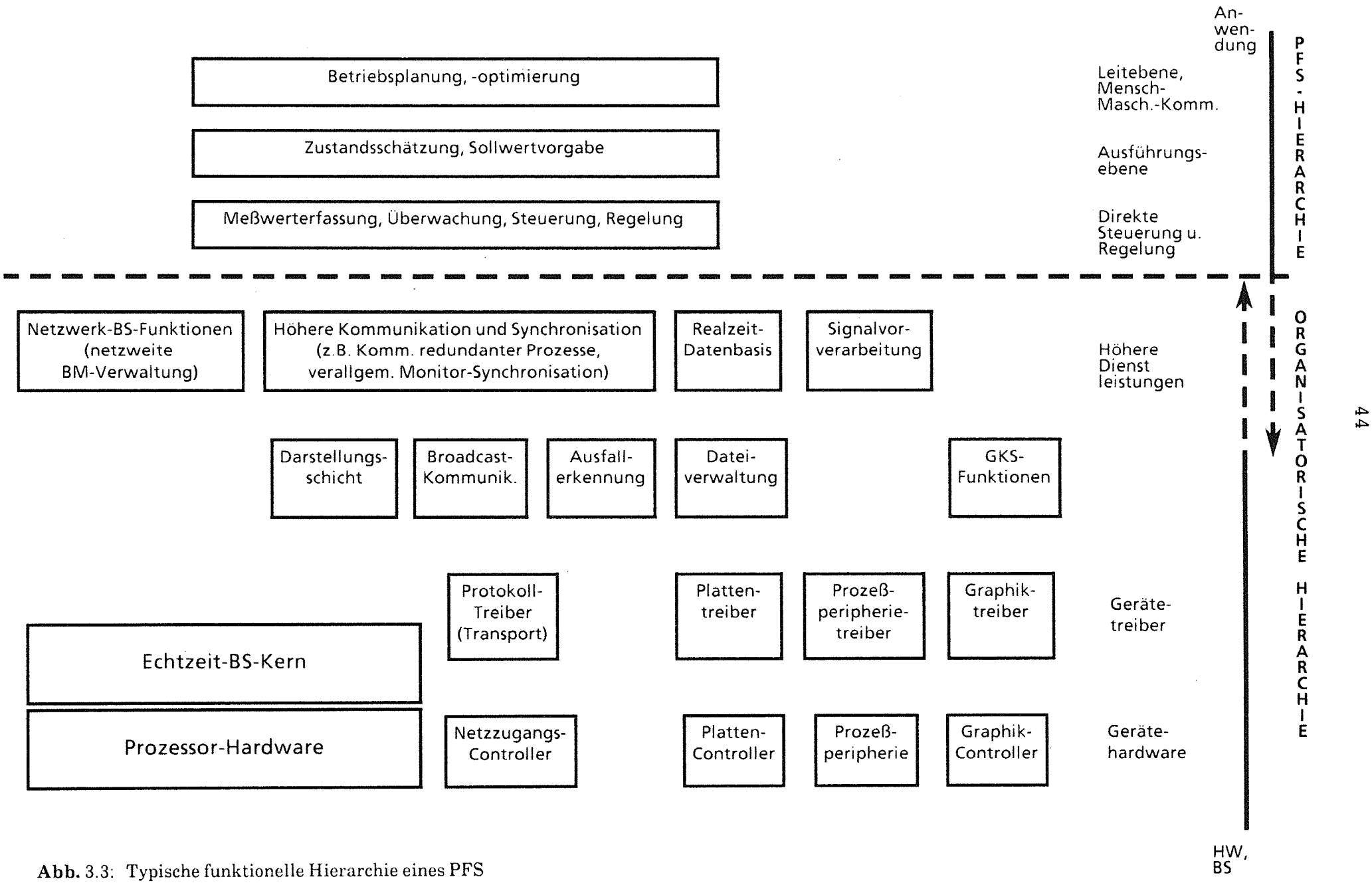



**Abb.** 3.3: Typische funktionelle Hierarchie eines PFS

(a) **Hardware**
Die Dienstleistungen umfassen neben den Rechner-Befehlssätzen die Operationen des Unterbrechungswerkes, der Echtzeituhr, die Adreßraumoperationen (einschließlich Kontextbehandlung), sowie die Programmier-Schnittstellen zu allen E/A-Gerätecontrollern einschließlich Rechnernetzzugang

(b) **Echtzeit-BS-Kern**
Die Dienste umfassen die Verwaltung, Kommunikation und Synchronisation lokaler (Rechen)Prozesse einschließlich zeitbasierender Synchronisation (Setzen und Aufheben zeitlicher Wartebedingungen), und die Zuteilung der Rechner-Betriebsmittel (Prozessor und Arbeitsspeicher, Adreßraumverwaltung, Kopplung von E/A-Geräten und externer Unterbrechungssignale an Rechenprozesse).

(c) **Geräte-Treiber**
Zuteilung, Parameterversorgung, Überwachung, Endebehandlung u. Fehleranalyse von E/A-Geräteaufträgenüber genormte Benutzerschnittstellen, die die physikalischen Geräteeigenschaften verbergen.

(d) **Höhere Betriebssystem- und Dienstleistungsschichten**
Hierzu zählen z.B.

- Kommunikationsprotokolle nach dem ISO-Schichtenmodell /ISO 85/ (z.B. Sitzungs- oder Darstellungsschicht), Dateiverwaltungs- und Datenbank-Funktionen
- "Höhere" Kommunikations- oder Synchronisationsdienste zur Unterstützung spezieller Anwendungen (z.B. redundante Systeme) /KLM 82//MAA 84//DKO 86//KES 77//FAP 83/
- Netzwerk-Betriebssystemdienste, welche lokale Rechner-Betriebsmittel oder BS-Dienste unter einer möglichst homogenen Sicht netzweit anbieten

(e) **Anwendungsebenen**
Die Anwendung wird häufig in Prozeßführungs- oder Regelungsebenen eingeteilt, die durch Ausdehnung von Einflußbereichen auf immer größere funktionale Einheiten des TP entstehen.Z.B. in der Fertigungstechnik:
**Leitebene**: Planung/Koordinierung von Fertigungsaufträgen, die mehrere Bearbeitungsstationen (Werkzeugmaschinen) und Transportmittel benötigen
**Ausführungsebene**: Steuerung/Überwachung einzelner Aufträge, Zerlegung in elementare Transport-, Hantierungs- und Montageoperationen
**Regelungsebene**: Ausführung/Überwachung der Elementaroperationen durch Verarbeitung der Meßsignale von Positions- und Geschwindigkeitsgebern, und Ausgabe der Stellsignale an die Elektromotoren.

Während die Funktionen der Schichten (a)-(c) i.a. in jeder Anwendung bzw. jedem Rechnerknoten existieren, sind die höheren Funktionen i.a. nur zu einem Teil vorhanden.

### (3) SW-Organisation, Auftragsfluß

Die koordinierte Abwicklung von Diensten mit i.a. mehreren aktiven, nebenläufigen Dienstanforderern und Diensterbringern erfolgt über **Austauschobjekte** (vgl. Abb. 3.4), z.B. Botschaftenkommunikation über Mailbox oder Monitor-Synchronisation, die durch die Basiskommunikation/-synchronisation der lokalen BS-Kerne unterstützt werden.
Die Eigenschaften dieser Austauschobjekte sind beliebig (vgl. 3.3.1).

Entscheidend für die Anwendbarkeit des (in den BS-Kern) integrierten Simulationsansatzes ist jedoch folgende **Bedingung**

(BI)Alle nicht rein prozeßlokalen Aktionen, insbesondere Zugriffe auf gemeinsame Rechnerbetriebsmittel, werden letztlich (i.a. unter Zwischenschaltung der Schichten (c)-(e)) über Dienste der BS-Kerne (b) abgewickelt, und nicht unter direktem Zugriff auf die Hardware (a). Jeder BS-Kern hat die vollständige Sicht uneingeschränkte Kontrolle über die Prozeßabläufe und Rechnerbetriebsmittel seines Einflußbereiches.

Bei Verwendung nebenläufiger höherer Programmiersprachen wie z.B. ADA /ADA 83/, MODULA2, CSSA /BMS 82/ bleiben natürlich (BS)-Prozesse und -dienste durch die Objekte und Operationen der Sprache (packages, Module, Agenten etc.) verborgen. Die Forderung an den Compiler lautet dann, daß alle Nebenläufigkeiten auf Sprachebene durch BS-Kern-Operationen realisiert werden. Dies gilt z.B. auch für die Bearbeitung von Hardware-Unterbrechungen.

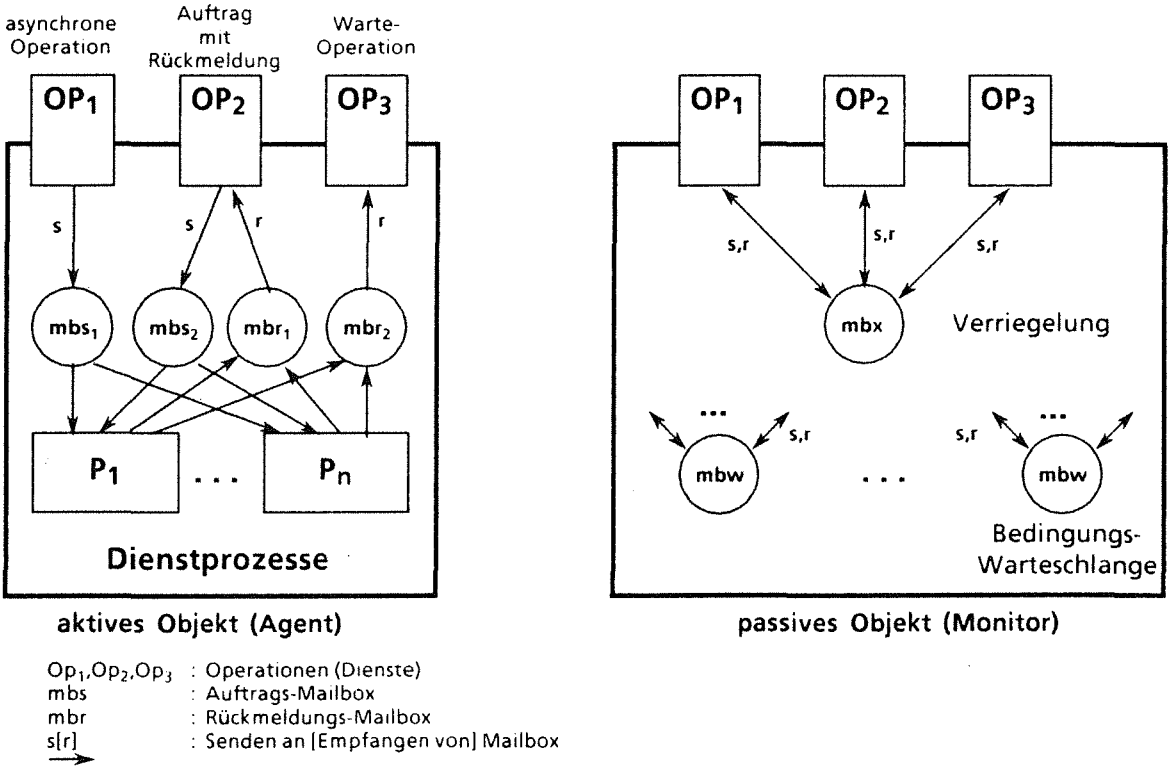



**Abb.** 3.4: Organisation der Auftrags- und Kontrollflüsse

### 3.1.3 Leistungsgrößen

Bei den PFSen als wichtigster Klasse von Anwendungen der integrierten Simulation lassen sich die Leistungsgrößen in drei Kategorien einteilen.

a) das **Führungsverhalten** des technischen Prozesses bzw. seiner Teilprozesse

b) das **Alter** der Ausgangsdaten im PFS relativ zu gewissen Eingangsinformationen (im Englischen 'port-to-port-time'), mit Antwortzeiten und Regler-Totzeiten als Spezialfällen,

c) die **Auslastung** der Rechnerbetriebsmittel (HW, SW) durch die Speicherung, den Transport und die Verknüpfung von Informationen.

Die Leistungsgrößen (b) und (c) dienen hauptsächlich zur Diagnose, als Indikatorvariablen und Einflußgrößen auf die primären Führungsgrößen a).

**Auswertungsarten**

Die Durchführung eines einzelnen Experiments liefert in einem Zeitraum $[T_0,T_1]$ für jede Leistungsgröße L n Einzelbeobachtungen $L(t_1),...,L(t_n)$ $(T_0<t_1<..<t_n<T_1)$. Folgende Arten der **Auswertung** bzw. Verarbeitung der Einzelwerte L(ti) werden angewandt.

**T- transienter Verlauf**
Die Zeitreihe $L(t_1),..,L(t_n)$ wird unverarbeitet, als Rohdatenverlauf, präsentiert. Die Einflüsse stochastischer Störgrößen oder stochastischer Elemente in der Simulation des PFS sollten dann aber durch Experimentwiederholungen mit unterschiedlichen Voreinstellungen der Zufallszahlen und eine Signalanalyse im Frequenzbereich kompensiert werden.

**M- Mittelwert**
Aus L(t1),..,L(tn) wird ein stationärer Mittelwert **L**, oder ein zeitliches Integral über den Verlauf von L berechnet.

**V- Empirische Verteilungsfunktion**
$L(t_1),..,L(t_n)$ werden als unabhängige Realisationen einer Zufallsvariablen L aufgefaßt, deren Verteilungsfunktion F(L) bzw. Wahrscheinlichkeitsdichte (Histogramm) näherungsweise durch Klasseneinteilung des Wertebereiches von L bestimmt wird. Auf die Definition geeigneter Konfidenzmaße für die Verteilungsfunktion und die Bestimmung der für einen gegebenen Konfidenzgrad notwendigen zeitlichen Dauer der Experimente gehen wir allerdings nicht ein (vgl. z.B. /Zei 76/, Anhang C).

Im folgenden werden die Leistungsgrößen aus (a)-(c) genauer definiert. Die Realisierung der benötigten Auswertungsinstanzen wird nicht behandelt.

### 3.1.3.1 Führungsgrößen (Regelgrößen)

**(A) Kontinuierliche Prozesse**

Als gebräuchlichstes Leistungsmaß des Führungsverhaltens dienen quadratische Gütekriterien /FOE 78/, die an äquidistanten Stützstellen $T_0+k\Delta$, $k=0,..,N$ die Regelabweichungen, ggf. auch die Geschwindigkeit des Ausregelns und die aufgewendete Stellenergie im Intervall $[T_0,T_0+N\Delta]$ bewerten:

$$G_N := \sum_{k=0}^{N} k * e^T[k] + u^T[k]\,Ru\,[k]$$

| | |
|---|---|
| $e := z - z^{soll}$ | Regelabweichung $(e[k] := e(T_0+k\Delta), k=0..N)$ |
| $z$ | Regelgröße (Istwert) $\in \mathbf{R}^r$ |
| $z^{soll}$ | Regelgröße (Sollwert) $\in \mathbf{R}^r$ |
| $Q \in \mathbf{R}^{r,r}$, $R \in \mathbf{R}^{m,m}$ | |

wobei Q, R positiv definite, symmetrische Matrizen, durch die die einzelnen Komponenten der Vektoren e bzw. u gewichtet werden. $G_N$ hat für $N->\infty$ nur dann einen endlichen Grenzwert, wenn die Regelabweichungen und Stellgrößen für $N->\infty$ verschwinden, also nicht bei ständig einwirkenden externen Störgrößen. In letzterem Fall ist stattdessen der Mittelwert $G_N := 1/N*G_N$ zu bilden.

**(B) Diskrete Prozesse**

Hier ist keine allgemeine Formel für die Leistungsgrößen anzugeben; diese hängen stark von der speziellen Anwendung ab. Wir führen nur einige Beispiele für Regelgrößen an:

(1) (betreffend die Bewertung von Verfahren und Strategien des PFS)
Zielgrößen der Betriebsführung, z.B. Durchsatz erfolgreich bearbeiteter Werkstücke, Auslastungen von Maschinen, Transportsysteme oder Lager. Die Bewertung und Optimierung von Strategien anhand solcher Zielgrößen wird von den meisten, z.B. aus der Fertigungstechnik bekannten Simulationssystemen verfolgt.

(2) (betreffend die Realisierung und den Test des PFS) unerlaubte Zustandsübergänge des technischen Prozesses (z.B. Verlust eines Werkstücks, Kollision einer Handhabungseinrichtung, Verlassen des vorgegebenen Bewegungsraumes einer Transporteinrichtung) und dadurch verusachte Stillstandszeiten (Notabschaltungen/Wiederanläufe). Gründe dafür könnten sein, daß zeitliche Restriktionen durch das PFS nicht eingehalten wurden oder Situationen im technischen Prozeß aufgetreten sind, die in der Spezifikation oder Implementierung des PFS nicht berücksichtigt wurden (z.B. unerwartete Position eines Werkstücks).

Die Maßeinheit für die Leistungsgrößen ist in allen Fällen: Anzahl Ereignisse pro Zeiteinheit ("Durchsatz"), oder Gesamtdauer pro Zeiteinheit ("Auslastung").

### 3.1.3.2 Alterung

Antwortzeiten und Reaktionszeiten spielen im PFS auf allen Ebenen eine zentrale Rolle: von der HW-Reaktionszeit auf einzelne Unterbrechungssignale bis zur Signaldurchlaufzeit der gesamten Regelungshierarchie. Während die Führungsgrößen und die Betriebsmittel-Auslastungen sich immer einzelnen Verarbeitungsinstanzen, nämlich Prozeßelementen (3.1.1) oder Betriebsmitteln, zuordnen lassen, sind die Antwortzeiten den Datenobjekten bzw. Datenpfaden zugeordnet. Als Oberbegriff zur Antwortzeit wird das **Alter** der Datenobjekte eingeführt, das die Änderungszeitpunkte beliebiger, funktionell abhängiger Ein- und Ausgangsdaten voneinander in Beziehung setzt. In PFSen treten allgemeinere Formen der Abhängigkeit von Daten auf als z.B. Transaktionen darstellen.

- Transaktionen stellen eine i.w. lineare Struktur mit eindeutigem Beginn (Benutzer!), Ende und Bearbeitungszustand dar; eine Transaktion kann sich zwar in nebenläufige Subtransaktionen aufspalten, diese müssen aber alle beendet sein, bevor die Haupttransaktion fortfährt (synchroner Anforderungs-/Rückmeldungsmechanismus in allen beteiligten Instanzen).
- In PFSen sind dagegen i.a. mehrere Eingangsdaten über **asynchron** arbeitende Verarbeitungsinstanzen (z.B. Regelungsebenen) netzartig mit mehreren Ausgangsdaten verknüpft. Während die untersten Ebenen mit der physikalischen Meßwerteingabe und Stellwertausgabe betraut sind, operieren die höheren Ebenen mit aggregierten, geschätzten Zustandsgrößen als Eingabe und produzieren Sollvorgaben für diese Zustände als Ausgabe. Die einzelnen Ebenen arbeiten weitgehend autonom, und der Aggregationsprozeß vollzieht sich in immer gröberen Zeitrastern. Gesucht ist dann z.B. die Gesamtdurchlaufzeit, nach der eine Prozeßinformation (Meßwert) die ihrer Wichtigkeit entsprechende Anzahl von Regelungsebenen durchlaufen und die aktualisierten Prozeßzustände sich umgekehrt über eine Hierarchie von Sollwerten schließlich in neuen Stellwerten niedergeschlagen haben.

Jedem Datenobjekt längs eines zu überwachenden Datenpfades wird der Zeitstempel der Information zugeordnet, auf der es letztlich beruht (**Basiszeit** /DKO 86/, i.f. wird vereinfachend **eine** skalare Basiszeit angenommen, es sind aber auch vektorielle Basiszeiten, für unterschiedliche Eingabedaten, denkbar). Ein Datenobjekt eines solchen "alternden Datentyps" <t> besteht also aus

- Nutzdaten d eines beliebigen Grunddatentyps <t> (Skalar, Verbund, Nachricht)
- Zeitstempel tb (Basiszeit).

Folgende die Basiszeit betreffenden Operationen sind auf einem solchen Datenobjekt definiert (vgl. Abb. 3.5).

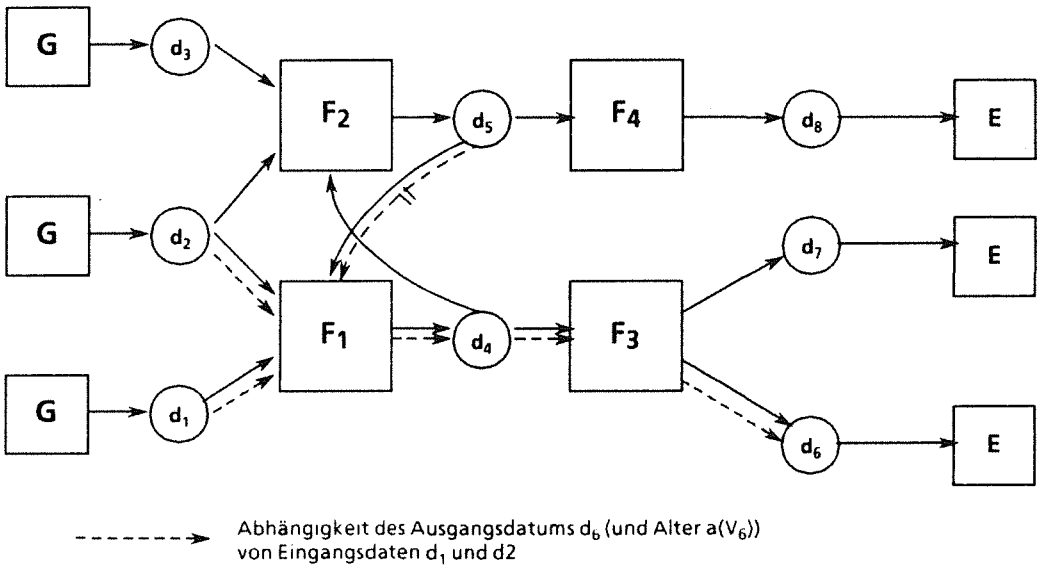



**Abb.** 3.5: Alterung von Datenobjekten

(1) **Generierung (G)**

Ein Datenobjekt d wird neu erzeugt oder entsteht aus Eingabedaten, deren Basiszeiten selbst nicht interessieren oder undefiniert sind. d markiert also den Beginn des auszuwertenden Datenweges.

d.tb := TIME; ( Zeitpunkt der Generierung )

(2) **Verknüpfung (F)**

Ein Datenobjekt d entsteht als Ergebnis einer Verknüpfung F (arithmetische Operation, Funktionsprozedur, Transportoperation oder Zuweisung) von Eingabedaten, deren Basiszeiten selbst definiert sind:

$d := F(d_1,...,d_k)$;

Zu jeder solchen Verknüpfung F ist eine Funktion Ft zu spezifizieren, die die Basiszeit des Ergebnisses als Funktion der Basiszeiten seiner Eingabedaten festgelegt:

$Ft : T^k -> T$, $d.tb = Ft(d_1.tb,...,d_k.tb)$

Die wichtigsten Beispiele für Ft sind:

$Ft(t_1,...,t_k) := \max\{t_i\}$ aktuellste Eingabe bestimmt die Basiszeit

$Ft(t_1,...,t_k) := \min\{t_i\}$ am wenigsten aktuelle Eingabe bestimmt die Basiszeit

$Ft(t_1,...,t_k) := t_j$ f. ein $j \leq k$ nur Eingabe j ist relevant

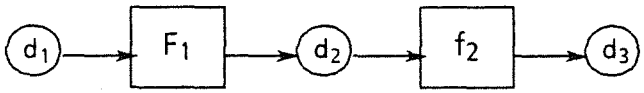
(3) **Auswertung (E)**

Das Alter einer Variablen v eines alternden Datentyps <t> wird aus der Basiszeit des aktuellen Variablenwertes d bestimmt. v ist Endpunkt des auszuwertenden Datenpfades.

a (v) : = TIME - d.tb ( TIME Zeitpunkt der Auswertungsoperation )

Das Alter einer Variablen v am Ende eines Datenpfades hat einen sägezahnartigen Verlauf, wobei die Sprungstellen die Aktualisierungen des Variablenwertes entsprechen. Abb. 3.6 zeigt als Beispiel einen sehr einfachen linearen Datenpfas (pipeline) mit n hintereinandergeschalteten, durch Puffer verbundenen Verarbeitungsinstanzen, für die das Alter der Ausgangsdaten sich leicht von Hand berechnen läßt; i.a. ist aber eine rechnergestützte Auswertung erforderlich.

**Abb. 3.6:** Alter der Ausgangsdaten einer linearen Pipeline mit n Bearbeitungsinstanzen F(1),..,F(n) im stationären Zustand
b(i): Bearbeitungszeiten für F(i) ( **konstant, deterministisch** angenommen)
Zahlenwerte in Abb.: n=2, b(1)=3, b(2)=5, c(1)=2




a) Zyklische, sequentielle Bearbeitung

**loop**
**for** i:=1 **to** n **do** v(i+1):= F(i)(v(i));
**end loop**

$$a_{min}(v(n+1)) = \sum_{i=1}^{n} b(i);$$

$$a_{max}(v(n+1)) = 2^* \sum_{i=1}^{n} b(i);$$

b) Bearbeitung durch autonome Bearbeitungsinstanzen F(i) mit Pufferung der Zwischenergebnisse in FIFO Kommunikationsobjekten m(i) ($1 \leq i < n$) zwischen F(i), F(i+1).
c(i): Kapazität von m(i)

T(i): ($1 \leq i \leq n$)
**loop**
rec_msg (m(i-1), v(i));
v(i+1):= F(i)(v(i))
send_msg (m(i), v(i+1));
**end loop;**

Sei bmax = max{b(i)|$1 \leq i \leq n$}, j = min{i|b(i) = bmax}

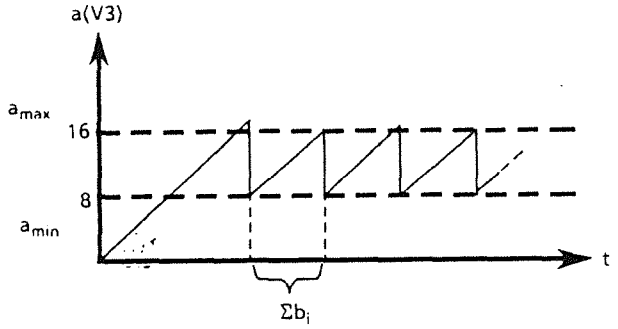
$$a_{min}(v(n+1)) = \left(\sum_{i=1}^{j-1} c(i)+j\right) bmax + \sum_{i=j+1}^{n} b(i)$$

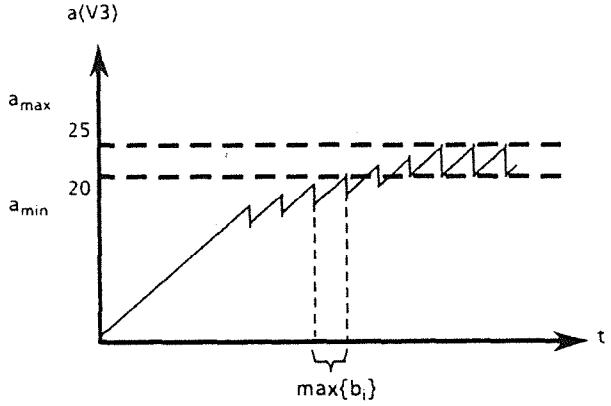
$$a_{max}(v(n+1)) = a_{min}(v(n+1)) + bmax$$

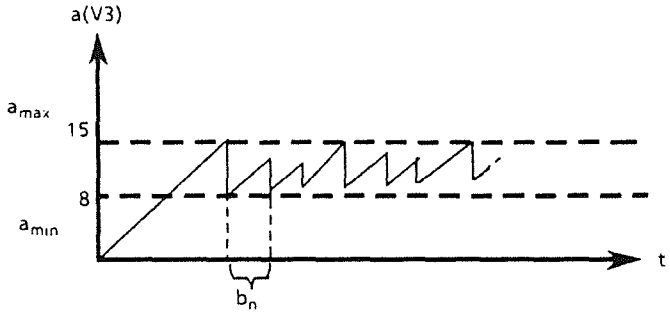
c) Wie b), aber m(i) Zustands-Mailbox (nichtblockierendes Lesen und Überschreiben des aktuellen Wertes d(i))

$$a_{min}(v(n+1)) = \sum_{i=1}^{n} b(i);$$

$$a_{max}(v(n+1)) = 2^* \sum_{i=1}^{n} b(i) - \sum_{i=1}^{n-1} ggt(b(i),b(i+1)) \text{ falls } b(i) \in N$$



Aus der Folge $(T_i,d_i)$ der Aktualisierungszeitpunkte und -werte einer Variablen v werden sowohl die empirische Verteilungsfunktion (Typ V) als auch folgende stationären Zielgrößen (Typ M) bis zu einem Zeitpunkt T ermittelt:

$\mathbf{a}(v,T) := 1/T^* \sum_{T_i \leq T} ((T_i + T_{i-1})/2 - d_{i-1}.tb)^*(T_i - T_{i-1})$ **Alter** (Mittelwert bzw. Zeitintegral)

$\mathbf{a}_{max}(v,T) = \max \{T_i - d_{i-1}.tb \mid T_i \leq T\}$ Maximales Alter

$\mathbf{a}_{min}(v,T) = \min \{T_i - d_i.tb \mid T_i \leq T\}$ Minimales Alter

$\mathbf{ar}_{max}(v,T) = \max \{T_i - d_i.tb \mid T_i \leq T\}$ Maximale Antwortzeit
(Antwort: Änderung am Datenausgang v)

Das Alter **a**(v,T) stellt ein integriertes Maß aus Antwortzeit und Durchsatz für den Datenausgang v dar, das sowohl die Aktualität der Ausgangsdaten relativ zu den Eingangsdaten, als auch die Häufigkeit der Aktualisierungen berücksichtigt.

### 3.1.3.3 Betriebsmittel-Auslastung

Auslastungswerte für Betriebsmittel lassen sich nach folgenden Kriterien einteilen:

- Die Auslastung bezieht sich auf feste **Zeitintervalle** $[T_i,T_{i+1}]$ eines Beobachtungszeitraums (**intervallbezogene Auslastung**) oder auf bestimmte, i.a. regelmäßig wiederkehrende Aufträge (**aufgabenbezogene Auslastung**) (Kriterium I|A)
- Es handelt sich um ein physikalisches (HW-)Betriebsmittel oder um ein logisches (SW-gestütztes) Betriebsmittel (Kriterium P|L). In beiden Fällen kann sowohl ein simuliertes als auch ein reales, von realen Testobjekten in Anspruch genommenes Betriebsmittel vorliegen.
- Es handelt sich bei HW-Betriebsmitteln um ein zeitlich aufgeteiltes (z.B. CPU, E/A-Gerät) oder um ein räumlich aufgeteiltes (Arbeitsspeicher, Massenspeicher) (Kriterium Z|R).

Folgende Auslastungswerte werden ausgewertet:

(1) **Intervallbezogene Auslastung von HW-Betriebsmitteln**

BIPZ(r,i) := (Σ der Aktivzeiten des zeitlich aufgeteilten HW-BM r innerhalb $[T_i,T_i+1]$)/$(T_{i+1}-T_i)$

$$BIPR(r,i) := 1/((T_{i+1}-T_i)\,c(r)) * \int_{T_i}^{T_{i+1}} b(r)dt$$

(b(r) Belegung des räumlich aufgeteilten HW-BM r
c(r) Kapazität des räumlich aufgeteilten HW-BM r $(b(r) \leq c(r)$

$(0 \leq BIPZ(r,i) \leq 1,$
$0 \leq BIPR(r,i) \leq 1)$

**(2) Intervallbezogene Auslastung logischer Betriebsmittel**

Ein logisches, i.a. geographisch auf mehrere Rechner verteiltes Betriebsmittel r stellt eine Menge von Dienstleistungen d1,...,dn zur Verfügung, die durch Aufträge beansprucht werden. I.a. können mehrere Aufträge simultan in Bearbeitung sein. Die Bearbeitung von Aufträgen stellt keine "reinen" Bedienzeiten wie bei HW-Betriebsmitteln dar, sondern enthält Wartezeiten auf untergeordnete Betriebsmittel.
Jedem Auftrag req zum Dienst req.d entspricht ein eindeutiges Ankunftereignis (z.B. Ankunft einer Auftragsbotschaft, Zeitpunkt req.ta) und eine eindeutige Auftrags-Beendigung (z.B. Rückmeldung an Auftraggeber, oder Weiterleiten des Ergebnisses an Dritte, Zeitpunkt req.te). Logische Betriebsmittel können räumlich verteilt sein (z.B. Protokollschichten); Ankunft- und Ende-Ereignis können auf verschiedenen Rechnern stattfinden.

$A_i(r,d) := \{t \mid T_i \leq t \leq T_{i+1} \wedge \exists\ req: req.d = d \wedge req.ta \leq t \leq req.te\}$

Aktivzeiten für Dienst d und logisches BM r innerhalb $[T_i, T_{i+1}]$

$BIL(r,d,i) := \mu(A_i(r,d))/(T_{i+1}-T_i)$ intervallbezogene Auslastung für (r,d)

Diese Größe stellt ein anwendungsorientiertes Maß für die Belastung/Überlastung von Dienstleistungsinstanzen dar. Die Ursache einer Überlastung kann sein, daß untergeordnete Betriebsmittel überlastet sind, oder daß r bei der Bedienung eine zu geringe Priorität erhält.

**(3) Aufgabenbezogene Belastung**

Die folgende Größe mißt die Gesamtkosten (Verwaltungsaufwand) an Hardware-Betriebsmitteln, die durch logische BM verursacht werden.

$BALZ(r,d,req,j) :=$ Σ der Aktivzeiten des zeitlich aufgeteilten HW-BM j durch Auftrag req für Dienst d des logischen BM r

Dabei werden alle Aktivzeiten berücksichtigt, die entweder bei der Bearbeitung von req in r selbst, oder in untergeordneten Betriebsmitteln r' entstehen, die zur Bearbeitung von req beansprucht werden. Eine effektive Belastung der HW-BM j durch Aufträge vom Typ (r,d) erhält man, indem man einen über alle Aufträge gemittelten Wert **BALZ**(r,d,j) mit der mittleren Dienst-Ankunftsrate **ANKR** (r,d) multipliziert:

BA (r,d,j) := **BALZ**(r,d,j) * **ANKR** (r,d)

(effektive aufgabenbezogene Belastung des HW-BM j durch Dienst d, logisches BM r).

### 3.2 Entwicklungsphasen und Experimente

Als grundlegendes Leitbild des Entwicklungsprozesses der PFSe dient ein Phasenmodell, das unter dem Namen **"operationaler Entwurf"** in /ZAV 84/,/ZAS 86/ vorgestellt wurde. Danach resultiert jeder Entwurfsschritt in einem ausführbaren und hinsichtlich seines funktionellen und zeitlichen Verhaltens bewertbaren Programm, wobei die Anforderungen und Randbedingungen der eigentlichen Einsatzumgebung schrittweise einbezogen werden. Anders als in /ZAS 86/, werden nicht nur die Entwurfsphase, sondern auch Realisierung, Test und Wartung unterstützt.
Eine grobe, schematische Einteilung in vier Phasen mit den wichtigsten Entwurfsentscheidungen, Leistungsgrößen und Experimenten jeder Phase wird i.f. für die betrachtete Klasse der PFS-Anwendungen beschrieben und in Tabelle (3.1) am Ende dieses Abschnitts zusammengefaßt.

#### 3.2.1 Systemanalyse und Anforderungsspezifikation

Die zentrale Aufgabe besteht in einer möglichst genauen und realitätsnahen Analyse und Modellbildung des technischen Prozesses, welche sowohl die Grundlage für die zu entwickelnde Prozeßsteuerung/-regelung, als auch für deren Leistungsbewertung bildet und über die Qualität der Ergebnisse letztlich entscheidet. Die Analyse des technischen Prozesses umfaßt nach 3.1.1

- die Festlegung aller Meß-, Stell-, Regel- (Ziel-)größen sowie Störgrößen, - die mathematische Modellbildung, d.h. die Herleitung eines Zustandsraummodells zur Beschreibung des dynamischen Verhaltens,
- die Prozeßidentifikation, d.h. Anwendung von Verfahren zur optimalen Schätzung von Parametern innerhalb der gewählten mathematischen Struktur mit Hilfe von Meßdaten des realen technischen Prozesses /ISE 77/.

Für die Prozeßführung sind die in 3.1.3.1 allgemein charakterisierten **Führungs- bzw. Regelziele** präzise festzulegen.

Simulationsexperimente am ungeregelten technischen Prozeß, die Teil dieser Phase sind, tragen zur Klärung der Anforderungen des PFS bei. Der Schwerpunkt bei der Werkzeugunterstützung entfällt in dieser Phase jedoch nicht auf die Simulation selbst, sondern auf das übergeordnete Entwicklungssystem (z.B. bei der Prozeßmodellentwicklung, bei der Erstellung von Pflichtenheften, Projektmanagement /TRB 84/, /LAL 84/), dessen Konzeption und Realisierung nicht Gegenstand dieser Arbeit ist.

### 3.2.2 Operationaler Entwurf (Stadium O)

An die Analyse des technischen Prozesses und die Festlegung der Regelziele schließt sich die Entwicklung und Auswahl von Verfahren für die Prozeßführung an. Hierbei kommen zwei grundlegende Strukturierungs- bzw. Zerlegungsprinzipien zur Anwendung:

(1) **Geographische Zerlegung**

Der technische Prozeß sei nach physikalischen Kriterien in Teilprozesse $TP_1,..,TP_n$ zerlegt, wie in 3.1.1 beschrieben.

(2) **Hierarchische Zerlegung**

Der technische Prozeß wird hierarchisch in **Prozeßebenen** $E_1,..,\ E_m$ gegliedert. Ein Teilprozeß TP auf Ebene $E_i$ entsteht durch Zusammenfassung (Kopplung) von k Komponenten $TP_1,...,\ TP_k$ auf der nächsttieferen Beschreibungsebene.

In entsprechender Weise wird das PFS in Führungskomponenten $P_i$ und PFS-Ebenen $PF_1,...,PF_m$ zerlegt. Jeder Teilprozeß $T_i$ auf Ebene i hat seine eigenen Führungsgrößen, und daraus sind Regelziele/Sollvorgaben für die Führungskomponente $P_i$ festzulegen. Jedes $P_i$ besteht i.d.R. aus zwei Hauptkomponenten: Schätzung nicht meßbarer Zustandsinformation (z.B. Beobachter bei kontinuierlichen Systemen, Zustandsautonmat bei diskreten Systemen), und Ermittlung von Sollvorgaben bzw. Stellwerten. Ein top-down-Entwurf des PFS mit ebenenweiser Bewertung ist dann möglich, wenn auch entsprechend vergröberte Prozeßmodelle zur Verfügung stehen, die als Schnittstelle aggregierte Prozeßzustandsinformation bzw. Sollwerte anbieten, wobei alle unteren Ebenen als geschlossene ("bereits erfolgreich geregelte") Teilsysteme modelliert werden.
Algorithmisch ist jedes $P_i$ durch zyklische (Rechen)Prozesse darstellbar, die auf ihre Eingabeschnittstellen (Meß-, Stell-, Schätz- oder Sollwerte) zugreifen, die Eingabedaten entsprechend den Verarbeitungsalgorithmen in die entsprechenden Ausgabedaten transformieren und diese an ihre Ausgabeschnittstelle übergeben. Die wesentlichen Entwurfsentscheidungen (Entwurfs"variablen") sind

(E1) die Informationsstruktur des PFS, d.h. welche Sollwerte, Stellwerte, Zustandsmeldungen, Aufträge oder Rückmeldungen zwischen den PFS-Komponenten auszutauschen sind (Ein-/Ausgabe-SSe)

(E2) die Verarbeitungsalgorithmen (Schätz-, Steuer- oder Regelalgorithmen) und ihre Verfahrensparameter

(E3) die Art der Zugriffsoperationen auf die Ein-/Ausgabeschnittstellen (E1), z.B.

- ob der Informationsaustausch auf Initiative des Senders, oder auf Anforderung des Empfängers erfolgt, oder beides vorgesehen wird
- ob die Ein-/Ausgaben zeittakt-gesteuert oder ereignisgesteuert erfolgen oder beides vorkommt
- inwieweit Sender/Empfänger synchronisiert sind

**Zeitliche Spezifikation:**

Die Entwurfsentscheidungen (E2),(E3) enthalten als zeitliche Parameter die Abtastzeiten, in denen PFS-Komponenten auf den Zustand ihrer Umgebung reagieren.

**Ergebnisse**

Die Experimente sollen eine vergleichende Bewertung und Auswahl von Prozeß-steuerungs-/Regelungskonzepten ohne Berücksichtigung von Randbedingungen ihrer Realisierung (zentrale oder verteilte Implementierung, Laufzeiten, Speicherbedarf, E/A-Operationen) ermöglichen.

Hierzu gehört auch die Bestimmung der zeitlichen Soll-Dynamik des PFS, sowohl der globalen Antwortzeiten als auch individuellen Abtastzeiten, die eine befriedigende Einhaltung der Führungsziele gewährleistet. Über die Diagnose-Beziehung

| | | | | |
|---|---|---|---|---|
| Abtastparameter zus. mit (E1)-(E3) (Entwurfsgröße) | -----> | Alterung, Antwortzeit (sekundäre Leistung, Diagnosegröße) | -----> | Führungsgröße (primäre Leistungsgröße) |

soll insbesondere auch die Empfindlichkeit (Sensitivität) der Führungsgrößen gegenüber Änderungen des Zeitverhaltens untersucht werden.

### 3.2.3 Entwurf in virtueller Betriebsmittelumgebung (Stadium VB)

Verschiedene **Implementierungsstrategien** für das PFS werden entwickelt und bewertet. Durch die Zuordnung der "abstrakten" Prozesse in 3.2.2 zu möglichen HW-Konfigurationen

- Zielrechner
- Kommunikationssysteme (DMA-fähige Rechnerkopplungen, Realzeit-LAN, Prozeßbusse)
- Prozeßperipherie
- Hintergrundspeicher (z.B. als Realzeit-Datenbasis)

wird sowohl die interne Verarbeitung (E2), als auch die Kommunikationsoperationen (E3) konkretisiert (als lokale oder abgesetzte Kommunikation oder Prozeßperipherie-E/A) und unter Berücksichtigung der Leistungscharakteristika dieser BM zeitlich bewertet.

Es ist eine HW-Konfiguration und Aufteilung der Prozesse zu finden, so daß die Führungsziele auch unter den im Vergleich zu 3.2.2 realistischeren Bedingungen erreicht werden. Konfiguration und Verteilung der Prozesse sind i.a. durch viele -von der Leistungsbewertung unabhängige- Randbedingungen eingeschränkt, unter anderem

- Zuverlässigkeit und Verfügbarkeit von Funktionen und Daten, die nur durch Redundanz von HW- und SW-BM wie Rechnern, Prozessen, Kommunikationswegen, Sensoren und Stellgliedern zu erreichen ist, falls nicht das Regelkonzept selbst (3.2.1) den Ausfall gewisser Zweige der Rückführung toleriert ("robuste Regelung").

- geographische Randbedingungen (Verkabelungsaufwand, Notwendigkeit von Frontend-Rechnern, einsetzbare Kommunikationssysteme)
- Kompatibilität mit Schnittstellen/Standards.

Bei der Untersuchung und Bewertung verschiedener Verteilungen bzw. Allokationen wird die Menge der zu verteilenden Rechenprozesse sich i.a. vergrößern:

- redundante Vervielfachung von Prozessen
- interne Parallelisierung eines Prozesses (Teilschritte, die disjunkte Ausgangsdaten erzeugen)
- Zerlegung sequentieller Bearbeitungsschritte in eine Kette von Unterprozessen, die asynchron aktiviert werden (pipelining). Dies führt wegen des zusätzlichen Nachrichtenaustausches zu einem schlechteren Antwortzeitverhalten im günstigsten Fall, aber wegen der überlappenden Auftragsbearbeitung zu einer besseren Antwortzeit im ungünstigsten Fall, also zu einer Verringerung der Varianz der Antwortzeit (vgl. 3.1.3.2, Abb. 3.6).

**Zeitliche Spezifikation**

Die wesentlichen Einflußgrößen sind die **Zuordnung** der Funktionen bzw. Prozesse zu den BM und ihre **Verbrauchsspezifikation** (CPU-Zeit, Umfang der zu speichernden/zu übertragenden Informationen etc.). Es handelt sich um reinen Verbrauch ohne Wartezeiten im Sinne der aufgabenbezogenen Belastung 3.1.3.3 (3). Als Anhaltspunkte dienen die Komplexität der Funktionen in Abhängigkeit von ihren Eingangsparametern, die relative BM-Schätzung der Elementaroperationen und die Verfeinerungskonsistenz der BM-Angaben, z.B. bei Aufspaltung eines Prozesses in eine Kette von Unterprozesse. Nicht wesentlich ist eine präzise absolute Schätzung des BM-Bedarfs, z.B. der Anzahl der Instruktionen.

**Ergebnisse**

Aus den zulässigen bzw. sinnvollen HW-Konfigurationen wird eine geeignete ausgewählt und dimensioniert und eine Verteilung bzw. Zuordnung des gegenüber 3.2.2 verfeinerten Prozeßsystems auf die Rechner spezifiziert. Wichtige Anhaltspunkte hierfür liefert die Diagnose-Beziehung

| | | | | |
|---|---|---|---|---|
| BM-Spezifikation | -----> | BM-Auslastung | -----> | Antwortzeit oder Führungsgröße |
| (Entwurfsgröße) | | (sekundäre Leistung, Diagnosegröße) | | (primäre Leistungsgröße) |

Bei starken Abweichungen der Antwortzeiten (Alterung) gegenüber dem idealisierten Sollverhalten von 3.2.2 wird die Auslastung der auf einem Datenfluß benötigten HW-BM untersucht. Wichtigstes Betriebsziel ist die Erhaltung einer ausreichenden Kapazitätsreserve dieser BM auch unter Spitzenlast; hierzu müssen die Verteilungsfunktionen für Antwortzeiten/Auslastungen bestimmt werden.

### 3.2.4 Implementierung, Test, Wartung in realer BM-Umgebung (Stadium R)

Da in 3.2.3 von Einflüssen der SW-Implementierung auf einer Zielmaschine noch abstrahiert wird, besitzen die Ergebnisse (Antwortzeiten, Regelgüte) keine absolute Gültigkeit. Selbst eine qualitative Aussage über zwei Entwürfe A,B kann sich im Lichte der Implementierungen A' und B' ins Gegenteil umkehren. Dies vorhersagen zu wollen hieße, die Implementierung auf der ausgewählten Zielkonfiguration mit den dort verfügbaren Zielsprachen und SW-Subsystemen selbst vorwegzunehmen. Erst hier wird für den Entwickler die Tatsache relevant, daß Rechenprozesse Prozesse im Sinne eines bestimmten BS sind, bzw. bestimmte Dienste nur hier verfügbar sind.

Beispiele für Entwurfsentscheidungen und Einflußgrößen, die speziell die Leistung betreffen, sind

- Implementierung abstrakter Datentypen (z.B. bei der Realisierung höherer Kommunikationsdienste oder anderer SW-gestützter Betriebsmittel)
  - als Monitor (prozedurorientierte Schnittstelle) mit gegenseitigem Ausschluß der Aufrufer
  - als Agent (oder "Sekretär" /WET 84/) mit asynchroner Schnittstelle und einem oder mehreren identischen Bedien-Prozessen
- Festlegung von Tuning-Parametern (Abstimmung von Zyklusperioden und Zyklus-offsets zyklischer Prozesse, Prioritäten, Zeitüberwachungsschranken, Pufferdimensionierungen etc.).
- Optimierungen der Kommunikationsstruktur, z.B. Durchgriffsfunktion von höheren Regelungsebenen auf tiefere in Ausnahmesituationen o.ä.

#### Zeitliche Spezifikation

Eine explizite Spezifikation des BM-Verbrauchs erübrigt sich, falls die Experimentmaschine zugleich Zielmaschine ist: der Verbrauch ergibt sich implizit, durch die reale Inanspruchnahme der SW- und HW-BM der Experimentmaschine durch die PFS-Implementierung.

#### Ergebnisse

Es wird überprüft, ob das in 3.2.3 prognostizierte Leistungsverhalten durch die Implementierung, nach deren Optimierung bzw. Tuning, tatsächlich erreicht wird. Ein wesentlicher Punkt liegt in der Untersuchung transienter Vorgänge in den Regelgrößen- und Antwortzeit-Verläufen (Einschwingvorgänge, wichtige Zustandsänderungen im technischen Prozeß, Ausfälle und Wiederanläufe von PFS-Komponenten).

### 3.2.5 Diskussion

Einige übergreifende Gesichtpunkte des skizzierten Phasenmodells sind i.f. noch zu diskutieren.

(1) Die Übergänge zwischen den Phasen erfolgen i.a. nicht für alle Subsysteme gleichzeitig. Für das PFS-Konzept essentielle Komponenten können frühzeitig implementiert, oder wiederzuverwendende Teile als reale Komponenten übernommen werden, während andere Komponenten im Experiment noch in simulierter BM-Umgebung ablaufen (VB), und wieder

andere, z.B. die Weitergabe von Produktionsdaten an einen Betriebsführungsrechner, noch zu entwerfen sind (O).

(2) In den Phasen O und VB handelt es sich aus der Sicht des Entwicklers tatsächlich um einen **Entwurf (Simulationsmodell)** des PFS, weil wesentliche Voraussetzungen und Randbedingungen einer **Implementierung** oder gar Inbetriebnahme noch gar nicht berücksichtigt werden (z.B. Bedienung der realen Wandlerperipherie, Forderungen nach hoher Zuverlässigkeit, Verfügbarkeit (Redundanz) und Effizienz, echt verteilte Systemzustandsinformation, Realzeit-Datenbasis für historische Prozeßdaten, siehe Tab. 3.1).
Wegen dieses Überraschungspotentials müssen die einzelnen Phasen - wie bei jedem Phasenmodell - i.a. **iterativ** durchlaufen werden.

(3) Trotzdem resultiert jede Phase in der Vorgabe einer ausführbaren Struktur für die gesamte Prozeßführungsaufgabe, z.B. als kooperierendes Prozeßsystem, und nicht in der Vorgabe von Anforderungen an das Ein-/Ausgangsverhalten, also einer **Spezifikation** im Sinne des requirement-engineering /ZAV 84/. Ohne auf die Vor- und Nachteile beider Ansätze einzugehen, wurde der operationale Ansatz aus pragmatischen Gründen gewählt, weil er durch die **Ausführbarkeit** der Entwürfe einer frühzeitigen Leistungsbewertung entgegenkommt. Beim requirement-engineering-Ansatz wird erst in der letzten Phase (Realisierung) das dynamische Verhalten sichtbar und bewertbar. Der interessierte Leser sei für eine weiterführende Diskussion auf die eingangs zitierten Quellen /ZAV 84/,/ZAS 86/ verwiesen; in /LFL 87/ finden sich z.T. ähnliche Überlegungen, wobei der Aspekt der Leistungsbewertung allerdings eine untergeordnete Rolle spielt.

## 3.3 Simulationskonzept

Ein Hauptmerkmal der Experimente ist die Kooperation realer Testobjekte mit einer simulierten Umgebung. Dieses Konzept wird nun genauer spezifiziert: in 3.3.1 wird die simulierte Umgebung (SU) durch die Übertragung einer vorgegebenen Ablaufsemantik kommunizierender Prozeßsysteme auf virtuelle Zeit eingeführt. In 3.3.2 wird der geschlossene Wirkungskreis aus einer simulierten Umgebung und einem realen Testobjekt (RTO) zunächst allgemein als gekoppeltes I/O-System definiert und eine Ablaufsteuerung hierfür angegeben.
In 3.4 wird das Konzept auf den Anwendungskontext zugeschnitten, in dem allein Echtteilesimulation i.f. interessiert: das RTO als Echtzeitrechensystem und die Ablaufsteuerung als Erweiterung der Echtzeitbetriebssysteme und der Rechnernetzkommunikation.

### 3.3.1 Verallgemeinerte prozeßorientierte Simulation

#### 3.3.1.1 Grundelemente der Simulation

Aufgrund der Erweiterungseigenschaft (T7) in 1.3 soll das Simulationskonzept von **gegebenen** Zielmaschinen ausgehen, sofern diese kommunizierende Prozeßsysteme in irgendeiner Form unterstützen, statt **eine** (spezielle) Simulationsmaschine **neu** zu definieren.
Ausgehend von Prozessen und Aktionen, deren normale (Echtzeit-)Semantik vorgegeben sei,. z.B. durch ein Hoare'sches Axiomensystem, wird die virtuelle Zeit, die eine (partielle) Ordnung der Aktonen definiert, als neue Prozeßkomponente spezifiziert, und die Wirkung der Aktionen auf sie durch neue Vor- und Nachbedingungen. Dazu dient ein allgemeines **Schema** mit 5 Klassen von Aktionen und Vorschriften zum Führen der virtuellen Zeit. Durch Anwendung dieses Schemas auf eine konkrete Zielmaschine (d.h. Einteilung ihrer Aktionen in die Klassen) und aus der normalen Ablaufsemantik erhalten wir die Semantik in virtueller Zeit (Abb. 3.7).
Aus einem BS-Kern, der die Echtzeit-Ablaufsemantik implementiert, entsteht später ein zugehöriger Simulator, der die virtuelle Zeit in geeigneter Weise im Prozeßzustandsmodell verankert (Kap. 4).
Ein Prozeß p ist der Ablauf eines sequentiellen Programms über einem privaten Adreßraum mit folgenden Komponenten:

c(p) $\in$ C(p) Kontrollzustand ("Befehlszähler")

d(p) $\in$ D(p) private dynamische Daten (Programmvariable)

tv(p) $\in$ T virtuelle Zeit

Prozeßzustände (Kontrollzustand, Variablenwerte, und auch die virtuelle Zeit) werden durch **Aktionen** transformiert, z.B. einfache Maschinenbefehle Operationen höherer nebenläufiger Programmiersprachen, BS-Dienste. Aktionen seien **atomar**. D.h. Aussagen über Prozeßzustände sind nur zu Beginn, oder, falls die Aktion terminiert, an ihrem Ende sinnvoll.
**Lokale** Aktionen eines Prozesses beeinflussen nur den eigenen Zustand, **globale** betreffen den

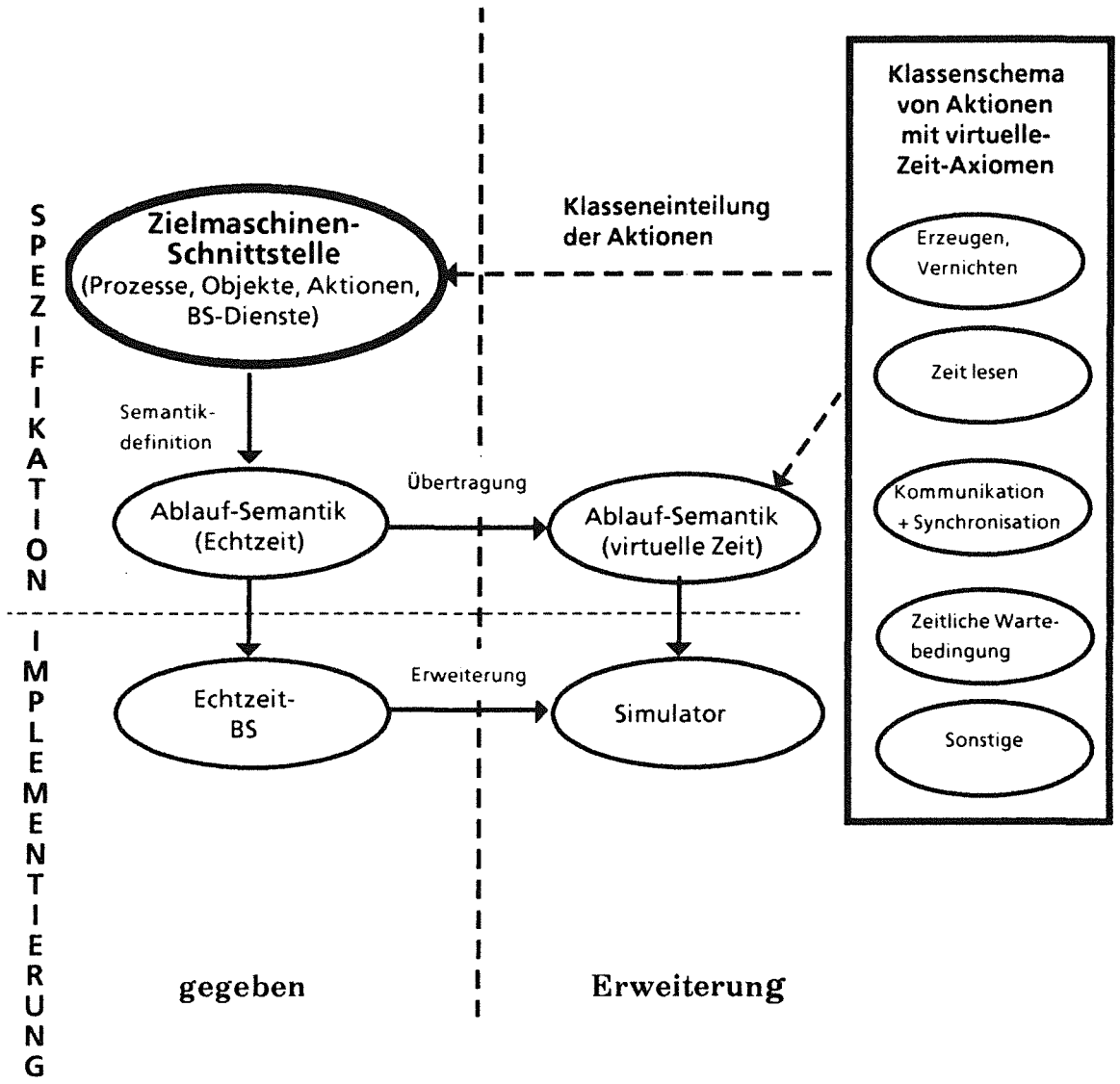



**Abb.** 3.7: Erweiterungseigenschaft

Zustand mehrerer Prozesse. Zur gegenseitigen Prozeßbeeinflussung stehen vor allem **Austauschobjekte** mit zugehörigen globalen Operationen zur Verfügung (3.1.2). Eine Vielzahl unterschiedlicher Arten von Austauschobjekten sind denkbar und i.f. zugelassen:

- Kommunikationsobjekte für Botschaftenaustausch (Operationen send__msg, rec__msg) mit
  - unterschiedlichen Synchronisationseigenschaften (z.B. synchrones, asynchrones, begrenzt asynchrones Senden)
  - unterschiedlichen Kommunikationstopologien;
    Kommunikationsobjekte können kommunizierenden Prozeßpaaren (p,q) fest zugeordnet sein wie in CSP (1:1-Kommunikation), oder jedem einzelnen Prozeß als privates Kommunikationsobjekt (n:1-Kommunikation, port-Kommunikation), oder unabhängig von Prozessen existieren (mailbox, n:m-Kommunikation);
- Entries für Rendezvous-Kommunikation (Operationen call, accept)
- reine Synchronisationsobjekte ohne Informationsaustausch (z.B. Semaphore, P- und V-Operationen)

- Monitore mit u.U. komplexen internen Wartebedingungen (Operationen signal, wait).

Im folgenden werden Programm-Annotationen ausgedrückt verwendet, also Prädikate über Prozeßzuständen vor und nach einer Aktion (vgl. z.B. /LAS 85/). Dabei bedeute

at (p, <a>) : p (bzw. c(p)) befindet sich unmittelbar vor oder in Aktion <a>

Wenn aus dem Zusammenhang klar ist, welcher Prozeß p gemeint und welche Aktion <a> als nächste auszuführen ist, wird abgekürzt:

$\{P\} \equiv (at(p, <a>) \Rightarrow P)$

wobei P ein beliebiges Prädikat, das nur von d(p),tv(p), aber nicht von c(p) abhängt.

{P} <a> {Q} bedeute:

wenn die Variablenwerte von Prozeß p unmittelbar vor Aktion <a> das Prädikat P erfüllen, so gilt Q unmittelbar nach <a>, falls <a> terminiert. In Annotationen zu lokalen Aktionen kommt nur der aufrufende Prozeß vor; in Annotationen für globale Aktionen i.a. mehrere Prozesse.
Die Wirkung von Prozeßaktionen auf die **virtuelle Zeit** wird nun beschrieben. Eine Aktion, die die virtuelle Zeit eines Prozesses nicht verändert, heißt im folgenden **zeittransparent** oder **zeitlos.**

### 3.3.1.2 Klassenschema der Aktionen

**Klasse (K1) Erzeugen, Initialisieren, Starten und Vernichten**
Obwohl man von einer statischen Prozeßmenge hätte ausgehen können, soll - auch mit Blick auf die spätere Realisierung - die dynamische Kreierung und Initialisierung eines Prozesses durch eine Operation explizit vorgesehen werden:

$\{tv(p) = T\}$
create__process(q, start__c, start__d)

q: Bezeichner für erzeugten Prozeß
start__c: Verweis auf die erste Aktion von q ("Einsprungadresse")
start__d: Startwerte für Variablen d(q)

$\{tv(p) = T \wedge tv(q) = T \wedge at(q, start\_\_c) \wedge d(q) = start\_\_d\}$

Die Startoperation ist also für den erzeugenden Prozeß p zeittransparent und überträgt dessen virtuelle Zeit auf q (T ist eine freie, implizit allquantifizierte Variable, d.h. das Prädikat gilt für beliebige Werte von T).

Die zugehörigen Vor- und Nachbedingungen für das **Vernichten** eines Prozesses q durch einen anderen p lauten

{tv(p) = T}

delete__process (q)

{tv(p) = T ∧ ¬ at(q, <a>) ∀ <a> }

Ähnliche Vor- und Nachbedingungen können für das Erzeugen und Verzichten von Austauschobjekten formuliert werden.

**Klasse (K2) Lesen der Zeit**

Ein Prozeß kann seine eigene virtuelle Zeit abfragen und z.B. für zeitabhängige Programmentscheidungen oder zur Zeitstempelung von Leistungsgrößen verwenden; diese Operation verläuft ebenfalls zeittransparent.

{tv(p) = T}

clock__read (m); m: lokale Zeit-Variable

{tv(p) = T ∧ m = T*)}

**Klasse (K3) Kommunikations- und Synchronisationsoperationen**

Zunächst wird das Grundprinzip der virtuellen Zeitführung für beliebige Austauschobjekte dargestellt. Im Detail werden dann die Annotationen für Botschaftenkommunikation angegeben. Ein Austauschobjekt K stelle Operationen $op_1,..,op_j$ zur Verfügung. An jede Operation $op_i$ ist eine logische Bedingung $c_i$ (u.U. ≡true) geknüpft, die zu ihrer Terminierung erfüllt sein muß. Z.B. muß eine Botschaft bestimmten Typs oder Absenders für eine Empfangsoperation vorliegen, oder ein empfangsbereiter Prozeß für eine Sendeoperation existieren, oder eine explizit in der Aktion angegebene Wartebedingung (z.B. beim Monitoraufruf) erfüllt sein. Die Operationen $op_i$ und Bedingungen $c_i$ können recht komplex sein, aber die Zuordnung der virtuellen Zeit verläuft immer nach folgendem Grundschema:

(a) Wenn für Prozeß p die Bedingung $c_i$ beim Aufruf von $op_i$ erfüllt ist, läuft opi zeitlos ab.

(b1) Wenn die Bedingung nicht sofort, aber später erfüllt wird, so kann dies nur infolge einer Operation $op_j$ eines anderen Prozesses q auf K geschehen, die selbst terminiert, weil ihre Bedingung $c_j$ erfüllt ist (Fall (a)). Dann wird Prozeß p mit der virtuellen Zeit des Prozesses q unmittelbar vor $op_j$ fortgesetzt.

*) Der Zweck der Annotationen besteht darin, die Wirkung von Aktionen auf möglichst direkte Weise zu **spezifizieren**. Will man auch **Programmbeweise** führen - was nicht Ziel dieser Arbeit ist - so empfiehlt es sich, wie in /LAS 85/ sog. **Satisfaktionsbedingungen** für die Aktionen (K1) bis (K5) für beliebige Vor- und Nachbedingungen P,Q aufzustellen. Um
{P} clock__read(m) {Q}
nachzuweisen, ist z.B. zu zeigen
$P \Rightarrow Q^m_{tv(p)}$
(Schreibweise: alle freien Vorkommnisse der Variablen m im Prädikat Q sind durch den Ausdruck tv(p) zu ersetzen). clock__read (m) entspricht also einer Zuweisung
m:= tv(p).

Ein solches Rendezvous ist möglich, wenn das Prädikat

$$\text{exchange}(j,q,i,p) \equiv (at(p,op_i(in',s)) \wedge tv(p) \leq tv(q) \wedge (c_i(in',s)^{(s,out)}_{op_j(in,s)}$$

für gewisse p,i erfüllt ist.

Die Wirkung der Operation $op_j$ sei dabei als Funktion $op_j : IN \times S \rightarrow S \times OUT$ beschrieben, wobei S ein geeignet gewählter Zustandsbereich von K und IN bzw. OUT Menge von Ein-/Ausgabeparametern der Operation auf K.

(b2) Wenn die Bedingung niemals erfüllt wird, dann ist die Zeit tv(p) nach $op_i$ undefiniert, weil $op_i$ nicht terminiert; es sei denn, eine endliche zeitliche Befristung t__d ist an den Wartezustand geknüpft. Läuft diese ab, wird der Aufrufer mit der virtuellen Zeit tv(p) + t__d und dem Resultat <timeout> fortgesetzt. Die Operation ist davon abgesehen wirkungslos.

Diese drei Fälle sind charakterisiert durch

(Fall a)

$$\{tv(q)=T \wedge c_j(in,s) \wedge \forall\, i,p: \neg \text{exchange}(j,q,i,p) \wedge P^{(s,out)}_{op_j(in,s)}\}$$

$$\frac{\text{Prozeß q}}{op_j(in,s)}$$

$\{tv(q)=T \wedge P\}$ P: beliebige Nachbedingung (Prädikat)

(Fall b1)

$$\{tv(q)=T \wedge c_j(in,s) \wedge \exists i,p: (\text{exchange}(j,q,i,p) \wedge P^{(s,out')}_{op_i(in',op_j(in,s))} \wedge Q^{(s,out)}_{op_j(in,s)}\}$$

$$\frac{\text{Prozeß q}}{op_j(in,s)} \qquad \frac{\text{Prozeß p}}{op_i(in',s)}$$

$\{tv(q)=T \wedge P\}$ $\{tv(p)=T \wedge Q\}$ Q: beliebige Nachbedingung

(Fall b2)

$$\{tv(q)=T \wedge \neg\, c_j(in,s) \wedge P^{out}_{<timeout>}\}$$

$$\frac{\text{Prozeß q}}{op_j(in,out,t\_\_d)}$$

$\{tv(q)=T+t\_\_d \wedge P\}$

Als konkretes Beispiel wird jetzt eine Mailbox mb mit folgenden Operationen betrachtet:

**rec__msg** (mb, m, P, t__d)
blockiert den Empfänger solange, bis eine Prädikat P erfüllende Nachricht msg eintrifft, die in m bereitgestellt wird, längstens aber für eine Zeitdauer t__d.

**send__msg** (mb, msg)
übergibt eine Nachricht msg an Mailbox mb, deren Kapazität auf cm Nachrichten mit $0 \leq cm \leq \infty$ begrenzt ist. Ein Sender werde blockiert, wenn bereits cm Nachrichten in mb vorhanden sind.

**Hilfsbezeichungen und -funktionen:**
$s_{mb} \in F(M)$ : Zustand von mb, wobei
M: Wertebereich der Nachrichten $\cup \{<\text{timeout}>\}$
F(M): Menge der Folgen mit Elementen in M
$|s_{mb}|$: Anzahl der Elemente der Folge $s_{mb}$
insert: $F(M) \times M \rightarrow F(M)$ Einfügen eines Elementes in Folge
remove: $F(M) \times M \rightarrow F(M)$ Ausfügen eines Elementes aus Folge
Psat: $F(M) \rightarrow \{true, false\}$,
$Psat(s_{mb}) = true <=> \exists\ m \in s_{mb}$ so daß P(m) gilt

Für das Ergebnis der receive-Operation gilt lokal aus der Sicht des Empfängerprozesses p:

$\{tv(p) = T\}$

rec__msg (mb,m,P,t__d)

$\{tv(p) \leq T + t\_\_d \wedge P(m) \vee tv(p) = t\_\_d \wedge m = <\text{timeout}>\}$

Mit Hilfe des Zustands $s_{mb}$ und der anstehenden Sendeoperationen an der Mailbox mb lassen sich die möglichen Austauschsituationen genauer charakterisieren.

### Empfangsoperation

Es existiert eine passende Nachricht in mb, aber kein wartender Sender (Fall (a))

$\{tv(p) = T \wedge Psat(s_{mb}) \wedge s' = s_{mb} \wedge \forall q: \neg\ at(q, send\_\_msg(mb,..))\}$

rec__msg (mb,m,P,t__d)

$\{tv(p) = T \wedge P(m) \wedge s_{mb} = remove(s',m)\}$

Es existiert ein an mb blockierter Sender q (Fall (b1))

$\{tv(p) = T \wedge (Psat(s_{mb})\ P \vee (msg)) \wedge s' = s_{mb}\}$ $\{tv(q) = T' \leq T\}$

rec__msg (mb,m,P,t__d) send__msg (mb,msg)

$\{tv(p) = T \wedge P(m) \wedge s_{mb} = remove(insert(s',msg),m)\}$ $\{tv(q) = T\}$

$s' = s_{mb}$ gilt nur, falls $\neg\ Psat(s_{mb})$!

### Sendeoperation

Es ist Platz in mb vorhanden, aber keine wartenden Empfänger, die mit Nachricht msg fortgesetzt werden können (Fall (a))

$\{tv(p)=T \wedge |s_{mb}|<cm \wedge s'=s_{mb} \wedge \forall q: \neg(at(q, rec\_msg(mb,P,..)) \wedge P(msg))\}$
send_msg (mb,msg)
$\{tv(p)=T \wedge s_{mb}=insert(s',msg)\}$

Es existieren wartende Empfänger q, die mit Nachricht msg fortgesetzt werden können (Fall (b2))

| | |
|---|---|
| $\{tv(p)=T \wedge P(msg) \wedge s'=s_{mb}\}$ | $\{tv(q)=T' \le T \le T'+t\_d\}$ |
| send_msg (mb, msg) | rec_msg (mb,m,P,t_d) |
| $\{tv(p)=T \wedge s_{mb}=remove(insert(s',msg),m)\}$ | $\{tv(q)=T \wedge P(m)\}$ |

**Kasse (K4) Explizite Zeitverzögerung**

Außer durch Wartebedingungen an Kommunikationsobjekten, kann die virtuelle Zeit eines Prozesses durch eine explizite Zeitverzögerung fortschreiten:

| | | |
|---|---|---|
| $\{tv(p)=T \wedge \Delta \ge 0\}$ | | $\{tv(p)=T\}$ |
| delay_for ($\Delta$) | oder | delay_until (T') |
| $\{tv(p)=T+\Delta\}$ | | $\{tv(p)=max(T,T')\}$ |

Es ist denkbar, die zeitliche Verzögerung eines Prozesses durch einen anderen Prozeß vorzeitig aufzuheben (dies ist dann eine globale Aktion).

| | |
|---|---|
| $\{tv(p)=T\}$ | $\{tv(q) \le T \le tv(q)+\Delta\}$ |
| reschedule (q) | delay_for ($\Delta$) |
| $\{tv(p)=T\}$ | $\{tv(q)=T\}$ |

**Klasse (K5) Alle übrigen Aktionen**

Für **alle** Aktionen <a>, die **nicht** (K1) bis (K4) angehören, gilt:

| | |
|---|---|
| $\{tv(p)=T \wedge wp(<a>,P)\}$ | wp(<a>,P): schwächste Vorbedingungen für <a>, mit |
| <a> | der die Nachbedingung P gilt ('weakest |
| $\{tv(p)=T \wedge P\}$ | precondition') |

Diese scheinbar triviale Forderung besagt, daß die virtuelle Zeit **nur** infolge einer der Wartebedingungen von (K3) oder (K4) fortschreiten kann, also unter vollständiger Kontrolle der Prozesse ist. <a> kann eine umfangreiche, den Prozeß p u.U. blockierende Operation sein (z.B. Anforderung von Arbeitsspeicher, Ein-/Ausgabeoperation, allgemein: betriebsmittel-belegende Operation).

### 3.3.1.3 Chronologische Reihenfolge der Aktionen

Mit (K1)-(K5) allein ist allerdings der korrekte Ablauf einer prozeßorientierten Simulation noch nicht **vollständig** beschrieben. Da die virtuelle Zeit die zeitliche Dynamik eines realen Systems nachbildet, muß das Ergebnis immer dem einer chronologischen Reihenfolge zumindest der

globalen Operationen auf Austauschobjekten, z.B. Senden und Empfangen von Nachrichten, entsprechen. Daher muß überall dort, wo aufgrund der bisherigen Bedingungen eine **globale** Aktion eines Prozesses p zur Zeit T - in ihrer **Nachbedingung** stehenden Zeit - terminieren könnte, folgende **zusätzliche "und"-verknüpfte Vorbedingung** gelten.

**(GC) Global chronologische Reihenfolge**

$\forall q \neq p$:

$\exists$ K,$op_i$: (at(q,$op_i$(in,out,t__d)) $\wedge$ $\neg c_i$(in,s) $\wedge$ tv(q) + t__d $\geq$ T)

( blockiert an Austauschobjekt mindestens bis Zeit T )

$\vee$ at(q, delay__for($\Delta$)) $\wedge$ tv(q) + $\Delta \geq$ T

(explizite Zeitverzögerung)

$\vee$ tv(q) $\geq$ T

( für alle sonstigen Operationen )

Dies ist zu beachten bei allen Operationen auf Austauschobjekten, Aufheben von Zeitverzögerungen, Löschen von Prozessen; ferner bei Zeitverzögerungen, wenn sie aufhebbar oder wenn Prozesse löschbar sind.

Durch (GC) wird gewährleistet, daß globale Operationen zu (K1), (K3) und ggf. (K4) jedes Prozesses p mit Zeit tv(p) = T erst **nach** allen Aktionen anderer Prozesse q mit tv(q) < T terminieren können. Z.B. kann ein Prozeß q erst dann vernichtet werden, wenn garantiert ist, daß er keine Aktionen mehr ausführen kann, die in virtueller Zeit vor dem Zeitpunkt T seiner Vernichtung liegen. Weil die virtuelle Zeit eines einzelnen Prozesses aufgrund (K1)-(K5) offensichtlich monoton wächst, terminieren insbesondere alle Operationen an ein und demselben Austauschobjekt, wenn überhaupt, in chronologischer Reihenfolge. Lokale Aktionen verschiedener Prozesse können dagegen in beliebiger Reihenfolge terminieren.

Für den oben betrachteten Spezialfall der Botschaftenkommunikation lauten die ersten beiden Zeilen von (GC) speziell

$\exists$ mb, P, t__d : rec__blocked (q,mb,P,t__d) $\wedge$ tv(q) + t__d $\geq$ T

v $\exists$ mb, msg : send__blocked (q,mb,msg)

mit

rec__blocked (p, mb, P, TIO) $\Leftrightarrow$ (at(p, rec(mb,m,P,TIO)) $\wedge$ $\neg$ Psat($s_{mb}$) $\wedge$
$\forall q'$: $\neg$ (at(q',send (mb,msg)) $\wedge$ P(msg)))

send__blocked (p, mb, msg) $\Leftrightarrow$ (at(p, send(mb,msg)) $\wedge$ $|s_{mb}| = cm$ $\wedge$
$\forall q'$: $\neg$ (at(q',rec(mb,m,P,TIO)) $\wedge$ (P(msg) $\vee$ Psat(mb))

#### 3.3.1.4 Diskussion

**(1) Modellierungssprachen**

Das Aktionenschema 3.3.1.2, 3.3.1.3 liefert eine allgemeine Vorschrift wie Prozeßsysteme für eine beliebige **Zielmaschine** (z.B. Echtzeit-BS) in virtueller Zeit auszuführen sind. Sie bedarf noch der konkreten Auslegung (vgl. 3.4.3.1). Z.B. könnte ein blockierender Betriebssystemdienst **wahlweise** (K5) (zeittransparente Aktion) **oder** (K3) (nicht zeittransparenter Synchronisationsdienst) zugeordnet werden.

Aber auch die zeitliche Semantik bekannter Simulationssprachen /FRA 77/ /ZEI 84a/ /OVN 85/ könnte mit (K1)-(K5), (GC) beschrieben werden. Das bekannteste Beispiel ist das Koroutinenmodell der PROCESS CLASSES von SIMULA /ROH 73/. SIMULA kennt allerdings keine echte Nachrichtenkommunikation, sondern nur die Synchronisation von Koroutinen. Da diese auf Variablen anderer Koroutinen direkt zugreifen können, ist die Bezeichnung "Prozeß" (privater Adreßraum!) für Koroutinen strenggenommen unzutreffend.

(K1) Erzeugen von Koroutinen
**ref** (B) A; ( B sei Unterklasse von PROCESS CLASS )
A:- **new**(B);
(K2) Uhrzeit lesen: **time**
(K3) Synchronisation: **passivate, activate, reactivate**
(K4) Zeitverzögerung, -rücknahme: **hold, cancel**

Weitere Beispiele von Diensten in Simulationssprachen finden sich in /FRA 77/,/OVN 85/ (zu (K4)), /BCM 87/ oder /WEM 88/ (MODSIM), wobei als Kommunikations- und Synchronisationsoperationen unter (K3) die Konstrukte

wait_until <boolesche Bedingung>
wait <timeout> for <message_type>

hervorzuheben sind. Simulationsspezifische Erweiterungen einzelner höherer Programmiersprachen, vor allem PASCAL und ADA, sind in diesem Zusammenhang auch zu nennen. /POH 87/ beschreibt z.B. die Semantik des ADA-Rendezvous (ohne zeitliche Befristung) in virtueller Zeit.

**(2) Verteilungsaspekt**

Bisher wurde nicht spezifiziert, ob die Prozesse quasiparallel oder echt parallel ausgeführt werden. In der Tat soll beides zugelassen werden, und die durch (K1)-(K5),(GC) spezifizierten zeitlichen Bedingungen müssen unabhängig davon erfüllt sein, wie die Prozesse auf Prozessoren verteilt sind.

**(3) Allgemeingültige Simulation und discrete-event-Systeme**

Durch (K1)-(K5),(GC) ist das Ein-/Ausgangsverhalten eines Prozeßsystems bei gegebenen Anfangszuständen im wesentlichen (bis auf die Reihenfolge zeitgleicher Aktionen) eindeutig

festgelegt.

Tatsächlich läßt sich jedes Prozeßsystem $(P_1,...,P_n)$ mit Operationen (K1)-(K5) durch ein verhaltensgleiches discrete-event-System DEVS, bzw. durch ein Netz von discrete-event-Komponenten $DEVN = (D_1,...,D_n)$ realisieren*).

Wichtig ist vor allem die umgekehrte Fragestellung, ob sich jedes DEVN bzw. DEVS durch ein verhaltensgleiches System von Prozessen mit den Mitteln einer gegebenen Zielmaschine, d.h. der zugehörigen Simulationssprache realisieren läßt.

**Def. 3.1 (Allgemeingültige prozeßorientierte Simulation)**

Eine prozeßorientierte Simulationssprache S heißt allgemeingültig, wenn jedes DEVN durch ein verhaltensgleiches endliches System von Prozessen in S darstellbar ist (vgl. (T8) in 1.3).

Dies gilt dann, wenn

- die virtuelle Zeit für jeden Prozeß lesbar ist (K2), **und**
- Warten auf ein Ereignis, z.B. Nachricht, zeitlich begrenzbar (K3), **oder**
  Warten auf Zeit durch andere Prozesse aufhebbar ist (K4).

Solche Dienste sind in vielen Echtzeit-BSen bereits vorhanden und dann nur von Echtzeit- in virtuelle-Zeit-Semantik zu übertragen (3.4.3).

Daß die genannten Dienste hinreichend sind, wird in 4.2.1.3 konstruktiv gezeigt, indem beliebige DEVNe mit ihrer Hilfe nachgebildet werden.

Daß sie auch notwendig sind, kann recht einfach durch ein Gegenbeispiel gezeigt werden: eine Sprache, die außer (K2) nur reine Botschaftenkommunikation mit unbefristetem Warten (K3) und eine nicht aufhebbare Zeitverzögerung HOLD (K4) enthält, deckt nur eine **echte Unterklasse** der DEVSe bzw. DEVNe ab. Danach sind viele der in der Literatur angegebenen Simulations-Erweiterungen von PASCAL, ADA oder SMALLTALK (z.B. /BEZ 87/), nicht

*) Kommunizieren die P1,...,Pn ausschließlich über Mailboxes, und definiert man für einen Prozeß p eine Menge von Mailboxes MB: = (mb(1),...,mb(k)) als Einzugsbereich, so kann man eine verhaltensgleiche discrete-event-Komponente d mit folgenden Ein-/Ausgabe- und Zustandsbereichen konstruieren:

$$X = Y = \prod_{i=1}^{k} PAR(i) \times \{1,..,n\}$$

PAR(i) Wertebereich aller Parameterwerte von Operationen an Mailboxes (einschließlich Operationsbezeichnung)

$$S = C(p) \times D(p) \times \prod_{i=1}^{k} S(i)$$

S(i) Zustandswertebereich der Mailbox mb(i)

Dies soll hier aber nicht weiter detailliert werden, man vgl. /ZEI 76/ Kap.9 oder /ZEI 84/ Kap.6 für ähnliche Konstruktionen.

allgemeingültig, aber z.B. die gängigen Simulationssprachen SIMULA (wegen der CANCEL- und REACTIVATE- Dienste) und auch GPSS oder SIMSCRIPT.

Die Allgemeingültigkeit gewährleistet also, daß der integrierte Simulationsansatz die Grundfähigkeiten konventioneller Simulationssprachen umfaßt (vgl. 2.3.3). Sie wird z.B. bei der Modellierung technischer Prozesse konkret benötigt.

**(4) Zusammenhang mit Lamport's Logischen Uhren /LAM 78/**

Wie verhält sich ein System kooperierender Prozesse in virtueller Zeit zu einem Prozeßsystem mit logischen Uhren ?

Zunächst sei eine partielle Ordnung "vor" ("->") der Aktionen der Prozesse wie in /LAM 78/ definiert. Ferner läßt sich jeder Aktion $<a>$ eines Prozesses p eine Uhrzeit $C(<a>)$ zuordnen:

$at(p,<a>) \Rightarrow C(<a>) = tv(p)$

Anhand (K1)-(K5) kann man leicht nachprüfen, daß dann auch

$a -> b \Rightarrow C(<a>) \leq C(<b>)$ (statt "<" wie in /LAM 78/)

gilt. "<" ließe sich ebenfalls erreichen, indem man z.B. fordert, daß nach jeder zeitlosen Operation eines Prozesses ein delay_for($\Delta$) mit $\Delta > 0$ stehen muß. (K1)-(K5) erfüllen damit die Konsistenzbedingungen für logische Uhren.

Dennoch besteht ein entscheidender Unterschied zwischen **prozeßorientierter Simulation** und einem Prozeßsystem mit **logischen Uhren**:

- logische Uhren sind Zeitmarken zu einem **gegebenen** Ablauf eines Prozeßsystems, die konsistent zur Halbordnung der Aktionen sein müssen.
  Es gibt i.a. viele, im Berechnungsergebnis verschiedene Abläufe eines Prozeßsystems mit denselben Zeitmarken.
- virtuelle Zeit dagegen **ordnet** selbst die Aktionen; **gesucht** sind genau die Abläufe, deren Berechnungsergebnis identisch dem einer **chronologischen** Ausführungsreihenfolge (GC) ist.

### 3.3.2 Echtteilesimulation in Nicht-Echtzeit-Umgebung

I.f. werden Experimente mit zwei gekoppelten Hauptkomponenten betrachtet:

- einem sog. **realen Testobjekt** (RTO) unbekannter interner Struktur, das als I/O-System beschreibbar sei
- einer **simulierten Umgebung** (SU) als kommunizierendes Prozeßsystem nach 3.3.1, also auch ein I/O-System, das eine zugehörige reale, in Echtzeit operierende Umgebung (RU) modelliere.

Ein Teil der Eingabe, die die Prozesse von SU über die Austauschobjekte beziehen, sind dabei Ausgabesignale des RTO, und ein Teil ihrer Ausgabe dient als Eingabesignale für das RTO. RTO

tausche mit SU im simulierten Wirkungskreis dieselben Arten von Daten und Signalen aus wie mit RU im realen WK (die Wertebereiche seien $Y_{ru}$ (RTO->(R- bzw. S)U) bzw. $Y_{ur}$ (R- bzw. S)U->RTO)). Die Kopplung zwischen RTO und RU bzw. SU besteht aus **zeitdiskreten** Signalverläufen:

$\omega_{ru} \in (Y_{ru},T)^e$ Ausgabe von RTO an RU im realen Wirkungskreis RR: = (RU,RTO)
$\omega_{ur} \in (Y_{ur},T)^e$ Ausgabe von RU an RTO im realen Wirkungskreis RR
$\omega_{rs} \in (Y_{ru},T)^e$ Ausgabe von RTO an SU im Experiment-Wirkungskreis RS: = (SU,RTO)
$\omega_{sr} \in (Y_{ur},T)^e$ Ausgabe von SU an RTO im Experiment-Wirkungskreis RS

**Def. 3.2 (Echtteilesimulation in Nicht-Echtzeit-Umgebung)**

Ein Experiment mit dem gekoppelten System RS : = (SU,RTO) heißt Echtteilesimulation in Nicht-Echtzeit-Umgebung (im Unterschied zur Echtteilesimulation in Echtzeitumgebung von 2.2.1.2), wenn gilt:

(R1) Die **reale zeitliche Dynamik** der Signalverarbeitung durch RTO, und nicht eine nachgebildete Zeit, bestimmt die zeitliche Komponente der Signale in einem Eingabeverlauf $\omega_{rs}$ für SU.

(R2) SU simuliert RU **nicht** in Echtzeit (nach Def. 2.6), d.h. die zeitliche Komponente eines Signals in $\omega_{sr}$ ist die virtuelle Zeit tv(p) des Prozesses p, der diese Ausgabe erzeugt, unabhängig von der Echtzeit, zu der sie erzeugt wird.

(R3) Die Simulationsergebnisse sollen übertragbar auf die Realität sein. Das bedeutet: wenn SU die Echtzeitumgebung RU als offenes I/O-System korrekt in virtueller Zeit nachbildet, dann simuliert RS auch korrekt das geschlossene System RR.

(R3) wurde bereits in 1.3 (T6) und 2.3.3 gefordert (Interferenzfreiheit).
I.a. simuliert SU die reale Umgebung des Einsatzes nicht exakt, sondern nur in vergröberter Form. SU wird später durch verfeinerte Modelle SU' ersetzt. Dadurch ändert sich i.a. das Gesamtsystemverhalten (RTO, SU'). (R3) garantiert nun, daß solche Änderungen dem veränderten **Ein-/Ausgangsverhalten** von SU' zuzuschreiben sind, und nicht zufällige Verhaltensänderungen desselben RTO unter verschiedenen Implementationen desselben Umgebungsmodells sein können.
Korrekte Simulationen SU derselben realen Umgebung RU unterscheiden sich in ihrem Echtzeitverhalten. Das Kernproblem, um (R3) zu erreichen, liegt also darin, das RTO invariant unter **Geschwindigkeitsunterschieden** der Umgebungen zu führen, mit denen es gekoppelt ist. Wenn seine Eingabeverläufe $\omega_{sr}$ (in RS), $\omega_{ur}$ (in RR) einander über eine bekannte Zeittransformation zwischen Echtzeit im Experiment und Echtzeit im Einsatz eindeutig entsprechen,

sollen auch die zugehörigen Ausgabeverläufe $\omega_{rs}$, $\omega_{ru}$ einander eindeutig über dieselbe Zeittransformation entsprechen.

Gegenbeispiele sind etwa Zeitverschiebungen von Signalen in $\omega_{sr}$ gegenüber $\omega_{ur}$, die zu (wertemäßig) verschiedenen Ausgaben $\omega_{rs}$, $\omega_{ru}$ führen können (z.B. "timeout"-Reaktionen), wenn das RTO von ihnen nichts weiß, oder verborgene Kopplungen zwischen RTU und RU bzw. SU, etwa über gemeinsame Betriebsmittel, deren Zustand durch die Schnittstelle $\omega_{ur}$, $\omega_{sr}$ gar nicht erfaßt wird. Um Probleme wie im ersten Beispiel zu vermeiden, muß die Testumgebung die Zeittransformation zwischen Realität und Simulationsexperiment **explizit** berücksichtigen. Ist SU "schneller" als RU, kann zwar die Simulation notfalls auf Echtzeit verzögert werden, ist SU aber "langsamer" als RU, muß das RTO entsprechend "gebremst" werden, z.B. durch Restaurieren früherer Zustände.

Um zu zeigen, daß (R1)-(R3) erfüllbar sind, wird ein operatives Modell einer Ablaufsteuerung für eine solche Echtteilesimulation skizziert. Dazu müssen RTO und SU folgende (systemtheoretisch allgemein gefaßte) Voraussetzungen (B1)-(B5) erfüllen.

(B1) Das RTO ist ein **zeitinvariantes** I/O-System (/ZEI 76/, Kap. 9.7). D.h. gegeben ein Anfangszustand s und ein beliebiger Eingangsverlauf $\omega$, ist der erreichte Endzustand unabhängig vom absoluten Anfangszeitpunkt von $\omega$.

$\omega:[t0,t1]->Y_{ur}$, $\omega':[t0+\tau,t1+\tau]->Y_{ur}$,

$\omega(t)=\omega'(t+\tau)\ \forall t\in[t0,t1] \Rightarrow \delta(s,\omega)=\delta(s,\omega')$

(B2) Der i.a. nicht direkt meßbare interne Zustand s des RTOzum Zeitpunkt t läßt sich bei bekanntem Eingangsverlauf $\omega_{sr}$ durch Beobachtung dsr Ausgangsverlaufs $\omega_{rs}$ für ein gewisses Zeitintervall [t,t+tb] **rekonstruieren**.
D.h. es existiert $tb>0$ und eine Funktion $B: (Y_{ru},tb)\times(Y_{ur},tb)->S$, so daß für alle s gilt:

$s(t)=B(\omega_{rs}|_{[t,t+tb]},\omega_{sr}|_{[t,t+tb]})$

(B3) Ein beliebiger Anfangszustand $s=s(t)$ von RTO läßt sich durch Anwenden eines geeigneten Eingabeverlaufs $\omega_{sr}$ in [t,t+tr] in jeden gewünschten Zielzustand s(t+tr) transformieren. D.h. es existiert $tr>0$ und eine Funktion $R: (Y_{ru},tr)\times S->(Y_{ur},tr)$, so daß für alle $s:=s(t)$, $s1:=s(t+tr)$ gilt:

$s1 = \delta(s, R(\omega_{rs}|_{[t,t+tr]},s1))$.

R ist eine Rückstellfunktion, die aus dem Ausgangsverlauf $\omega_{rs}$ den nötigen Eingangsverlauf $\omega_{sr}$ zur **Restaurierung** des gewünschten Zustands s berechnet.

Für den Spezialfall linearer, zeitinvarianter RTOe sind (B2) und (B3) äquivalent zur Zustands-Beobachtbarkeit und -Steuerbarkeit /FOE 78/, wobei $\omega_{rs}$ die Rolle der Meßgrößen und $\omega_{sr}$ der Stellgrößen spielt.

Von der simulierten Umgebung (SU) wird erwartet

(B4) wenn SU im Experiment eine korrekte und vollständige Darstellung $\omega_{ru}$ des Eingangsverlaufs in der Realität (RU) empfängt, so produziert SU bei gleichem Anfangszustand in endlicher Zeit auch eine korrekte und vollständige Darstellung $\omega_{ur}$ des

Ausgangsverlaufs , also

$\omega_{ur} = f_{RU}(\omega_{ru})\ \forall\ t0,t1, \forall\ \omega_{ru}:[t0,t1\ ] \mapsto Y_{ru}$,

wobei $f_{RU}$ die Ein-/Ausgangsfunktion des I/O-Systems RU in dem betrachteten Anfangszustand ist (vgl. Def. 2.2).

SU simuliert also RU als offenes System korrekt.

(B5) Für jedem Zeitpunkt t1 ermittelt SU in endlicher Zeit eine untere zeitliche Schranke tv_su>t1, wann frühestens die nächste Ausgabe von RU an RTO erfolgt, sofern keine Eingabe von RTO zuvorkommt:

$tv\_su = \min\{t > t1 | \omega_{ur}(t) \neq \varnothing \wedge \omega_{ru}|_{(t1,t)} = \varnothing|_{(t1,t)}\}$.

Für ein Prozeßsystem SU nach 3.3.1.3 errechnet sich tv_su z.B. wie folgt

$tv\_su := \min_{q} \{t(q)\}$

$$\text{mit } t(q) := \begin{cases} tv(q) + t\_d & \text{falls rec\_blocked (q,mb,P,t\_d)} \\ \infty & \text{falls send\_blocked (q,mb,msg)} \\ tv(q) + \Delta & \text{falls at(q, delay\_for}(\Delta)) \\ tv(q) & \text{sonst} \end{cases}$$

tv_su ist wegen der globalen Bedingung (GC) eine untere Schranke für den Terminierungszeitpunkt tv(p) der nächsten Aktion, insbesondere Ausgabeoperation, eines jeden Prozesses p, solange kein sende- oder empfangsblockierter Prozeß durch eine externe Kommunikationsoperation (Eingabe von RTO) zu einem Zeitpunkt $T \leq tv(p)$ deblockiert wird.

**Ablaufsteuerung**

Abb. 3.8a zeigt das Zustandsdiagramm dern Ablaufsteuerung mit vier Zuständen (Phasen), das i.f. diskutiert wird, Abb. 3.8b die entsprechenden zeitlichen Abläufe in Experiment und Realität.
te bezeichne im folgenden einen Zeitpunkt (Echtzeit) im Experiment (RS), tv den zugehörigen Zeitpunkt im realen Wirkungskreis (RR), ts:te→tv die zeitliche Zuordnung zwischen beiden.
Im folgenden werden die den $\omega_{rs}$, $\omega_{sr}$ in RS nach zeitlicher Transformation entsprechenden Verläufe $\omega'_{rs} \in (Y_{ru},T)^e$, $\omega'_{sr} \in (Y_{ur},T)^e$ in RR betrachtet. Zwischen ihnen besteht der Zusammenhang

$\omega'_{rs} \circ ts = \omega_{rs}$

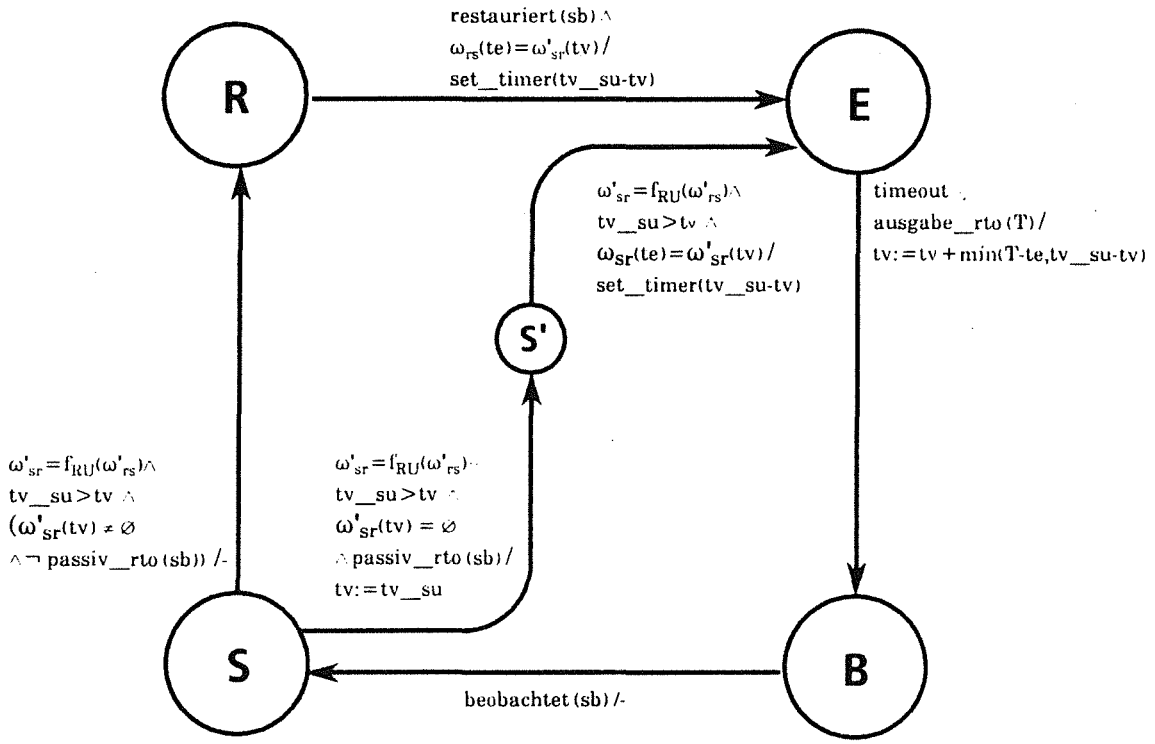
$\omega'_{sr} \circ ts = \omega_{sr}$.

Gewisse Zeitintervalle, die im Experiment nur der Simulation von SU oder der Beobachtung/Restaurierung des Zustandes des RTO dienen und während der tv konstant bleibt, werden so aus $\omega_{rs}$, $\omega_{sr}$ "ausgeblendet" und andere, während der RTO in Echtzeit arbeitet (R1) und ts eine Identitätsabbildung ist, unverändert nach $\omega'_{rs}$, $\omega'_{sr}$ kopiert. (R3) ist gezeigt, wenn stets

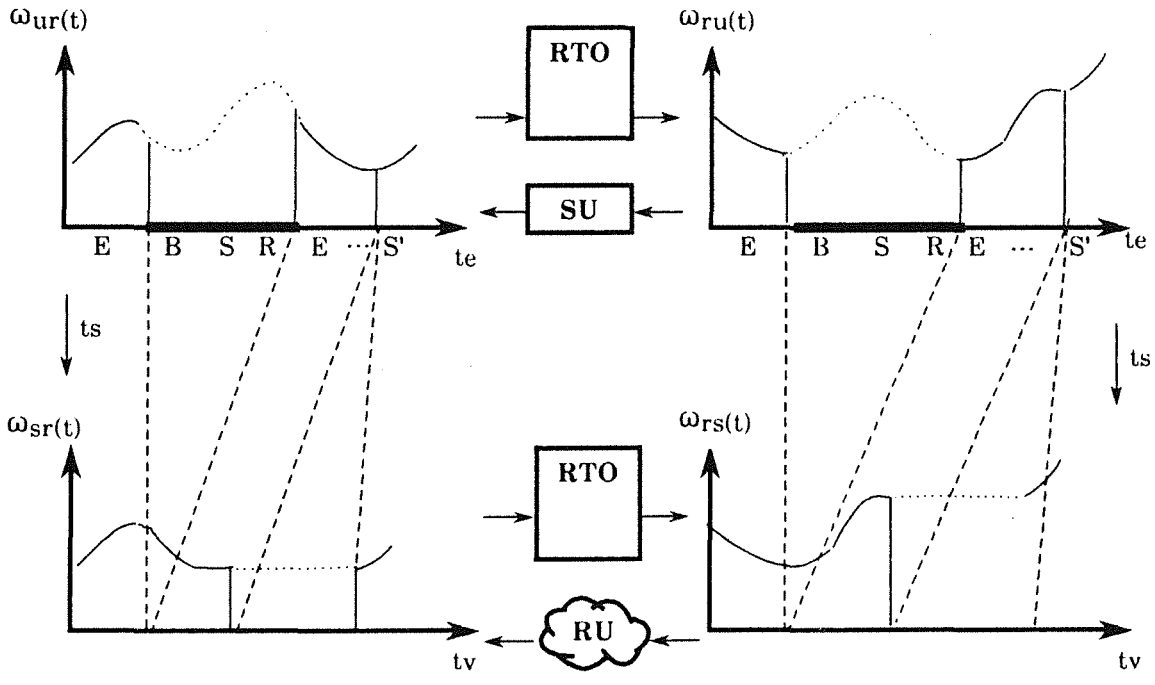
$\omega'_{rs} = \omega_{ru}$

$\omega'_{sr} = \omega_{ur}$,

also Übereinstimmung mit den tatsächlichenVerläufen im realen Wirkungskreis (RR) gilt.




a) Zustandsdiagramm




b) Zeitlicher Ablauf

**Abb.** 3.8: Prinzip einer Echtteile-Simulation in Nicht-Echtzeit-Umgebung

Zu Beginn des Experiments (tr: = t0) sollen folgende Voreinstellungen gelten:

- Zustand und Eingabe von SU repräsentieren RU zu Zeit t0:
  $s_{SU}(te) = s_{RU}(t0)$; $\omega_{rs}(te) = \omega_{ru}(t0)$ (also $\omega'_{rs}(t0) = \omega_{ru}(t0)$)
- Zeitschranke tv__su > t0 für die nächste Ausgabe von RU an RTO sei berechnet (B5).
- Der Zustand von RTO repräsentiert den in realer Umgebung zur Zeit t0:
  $s_{RS}(te) = s_{RR}(t0)$.
- RTO wird mit der Eingabe $\omega_{ur}(t0)$ beaufschlagt, d.h.
  $\omega_{sr}(te) = \omega_{ur}(t0)$ (also $\omega'_{sr}(t0) = \omega_{ur}(t0)$).

**Experimentphase (Echtzeitphase) (E)**

Die Experimentphase besteht darin, von nun an (Zeitpunkt te) RTO ohne Eingabe "laufen" zu lassen, bis entweder (vgl. E→B, Abb. 3.8a)

a) eine Zeitspanne der Dauer tv__su__tv in Echtzeit verstrichen ist (timeout), oder
b) Ausgabe (T) wahr wird, d.h. RTO zuvor eine Ausgabe erzeugt:
   $ausgabe(T) \equiv (te \leq T < te + tv\_\_su - tv) \wedge \omega_{rs}(T) \neq 0 \wedge \forall t, te \leq t < T: \omega_{rs}(t) = \varnothing$

In beiden Fällen wird - gemäß (R1) - tv genau synchron mit der in Echtzeit abgelaufenen Zeit $\Delta$ inkrementiert:

$\Delta = \min(T\text{-}te, tv\_\_su\text{-}tv)$.

An den $\omega_{rs}$ zugeordneten Verlauf $\omega'_{rs}$ wird das Stück $\omega_{rs|[te,te+\Delta]}$ angehängt. Es gilt:

$$\omega'_{rs}(t) = \begin{cases} \omega_{ru} & t \leq tv\text{-}\Delta \\ \varnothing & t\text{-}\Delta < t < tv \\ \omega_{rs}(t) & t = tv \end{cases}$$

Danach erfolgt ein Übergang in die Beobachtungsphase (B).

**Beobachtungsphase (B)**

tv ist in jedem der beiden Fälle a) und b) unter Phase E der letzte Zeitpunkt, bis zu dem die Ausgabe der SU (bzw- RU) bekannt ist. Die Regie sollte daher wieder an SU übergehen, um den weiteren Verlauf von $\omega_{sr}$ ab tv zu berechnen. Da dies nicht in Echtzeit geschieht (R2), werden Zustands- und Ausgabeverlauf des RTO unter (RS) von nun an anders aussehen als unter (RR). Um so wichtiger ist es, den letzten für (RR) gültigen Zustand $sb: = s_{RS}(te) = s_{RR}(tv)$ zu konservieren. Wegen (B2) gelingt dies durch Beobachtung von $\omega_{rs}$ für tb Zeiteinheiten (die Eingabe $\omega_{sr}$ für RTO sei währenddessen leer).

Am Ende gilt das Prädikat

$beobachtet(sb) \equiv (sb = B(\omega_{rs}|_{[te\text{-}tb,te]}, \varnothing|_{[te\text{-}tb,te]}))$.

**Simulationsphase (S)**

SU verarbeitet zunächst $\omega'_{rs}|_{[tv-\Delta,tv]}$ zu $\omega'_{sr}|_{[tv-\Delta,tv]}$. hier kann aufgrund der Definition von tv__su und der Bedingungen a) und b) der vorangegangenen Experimentphasen, wenn überhaupt, nur am letzten Zeitpunkt tv ein echter (nichtleerer) Ausgabewert hinzukommen. Nachdem SU eine neue Schranke tv__su > tv (Vorausschau) für die nächste Ausgabe an RTO berechnet hat, wird im **Normalfall** der Zustand sb restauriert (S→R) und mit der letzten Ausgabe $\omega'_{sr}(tv)$ eine neue Echtzeitphase für RTO begonnen (R→E).

In einem **Sonderfall** kann der gesamte nächste Zyklus S→R→E→B→S→R **übersprungen** und direkt tv: = tv__su gesetzt werden (S→S' in Abb. 3.8a), wenn nämlich

a) $\omega'_{sr}(tv) = \emptyset$ (Keine Eingabe für RTO)

b) passiv__rto(sb) $\equiv (\delta_{RTO}(sb,\emptyset|_{[tv,t]}) = sb \; \forall \; t \geq tv)$,
also der Zustand von RTO **ohne** externe Eingabe unverändert bleibt.
Denkbar ist z.B., daß RTO im Zustand sb zeitlich unbegrenzt auf einen Auftrag wartet.I.a. aber ist die Bedingung passiv__rto schwer nachzuprüfen, da der Zustand eine modellhafte Abstraktion darstellt und nicht direkt meßbar ist.

**Restaurationsphase (R)**

Der rekonstruierte Zustand sb von RTO wird, unabhängig vom momentanen Zustand, durch einen geeigneten Eingabeverlauf $\omega_{sr}$ wiederhergestellt. Dazu wird nur der Ausgabeverlauf $\omega_{rs}$ benötigt. Sobald

restauriert(sb) $\equiv (\omega_{sr}|_{[te-tr,te]} = R(\omega_{rs}|_{[te-tr,te]}, sb))$,

gilt wegen (B3) $s_{RS}(te) = sb = s_{RR}(tv)$.

RTO erhält nun die letzte Eingabe $\omega_{sr}(te) = \omega'_{sr}(tv)$ und läuft in Echtzeit weiter, wobei ein Wecker für die Dauer des Vorhersagezeitraums von SU gestellt wird (set__timer(tv__su-tv) in Abb. 3.8a, R→E und S'→E).

Den Zustandsübergängen lassen sich folgende Invarianten zuordnen
(ohne formalen Beweis, der aber mit Hilfe der Zeitinvarianz von RTO (B1), der Automateneigenschaft von I/O-Systemen (Def. 2.2) und der Minimalität von tv__su relativ einfach zu führen ist).

(tv' Hilfsvariable, die den letzten Wert vor einer Änderung von tv festhält)

{I1} ≡ ( R→E, S'→E )

$\omega'_{rs}|_{[t0,tv]} = \omega_{ru}|_{[t0,tv]} \wedge \omega'_{sr}|_{[t0,tv]} = \omega_{ur}|_{[t0,tv]}$)
$\wedge\, s_{RS}(te) = s_{RR}(tv) \wedge \omega_{sr}(te) = \omega_{ur}(tv)$
$\wedge\, tv\_\_su = \min\{t > tv \mid \omega_{ur}(t) \neq \varnothing \wedge \omega_{ru}|_{(tv,t)} = \varnothing|_{(tv,t)}\}$
$\wedge\, tv' = tv$

{I2} ≡ ( E→B )

$tv' < tv \leq tv\_\_su$
$\wedge \omega'_{rs}|_{[t0,tv]} = \omega_{ru}|_{[t0,tv]} \wedge \omega'_{sr}|_{[t0,tv']} = \omega_{ur}|_{[t0,tv']}$)
$\wedge\, s_{RS}(te) = s_{RR}(tv)$
$\wedge\, \omega_{rs}|_{(te-\,tv+tv',te]} = \omega_{ru}|_{(tv',tv]}$
$\wedge\, \omega_{rs}(t) = \varnothing\ \forall t: tv' < t < tv$

{I3} ≡ ( B→S )

$tv' < tv \leq tv\_\_su \wedge sb = s_{RR}(tv)$
$\wedge \omega'_{rs}|_{[t0,tv]} = \omega_{ru}|_{[t0,tv]} \wedge \omega'_{sr}|_{[t0,tv']} = \omega_{ur}|_{[t0,tv']}$)
$\wedge\, \omega'_{rs}(t) = \varnothing\ \forall t: tv' < t < tv$

{I4} ≡ ( S→R, S→S' )

$tv' < tv < tv\_\_su \wedge sb = s_{RR}(tv)$
$\wedge \omega'_{rs}|_{[t0,tv]} = \omega_{ru}|_{[t0,tv]} \wedge \omega'_{sr}|_{[t0,tv]} = \omega_{ur}|_{[t0,tv]}$)
$\wedge\, \omega'_{rs}(t) = \varnothing\ \forall t: tv' < t < tv$
$\wedge\, tv\_\_su = \min\{t > tv \mid \omega_{ur}(t) \neq \varnothing \wedge \omega_{ru}|_{(tv,t)} = \varnothing|_{(tv,t)}\}$

Aus (I1) folgt durch Induktion über die Anzahl der Zustandsübergänge nach R, daß die Verläufe $\omega'_{rs}$ und $\omega_{ru}$, $\omega_{sr}$ und $\omega_{ur}$ bis zu jedem beliebigen Zeitpunkt tv übereinstimmen, also die Übertragbarkeit der Ergebnisse (R3).

**Bemerkungen**

(1) Die skizzierte Ablaufsteuerung ist die wohl einfachste, die (R1)-(R3) erfüllt, aber keineswegs die einzig mögliche. Eine Alternative wäre z.B. ein echt paralleler Ablauf von Simulationsphase S und Experimentphase E; dies setzt voraus, daß außer den früheren Zuständen von RTO auch Eingabehistorien für RTO gespeichert und bei Restauration des Zustands von RTO echtzeitgetreu wieder abgespielt werden. In der Ablaufsteuerung in Abb. 3.7a ist das nicht erforderlich, weil sowohl RTO als auch SU immer nur bis zum nächsten Interaktionszeitpunkt laufen.

(2) Das geschilderte Verfahren soll den Blick dafür schärfen, was eine Integration **beliebiger** realer Testobjekte, z.B. auch technischer Prozesse in eine Nicht-Echtzeitumgebung erfordert. Praktikabel ist das Verfahren für solch allgemeine Systeme i.a. natürlich nicht, denn:

- Mit Ausnahme der Beobachtbarkeit und Steuerbarkeit von linearen, zeitinvarianten Systemen gibt es keine effektiven, d.h. algorithmisch durchführbaren Verfahren für die Zustandsrekonstruktion/-restauration. Dies gilt bereits für nichtlineare DGSe und erst recht für allgemeine I/O-Systeme.
- Selbst wenn (B2),(B3) erfüllt sind, können während der Beobachtungs- und Simulationsphasen unerwünschte oder sogar gefährliche Betriebszustände im technischen Prozeß erreicht werden, die durch zusätzliche Stellgrößentrajektorien während der Umgebungssimulation beherrscht werden müßten.
- Unbekannte externe Störgrößen, die auf RTO einwirken, können die Zustands-rekonstruktion/-restauration ungenau oder unmöglich machen.
- Das Verfahren ist insbesonders bei feiner zeitlicher Granularität der Ereignisse in SU sehr aufwendig und ineffizient.

Ganz anders sieht es hingegen aus, wenn RTO **anhaltbar** bzw. **fortsetzbar** ist, d.h. wenn es zwei ausgezeichnete Kontrollsignale c__a (Anhalten), c__f (Fortsetzen) in $Y_{ur}$ gibt, so daß gilt

$\delta(s, w\_\_a) = s \; \forall s \in S, w\_\_a:[t0,t1] \rightarrow Y_{ur}$ mit

$$w\_\_a(t) = \begin{cases} c\_\_a & \text{falls} \quad t = t0 \\ \text{beliebig} \neq c\_\_f & \text{sonst} \end{cases}$$

$\delta(s, w\_\_f) = \delta(s, w) \; \forall s \in S, w\_\_f, w:[t0,t1] \rightarrow Y_{ur}$ mit

$$w\_\_f(t) = \begin{cases} c\_\_f & \text{falls} \quad t = t0 \\ w(t) \text{ (beliebig)} & \text{sonst} \end{cases}$$

Das Verfahren der Ablaufsteuerung vereinfacht sich durch den Wegfall der Phasen B, R, aber die Beweistechnik bleibt dieselbe.

Während die Anhaltbarkeit für kontinuierliche Systeme noch fragwürdiger erscheint als (B2) und (B3), ist sie offensichtlich für **Rechensysteme** mit Hilfe von HW-und SW-Unterbrechungen und weiterer Maßnahmen sehr gut erfüllbar.

## 3.4 Grobarchitektur der Experimentumgebung

### 3.4.1 Zielprozesse und Wirtsprozesse

Das Konzept der Echtteilesimulation wird nun auf Echtzeit-Rechensysteme als reale Testobjekte (zentrale oder verteilte Zielmaschine + BS-Kern + Anwendung, vgl. 3.1.2) mit folgenden Ein-/Ausgangs- und Zustandsgrößen angewandt (vgl. 3.1.1, Abb. 3.1).:

I : (Eingangsgrößen, -signale)
z.B. Meßwerte yy vom technischen Prozeß, Unterbrechungssignale ya von E/A-Geräten, Nachrichten/Aufträge k von anderen Subsystemen des PFS, Bedieneranforderungen s

O : (Ausgangsgrößen, -signale) z.B. Stellwerte uu an den technischen Prozeß, Meßwertanforderungen yr und sonstige Geräte-Aufträge, Nachrichten/Aufträge k an andere Subsysteme des PFS, Bedienerausgaben b

S : (Zustandsgrößen)
z.B. Werte sämtlicher rechnerinterner Variablen und Zustände der E/A-Geräte.

Ob die deterministische Sichtweise eines I/O-Systems, daß jedem Anfangszustand s und Eingabeverlauf $x \in (I,T)$ eindeutig ein Ausgabeverlauf $y = F(s,x) \in (O,T)$ entspricht, angemessen ist, kann nicht allgemeingültig beantwortet werden. In der Praxis kommt es darauf an, möglichst viele Einflußgrößen auf den Ablauf des realen Testobjekts, die man bei vergröberter Modellierung als "stochastisch" annehmen würde, **explizit** W in I,O oder S aufzunehmen und im Experiment zu kontrollieren, um dadurch die o.g. Idealisierung möglichst gut anzunähern.
Da nach 3.1.2 und 3.3.1 sowohl das RTO als auch das abstrakte Modell der SU als kooperierende Prozeßsysteme vorliegen und beide in einem ablauffähigen Experiment auf BS-Prozesse einer unterliegenden Ablaufmaschine abgebildet werden, liegt folgendes hybrides Prozeßkonzept nahe.

**Def. 3.3 (Ziel- und Wirtsprozesse)**

Ein **realer Zielprozess (R-Prozess)** $(p, Z_M)$ ist ein als reales Testobjekt nach 3.3.2 zu bewertender Rechenprozeß p mit $Z_M$ als Zielmaschine (BS-Kern).

Ein **Wirtsprozeß (W-Prozeß)** p ist ein ablauffähiger Rechenprozeß, dessen Verhalten durch die prozeßorientierte Simulation nach 3.3.1 beschrieben ist, auf einer beliebigen Ablaufmaschine.

Beim Zielprozeß ist der Bezug zur Zielmaschine wesentlich, da sein Zeitverhalten von der Zielmaschine abhängt, während das Zeitverhalten eines Wirtsprozesses allein durch die Semantik der Systemdienste (K1)-(K5),(GC) (3.3.1) gegeben ist, also nicht von der konkreten Simulationsmaschine abhängt.

**Der erweiterte Systemkern für die Experimentumgebung** ist die Instanz, die die Echtteilesimulation nach 3.3.2 steuert, d.h.

a) Ausführung aller R-Prozesse als reales Testobjekt
b) korrekte Simulation der W-Prozesse

c) Realisierung der Kommunikation und Synchronisation zwischen a) und b), die sich aus dem Nicht-Echtzeit-Charakter von b) ergibt.

Für a) wird die Zielmaschine des realen Testobjekts als Teilmenge benötigt, während b) und c) durch die Erweiterungen zu leisten sind.

Sowohl die Menge der R-Prozesse kann leer sein (reine Simulation), als auch die der W-Prozesse (reine Echtzeitumgebung). Sind in einem Experiment beide nichtleer, so ist die Schnittstelle der R-Prozesse zumindest zu Teilen ihrer externen Echtzeitumgebung ersetzt durch Schnittstellen zu den W-Prozessen. Diese besteht aus speziellen Kommunikationsobjekten, genannt **RW-Kommunikationsobjekten**. Sie repräsentieren entweder die Kommunikation mit anderen Rechenprozessen auf demselben oder einem anderen Rechner, oder dem technischen Prozeß, also letztlich

- lokale Prozeßkommunikation
- E/A-Gerätekommunikation, speziell Gerätebeauftragung, Warten auf Geräterückmeldung sowie Empfangen asynchroner Unterbrechungssignale.

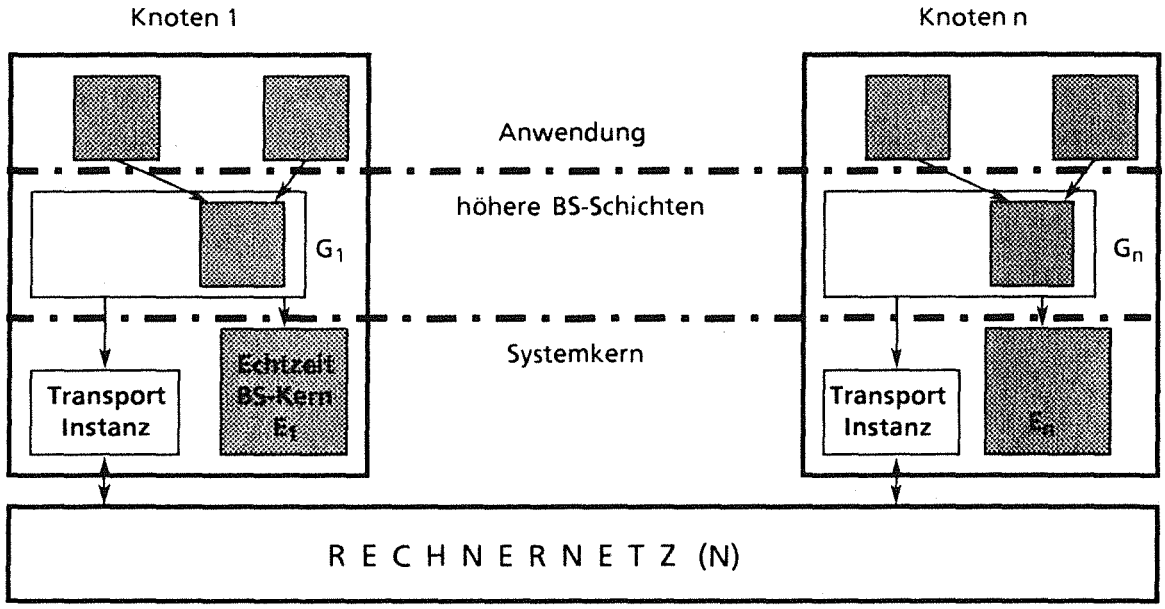
### 3.4.2 Hauptkomponenten und Schnittstellen des Systemkerns (horizontale Architektur)

I.a. ist das reale Testobjekt auf mehrere Rechner verteilt. Ein verteiltes Zielsystem ist nach 3.1.2 grob gegliedert in

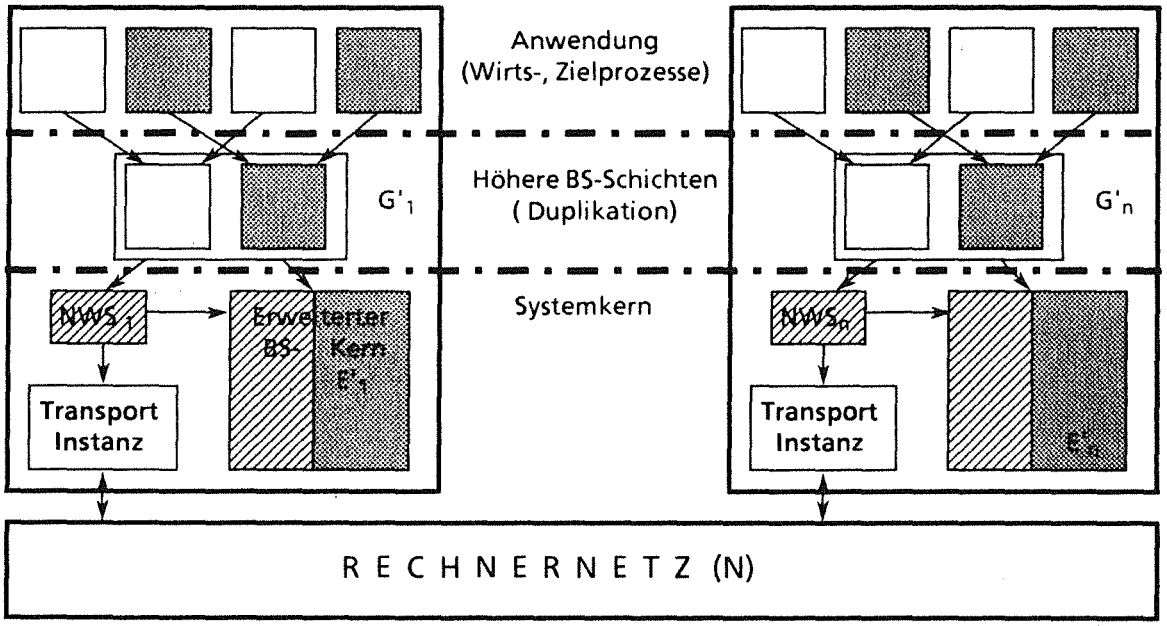
- einen Monoprozessor-Echtzeit-BS-Kern (E) für jeden Rechner,
- in Kommunikationsmedium mit Netzzugang (N) in jedem Rechner für den physikalischen Nachrichtentransport zwischen beliebigen Rechnern,
- die höheren Betriebssystem-Schichten (G) mit Kommunikationsprotokollen, Dateiverwaltung, ggf. netzweiten BS-Diensten, Sprach-Laufzeitsystemen, die (E) und (N) benutzen und ihrerseits die verteilten Anwendungen unterstützen (Abb. 3.9a).

Die verteilte Experimentumgebung besteht aus folgenden Hauptkomponenten (Abb. 3.9b), von denen die ersten beiden den Systemkern bilden:

- **Erweiterungen** (E') aller **Monoprozessor-BS-Kerne** (E) zur integrierten Simulation
  Die **Schnittstelle** von E' besteht aus
  (a) der Schnittstelle von E
  (b) zusätzlichen Systemdiensten zur prozeßorientierten Simulation (3.3.1)
  (c) zusätzlichen Systemdiensten für RW-Kommunikationsobjekte (3.4.1)
- Instanzen für die zeitliche Synchronisation der Rechner, genannt **Netzwerksynchronisation** (NWS), die sowohl das Rechnernetz (N) benutzen, als auch eng mit den lokalen BS-Kernen (E') kooperieren.
  Die **Schnittstelle** der NWS für die höheren Schichten ist identisch mit der Netzzugangs-Schnittstelle von (N): Senden und Empfangen von Dateneinheiten an beliebige, über das Netz erreichbare Rechner. Die neuen Dienste für die integrierte Simulation konzentrieren




a) Verteiltes Zielsystem




b) Erweiterung zum verteilten Simulations- und Zielsystem

NWS = Netzwerksynchronisation

reales Testobjekt (R-Prozesse)

simulierte Umgebung (W-Prozesse)

Systemkernerweiterungen

**Abb.** 3.9: Horizontale Architektur zur integrierten Simulation

sich auf den Monoprozessor-BS-Kern. Die NWS erfüllt nur eine für ihre Benutzer transparente Synchronisationsaufgabe.

- höheren Betriebssystem-Schichten (G'), die ihren unterschiedlichen Benutzern (R- und W-Prozessen) die Funktionen von G, und ggf. auch die zusätzlichen Schnittstellendienste von (E') für die prozeßorientierte Simulation und RW-Kommunikation auf Netzebene anbieten. Z.B. könnten die in 3.3.1 besprochenen Kommunikationsdienste unter Erhaltung ihrer zeitlichen Semantik von rechnerlokalen auf globale, lokations-transparente mailboxes ausgedehnt werden. G' nimmt dieselben Rechnernetzdienste in Anspruch wie G, aber über die NWS statt über N.
- einem übergeordneten Entwicklungs- und Auswertungssystem (3.5).

Das **Hauptproblem** für E' besteht in der korrekten Ausführung des realen Testobjektes, das im Experiment mit den W-Prozessen um gemeinsame Rechner-Betriebsmittel (CPU, ASP und lokale E/A-Geräte) konkurriert, im Echtzeiteinsatz dagegen nicht.
**Erleichternd** wirkt sich die Tatsache aus, daß E' für seinen begrenzten Einflußbereich Monoprozessor eine vollständige Sicht und Kontrolle der Prozeß- und BM-Zustände ausübt, daß eine zentrale Zeitführung realisierbar ist und daß - abgesehen von E/A-Gerätetätigkeit - nur eine Pseudoparallelität der Abläufe besteht.
Das **Hauptproblem** für die NWS besteht in der zeitlichen Synchronisation autonomer echt verteilter R- und W-Prozesse nach der globalen Bedingung (GC) in 3.3.1, wobei keine zentrale Uhr und keine zentrale Kontrolle der Prozeßzustandsinformation existiert.
**Erleichternd** wirkt sich aus, daß Interferenzprobleme zwischen R- und W-Prozessen auf unterschiedlichen Rechnern nicht auftreten, weil die Rechnerbetriebsmittel bis auf das Kommunikationsmedium selbst disjunkt sind (in 3.1.2 wurde von gemeinsamem Speicher abgesehen!). Daher ist es für einen kommunizierenden Prozeß letztlich belanglos, ob der abgesetzte Kommunikationspartner eine rein simulierte Umgebung oder eine mit realen Testobjekten darstellt.
Bemerkenswert ist schließlich, daß die Schicht (G') in einfachen Fällen keinen zusätzlichen Entwicklungs- und Implementierungsaufwand gegenüber (G) bringt. Wenn diese Schichten durch Dienstleistungsprozesse realisiert sind, müssen diese z.B. nur dupliziert werden (R-Dienstleistungsprozesse für R-Prozesse als Benutzer, W-Dienstleistungsprozesse für W-Benutzer).

Die Architektur der verteilten Experimentumgebung nutzt also die der zugrundeliegenden verteilten Zielmaschine entscheidend aus.

In den folgenden beiden Abschnitten 3.4.3, 3.4.4 werden die Anforderungen an die Hauptkomponenten des Systemkerns, E' und NWS, beschrieben. Auf die Benutzung durch die höheren Schichten in den unterschiedlichen Entwicklungsstadien einer Echtzeitanwendung wird in 3.4.5 eingegangen.

### 3.4.3 Erweiterter Monoprozessor-BS-Kern

#### 3.4.3.1 Anforderungen an die Systemdienste

**a) R-Prozesse**

(AR1) Die für die Prozesse unter E vorhandenen Leistungen, d.h. die Objekte und Operationen bzw. Systemdienste, sind als die für die R-Prozesse unter E' gültigen zu übernehmen (Erweiterungseigenschaft).

Eine grobe Abgrenzung dieses Funktionsumfangs wurde in 3.1.2 (2)(b) gegeben, man vgl. hierzu auch /WET 84/,/LEV 81/,/FAE 79/ oder Unterlagen realisierter BSe, z.B. /INT 85a/, RSX11S, THOTH /CMM 79/, MOBS /SLR 81/ etc. Mit E' sollen ja alle Zielsysteme als RTO bewertet werden können, die für E entwickelt werden.

R-Prozesse benötigen ein "Fenster", über das im Experiment die Interaktionen mit der simulierten Umgebung erfolgt: RW- Kommunikationsobjekte.

(AR2) RW-Kommunikationsobjekte müssen außer dem Datenaustausch die wesentlichen Synchronisationseigenschaften*) bei der Kopplung von Rechenprozessen, z.B. Geräte- oder Protokolltreiber, mit ihrer realen Umgebung (technischer Prozeß, abgesetzter Kommunikationspartner) über Peripheriegeräte nachbilden können. Sie stellen die einzige mögliche Interaktion zwischen den R- und W-Prozessen dar.

**b) W-Prozesse**

Zu klären sind folgende Fragen:

- Welche der in E vorhandenen Leistungen von E werden auf W-Prozesse übertragen?
- Welche zeitliche Semantik wird den Systemdiensten gegeben?
- Sind über die Leistungen von E und (AR2) hinausgehende Systemdienste für W-Prozesse vorzusehen, und welche?

Dazu

(AW1) Alle in E vorhandenen, für einen sicheren und komfortablen Mehrprozeßbetrieb notwendigen Leistungen (genannt **Basisdienste**) müssen auch den W-Prozessen zur Verfügung stehen. Dazu zählen insbesondere die Verwaltung und (dynamische) Zuteilung der Rechnerbetriebsmittel wie Arbeitsspeicher, Hintergrundspeicher oder Ein-/Ausgabegeräte, sowie das dynamische Erzeugen und Vernichten von Prozessen.

(AW2) Jeder Systemdienst für W-Prozesse wird entweder in die Klasse (K4) der Zeitverzögerungen, oder in die Kommunikations- und Synchronisationsoperationen (K3) oder die zeitlos verlaufenden Aktionen (K1),(K2) und (K5) von 3.3.1 eingeteilt und seine zeitliche Semantik so spezifiziert. Sende- und Empfangsoperationen auf RW-

*) z. B. synchrone, asynchrone begrenzt asynchrone Kommunikation, Verhalten bei Überlastung des Kommunikationskanals (vgl. 4.1.3)

Kommunikationsobjekten gehören in (K3), lokale Rechenoperationen ohne BS-Interaktion verlaufen immer, die Inanspruchnahme von ASP und E/A in der Regel zeitlos (K5).

(AW3) Zusätzliche Dienste sind genau dann erforderlich, wenn das Simulationskonzept nach Anwendung von (AW1), (AW2) noch nicht allgemeingültig (3.3.1, Def. 3.1) ist; hinzuzufügen sind ggf. zeitlich begrenztes Warten auf Ereignisse oder vorzeitig aufhebbare zeitliche Wartebedingungen.

Die Vorschrift (AW1) läßt noch einen gewissen Spielraum, denn i.a. verfügt ein Echtzeit-BS-Kern über Dienste, die über die Basisdienste hinausgehen, aber für eine allgemeingültige Simulation nicht notwendig sind. Hierzu zählen **realzeit-orientierte Dienste**, die speziell für die Effizienz und das Echtzeitverhalten einer auf BS-Kern E ablaufenden Echtzeitanwendung wichtig sind, z.B. optimierte Formen der Prozeßkommunikation und -synchronisation (wie Ereignissynchronisation oder kritische Abschnitte (regions, /INT 85a/), oder Dienste, die die Prioritäten und die Zuteilung (scheduling) von Prozessen beeinflussen. Diese auf W-Prozesse und virtuelle Zeit zu übertragen, hat Vor- und Nachteile:

- Dafür spricht, daß die strikte Anwendung der Erweiterungseigenschaft genau dies fordert. Über ein und demselben Quellprogramm können dann wahlweise R- oder W-Prozesse erzeugt werden. Dies erleichtert z.B. die Duplizierung von Dienstleistungsprozessen (vgl. 3.4.2, Schicht G), aber auch den nahtlosen Übergang von PFS-Modellen zu realen Testobjekten: **jedes** Echtzeitprogramm, das Dienste von E benutzt, ist auch ein zulässiges Modell, d.h. kann als W-Prozeß ausgeführt werden (vgl. 2.3.3).
- Dagegen spricht, daß die Übertragung der Semantik solcher Dienste auf virtuelle Zeit zu unerwarteten oder wenig sinnvollen Resultaten führen kann. Dies gilt z.B. für scheduling-Dienste, mit denen zur Laufzeit die Zuteilungsreihenfolge unter mehreren rechenbereiten (R-)Prozessen verändert wird (erzwungener Prozessorentzug, Änderung der Priorität). Übertragen auf W-Prozesse würden solche Dienste dynamische Änderungen der Zuteilungsreihenfolge oder Präemptionen zwischen W-Prozessen spezifizieren, die alle zum selben Zeitpunkt auf der virtuellen Zeitachse rechenbereit sind. Zwar gibt es solche Funktionen auch in existierenden Simulationssprachen (z.B. ACTIVATE <PRIOR> in SIMULA), doch kann man über ihren Wert für die Modellierung sicherlich streiten.

### 3.4.3.2 Anforderungen an die Ablaufsteuerung

Die Ablaufsteuerung der Echtteilesimulation nach 3.3.2 wird konkretisiert für den Fall, daß reales Testobjekt und simulierte Umgebung quasiparallel auf einem Rechner ausgeführt werden.
Anstelle der reinen Echtzeituhr des Echtzeit-BS-Kerns E verfügt der erweiterte BS-Kern E' über **eine** (zentrale) **virtuelle Uhr**, die - falls W-Prozesse im Experiment sind - **nicht** mit der Echtzeit korrespondiert, sondern die Echtzeit im realen Wirkungskreis nachbildet, also die Rolle von tv in 3.3.2 übernimmt. Die Aufgabe von E' besteht nun darin, die Steuerung der **virtuellen Uhr und die Prozessorzuteilung** der R- und W-Prozesse so zu koordinieren, daß folgende Anforderungen erfüllt werden (das Attribut "Zeit", "Zeitpunkt" bzw. "zeitlich" bezieht sich i.f. auf die virtuelle Uhr).

(AV) Die virtuelle Zeit im Experiment ist monoton wachsend, d.h. die Reihenfolge aller

Operationen im Experiment - ob aus dem realen Testobjekt oder der simulierten Umgebung kommend - ist dieselbe wie im realen Wirkungskreis.

(AW4) Die Aktionen der W-Prozesse finden exakt zu den durch tv(p) in 3.3.1 vorgeschriebenen Zeitpunkten statt.

Die R-Prozesse in simulierter Umgebung entsprechen in ihrem funktionellen und zeitlichen Verhalten derselben Prozeßmenge in realer Umgebung. Dies soll auf folgende Weise erreicht werden:

(AR3) Die R-Prozesse werden unter denselben Zuteilungsregeln **ausgeführt** wie im Echtzeiteinsatz, und die Virtuelle Uhr ist während der Ausführung **echtzeit-synchron**, d.h. R-Prozesse werden als "meßbare" und nicht als in ihrem Zeitverhalten nachgebildete Komponenten behandelt.

(AR4) Der zeitliche Ablauf der R-Prozesse hängt zwar von Art und zeitlichem Verlauf der an der RW-Kommunikations-Schnittstelle durch die SU durchgeführten Operationen ab, aber **nicht** davon, **wie** diese Umgebung simuliert wird **(Interferenzfreiheit)**.

Zu berücksichtigen ist dabei natürlich die Art der erlaubten Kommunikationsoperationen an der Schnittstelle zwischen realem Testobjekt und realer bzw. simulierter Umgebung. So zeigt sich, daß die Asynchronie zwischen Sendern und Empfängern explizit begrenzt werden muß, was aber die in (AR2) geforderte Flexibilität und Realitätsnähe der Kommunikations-Schnittstelle nicht beeinträchtigt. Vor allem aber stellt (AR4) besondere Anforderungen an die Prozeß- und Betriebsmittelzuteilung der BS-Kernerweiterung E' (Kap. 4.2).

**Bemerkungen**

- **Abschwächung, Einschränkung** von (AR4):
  Die Interferenzfreiheit läßt sich zwar theoretisch exakt, in der praktischen Implementation aber nur näherungsweise erfüllen, weil die Nahtstellen zwischen R- und W-Prozessen im Ablauf (räumlich und zeitlich) nicht "punktförmig" sind. Die Fehler lassen sich aber durch zeitliche Korrekturterme fast vollständig kompensieren.
- Es besteht ein Konflikt zwischen dem Wunsch, möglichst viele Komponenten als reales Testobjekt zu integrieren, und die Verfälschung aller vorhandenen RTO durch ihre Umgebung zu minimieren. Ein wichtiger Vorteil der integrierten Simulation liegt gerade darin, flexibel zwischen Simulationsmodell und RTO wählen und damit eine Abwägung im o.g. Zielkonflikt treffen zu können. (Beispiel: E/A-Geräte, vgl. 4.2.5.5).
- (AR2),(AR4) gelten nicht nur für die Aktionen der R-Prozesse auf Anwendungsebene, sondern selbstverständlich auch für die unter der Regie des RTO ausgeführten BS-Kern-Funktionen. Insofern verläuft die Schnittlinie zwischen Testobjekt (Zielmaschine) und Testumgebung (Simulator der W-Prozesse) mitten durch E' hindurch.
- (AV) (zentrale, monotone Uhr) stellt strenggenommen keine Anforderung, sondern bereits eine Entwurfsentscheidung dar. Nach 3.3.1, 3.3.2 ist auch eine dezentrale Zeit denkbar.

Eine Untersuchung ergab aber, daß dies in einer Einprozessorumgebung keine Effizienzvorteile bringt, aber den Entwicklungsaufwand für die Kernerweiterung stark erhöht und damit die Wirtschaftlichkeit des Konzeptes gefährdet.

### 3.4.3.3 Zeitmodi

Eine zentrale virtuelle Zeit nach 3.4.3.2 kann als Abbildung **aller** Aktionen von Prozessen eines Experimentrechners in einen Zeitbereich T verstanden werden.

$t_v : A \rightarrow T$

Durch eine externe Zeitreferenz

$t_e : A \rightarrow T$

wird andererseits allen Aktionen im Experiment der Echtzeitpunkt ihres Vorkommens zugeordnet (z.B. Universal Standard Time oder Zeitdienst der Deutschen Bundespost). Die Granularität der physikalischen Zeit sei fein genug, so daß sie eine totale Ordnung aller interessierenden Aktionen induziere:

f. alle Aktionen A1, A2: $A1 \neq A2 \Rightarrow t_e(A1) \neq t_e(A2)$

Weil $t_e$ injektiv ist, wird auf dem Bildbereich $T' := im(t_e) \subset T$ durch

**Def. 3.4**

$t_s : T' \rightarrow T$ **(Bewertungszeitfunktion)**

$ts(t) := t_v(t_e^{-1}(t))$

der Verlauf der virtuellen Zeit $t_v$ direkt als Funktion der Echtzeit beschrieben, statt als Funktion der Aktionen. Dies eignet sich besonders zur anschaulichen graphischen Darstellung, welchem Zeitpunkt im Zielsystem jeder Zeitpunkt im Experiment entspricht (Abb. 3.10).

**Beispiel (a)**

Bei jeder zeitdiskreten Simulation oder Emulation ist ts eine Treppenfunktion (Abb. 3.10a).

**Beispiel (b)**

Bei der (skalierten) Echtzeitsimulation (Def. 2.6) sowie bei allen Verfahren zur Leistungsmessung besteht ein linearer Zusammenhang zwischen Echtzeit und virtueller Zeit (Abb. 3.10b).

$ts(t) := T_0 + d*t$ $\quad$ $d > 0$ Skalierungsfaktor
$d = 1$ bei reiner Echtzeitsimulation bzw. Meßsystem

Bei der integrierten Simulation wechselt die Form des Zeitverlaufs innerhalb eines Experimentrechners dynamisch zwischen Teilfunktionen der Art (a) und (b) (Abb. 3.10c). Dieser Wechsel wird durch die **Zeitmodi** ausgedrückt: den F-Modus, den R-Modus und den D-Übergang.

**Def. 3.5 (Zeitmodi)**

Der **F-Modus** (freeze)

gilt in einem Echtzeitintervall [T0,T1], falls $ts(t) = const \ \forall \ t \in [T0,T1]$

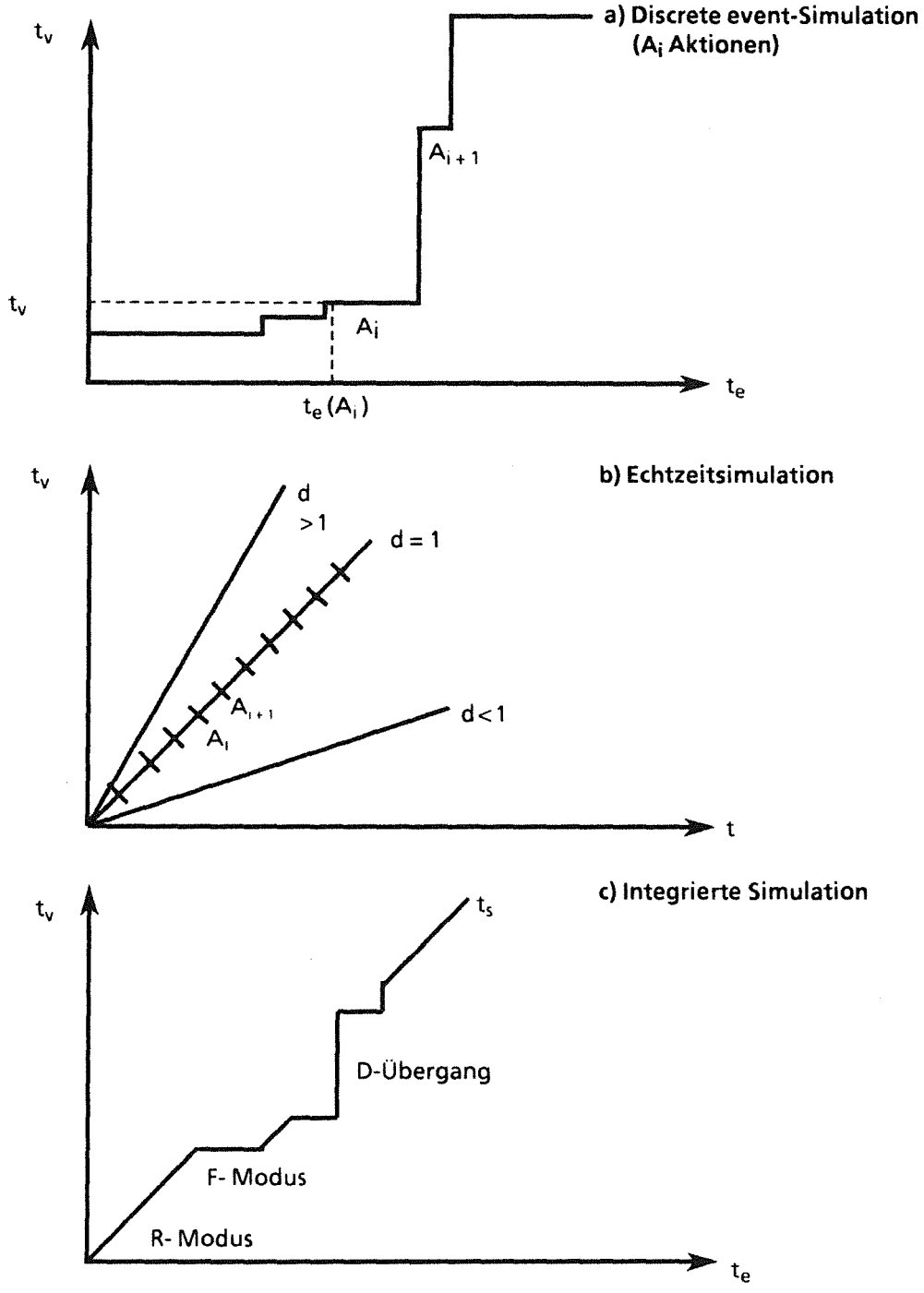



**Abb.** 3.10: Bewertungszeitfunktionen

Der **R-Modus** (real time)

gilt in einem Zeitintervall [T0,T1], falls ts(t1)-ts(t0) = t1-t0 $\forall$ t1,t0$\in$[T0,T1]

**D-Übergang** (discrete event)

liegt zu einem Zeitpunkt t vor, wenn t Unstetigkeitsstelle von ts und ts(t) > ts(t') $\forall$ t' < t.

Offenbar ist die virtuelle Zeit monoton wachsend, wenn ts aus diesen drei Zeitmodi zusammengesetzt ist.

### 3.4.4 Netzwerksynchronisation

#### 3.4.4.1 Aufgabenstellung

Für die integrierte Leistungsbewertung **verteilter** Zielsysteme (PFSe) ergibt sich folgendes Bild:

(1) Das Ziel der Experimente besteht darin, am Ende das vollständig implementierte PFS als reales Testobjekt zu bewerten. Da ein Experimentrechner sinnvollerweise nur diejenigen Rechenprozesse als R-Prozesse bewertet, für die er selbst Zielrechner ist (vgl. 3.6.1), muß die Experimentrechnerkonfiguration die Menge der Zielrechner enthalten, also selbst verteilt sein.

(2) Jeder Experimentrechner $E_i$ realisiert nach 3.3.2, 3.4.1 ein i.a. gemischt reales/simuliertes Testobjekt, das insgesamt als I/O-System $K_i$ mit zeitdiskreten Ein-/Ausgabeverläufen gelten kann.

(3) In **jedem** Experiment gibt es W-Prozesse, also nicht in Echtzeit operierende Komponenten, auf mindestens einem Experimentrechner. Die virutellen Uhren (Abb. 3.10) verschiedener Experimentrechner geben infolgedessen i.a. verschieden, m.a.W.: zu **einem** Zeitpunkt im **Experiment** repräsentieren die $K_i$ **unterschiedliche** Zeitpunkte in der **Realität**.

Diese Problematik führt zwangsläufig auf die **verteilte discrete-event-Simulation** und ihre Verfahren (VVS) /MIS 86//PEA 80//CHM 81//REY 82//JES 83//ZEI 85//KUM 86//UNG 88/. Hierunter versteht man

- die Zerlegung eines komplexen Simulationsmodells in autonome Modellkomponenten*), welche parallele Prozesse im realen System repräsentieren
- die verteilte Ausführung dieser Modellkomponenten, wobei jede über eine eigene Simulationsuhr verfügt und mit den übrigen über zeitgestempelte Ereignisnachrichten kommuniziert, durch die der Empfänger die notwendige zeitliche Zuordnung der Ereignisse erhält.

Als Modellkomponenten werden in der Literatur discrete-event-Systeme, meist Bedienstationen in Warteschlangennetzen, jedoch keine in die Simulation integrierten realen Testobjekte betrachtet.

*)In der Literatur finden sich neben der Verteilung der Modellkomponenten auch Ansätze zur Verteilung eines Simulationssystems nach funktionellen Gesichtspunkten /WYA 85//COG 88/: Hilfskomponenten wie Lastgeneratoren, interaktive Experimentsteuerung, Zufallszahlenerzeugung, Archivierung, statistische oder graphische Auswertung werden auf separate Rechner ausgelagert, während die Simulationsmodelle und die Zeitliste selbst zentralisiert bleiben. Nun lassen sich aber alle diese Hilfskomponenten auch als spezielle Modellkomponenten auffassen (in /ZEI 84a/ als Generatoren, Akzeptoren oder Transducer bezeichnet). Jedes allgemeine Verfahren zur Modellverteilung ist also prinzipiell auch auf die Verteilung der Support-Funktionen anwendbar, aber nicht umgekehrt.

Sind die Modellkomponenten I/O-Systeme, so stehen die zwischen ihnen ausgetauschten Nachrichtenfolgen für Ein-/Ausgabehistorien $xi \in (XI, T_{inp})$, $xo \in (XO, T_{out})$. Die in 3.4.2 erwähnte Netzwerksynchronisation muß sicherstellen, daß alle Historien xi, xo korrekt nach der Definition von I/O-Systemen verknüpft sind:

(3.6) $xo(t) = \lambda(\delta(s_i, xi|_{[T0,t]}))\ \forall\ t \leq \min(T_{inp}, T_{out})$

$\delta$ lokale Zustandsübergangsftn. von $K_i$

$\lambda$ lokale Ausgabeftn. von $K_i$

und zwar **unabhängig** von

- der Art der Verteilung der $K_i$ auf verschiedene Experimentrechner
- der realen zeitlichen Dauer der Simulation und den Nachrichten-Laufzeiten

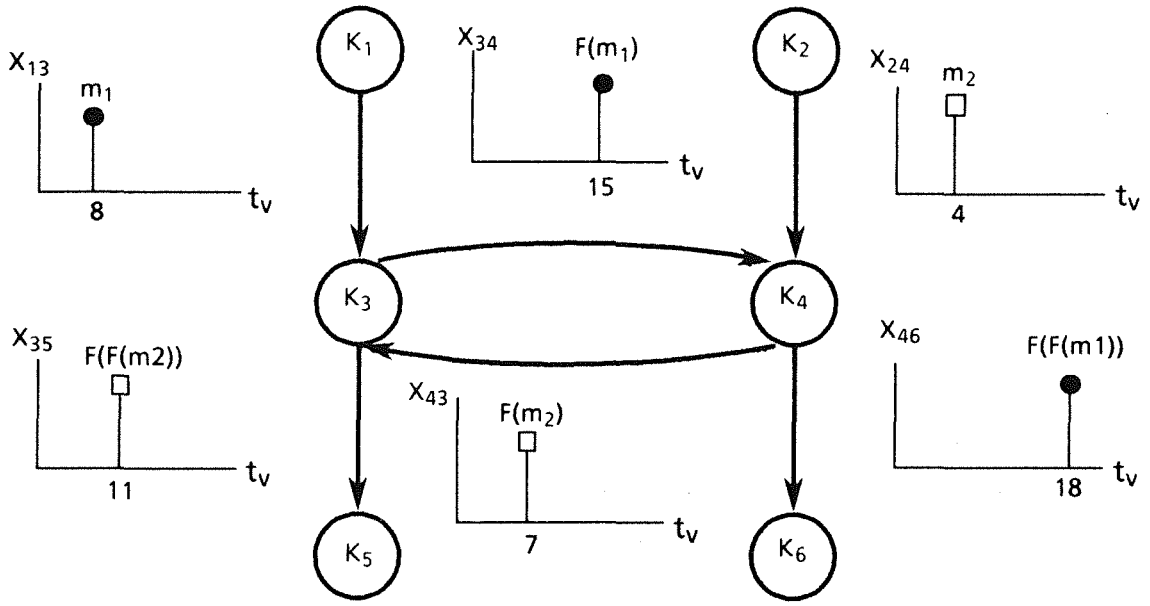
(**schwache Korrektheit** eines VVS).

Die NWS könnte z.B. durch ein blockierendes Nachrichtenprotokoll die chronologische virtuelle-Zeit-Reihenfolge aller Ereignisse netzweit erzwingen. Attraktiver ist es i.a., die zeitliche Entkoppelung der Komponenten zuzulassen. Dann besteht aber das Problem, daß Ausgabe-Nachrichten einer Komponente $K_i$ mit Zeitstempel T sich infolge nachträglich eintreffender, nicht berücksichtigter Eingabe-Nachrichten mit Zeitstempel $<T$ als falsch erweisen. Mit der zeitlichen Reihenfolge geht zwar nicht zwingend, aber oft auch eine kausale Abhängigkeit von Ereignissen einher.

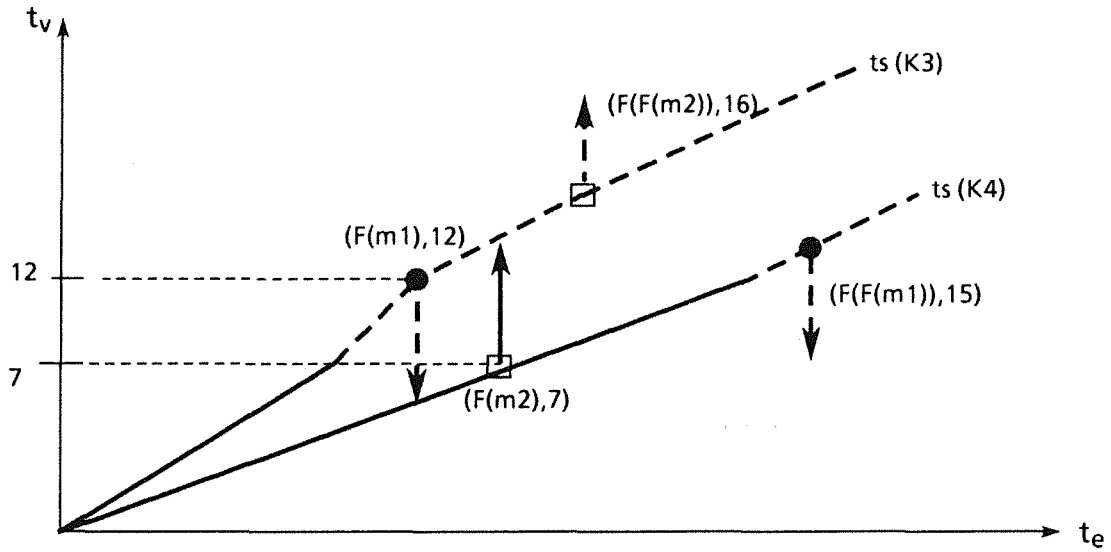
Abb. 3.11 zeigt ein einfaches Beispiel, in dem die Komponenten K1,..,K6 FIFO-Bedienstationen darstellen. Die Bearbeitungsaufgaben von K3 und K4 seien durch eine einzige Funktion F mit virtueller Bedienzeit 4 (für K3) bzw. 3 für (K4) modelliert, wobei K3 [K4] seine Ergebnisse abwechselnd nach K4 [K3] und K5 [K6] sendet (man vgl. dazu auch Programmbeispiel Abb. 3.12). K3 finde in seiner Eingangswarteschlange eine Auftragsnachricht (m1,t=8) von K1 vor. Wenn er diese sofort bearbeitet, ergibt sich inkorrekterweise ein simuliertes Bedienitintervall [8,12], denn im Beispiel trifft später noch ein Auftrag (F(m2),t=7) von K4 ein, dessen Bearbeitung K3 im Intervall [7,11] beschäftigt hätte. Der korrekte Ausgabezeitpunkt für F(m1) wäre demnach 15, nicht 12.

Ein VVS arbeitet **korrekt**, wenn es zusätzlich zur schwachen Korrekheit stets terminiert, d.h. wenn die $T_{inp}$, $T_{out}$ schließlich die vorgegebene Simulationszeit erreichen. Dazu muß ein VVS **verklemmungsfrei** sein, und die Folgen der Nachichten-Zeitstempel dürfen keine **endlichen Häufungspunkte** besitzen (der Zeitbereich ist i.a. reellwertig).

Ein VVS heißt **allgemeingültig**, wenn es korrekt für beliebige gekoppelte discrete-event-Systeme funktioniert.




a) Komponenten-Topologie und Nachrichtenhistorien




b) Verletzung der chronologischen Reihenfolge der Nachrichtenverarbeitung
(- - - : inkorrekte Berechnung oder Ausgabenachricht)

**Abb.** 3.11: Beispiel zur verteilten Simulation (Programmformulierung in Abb. 3.12)

### 3.4.4.2 Anforderungsprofil

An ein VVS werden folgende Anforderungen gestellt

**(AN1) Korrektheit, Allgemeingültigkeit**

Die Mehrzahl der publizierten VVS ist nicht allgemeingültig; oft wird auch nicht die Klasse der Modelle exakt spezifiziert, für die ein Verfahren korrekt arbeitet. Es existieren keine

rechnergestützten Werkzeuge, um die Zulässigkeit der Modelle für ein bestimmtes VVS zu prüfen. Dies müßte der Benutzer (Anwendungsentwickler) leisten, ebenso ggf. die Anpassung oder den Austausch der NWS im BS-Kern.
Dieser Aufwand wird mit zunehmender Detaillierung des Zielsystems i.a. wachsen, ohne dem Entwicklungsprojekt direkt zugute zu kommen.
Es ist auch nicht gerechtfertigt, etwa durch einen Baukastenansatz, die Modell-/ Zielsytementwicklung so einzuschränken, daß nur zulässige Modelle eines bestimmten VVS erzeugt werden können (z.B. nur FIFO- Warteschlangensysteme wie in /KUM 86/). Zur Modellierung technischer Prozesse, deren Zustand von der Einhaltung harter Zeitbedingungen abhängt, präemptiver Bedienstrategien u.ä. wird die Allgemeinheit benötigt (vgl. 5.2.1).

**(AN2) Integrierbarkeit von realen Testobjekten (R-Prozessen)**
Bisher wurde zwar nur vorausgesetzt, daß die Modellkomponenten I/O-Systeme mit zeitdiskretem Ein-/Ausgabesignalverlauf sind - was reale Testobjekte nach 3.4.4.1 (2) zuläßt - doch stellen manche VVS auch bestimmte Anforderungen an die interne Struktur der Modellkomponenten, z.B. daß alle Zustandsänderungen zu diskreten Zeitpunkten auftreten. Insbesondere wird oft eine Funktion ta:S->T angenommen, die durch Inspektion des aktuellen Zustands s - ohne diesen oder die Simulationszeit zu ändern - den Zeitpunkt des nächsten Zustandsübergangs liefert ('next-event-time'- Funktion, vgl. Def. 2.3).
Diese Funktion ist auf reale Testobjekte ohne ein Zurücksetzen nicht anwendbar: das nächste Ereignis (z.B. nächste Sende- oder Empfangsoperation eines R-Prozesses, nächstes Unterbrechungssignal eines E/A-Gerätes) kann nur durch **Ausführen** des Testobjektes bestimmt werden, wodurch Zustand und virtuelle Zeit irreversibel von (s,T) nach (s',$T+\Delta$) übergehen.

**(AN3) Programmier-Transparenz**
Sobald ein Simulationsmodell in konzeptionell parallele Teilmodelle zerlegt ist, sollte die physikalische Verteilung bzw. deren Änderung für den Modellentwickler/-programmierer transparent sein. In der Formulierung von Abb. 3.12 wurde implizit vorausgesetzt, daß die Komponenten K3 und K4 über Ein-/Ausgabe-Kommunikationsobjekte kommunizieren, die die logische Kommunikationstopologie nach Abb. 3.11a reflektieren, daß aber die physikalische Verteilung der Simulationsmodelle K1,..,K6 auf die Experimentrechner unbekannt ist.
Dazu müssen drei voneinander unabhängige Voraussetzungen erfüllt sein:

- das VVS muß systemtheoretisch eine beliebige Verteilung zulassen (AN1).
- die unterliegenden Synchronisationsalgorithmen der NWS müssen für die Anwenderprogramme transparent sein. Die an inp_mb wartende Komponente K3 in Abb. 3.12 kann z.B., sobald ihr der Auftrag (m,8) von K1 übergeben wird, diesen unbesorgt verarbeiten, ohne sicherstellen zu müssen, daß nicht doch noch ein Auftrag (m',7) von K4 eintrifft.

```
Beschreibung der Komponenten:
type request is        record
                           d : appl_data;
                           t : absolute_time;
                       end;

function F (in_msg: request) return request is
    (*Verarbeitungsalg. zur Auftragssimulation *)
```

```
comp K3 is
begin
inp_mb,k4_mb,k5_mb: mailbox;
in_msg,out_msg: request;
discr: boolean: = true;

loop
    rec_msg (inp_mb, in_msg);
    delay_for (4);
    out_msg.d : = F(in_msg);
    out_msg.t : = clock_read;
    if discr
    then
        send_msg (k4_mb, out_msg)
    else
        send_msg (k5_mb, out_msg);
    discr: = not (discr);
end loop;
end;
```

```
comp K4 is
begin
inp_mb,k3_mb,k6_mb: mailbox;
in_msg,out_msg: request;
discr: boolean: = true;

loop
    rec_msg (inp_mb, in_msg);
    delay_for (3);
    out_msg.d: = F(in_msg);
    out_msg.t : = clock_read;
    if discr
    then
        send_msg (k3_mb, out_msg)
    else
        send_msg (k6_mb, out_msg);
    discr: = not (discr);
end loop;
end;
```

**Abb.** 3.12: Programmbeispiel zu Abb. 3.11

- das Kommunikationssytem muß rein funktionell die lokations-transparente Kommunikation über mailboxes oder ports unterstützen. Wir setzen dies i.f. voraus.

**(AN4) Praktikabilität**

Dazu zählen Kriterien wie

(a) Laufzeiteffizienz

(b) Ressourcenbedarf (insbesondere ASP-Bedarf)

(c) Implementierungsaufwand und Portabilität.

Unter der Laufzeiteffizienz eines VVS versteht man das Verhältnis der Gesamtlaufzeiten im verteilten Fall und im zentralen Fall, um ein vorgegebenes Modell bis Zeit T zu simulieren (dieser Wert ist ggf. noch bzgl. der Anzahl der Prozessoren oder weiterer Parameter zu normieren). Die Effizienz hängt sowohl von der zeitlichen Autonomie ab, die die NWS den Komponenten gestattet, als auch vom Verwaltungsaufwand der NWS selbst (z.B. dem zusätzlichen organisatorischen Nachrichtenaufkommen). Untersuchungen der Effizienz eines bestimmten VVS oder der maximal erreichbaren Parallelität wie z.B. in /JEW 84//BEL 87//FUJ 88//REM 88/, werden in dieser Arbeit nicht durchgeführt.

Die Kriterien (b) und (c) spielen deshalb eine wichtige Rolle, weil die integrierte Simulation nicht als dediziertes Testbettsystem, sondern als Erweiterung zur Standard-BS-Software marktüblicher Mikro- und Minirechnersysteme vorgeschlagen wird. Daher sollten sich die Erweiterungen auch an diese Gegebenheiten anpassen und nicht selbst alle Anforderungen an die Zielrechner dominieren, und sie sollten sich leicht auf unterschiedlichen Zielrechnern implementieren lassen.

### 3.4.5 Benutzung des BS-Kerns als Simulations- und Zielmaschine (vertikale Architektur)

Als nächstes soll im groben Überblick gezeigt werden, wie die Systemkern-Schnittstelle in den einzelnen Entwicklungsphasen des Zielsystems nach Abschnitt 3.2 benutzt wird (Abb. 3.13). Damit knüpfen wir zugleich an Abschnitt 2.3 an (schrittweise Verfeinerung von PFS-Entwürfen zu einsatzfähigen Implementierungen).
Die Basisschicht bildet wieder die erweiterte Systemdienst-SSe eines Monoprozessor-BS-Kerns (3.4.3.1); aufgrund 3.4.4 kann die physikalische Verteilung auf mehrere BS-Kerne bzw. NWSen nun ignoriert werden.
Erst durch die höheren BS-Funktionen bzw. Laufzeitsysteme (Schicht G) wird eine Funktionalität erreicht, die den Simulationssprachen im wirtsorientierten Simulationsansatz (z.B. SIMULA einschließlich Ein-/Ausgaberoutinen) vergleichbar ist. Wenn sie im Experiment nur der Simulationsunterstützung dienen (z.B. Dateischnittstelle), sind ihre Leistungen zeitlich transparent. Werden sie durch Prozesse realisiert, handelt es sich um W-Prozesse. Z. Teil können dieselben Funktionen von Echtzeitanwendungen benutzt werden und gehören damit zum realen Testobjekt.

Auf der Schicht G setzt nun die **Modellierungsschnittstelle** (MSS) bzw. Modellierungssysteme auf, die die zur Leistungs**vorhersage** von PFSen in den Entwurfsstadien O (3.2.2) und VB (3.2.3) notwendigen Komponenten enthalten:

(A) **Umgebungsmodelle** für die zeitdiskrete Simulation kontinuierlicher und diskreter technischer Prozeßelemente (3.1.1); diese werden in allen Entwicklungsstadien (O,VB,R) i.w. in unveränderter Form verwendet

(B) **Funktionelle Modellierung**, d.h. in erster Linie Kommunikations- und Synchronisationsmechanismen für das PFS im Stadium O, die abstrakter und allgemeiner bzw. kompakter und leichter zu handhaben sind als die vom BS-Kern angebotenen. Ein gutes Beispiel sind etwa verallgemeinerte Monitore nach Kessels /KES 77/, deren Wartebedingungen zeitlich befristet und mit der zeitlichen Semantik von 3.3.1 versehen werden können.

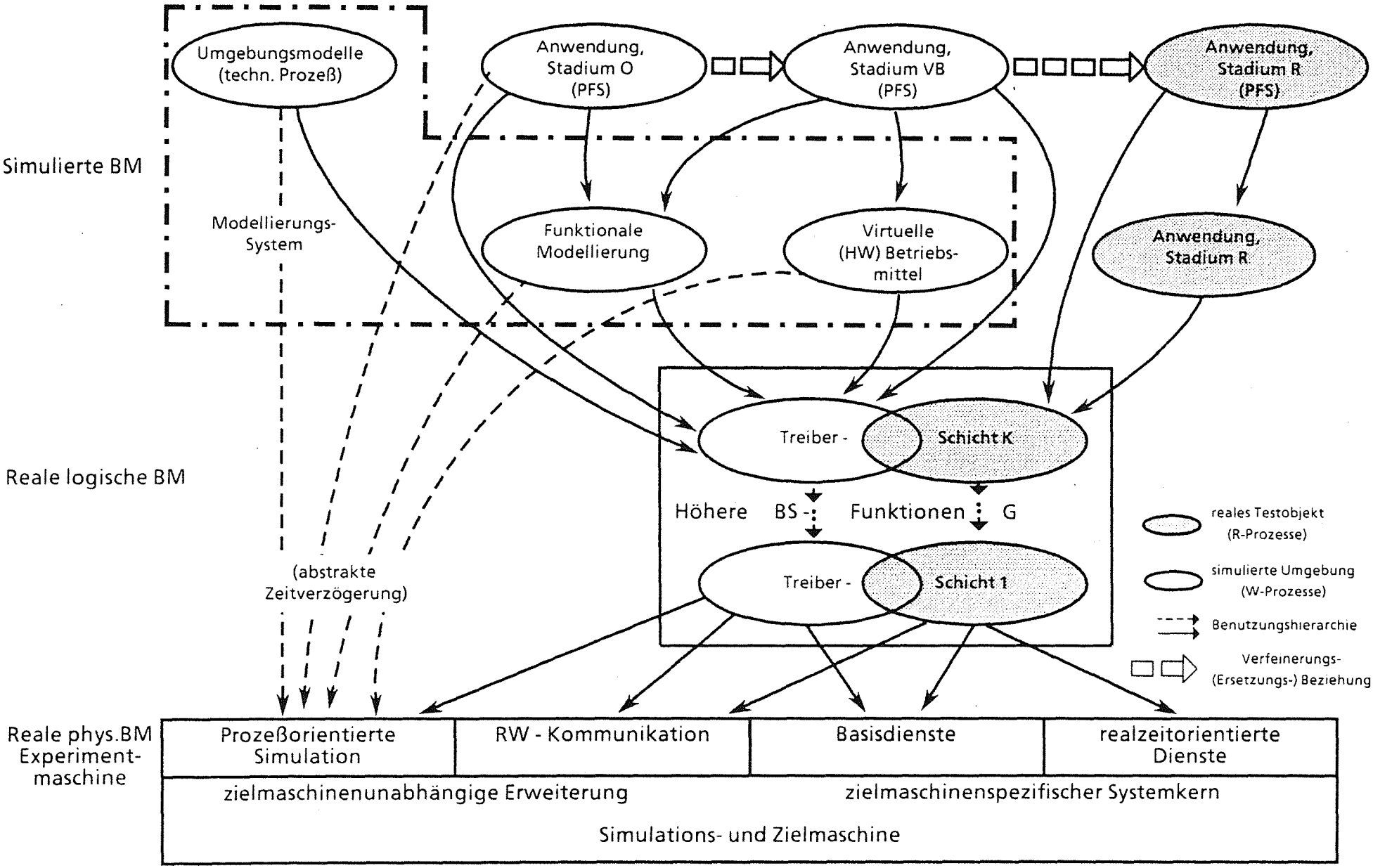



Abb. 3.13: Benutzung der Systemkern-Erweiterung als Simulations- u. Zielmaschine („vertikale Architektur")

Solche nur zu Demonstrationszwecken dienende, aus Effizienzgründen für den Realzeiteinsatz i.a. nicht tauglichen Funktionen werden später durch realzeittaugliche verteilte Funktionen ersetzt, die mit Hilfe der Schicht G oder der BS-Kern-Schnittstelle direkt die gewünschten Synchronisationseigenschaften implementieren (rechte Bildhälfte von Abb. 3.13).

(C) **Virtuelle Betriebsmittel-Umgebung**

Diese besteht aus Objekten zur Simulation verschiedener Zielrechner, Kommunikations-Medien ("virtuelles LAN") und E/A-Geräte (Prozeßperipherie, Hintergrundspeicher). Gemäß ihrem Verwendungszweck im Entwurfsstadium VB (3.2.3) müssen geeignete **Operationen**

- zum dynamischen Erzeugen und Lösen von Zuordnungen:
  Prozesse <-> virtuelleHW-Konfiguration
- zur Spezifikation des BM-Verbrauchs

und **Attribute** zur Beschreibung von Kapazität und Leistungscharakteristika sowie Bedienstrategien vorgesehen werden (vgl. 6.2.2).

Diese Komponenten werden durch W-Prozesse realisiert und ausschließlich von W-Prozessen benutzt. Wir haben nur wenige Beispiele für Modellkomponenten angeführt; jede Anwendung bringt i.a. neue Anforderungen. Die Modellierungs-Schnittstelle ist daher offen und erweiterbar zu gestalten.

### 3.4.6 Simulierte Zielprozesse

Das bisher skizzierte Schema der W- und R-Prozesse läßt einen nahtlosen Übergang vom Entwurfsstadium in virtueller BM-Umgebung (VB) zur realen Ablaufumgebung (R) bei der Entwicklung und Bewertung des PFS noch nicht zu. Dies wird deutlich, wenn man R- und W-Prozesse danach charakterisiert, wie die Benutzung der realen Betriebsmittel eines Experimentrechners in ihrem Zeitverhalten sichtbar wird. Für W-Prozesse ist diese Benutzung irrelevant und daher unsichtbar: alle Aktionen in 3.3.1, bei denen ein W-Prozeß BM der Experimentmaschine beansprucht (Aktionen der Klassen (K1),(K2) und (K5)), verlaufen zeitlich transparent. Die virtuelle Zeit schreitet nur als Folge einer expliziten Zeitverzögerung (K4) oder beim Warten auf ein Ereignis oder eine Botschaft (K3) fort. Die Konkurrenz der W-Prozesse läßt sich nur über virtuelle Betriebsmittel (3.4.5) nachbilden, deren Benutzung auf Botschaftenkommunikation zurückgeführt wird (Anforderung und Gewährung von Bedienwünschen) und über die BM-Hierarchie letztlich auf explizite Zeitverzögerungen (K4) führt ("nicht weiter spezifizierter, abstrakter" BM-Verbrauch, z.B. bei der Modellierung von HW-Betriebsmitteln).

Für R-Prozesse andererseits gilt ihr BM-Bedarf auf der Experimentmaschine in **Art** und **Umfang** als repräsentativ für den Echtzeiteinsatz.

Diese Polarität zwischen R- und W-Prozessen paßt zu einem SW-Entwicklungsprozeß, in dem Rechenprozesse erst nach **abgeschlossener** Implementierung auf ihren Zielrechnern als R-Prozesse installiert und bewertet werden, und zwar möglichst alle Prozesse eines Zielrechners gemeinsam, damit dessen Last repräsentativ ist. Eine größere Flexibilität ist nach 3.2.5 wünschenswert: ein Prozeß wird z.B. auf seinem Zielrechner noch vervollständigt oder bestimmte Algorithmen werden dort noch optimiert. Er benutzt im Experiment also bereits die BM seines späteren **Zielrechners**, aber der **Umfang** seines Bedarfs ändert sich noch. Für diese Situation wird der **simulierte Zielprozess (S-Prozess)** als Bindeglied zwischen R- und W-Prozessen eingeführt. Er konkurriert -anders als ein W-Prozeß- mit den R-Prozessen **sichtbar** um die BM CPU, ASP, E/A-Geräte des Zielsystems, spezifiziert seinen Bedarf -anders als ein R-Prozeß- aber **explizit** durch Dienste der Form

**work** (<bm_type>, <umfang>); (z.B. Anzahl CPU-Zyklen, ASP-Bedarf, ( Anzahl zu transportierender Worte etc.)

Ein S-Prozeß steht wie ein R-Prozeß für einen SW-Prozeß einer konkreten Zielmaschine, hat dieselben Attribute (z.B. Priorität) und benutzt dieselben Systemdienste. Ein S-Prozeß erfüllt die Funktion eines Platzhalters für einen R-Prozeß. Die Anforderungen an den Monoprozessor-BS-Kern in 3.4.3.1 werden also noch erweitert:

(AS) Für Realzeit-SW im Implementierungsstadium (S-Prozesse) sind Systemdienste zur Simulation von HW-Betriebsmittelverbrauch bereitzustellen, der einem tatsächlichen BM-Verbrauch von gleicher Art und Umfang eines ansonsten identischen R-Prozesses auf dieser Zielmaschine gleichkommt.

**Beispiele**

(1) **Synthetische Last**

P1,...,P5 seien Rechenprozesse, die einem gemeinsamen Zielrechner Z zugeordnet werden sollen, aber erst P1, P2 seien implementiert. Für P3,P4,P5 liegen die Schnittstellen sowie ein ausführbarer (aber noch nicht endgültiger) Algorithmus vor. Wenn P1,P2 als R-Prozesse und P3,P4,P5 als S-Prozesse auf Z bewertet werden, so stellt dies zwar nur eine Näherung des endgültigen Verhaltens dar, aber sicher eine bessere Repräsentation der Zielrechner-Last, als wenn entweder

- nur P1,P2 R-Prozesse auf Z und P3,P4,P5 Wirtsprozesse auf einem virtuellen Rechner, oder aber
- P1,...,P5 alle Wirtsprozesse auf einem gemeinsamen simulierten Rechner wären.

Im ersten Fall können P3,P4,P5 als Wirtsprozesse nicht zur Last des Zielrechners Z beitragen, im zweiten Fall kann das Zeitverhalten der Implementierung von P1,P2 nicht ausgenutzt werden.

(2) **Testhilfe nebenläufiger Systeme im Implementierungsstadium**
Bekanntlich lassen sich zeitabhängige Fehler bei der Synchronisierung paralleler Prozesse im Realzeittest u.a. deshalb schwer finden, weil sich eben doch oft eine nahezu deterministische Ablauffolge der Prozesse in den Testfällen einstellt und erst bei wesentlicher Änderung des Lastprofils - z.B. durch Systemerweiterung - plötzlich "unerklärliche" Fehler auftreten. Indem man die - funktionell mit der Einsatzversion identischen - Prozesse zu Testzwecken als S-Prozesse konfiguriert und ihren Zeitbedarf auf dem Zielrechner variiert, lassen sich zeitabhängige Fehler leichter aufdecken (ein solcher Test kann natürlich die Bemühungen um einen **Korrektheitsnachweis** nicht ersetzen!).

Tabelle 3.2 faßt die drei Prozeßklassen und ihre Anwendungen zusammen.

| KLASSE | BETRIEBSMITTEL-BEDARF UND -BEWERTUNG | BEISPIELE,ANWENDUNG | ZEITFÜHRUNG |
|---|---|---|---|
| REALER ZIELPROZESS (R) | identische BM in Zielsystem und Experiment, identischer Bedarf (reale BM, realer Bedarf) | voll implementierte PFS-Komponente auf Zielrechner | Echtzeit |
| SIMULIERTER ZIELPROZESS (S) | identische BM in Zielsystem und Experiment, i.a. anderer Bedarf (reale BM, synthetischer Bedarf) | SW-Komponente auf Zielrechner, aber z.B.<br>- funktionelle Vergröberung<br>- funktionell gleicher, aber weniger effizienter Prototyp<br>- Testversion (Variation des Zeitverhaltens) | virtuell |
| WIRTSPROZESS (W) | BM-Bedarf im Experiment irrelevant für Zielsystem (virtuelle BM, synthetischer Bedarf) | - simulierter technischer Prozeß<br>- Debugger- oder Auswertungskomponente<br>- virtuelles Betriebsmittel<br>- Anwendungs-SW in virtueller BM-Umgebung | virtuell |

**Tab**. 3.2: Bedeutung der R-, S- und W-Prozesse

### 3.5 Entwicklungs- und Auswertungssystem

Zur Modell- und Zielsystementwicklung, Definition, Steuerung und Auswertung von Experimenten wird eine übergeordnete Entwicklungs- und Auswertungsumgebung benötigt. Deren Funktionen werden i.f. nur grob vorgestellt, aber es werden keine detaillierten Konzepte hierzu entwickelt oder gar implementiert (man vgl. hierzu DESIGN /MUE 86/, JADE /UBC 84/, /SPO 84/).

Auch beim Entwicklungs- und Auswertungssystem wird das Konzept der funktionellen Erweiterung verfolgt. Vorausgesetzt wird also zunächst eine Entwicklungs- und Programmierumgebung für die Zielsysteme selbst, also für verteilte Echtzeitsysteme, deren Sprachen und Werkzeuge so **erweitert** werden sollen, daß **speziell** die die **Simulationsexperimente** betreffenden Anforderungen erfüllt werden. Es soll keine dedizierte und in sich abgeschlossene Entwurfs- und Simulationsumgebung aufgebaut werden wie z.B. in SARA /EFR 86/, PAISLEY /ZAS 86/ oder /SPO 84/, in der große Teile einer konventionellen Entwicklungsumgebung "neu erfunden" werden. Ein graphisches, mit einer komfortablen Bilddefinitionssprache ausgestattetes Prozeßinformationssystem (z.B. GROOPI /GHU 85/) kann als Simulationsgrafik im Prinzip ebenso verwendet werden wie als Leitstand für den realen technischen Prozeß. Eine Datenbank zur Verwaltung von Entwicklungsversionen und Konfigurationskontrolle des Zielsystems ist prinzipiell auch als Experiment-Datenbank tauglich.

Abb. 3.14 zeigt die Einbettung der Modell- und Zielrechner mit den in 3.4 grob beschriebenen Komponenten in das Gesamtsystem.

Die Standard-Entwicklungsumgebung besteht aus Editoren, Übersetzern, Bindern, down-loading (Verteilen ablauffähiger Objektmodule auf die Zielrechner) sowie Test-, Dokumentations- und ggf. Projektverwaltungs-Werkzeugen. Für Realzeitsysteme kommen u.U. weitere Werkzeuge zur Analyse der Zeit- und ggf. Zuverlässigkeitseigenschaften hinzu, z.B. Scheduling-Analysatoren /KDK 88//TOK 88/. Speziell für die integrierte Simulation und die Experimentdurchführung mit ihr sind drei Hauptkomponenten mit folgenden Funktionen notwendig (rechte Bildseite):

**(1) Entwicklungsumgebung**

- Modellierungsspezifische Erweiterungen der Programmiersprachen.
  Höhere, zur Entwicklung der PFSe verwendete Programmiersprachen wie MODULA2, ADA, PEARL, C sollten durch die erweiterte Laufzeitmaschine (3.4.2) als Modellierungssprachen unterstützt werden. Dazu sind einige, z.B. mittels Präprozessor- Ansätzen realisierbare Erweiterungen vorzunehmen.
  - Übertragung der Modellierungskonzepte des BS-Kerns auf Sprachebene (R-,S-,W-Tasks bzw. -module bzw. -packages);
    Schnittstellenanpassungen der Laufzeitroutinen der Sprachen zum BS-Kern, soweit Prozeßerzeugung und -kommunikation betroffen sind.

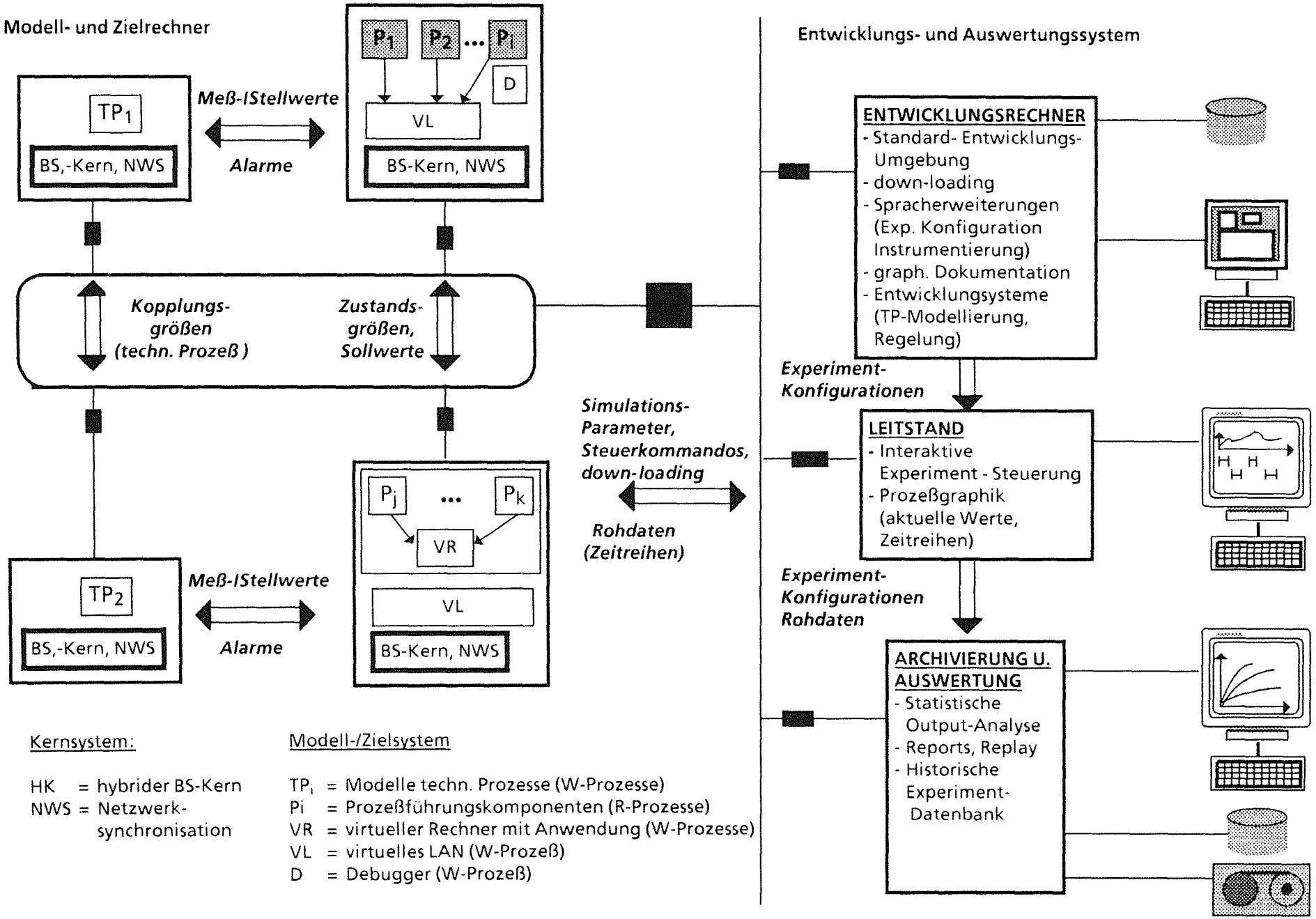



**Abb.** 3.14: Einbettung der Experimentrechnerkonfiguration in ein Entwicklungs- und Auswertungssystem

- Spezifikation virtueller BM-Konfigurationen (virtuelle Rechner- und Kommunikationstopologien);
  Zuordnung einer aus W-Prozessen bestehenden Anwendung im Stadium VB zu Komponenten einer virtuellen BM-Umgebung, einschließlich BM-Verbrauchsspezifikation der Anwendung
- Spezifikation der physikalischen Verteilung von realen Testobjekten und Modellen auf Experimentrechner, soweit der Systementwickler hierauf Einfluß haben soll:
  PFS-Anwendung ----> Experimentrechner (Stadium O)
  PFS-Anwendung ----> Zielrechner (Stadium R)
  Komponenten der MSS (3.4.5) ----> Experimentrechner
- Spezifikation einer Instrumentierung zur Leistungsgrößenerfassung, z.B. alternde Datentypen nach 3.1.3.2

- graphische Dokumentationshilfen (graphische Darstellung der Experimentrechnerkonfiguration, der virtuellen BM-Umgebung, der Objekt- bzw. Modulhierarchien, der Auftragsflüsse, z.B. durch Kanal-Instanz-Netze). Diese Informationen sind aus den statischen Strukturen der Modellierungs- bzw. Programmiersprachen ableitbar.
- Dialogkomponenten zur benutzerfreundlicheren Erstellung ausführbarer Modelle kontinuierlich-diskreter Prozesse (vgl. 6.2.1 sowie /BIL 84/,/MAR 84/)
- Entwicklungsumgebung für digitale Regelsysteme (Reglerentwurf, -analyse, Prozeßidentifikation);
  Schnittstelle zur Gewinnung realer Meßdaten des technischen Prozesses zwecks Prozeßidentifikation (in Abb. 3.14 nicht dargestellt)
- Numerische und statistische Unterprogrammbibliotheken (z.B. Zufallszahlengenerierung, numerische Integration)

**(2) Leitstand**

- Experimentsteuerung: Realisierung von Haltebedingungen und Dialogführung. Als Haltebedingungen sollten zeitliche Kriterien, Haltepunkte in Programmen und logische Prädikate (z.B. Ankunft von Botschaften gegebenen Typs an einer gegebenen Mailbox) möglich sein;
- dynamische Änderung ausgewählter Modellparameter an Haltepunkten (z.B. Tuning-Parameter des PFS, Störgrößen des technischen Prozesses)
- graphischer Arbeitsplatz (Prozeßgraphiksystem) zur Anzeige von Zeitverläufen der Regelgrößen, Antwortzeiten, BM-Auslastungen sowie zusätzlicher Testinformation, z.B. Gantt-Diagrammen des Prozeßablaufes; Darstellung des technischen Prozesses durch bewegte Objekte (Animation, /MAT 84/)
- Schnittstelle zum Archivierungs- und Auswertungssystem.

**(3) Archivierungs- und Auswertungssystem**

- Statistische Auswertung und Komprimierung von Rohdaten, z.B. Analyse auf Stationarität, Bestimmung empirischer Verteilungsfunktionen (Histogramme), Signalanalysen im Frequenzbereich (Fourier-Transformation);
- Spezielle Darstellungen von Experimentergebnissen für die off-line-Auswertung, z.B.
  - Summary-Reports
  - Vergleich und ggf. Verknüpfung der Ergebnisse mehrerer Experimente
  - wiederholtes Abspielen früherer Experimente mit Rücksetzmöglichkeit, ggf. in skalierter Echtzeit
- Experimentdatenbank zur Verwaltung von Experimentversionen, bestehend aus
  - Versionsname und -historie (Entwicklungsstand)
  - Experimentname
  - beteiligte Komponenten des Entwicklungsobjektes (PFS) und Umgebungsmodelle
  - Werte der parametrisierten Einflußgrößen (Modellparameter, ggf. mit Änderungshistorie im Experiment)
  - Experimentrechner-Konfiguration
  - erfaßte Leistungsgrößen und Experimentergebnisse
- Archivierung und Abfrage beliebiger historischer Informationen (Zeitreihen).

An der Schnittstelle zwischen Modellrechnern und Auswertungs-/Leitsystem werden zur Laufzeit zeitgestempelte, selbstbeschreibende Daten übertragen. Das Auswertungssystem ist in seiner Konzeption unabhängig von der Zeitführung der Modell-/Zielrechnerkonfiguration, basiert vorzugsweise auf einem Standard-BS (z.B. UNIX) und nimmt nicht teil an dem zwischen den Modellrechnern eingesetzten, speziellen Kommunikationsprotokoll der Netzwerksynchronisation. Für die Kommunikation mit dem Auswertungssystem kann ein anderes, einfacheres Protokoll verwendet werden (vgl. 5.2.4.2).

## 3.6 Begründung und weitere Anwendungen der Systemarchitektur

### 3.6.1 Begründung der Kernintegration

Der Ansatz der funktionellen Erweiterung konventioneller Echtzeit-BS-Kerne zu prozeßorientierten Simulatoren ist neuartig. Er zielt auf eine neue Generation von Echtzeit-Betriebssystemen. Um die Funktionsfähigkeit der Konzepte aber zu demonstrieren, muß ein existierender BS-Kern erweitert oder neu entwickelt werden. Implementierung und Test eines Pilotsystems sind infolge der hardwarenahen Programmierung erschwert. Daher ist es wichtig zu begründen, weshalb nicht eine andere SW-Architektur den Anforderungen in 3.4 gerecht werden kann.

Speziell sollen hier folgende Fragen untersucht werden:

(1) Weshalb wird das Simulationskonzept innerhalb des BS-Kerns realisiert, statt oberhalb?

(2) Weshalb bietet die Kern-Schnittstelle nur "einfache" Simulationsfunktionen für W-Prozesse an?

(3) Weshalb erfordert die Leistungsanalyse verteilter Zielsysteme verteilte Experimente?

Ausführlicher soll nur die Kernfrage (1) diskutiert werden.

Zu (2)

Nur dadurch, daß die Systemdienst-Schnittstelle **minimal** und **anwendungsneutral** gehalten wird, besteht eine Chance, daß eine funktionelle Erweiterung sich auch in Hersteller-BS-Kernen durchsetzt. Würde man z.B. versuchen, **anwendungsabhängige** Leistungen der Modellierungssysteme (3.4.5) auf BS-Kern-Ebene anzubieten, würde die Zahl der benötigten Dienste sehr stark anwachsen. Sicherheitsaspekte, erschwerte Programmierung und erschwerter Test sprechen ebenfalls dafür, den BS-Kern so kompakt wie möglich zu halten.

Zu (3)

Entscheidend ist, daß die Zuordnung der **R-Prozesse** zu Rechnern im Experiment identisch mit der im Einsatz sein sollte.Würde ein Experimentrechner R-Prozesse für einen anderen Zielrechner**typ** im Einsatz bewerten, so müßte z.B. mittels Cross-Compilern aus demselben Quellprogramm verschiedene Objektmodule (z.B. für INTEL sbc286 und M68020) erzeugt werden, deren Echtzeitverhalten im Ablauf gar nicht vergleichbar wäre .

Würde ein einziger Experimentrechner R-Prozesse ausführen, die im Einsatz auf mehrere Zielrechner verteilt sind, so müßte der Monoprozessor-BS-Kern eine verteilte, nicht mehr monotone Zeitführung einsetzen und im Grunde doch alle Probleme der verteilten Simulation bewältigen, aber ohne aus der Parallelität einen Nutzen ziehen zu können.

Zu (1)

Zunächst betrachten wir folgende naheliegende Architektur-Variante: ein Hersteller-BS-Kern (HBS abgekürzt) werde ohne Erweiterungen als Basismaschine sowohl für das RTO als auch für die simulierte Umgebung eingesetzt. Die zusätzlichen Anforderungen (Unterscheidung zwischen Ziel- und Wirtsprozessen und alle daraus resultierenden Konsequenzen für die Zeitführung, Zuteilung und Prozeßkommunikation) sollen durch einen 'Simulations-Controller' (SC) auf der Anwendungsebene erfüllt werden (Abb. 3.15). SC sei als normaler Anwenderprozeß unter HBS oder als Monitor realisiert. Alle für die Simulation wesentlichen Interaktionen, insbesondere auch die RW-Kommunikation und Zeitverzögerungen der W-Prozesse, werden unter Beteiligung von SC realisiert. Wir nehmen ferner folgendes einfache Ablaufmodell für die Prozesse an:

> Jeder rechnende Prozeß (R- oder W-) behält den Prozessor solange, bis er an einen Interaktionspunkt gelangt **(Koroutinenmodell).**

Interessant an diesem Modell ist, daß die wenigen Ansätze, die eine eingeschränkte Integration realer SW-Testobjekte in eine Simulation versucht haben - es sind dies i.w. /LOU 85/ (JADE) und /SAN 77/ (vgl. 2.4)- ein solches Koroutinenmodell voraussetzen müssen. Dann können die Prozesse mit Hilfe von SC scheinbar selbst - ohne Unterstützung durch HBS - die Regie für die Zeitführung übernehmen:

1) für jeden Prozeß wird eine eigene virtuelle Zeit tv (in SC) gehalten;
2) erreicht ein Prozeß einen **Interaktionspunkt** - und nur dann - geht die Kontrolle an SC über. Zwei Fälle sind möglich:

Fall 1 die notwendigen Bedingungen zur Terminierung der Operation, insbesondere auch globale zeitliche Bedingungen wie (GC) in 3.3.1, sind erfüllt: der Prozeß kann unmittelbar fortgesetzt werden;

Fall 2 Der Prozeß muß auf eine Botschaft oder auf eine globale zeitliche Bedingung warten. Er wird von SC als blockiert vermerkt und zusammen mit seiner aktuellen virtuellen Zeit tv eingetragen.SC wählt bisher blockierte Prozesse aus, deren Bedingungen inzwischen erfüllt sind, bestimmt tv und setzt einen Prozeß mit minimalem tv fort.

3) Bestimmung der virtuellen Zeit am Ende einer **interaktionsfreien Rechenphase** aus der Zeit tv bei seiner letzten Aktivierung:
   - bei W-Prozessen ergibt sich tv durch Summierung der seither ausgeführten DELAY's (evtl. 0)
   - bei R-Prozessen ergibt sich tv durch Addition der seit dem letzten Wiederaufnahmepunkt verstrichenen (gemessenen) Echtzeitspanne.

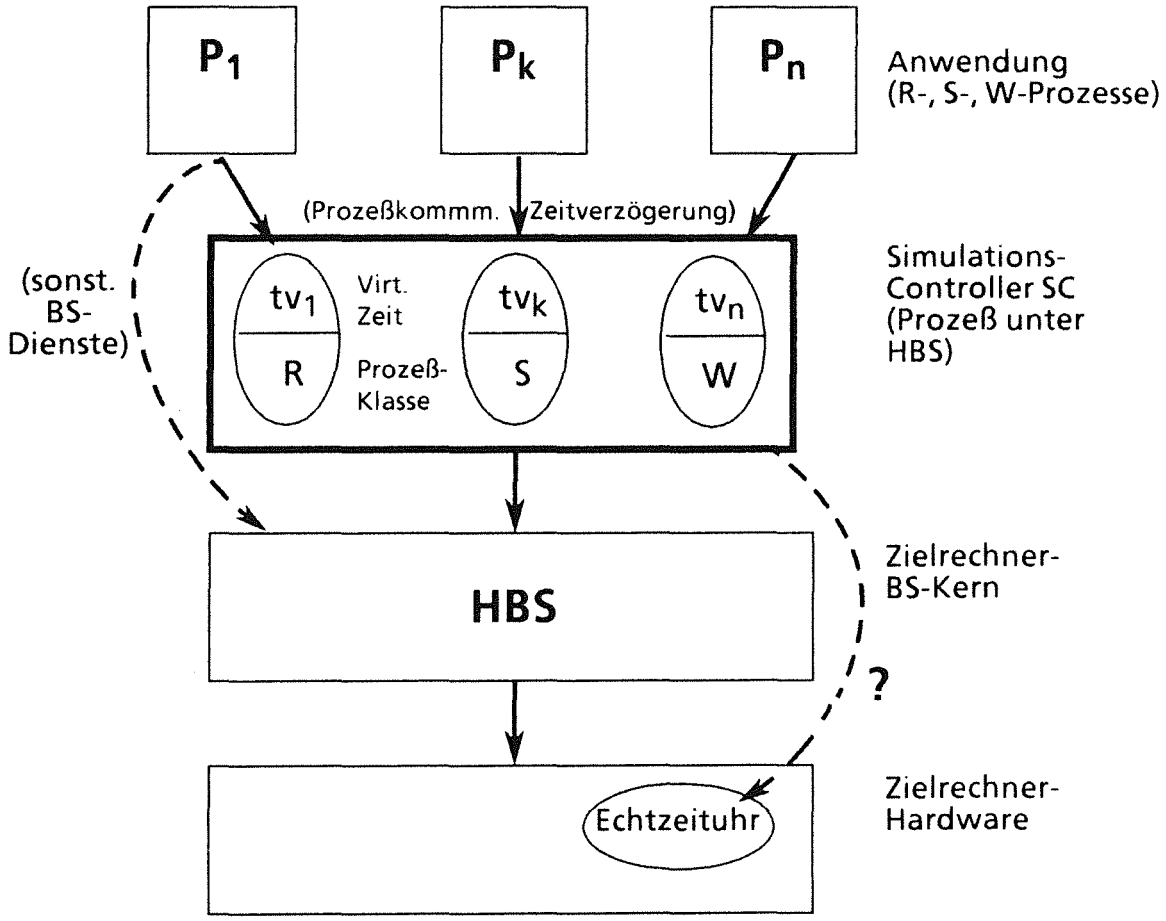



**Abb**. 3.15: Architektur-Variante ohne Erweiterung des BS-Kerns

Sicherlich handelt es sich hierbei um eine Art von Simulation, in die auch gemessene Zeiten einfließen. Das Problem dieses Ansatzes: er erlaubt keine belastbaren Aussagen über das tatsächliche Leistungsverhalten der R-Prozesse unter dem gegebenen HBS in einer den W-Prozessen verhaltensgleichen Echtzeitumgebung (zumindest die Anforderungen (AR1),(AR2) und (AR4) von 3.4.3 sind verletzt). Insbesondere:

- Die Zuteilungsmechanismen des realen Echtzeit-BS-Kerns (HBS) sind i.d.R. **unterbrechungsgesteuert**; Unterbrechungssignale der realen Umgebung müssen daher auch in der simulierten Umgebung auf Unterbrechungen mit gleichen Eigenschaften

führen; es ist nicht im voraus bekannt, ob ein Prozeß bis zum nächsten Interaktionspunkt rechnen wird.
- R-Prozesse synchronisieren sich neben Botschaften fast immer auch über Weckaufträge. Die Koordinierung solcher Echtzeit-Wartebedingungen mit den virtuellen Zeiten tv von SC gelingt durch obigen Mechanismus natürlich nicht.

Dies ist durch eine **gemeinsame** virtuelle Zeitachse für R- und W-Prozesse (3.4.3.3) möglich. D.h. solange die virtuelle Zeit tv für W-Prozesse angehalten ist, laufen auch keine Weckaufträge für R-Prozesse ab. Eine solche Zeitmodus-Steuerung ist aber auf der Anwendungsebene nicht korrekt realisierbar, da sie mit der BS-Kern-internen Verwaltung der Echtzeituhr kollidieren würde. Dieses Problem ist nur lösbar, indem alle Funktionen der Echtzeituhr zu einem abstrakten Datentyp zusammengefaßt werden, der von Anfang an sowohl auf die Bedürfnisse der R- als auch der W-Prozesse hin konzipiert wird (vgl. 4.2.2.3).

Neben der Zeitführung gibt es weitere Probleme, die auf der Anwendungsebene schwer oder gar nicht zu bewältigen sind:

- während W-Prozesse bei angehaltener virtueller Zeit rechnen, ist es trotzdem möglich, daß HBS auch Prozesse des realen Testobjektes ausführt (z.B. zur Ausnutzung der I/O-Wartezeiten von W-Prozessen). Dies führt zu einer inkorrekten zeitlichen Transparenz der R-Prozesse.
- die Konkurrenz der R- und W-Prozesse um Sekundärbetriebsmittel (ASP, E/A-Geräte, evtl. virtueller Speicher), die durch HBS verwaltet werden, kann zu ernsthaften Interferenzproblemen führen: das Verhalten der R-Prozesse ist abhängig vom Betriebsmittelbedarf (Implementierung!) der W-Prozesse.
- Die Entscheidung, **welcher** Zeitmodus (R-,F- oder D) überhaupt der richtige ist, ist auf Anwendungsebene nur schwer zu treffen. Ein D-Übergang erfordert z.B. einen vollständigen Überblick über die rechenbereiten und die auf zeitliche Bedingungen wartenden R-Prozesse. Selbst wenn ein Anwenderprozeß wie SC sich diese Information über HBS-Dienste beschaffen könnte, ist sie zu dem Zeitpunkt seiner Entscheidung u.U. nicht mehr zutreffend (Unterbrechbarkeit!).

Diese Einwände lassen sich alle auf einen gemeinsamen Nenner bringen: die Anforderungen an die Ausführung der R- und W-Prozesse in 3.4.3.2 kann nur die Instanz erfüllen, die jederzeit die vollständige Sicht der Prozeßzustände und die uneingeschränkte Kontrolle über die Vergabe der Rechnerbetriebsmittel besitzt, und das ist der BS-Kern, nicht ein einzelner (Anwender)Prozeß wie SC. Daher sollte zur Unterstützung der integrierten Simulation ein Echtzeit-BS-Kern (HBS) von Anfang an als hybrider Kern für beide Klassen von Prozessen konzipiert werden. Dadurch lassen sich die genannten, und weitere Probleme auf verhältnismäßig einfache und saubere Weise lösen. Letztlich ist es auch die ökonomischere Lösung: viele Funktionen eines "gewöhnlichen" BS-Kerns, die ein "aufgesetzter" Simulations-Controller zur Verwaltung der W-Prozesse replizieren müßte, z.B. Prozeßblockierung und -deblockierung, können von beiden Prozeßklassen gemeinsam verwendet werden.

### 3.6.2 Sprachlich heterogene Modellierungssysteme

Die mangelnde Eignung der block- oder warteschlangenorientierten reinen Simulationssprachen wie GPSS, SIMAN, INSIGHT oder SLAM /SCH 84//PEF 84//ORY 85//ROB 84/ zur Entwicklung verteilter DV-Systeme hat wesentlich dazu beigetragen, daß neben "gemischten" Simulations- und Programmiersprachen wie SIMULA auch viele bestehende höhere Programmiersprachen nachträglich um Pakete zur discrete-event-Simulation erweitert wurden (z.B. PASCAL /HUG 84/, ADA /ILU 84//ADE 83//WON 84/, T-PROLOG, PARLOG und SMALLTALK /BEZ 87/, dokumentiert u.a. im Konferenzband 'Simulation in Strongly Typed Languages' /SCS 84/ und in /LIU 87/). Die Erweiterungen umfassen Dienste für die Zeitverzögerung (HOLD bzw. DELAY), ein

Konzept für Nebenläufigkeit, z.B. Prozesse, Ereignisse, Transaktionen und deren Synchronisation, und ggf. statistische Grundfunktionen. Bei genauerer Betrachtung dieser Ansätze fällt auf

- Die neuen Konstrukte müssen auf die Basissprache aufgesetzt, d.h. mit den Mitteln der Sprache selbst implementiert werden, ob diese geeignet ist oder nicht. Transparenz und sichere Benutzung dieser Leistungen, Korrektheit und Effizienz ihrer Implementierung werden erschwert. Über Erfahrungen mit solchen Simulationspaketen in ADA berichten z.B. /DOB 84/ /BRY 82//POH 87/. /POH 87/ versucht das ADA-Rendezvous als Synchronisationsmechanismus auf virtuelle Zeit zu übertragen und kommt zu dem Schluß, daß die Leistungen des ADA-Laufzeitsystems, insbesondere des Standard-Rendezvous-Konzeptes, umgangen werden müßten, um eine korrekte Realisierung in ADA selbst zu erreichen.
- Es handelt sich um "Insellösungen"; die Simulationsmodelle können nicht als Teil eines übergeordneten Modells mit Komponenten in anderen Modellierungssprachen kooperieren.

Mit der funktionellen Erweiterung der über Rechnernetz gekoppelten Echtzeitbetriebssysteme zu verteilten Simulationsmaschinen ergeben sich folgende Perspektiven:

- Eine höhere, nebenläufige Programmiersprache S, die über geeignete Laufzeitprozeduren durch einen Echtzeit-Betriebssystemkern unterstützt wird (wie z.B. PEARL), kann durch den erweiterten Kern auch direkt als Simulationssprache unterstützt werden, was nur geringe Erweiterungen des Laufzeitsystems erfordert.

  Die Vorteile einer BS-Unterstützung werden zunehmend auch bei solchen Sprachen erkannt, die prinzipiell auch ohne diese implementiert werden könnten, wie z.B. das ADA-Projekt ASTERIX /BBK 86/ zeigt. In dem o.g. Beispiel /POH 87/ könnten z.B. die ADA-Sprachelemente für Rendezvous, einschließlich seiner zeitlich befristeten Sonderfälle, sowohl in der gültigen ADA-Semantik (Echtzeit) als auch in virtueller-Zeit-Semantik durch einen **gemeinsamen** Kern unterstützt werden.
- Sprachlich heterogene verteilte Simulationsanwendungen wie z.B. in /REY 88/ gefordert, sind auf dieselbe Weise realisierbar wie sprachlich heterogene verteilte Echtzeitanwendungen (die an die Entwicklungsumgebung - Übersetzer, Binder, Laufzeitbibliotheken - zu stellenden Anforderungen sind unabhängig von der virtuellen Zeit, man vgl. z.B. /FEN 87/).

Sicherlich ist ein solches Konzept auf weitere Fortschritte bei der Entwicklung von Netzwerk-BSen für heterogene Rechensysteme, der dazugehörigen sprachlichen Unterstützung und verteilter Anwendungen im allgemeinen angewiesen. Aber die Simulationstechnik wird Fortschritte auf diesen Gebieten solange gar nicht nutzen können, wie keine allgemein einsetzbaren Simulations-Betriebssysteme existieren, welche Basisfunktionen für Administration, Kommunikation, Synchronisation und zeitliche Verzögerung für unterschiedliche Sprachen anbieten, sondern diese Basisfunktionen in jeder Simulationssprache neu implementiert werden.

### 3.6.3 Interferenzarmes Testen verteilter Echtzeitsysteme

Unter Interferenz beim Testen von Echtzeit-Programmsystemem versteht man die Verfälschung des am Testobjekt zu beobachtenden Ablaufverhaltens durch die Testumgebung (z.B. Zeilenprotokoll, Debugger), man vgl. /AFH 85//LBR 85/. Gelegentlich werden Veränderungen des zeitlichen Ablaufes sogar bewußt herbeigeführt, um z.B. zeitabhängige Fehler bei der Synchronisation paralleler Prozesse durch Einfügen von Zeitverzögerungen zu maskieren. Umgekehrt verursacht ein Debugger selbst solche Zeitverzögerungen, und die Fehler können gerade dann verschwinden, wenn sie aufgedeckt werden sollen. Eine interessante empirische Untersuchung dieser Effekte (in ihren quantitativen Ergebnissen natürlich selbst der Interferenzproblematik unterworfen) bringt /GAI 86/.

Werkzeuge zur Programmbeobachtung/-Test sollen auch hohen Ansprüchen nach Komfort und Anwenderunterstützung genügen (vgl. etwa /BUM 88/):

- **on-line, interaktiv, anwendungsorientiert**: Auswertung von komplexen Datenstrukturen aus anwendungsorientierter Sicht (also nicht nur von Registern/Speicherzellen) und im laufenden Betrieb; der Benutzer kann diese Information an Haltepunkten anfordern, inspizieren, evtl. modifizieren und den Ablauf fortsetzen.
  Die Daten sollen also nicht nur zyklisch erfaßt und 'post mortem' analysiert werden wie in /GOF 88/ oder /TOK 88/.
- **realistische Lastsituation** und **realistische Betriebsdaten** im Test: das Testobjekt ist zumindest an ein dynamisches Modell des technischen Prozesses angekoppelt.

Bei debugging-Werkzeugen nach dem heutigen Stand der Technik sind die Interferenzprobleme vor allem in verteilten Systemen nicht beherrschbar.

- Läuft der Debugger als eigenständiger Systemprozeß unter dem Ziel-BS ab (z.B. dynamischer Debugger für RMX /INT 85b/) oder als anwendungsnaher Testtreiber/Monitorprozeß (wie z.B. in /TOK 88/), so hat er Zugriff auf alle Anwender-Datenstrukturen und Rechnerbetriebsmittel (z.B. Dateiverwaltung des BS, Netzkommunikation, Kommandointerpreter), was dem Verlangen nach Komfort entgegenkommt. Der Ablauf einzelner Prozesse verschiebt sich aber stark gegenüber dem ungetesteten Ablauf, selbst im Einprozessorfall, weil die Erkennung der Testereignisse und ihre Auswertung rechenzeitmäßig voll zu Buche schlagen.
- Werden die Testereignisse durch die HW erkannt (z.B. durch Unterbrechungsroutinen wie beim RMX system debug monitor /INT 85b/, oder durch in-circuit-Emulatoren), so ist die Ablaufverfälschung im Einprozessorfall zwar vernachlässigbar, aber Auswertung und Modifizierung der Daten erfolgen i.d.R. auf Wort-Ebene. Komplexe, zusammengesetzte Datenstrukturen sind zum Zeitpunkt der Unterbrechung i.a. gar nicht in einem aus Sicht

der Anwendung konsistenten Zustand, und die Spezifizierung der Testereignisse und ihre Auswertung erfolgen in einer separaten Kommandosprache, nicht der Sprache der Anwendung.

- In beiden Fällen verschiebt sich bei einem **verteilten** System der Ablauf des getesteten gegenüber dem der übrigen Rechner, ggf. kommt es zur Asynchronie der Echtzeituhren. In allen Fällen aber tritt Synchronisationsverlust mit dem zu steuernden realen oder in Echtzeit simulierten technischen Prozeß ein.

Der Debugger in Echtzeitsystemen findet in **Simulationsumgebungen** seine Entsprechung in den **interaktiven Haltepunkten (breakpoints)**. Hierbei gibt es - wegen der virtuellen Zeitführung - keine Interferenzprobleme. Zustand und Simulationszeit des gesamten, meist auf einem Rechner befindlichen Simulationsmodells bleiben erhalten. Leider sind die mit einer reinen Simulation erzielten Testergebnisse unvollständig und zum Teil nicht auf die Zielumgebung übertragbar; insbesondere durch das (Ziel)BS oder seine fehlerhafte Benutzung verursachte Fehler bleiben ausgeklammert.

Wiederum bietet die integrierte Simulation ein Hilfsmittel, um Vorteile aus beiden Welten zu ziehen: die reale Anwendung auf der Zielmaschine zu testen, aber die Interferenzprobleme stark zu reduzieren.

I.f. wird als Beispiel eine debugging-Anwendung betrachtet, in der der Nachrichtenverkehr des Testobjektes (PFS) über eine gegebene Mailbox mbx im Rechnerknoten K überwacht werden soll. Ähnliche Techniken sind aber auch anwendbar, um z.B. lokale Datenstrukturen eines Anwenderprozesses an Haltepunkten zu inspizieren. Die Testumgebung des PFS besteht aus folgenden Hauptkomponenten (vgl. Abb. 3.16):

- dem PFS in seiner verteilten Zielrechnerkonfiguration (nur R-Prozesse)
- dem simulierten technischen Prozeß (W-Prozesse)
- einem zusätzlichen Debugger-Prozeß (**W-Prozeß**) auf Knoten K, der auf das Eintreffen beliebiger oder durch spezielle Filterprädikate /LOU 85//HHK 85/ ausgewählter Nachrichten in mbx wartet, und im Bedienerdialog die Formatierung, Archivierung und ggf. Modifizierung der Nachricht durchführt. Der Debugger hat Zugriff auf die benötigte Datenperipherie (lokal oder auf das abgesetzte Entwicklungssystem über LAN).

Die Mailbox mbx ist als RW-Kommunikationsobjekt konfiguriert, damit sowohl die R-Prozesse als auch der Debugger sende- und empfangsberechtigt sind. Für diese Anwendung werden allerdings Kommunikationsoperationen mit spezieller Semantik benötigt (**Dämonenfunktion**, die gegenüber der in /SMI 84/ vorgeschlagenen Form allerdings modifiziert werden).
Die Funktion

```
demon_rec_msg     (
                  mbx  :  mailbox;
                  p    :  predicate;

            out   buf  :  message;
            out   rs   :  receive_id);
```

- blockiere den Empfänger (Debugger) solange, bis eine Nachricht vom Typ message, die p erfüllt, in mbx **eintrifft** (eine bei Aufruf von demon_rec_msg in mbx bereits lagernde Nachricht bleibt unbeachtet, anders als bei einer normalen receive-Operation).
- kopiere die Nachricht in buf, ohne sie zu **konsumieren**.

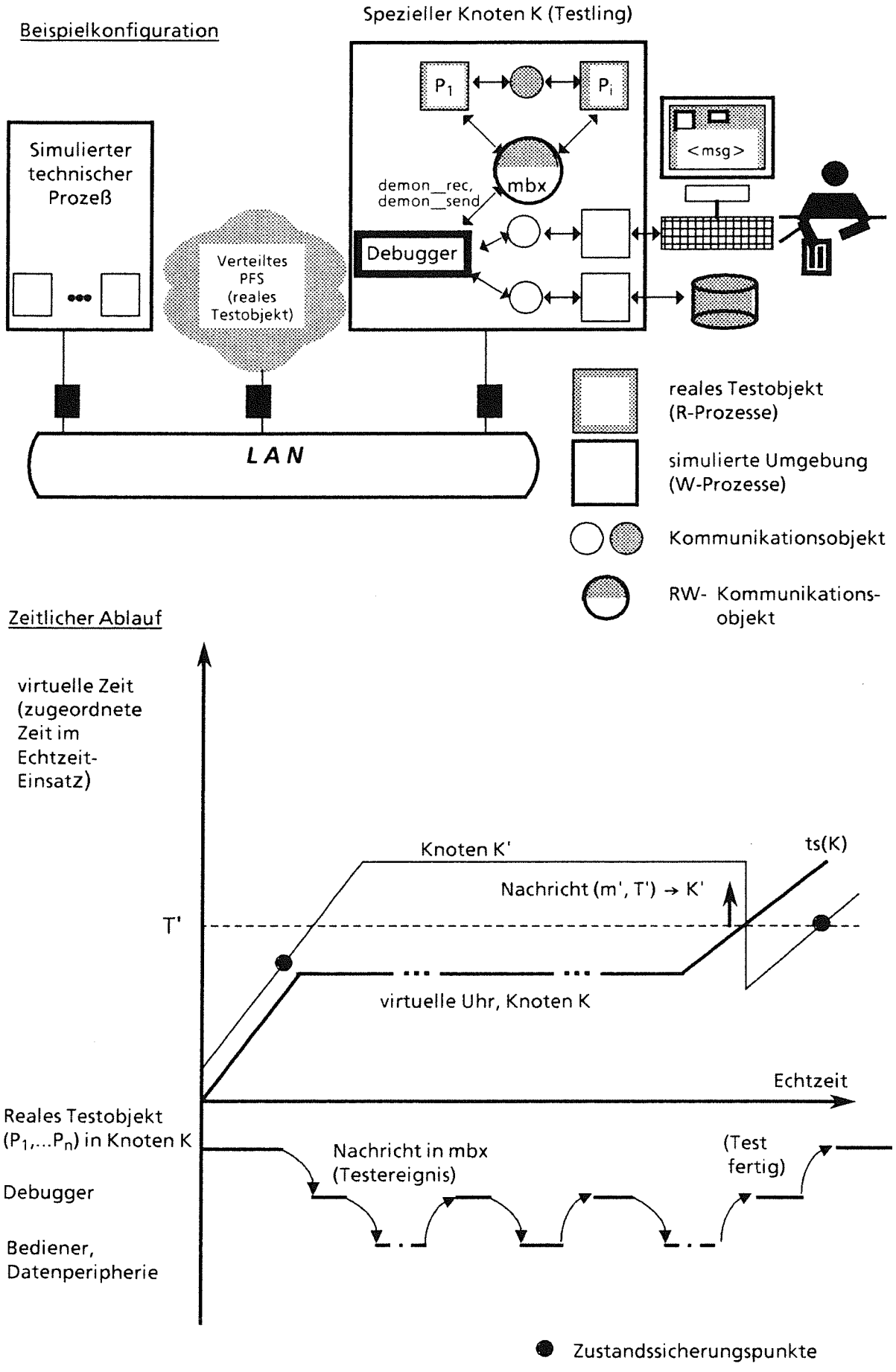



**Abb.** 3.16: Interferenzarmes Testen in verteilten Echtzeitsystemen

Durch

```
demon__send__msg   (
                   mbx  :  mailbox;
                   msg  :  message;
                   rs   :  receive__id);
```

sei es möglich, eine Nachricht ggf. geändert "auf denselben Platz" unter Bezugnahme auf eine frühere Leseoperation (demon__rec__msg, receive__id) zurückzuschreiben, falls sie nicht zwischenzeitlich konsumiert wurde. Damit wird erreicht, daß der Debugger keinen **funktionellen** Seiteneffekt auf die Kommunikations-Schnittstelle mbx ausübt, außer dem vom Bediener explizit gewünschten, und es wird verhindert, daß er bei unverändertem Zustand der Mailbox unnötig aktiviert wird.

Entscheidend für die Interferenzeigenschaften ist nun, daß der Debugger ein Wirtsprozeß in der integrierten Simulationsumgebung ist. Man betrachte

- ein System S1 = (PFS,TP,D) mit Debugger, wobei der Benutzer die Nachrichten nur inspiziert, aber nicht ändert, und
- ein System S2 = (PFS,TP) ohne Debugger.

Die Schnittstelle zwischen Debugger D und dem Restsystem (PFS, TP) besteht nur aus mbx (die Ein-/Ausgabeaktionen des Debuggers über die Datenperipherie laufen ohne direkte Kommunikation mit dem Testobjekt (PFS,TP) ab). In virtueller Zeit-Semantik schrumpfen die komplexen Debugger-Bediener-Interaktionen zu diskreten Punkten. Weil das reale Testobjekt mit D nur über mbx interferenzarm (3.4.3.2) gekoppelt ist, entspricht sein Ein-/Ausgabeverhalten einem Einsatzfall, in dem der W-Prozeß D fehlt, und stattdessen ein unsichtbarer Echtzeit-Prozeß zu diskreten Zeitpunkten auf mbx zugreift; d.h. aber, solange dieser die Nachrichten nur anschaut, einem **Einsatz ohne Debugger**.
Ohne im Detail vorwegzunehmen, wie dies realisiert wird (Kap. 4 und 5), sind i.w. zwei Mechanismen hieran beteiligt:

- **rechner-lokal** sorgt der **BS-Kern** während der Debugger-Tätigkeit dafür, daß
  - die rechnerlokale Zeit konstant bleibt;
  - die Prozeßzustände aller R-Prozesse konstant bleiben und insbesondere keine R-Prozesse rechnen
  - der Zustand aller sekundären Betriebsmittel (verfügbarer Arbeitsspeicher, E/A-Geräte) sich für R-Prozesse nicht **sichtbar** ändert.
- **netz-global** garantiert die **Netzwerksynchronisation**, daß alle Rechnerknoten nach der virtuellen Zeit der Nachrichten - also den Zeitverhältnissen im Einsatz - synchronisiert werden. Dazu vergleicht jede NWS die knotenlokale Uhr mit den Zeitstempeln ankommender Nachrichten. Eilen diese voraus, werden sie zeitverzögert zugestellt. Liegen sie zurück - wie in Abb. 3.16b die Nachricht m' von K - setzen der Knoten (K') und seine Uhr zu einem früheren Zustand zurück, aus dessen Sicht die "vergessene" Nachricht wieder in der Zukunft liegt.

Die Interferenzarmut dieses Mechanismus ist eine Konsequenz der allgemeinen Regeln der Kooperation und Betriebsmittelzuteilung von R-und W-Prozessen, und nicht speziell für den Debugger implementierter Mechanismen.

Die Vorteile der Anwendung von Nicht-Echtzeit-debugging-Methoden auf Echtzeitsysteme werden auch in /PAJ 84/ am Beispiel einer Steuerung für Kopierautomaten gezeigt; das Anhalten des Testobjekts wird hier durch eine HW-Schaltung (Unterbrechen der Taktversorgung der Gerätesteuerung) erreicht. Es handelt sich um eine zentrale, auf die spezielle Anwendung zugeschnittene Lösung.

Das Zurücksetzen eines Testobjektes, um zeitabhängige Fehler zu reproduzieren, wurde als Idee im Testsystem BUGNET /JBW 87/ für verteilte Anwendungen unter UNIX propagiert. Allerdings treten - konzeptbedingt - erhebliche, in harten Realzeitanwendungen nicht tolerierbare Ungenauigkeiten dabei auf. Gründe dafür sind das Fehlen einer virtuellen (echtzeitunabhängigen) Zeit, die fehlende Integration in die BS-Software (vgl. 3.6.1) und vor allem die Tatsache, daß ein verteiltes System immer als Ganzes - d.h. alle Komponenten simultan - zurückgesetzt werden muß, was kaum realisierbar ist (vgl. Kap. 5.3).

## 4. Erweiterter Monoprozessor-BS-Kern (HYBRIS)

In diesem Kapitel werden die wichtigsten Schritte bei der Erweiterung von Echtzeit-BS-Kernen zur Simulation beschrieben:

- die Erweiterung der Kern-Schnittstelle (Systemdienste, 4.1)
- die Ablaufsteuerung, insbesondere Zuteilungsregeln und Zeitführung (4.2)
- die Systemarchitektur, also die Hierarchie der (internen) BS-Objekte der Kernerweiterung im Vergleich zum reinen Echtzeit-Kern (4.3).

Der erweiterte BS-Kern wird im folgenden auch HYBRIS (**hyb**rid kernel for **r**eal time execution and **i**ntegrated **s**imulation) genannt.

Um Mißverständnisse zu vermeiden, wird noch auf zwei Punkte hingewiesen:

- **Funktionelle Erweiterung und Code-Erweiterung**
  Es wird gezeigt, wie eine gegebene funktionelle **Spezifikation** eines Echtzeit-BS-Kerns E zu einer Spezifikation E' für die integrierte Simulation erweitert wird, und wie bei **geeigneter** (objektorientierter) Realisierung des Gesamt-BS-Kerns auch ein vertretbarer Aufwand für due Erweiterung erzielt werden kann. Es wird **keine** Anleitung gegeben, wie der bereits ausgelieferte Code eines bestimmten Hersteller-BS-Kerns nachträglich erweitert wird. Dies würde voraussetzen, daß - abgesehen von der Verfügbarkeit des Quellcode und seiner Dokumentation - dessen Architektur der in diesem Kapitel beschriebenen entspricht. In der Regel wäre eine solche nachträgliche Erweiterung zu unwirtschaftlich und fehleranfällig.
- **Experiment und Einsatz**
  Im Echtzeit-Einsatz soll ebenfalls der erweiterte Kern E', und nicht die ursprüngliche BS-Version E eingesetzt werden. In der Einsatz-Konfiguration werden lediglich die Simulationsfunktionen nicht **benutzt**. Durch Verwendung des Kerns E' wird erreicht, daß das Verhalten des realen Testobjekts in Experiment und Einsatz identisch ist. Als Preis ist ein leichter Effizienzverlust (erhöhter overhead) gegenüber einem reinen Echtzeit-Kern E zu zahlen, der i.w. auf die erweiterte Kern-Schnittstelle (Prüfung von Zugriffsrechten) zurückzuführen ist.
  Dies läßt sich durch Verwendung spezialisierter BS-Versionen weitgehend vermeiden, was vor allem in Mehrrechnerumgebungen interessant ist (vgl. 4.3.2).

### 4.1 Kern-Schnittstelle

#### 4.1.1 Objekte, Operationen und Zugriffsrechte

Die BS-Architektur von E, wie die von E', sei objektorientiert /WET 84//INT 81//BOO 86/. D.h. Datenstrukturen zusammen mit einer Menge von Operationen zu ihrer Manipulation bilden die Einheiten der Modularisierung (Objekte genannt); der Zugriff auf die interne Datenstruktur eines

Objekts von außerhalb ist nur mit Hilfe der von ihm exportierten Operationen möglich.
Die meisten Objekte sind als 'type manager' organisiert: eine Objektvereinbarung legt einen **Objekttyp** fest, von dem mehrere Einzelobjekte (Instanzen) erzeugt werden können. Jede Operation enthält die zu manipulierende Objektinstanz als Argument.

Trotz ihrer aktiven Rolle können auch Prozesse (bzw. Prozeßinstanzen) als Objekte erzeugt, manipuliert und vernichtet werden. Daneben erzeugen Prozesse viele andere Arten von Objekten, z.B. Kommunikationskanäle, Betriebsmittelobjekte (Speicherbereiche, Seitentabellen, Objekt-Kataloge etc.) oder Zusammenfassungen von Prozessen und Betriebsmitteln zu Jobs wie in RMX 86 /INT 85a/ oder Teams in EOS /LIE 86/.
Als weitere "Dimension" der in E möglichen Prozeß- und Objekttypen kommt nun die Ausprägung als R-,S- und W-Prozesse (3.4) bzw. -objekte hinzu. Nun läßt sich diese Dreisortigkeit i.w. auf eine Zweisortigkeit zurückführen, weil ein S-Prozeß Zugriff auf dieselben Objekte und Operationen hat wie ein R-Prozeß, nur zusätzlich die Möglichkeit des simulierten CPU-Bedarfs.

Wichtig für die BS-Kern-Erweiterung E' sind nun zwei Entwurfsprinzipien

- **Symmetrie** der Objekttypen und Operationen für R- und W-Prozesse
- **Minimalität** der Interaktionen zwischen R- und W-Prozessen

W-Prozesse haben Zugriff auf dieselben Typen von Objekten und i.w. über die gleichen Operationen bzw. Verfahren wie R-Prozesse (vgl. (AW1), 3.4.3.1). Die Zugriffsstrukturen unter E lassen sich durch Relationen

(pt__i, ot__k, (op1,...,opj))

ausdrücken mit

pt__i : Typ des Zugreifers (Prozesses)
ot__k : Typ des referenzierten Objektes
(op1,...,opj) erlaubte Operationen auf ot__k, einschließlich Erzeugen, Löschen.

Diese Struktur wird dupliziert und die Zugriffsrechte spiegelbildlich auf die R- und W-Prozesse unter E' übertragen:

((pt__i,R), (ot__k,R), (op1,...,opj))
((pt__i,W), (ot__k,W), (op1,...,opj))

Konkret bedeutet dies

- Jede neu erzeugte Objektinstanz eines beliebigen Typs "erbt" das Merkmal R oder W von ihrem erzeugenden Prozeß;

- Ein Prozeß kann nur Objekte seiner Klasse über Operationen manipulieren.

Ein R- und S-Prozeß kann also z.B. nur R- oder S-Prozesse erzeugen, blockieren oder vernichten, ein W-Prozeß nur W-Prozesse.

| Prozeß \ Objekt | R | W |
|---|---|---|
| R oder S | c,d,* | -- |
| W | -- | c,d,* |

c = create
d = delete
* = beliebige Manipulationen

| Prozeß \ Kommunikationsobjekt | R | RW | W |
|---|---|---|---|
| R oder S | c,d,s,r | s,r | -- |
| W | -- | c,d,s,r | c,d,s,r |

c = create
d = delete
s = send
r = receive

Von dieser Regel gibt es genau eine Ausnahme: die in 3.4.1 erwähnten RW-Kommunikationsobjekte, über die Prozesse klassenübergreifend kommunizieren. RW-Kommunikationsobjekte werden zwar nur von W-Prozessen erzeugt und vernichtet, aber Prozesse aller Klassen sind sende- und empfangsberechtigt.

Da der BS-Kern E an der Kern-Schnittstelle den korrekten Zugriff auf Objekte ohnehin in der Regel abprüft, können die zusätzlichen Prüfungen leicht integriert werden. Geeignet zur Implementierung ist z.B. ein capability-Mechanismus /LIE 86/.

**Erläuterungen zur Minimalität der Interaktionen**

Durch die Minimierung der gemeinsamen Schnittstelle zwischen R- und W-Prozessen wird ein disziplinierender Effekt für die Systementwicklung erreicht, der beabsichtigt ist: daß R-Prozesse tatsächlich vollständige Implementierungen sind, die unverändert im Echtzeiteinsatz übernommen werden.

Als wesentliche Eigenschaft wurde in 3.4.3.2 die Interferenzfreiheit gefordert, d.h. das (zeitliche) Verhalten der R-Prozesse soll nur von dem Verhalten an der Schnittstelle zur simulierten Umgebung abhängen, und nicht von deren Implementierung. Das damit verfolgte Ziel, die Experimentergebnisse auf die Echtzeitumgebung übertragen zu können, ist aber nur dann erreichbar, wenn die Interaktionen im Experiment und Einsatz **funktionell** vergleichbar sind.
Hierbei ist es wichtig zu unterscheiden, auf welchen SW-Ebenen diese Interaktionen stattfinden. Auf der Anwendungsebene sind sehr vielfältige Interaktionen zwischen realem Testobjekt und simulierter Umgebung denkbar, aber hier geht es darum, welche Interaktionen durch Dienste des Monoprozessor-BS-Kerns realisiert werden müssen. Was sollte es z.B. bedeuten, daß ein W-Prozeß B die Priorität eines R-Prozesses A ändert, oder ein R-Prozeß A eine Seitentabelle für einen W-Prozeß B erzeugt? Dies wäre nur sinnvoll, wenn B im Echtzeiteinsatz ebenfalls für einen lokalen SW-Prozeß **desselben** Zielrechners wie A stünde; dann könnte aber B im Experiment ebenfalls als R- oder zumindest als S-Prozeß vertreten sein. Repräsentiert B dagegen einen beliebigen HW- oder SW-Prozeß der rechnerexternen Umgebung, und A bleibt im Echtzeiteinsatz derselbe, so kann A mit B auf der Ebene des BS-Kerns E nur über eine E/A-Geräte- bzw. Rechnernetz-Schnittstelle kommunizieren. Die Aufgabe der Schnittstelle zwischen R- und W-Prozessen im Experiment (RW-Kommunikationsobjekte) besteht genau darin, diese Geräteschnittstellen funktionell nachzubilden (vgl. (AR2), 3.4.3.1).

In diesem Zusammenhang ist auch noch die Schnittstelle und Zugriffsrechte auf die physikalischen E/A-Geräte im Experiment zu klären. Anders als bei den meisten Objekten ist die Gerätekonfiguration eines Experiments, d.h. die Menge der Geräteeinheiten und Unterbrechungseingänge **statisch**; diese "Objekte" können nicht wie z.B. Prozesse dynamisch erzeugt werden. Dennoch gilt bzgl. der Zugriffsrechte dieselbe Regel wie sonst: entweder sind auf ein E/A-Gerät nur R-Prozesse oder nur W-Prozesse zugriffsberechtigt. Im ersten Fall gehört das E/A-Gerät zur Echtzeitumgebung (reales Testobjekt, im Einsatz trifft dies auf alle Geräte zu), im zweiten Fall zur Nicht-Echtzeit-Umgebung.
Aber auch im zweiten Fall können R-Prozesse die Gerätefunktionen konkurrierend mit W-Prozessen benutzen. Die Lösung dieses scheinbaren Widerspruchs: in Wirklichkeit kommuniziert ein R-Prozeß (bzw. Treiber) nicht direkt mit dem Gerät, sondern über RW-Kommunikationsobjekte mit einem dem Gerät vorgeschalteten Hilfsprozeß, einem W-Prozeß (vgl. 4.2.5).

### 4.1.2 Übertragung vorhandener Dienste in virtuelle-Zeit-Semantik

Für diejenigen Dienste von E, welche auf W-Prozesse übertragen werden, ist die zeitliche Semantik noch zu spezifizieren (vgl. (AW2) in 3.4.3.1). Hierbei bleibt ein gewisser Ermessens-Spielraum, aber folgende Richtlinien erweisen sich in den meisten Fällen als ausreichend und sinnvoll.

(E1) Dienste von E, die den aufrufenden W-Prozeß niemals in einen Wartezustand versetzen (blockieren), sind zeitlich transparent (Klassen (K1), (K2) oder (K5) in 3.3.1).

(E2) Dienste, die eine Blockierung mit explizit begrenzter maximaler Dauer (Echtzeit) vorsehen und die auf W-Prozesse als virtuelle zeitliche Befristung interpretiert werden sollen, werden in die Klasse (K3) der Kommunikations- und Synchronisationsfunktionen eingeordnet. (Zeitliche Transparenz stünde im Widerspruch zur variablen zeitlichen Befristung).

(E3) Dienste in E, die nur eine zeitliche Verzögerung ohne zusätzliche (logische) Blockierungsbedingungen beinhalten, werden für W-Prozesse als explizite virtuelle Zeitverzögerung ((K4) in 3.3.1) übernommen.

(E4) Dienste, die zu einer Blockierung aufgrund momentan nicht verfügbarer, teilbarer, räumlich aufgeteilter Betriebsmittel führen (z.B. ASP-Segmente, Seitentabelle), sind zeitlich transparent ((K5) in 3.3.1).

(E5) Das Warten auf die Beendigung von Ein-/Ausgabeoperationen an E/A-Geräten ist zeitlich transparent (K5), wenn es sich um eine erwartete Rückmeldung zu einem zuvor ausgegebenen Geräteauftrag handelt (Bsp. Ein-/Ausgabeoperationen auf Hintergrundspeicher, Ausgaben auf Protokollperipherie).

(E6) Das Warten auf asynchrone Meldungen bzw. Geräteereignisse, die nicht durch einen Auftrag angefordert wurden und zu einem beliebigen Zeitpunkt oder auch gar nicht erfolgen können (z.B. Konsoleingabe, interrupt-fähige Digitaleingabe) wird wie Warten auf eine Nachricht ((K3) in 3.3.1) behandelt.
Da der Wartezustand in diesem Fall nicht durch einen Rechenprozeß, sondern ein **externes** Gerät beendet wird, setzt dies voraus, daß diesem Ereignis eine virtuelle Zeit zugeordnet wird (siehe 4.2.5.7).

### 4.1.3 Neue Dienste für W- und S-Prozesse

I.f. wird ein minimaler Satz von Diensten beschrieben, der die simulationsspezifischen Forderungen (AW3), (AR2) und (AS) (allgemeingültige Simulation, RW-Kommunikation, simulierter Betriebsmittelverbrauch) abdeckt.

**Initialisieren und Lesen der Uhrzeit**

**clock-init** (T: absolute_time); — Datentyp absolute_time beliebig, aber einheitlich, z.B. real oder (Tagesdatum, Uhrzeit)

Der zeitliche Beginn des Experimentes wird festgelegt.

**clock-read**: absolute_time;

Liefert die prozeßspezifische, aber nach 3.4.3.2 für alle R-, S- und W-Prozesse des Rechnerknotens einheitliche virtuelle Uhrzeit.

**Prozeßkommunikation und - synchronisation**

```
create-mbx      (mtyp   :  userclass;
                 c      :  capacity;
                 b      :  buffer_synchr;
                 s      :  strategy;
           out mb       :  mailbox_token);
```

Durch diesen Dienst wird ein Kommunikationsobjekt (Mailbox) mit der Identifikation mb (Resultatparameter) erzeugt.

mtyp legt für das erzeugte Kommunikationsobjekt fest, welche Klassen von Prozessen kommunizieren dürfen.

```
type        userclass  =  ( w, nur W-Prozesse
                            rw  beliebige Prozesse
                            );
```

R-Kommunikationsobjekte, über die ausschließlich R-Prozesse kommunizieren, werden durch E selbst (nicht durch die Erweiterung) zur Verfügung gestellt.
Die Kommunikation basiert auf einem begrenzt asynchronen Senden. Die Entkoppelung zwischen Sender und Empfänger wird durch den Parameter c, den zur internen Pufferung der Nachrichten bereitzustellenden Speicher, bestimmt (bounded buffer). Die möglichen Fälle sind

$c=\infty$ Asynchrones Senden ('no-wait-send'). Der Pufferspeicher wird nicht statisch begrenzt, sondern gesendet wird solange, wie überhaupt verfügbarer Speicher im System existiert.

$c<\infty$ Die Kapazität der Mailbox wird statisch auf c Einheiten begrenzt.

$c=0$ Die Kommunikation verläuft synchron nach dem Rendezvous-Konzept (es existiert kein Zwischenpuffer für Nachrichten).

Als Synchronisationseigenschaften b sind folgende Werte möglich:

```
type  buffer_synchr =  ( b,    Senden blockierend
                       fc,     Senden überschreibend
                               Empfangen konsumierend
                       fl,     Senden überschreibend
                               Empfangen lesend
                       );
```

Falls buffer_synchr=b und die Kapazität c der Mailbox erschöpft ist, wird ein Sender blockiert, bis der Kanal wieder aufnahmebereit ist. In den Fällen fc, fl und $c=0$ wird dann der Sender mit einem Fehlercode fortgesetzt, bei $0<c<\infty$ wird die älteste Nachricht in mb überschrieben. Im Fall fl kann dieselbe Nachricht mehrmals empfangen ("gelesen") werden, im Fall fc wird sie aus mb entfernt (konsumiert).

**Randbedingungen**

Für jedes RW-Kommunikationsobjekt ist zwingend vorgeschrieben, daß c **endlich** ist (statisch begrenzter Puffer). Die Notwendigkeit dieser Einschränkung wird in 4.2.4.3 noch begründet. Aber auch in der Echtzeitumgebung sind Kommunikationspuffer zwischen E/A-Geräten oder Transportmedien und Rechnern stets begrenzt, daher ist die Einschränkung sinnvoll. Die Optionen fc, fl sind dann aber notwendig, um völlige Geschwindigkeitsentkoppelung zwischen Rechnern und externen Datenquellen modellieren zu können, mit Datenverlust, falls Informationen schneller produziert als verarbeitet werden. Tab. (4.1) zeigt für jeden der vier Fälle $c=0/c>0$ und b/fl einen einfachen Anwendungsfall, der hierdurch beschrieben wird.

*) Verfeinerungen der Zugriffsstruktur eines RW-Kommunikationsobjektes (z.B. sendeberechtigt sind nur R- (W-), empfangsberechtigt nur W-(R-)Prozesse) sind zwar z.B. für die Schnittstelle zwischen Gerätetreibern und E/A-Geräten denkbar, aber es gibt auch Beispiele, in denen solche weitergehenden Restriktionen unerwünscht sind (z.B. debugger-Anwendungen wie in 3.6.3).

Durch die Strategie s wird ein Ordnungskriterium für die Nachrichten-Warteschlange (FIFO oder Priorität des Senders) und die Warteschlangen blockierter Prozesse (FIFO oder Priorität) vorgegeben.

**delete-mbx** (mb : mailbox_token);

vernichtet die durch mb referenzierte mailbox einschließlich der wartenden Nachrichten; wartende Prozesse werden mit einem speziellen Fehlercode fortgesetzt.

| c | buffer-opt. | fl | b |
|---|---|---|---|
| 0 | | Auftrag<br>Treiber ----------> Gerät<br><br>(unsynchronisiert,<br>d.h. ohne Warten<br>auf "Gerät frei" | Auftrag<br>Treiber <----------> Gerät<br>Fertigmeldung<br>(synchronisert) |
| > 0 | | bitserielle<br>Rechnerkopplung<br>Sender --------> Empf.<br><br>Gerät ----------> Int.-Rout.<br>Interrupt | bitparallele<br>Rechnerkopplung<br>Sender ----------> Empf.<br>(handshake) |

**Tab.** 4.1: Interpretationen der Kapazität/Überlaufoption eines RW-Kommunikationsobjektes

```
send-msg        (mb   :  mailbox_token;
                 buf  :  message;
                 gt   :  time_duration;
          out    rs   :  resultpar);

rec-msg         (mb   :  mailbox_token;
          out    buf  :  message;
                 wt   :  time_duration;
          out    rs   :  resultpar);
```

Mit Hilfe dieser beiden Dienste werden Nachrichten an Mailboxes gesandt und von dort empfangen.

buf: Identifikation (Adresse, Länge) eines Quell/Ziel-Speicherbereiches der Nachricht im Adreßraum des Aufrufers. Nachrichten sind auf BS-Kern-Ebene nicht typisiert.

```
type message = record
                 size  :  integer;
                 adr   :  array [1..size] of char;
               end;
```

wt: maximale Wartezeit eines Prozesses auf eine Nachricht
gt: maximale Wartezeit einer Nachricht auf einen Empfänger-Prozeß (Gültigkeitsdauer)
rs: Ergebnis der Kommunikationsoperation: Länge einer empfangenen Nachricht, Alter oder Generierungszeitpunkt der Nachricht, Erfolg der Operation (z.B. timeout)

Nachrichten werden explizit vom Sender- zum Empfängeradreßraum kopiert.

Die zeitliche Semantik der Nachrichtenkommunikation wurde in 3.3.1 i.w. bereits spezifiziert. Nachzutragen bleiben noch zwei Punkte:

- eine Sendeoperation im Fall buffer_synchr∈{fc,fl} ist für einen W-Prozeß als Sender immer zeitlich transparent, weil keine Blockierung möglich ist (4.1.2 (E1)) - eine zum Zeitpunkt T begonnene Sendeoperation, deren Nachricht bis T+gt nicht empfangen wurde, führt zur
  - Fortsetzung des Senders mit virtueller Zeit T+gt, falls dieser an mb sende-blockiert ist,
  - zur Vernichtung die Nachricht in mb zum Zeitpunkt T+gt, andernfalls.

Die Sonderfälle wt=0, gt=0, wt=∞, gt=∞ sind zulässig.

| Attribut / Komm. objekt | Kap. c | buffer-synchr | Wartezeit Empfänger | Wartezeit Botschaft |
|---|---|---|---|---|
| R | Mechanismen | durch E | vorgegeben | |
| RW | $0 \le c < \infty$ | b,fc,fl | $0 \le wt \le \infty$ | $0 \le gt \le \infty$ |
| W | $0 \le c \le \infty$ | b,{fc,fl} | $0 \le wt \le \infty$ | $0 \le gt \le \infty$ |

**Tab.** 4.2: Attribute von Kommunikationsobjekten

### Explizite Zeitverzögerung

In Anlehnung an 3.3.1 (K4) werden folgende Dienste bereitgestellt.

```
delay (<FOR  t: time_duration|
       UNTIL t: absolute_time>);
```

Der aufrufende (rechnende) W-Prozeß verzögert sich selbst für die angegebene Zeitspanne bzw. bis zur angegebenen Uhrzeit.

```
reschedule-process
                    (p   :   process__token;
          {<AT      t    :   absolute__time|
          AFTER     t    :   time__duration >});
```

Falls der Prozeß p in einer delay-Operation blockiert ist, wird die Wartebedingung aufgehoben; wenn AT|AFTER t spezifiziert ist, wird die zeitliche Wartebedingung für p mit der Absolut/Differenzzeit t neu festgesetzt. Dieser Dienst ist nur auf W-Prozesse anwendbar.

Die Wirkung der Dienste delay/reschedule kann zwar auch durch zeitüberwachte Nachrichtenkommunikation nachgebildet werden, doch werden sie aus Gründen des Modellierungskomforts und der -effizienz dennoch angeboten.

**Virtueller CPU-Zeitverbrauch**

Nach Abschnitt 3.4.6 können S-Prozesse (und nur diese) einen simulierten Bedarf an Betriebsmitteln auf der Ziel- und Experimentmaschine spezifizieren. Dieser Dienst wird nur für das Betriebsmittel CPU zur Verfügung gestellt (vgl. 4.2.3).

```
work    (                          Anzahl der benötigten Prozessorzyklen,
          c    :   integer);       umzurechnen in virt. zeitliche Dauer
```

### 4.1.4 Beispiel

Die Erweiterung der Systemdienst-Schnittstelle soll im Gesamtüberblick an einem einfachen, fiktiven Echtzeit-BS-Kern dargestellt werden. Eine konkrete BS-Erweiterung des BS-Kerns EOS /LIE 86/, der z.Teil ähnliche Dienste besitzt, wird in der Diplomarbeit /STU 88/ berichtet. Der vorhandene BS-Kern E verfüge über folgende Systemdienste.

```
create-process      (ldo   :   load__module;
                    ipar   :   initialparameter;
              out   p      :   process__token);
```

Über dem Ladeobjekt (Programm) ldo wird eine neue Prozeßinstanz, insbesondere ein Prozeßleitblock und -kontext, erzeugt und initialisiert und eine Referenz p als Resultat übergeben. Der Typ ladeobjekt enthält eine Adreßraumbeschreibung des zugehörigen Programms (Code-, Daten-, Kellersegment, Startadresse, ggf. Filename des Ladeobjektes auf Hintergrundspeicher) und wird hier nicht weiter konkretisiert.
Durch ipar kann der erzeugten Prozeßinstanz ein Satz von Parameterwerten mitgegeben werden (z.B. seine eigene Identität). Die neu erzeugte Prozeßinstanz ist zunächst 'inaktiv', d.h. wartet auf Aktivierung.

**terminate-process**

(p : process_token);

Die Prozeßinstanz p wird aus einem beliebigen Ausgangzustand in den Zielzustand 'inaktiv' überführt, ihre noch belegten BM werden freigegeben und ihr Prozeßleitblock vernichtet.

```
activate-process   (p    : process_token;
                    prio : integer;
        {<AT        t    : absolute_time DEADLINE d : absolute_time |
        AFTER       t    : time_duration DEADLINE d : time_duration |
        AFTER       t    : time_duration CYCLE    c : time_duration |
        ON          s    : signal DEADLINE        d : time_duration>};
```

Der Prozeß p wird entweder sofort, oder bei Eintreten eines Ereignisses (ON), oder zu einem bestimmten Zeitpunkt (AT) bzw. nach einer bestimmten Zeitspanne (AFTER) rechenbereit und ggf. zyklisch (CYCLE mit Periode c) reaktiviert. Sowohl die Priorität prio als auch die spezifizierte Frist d (DEADLINE) werden für die Prozessorzuteilung herangezogen.

**delay-for** (t : time_duration);

Der aufrufende Prozeß verzögert sich um die angegebene Zeitspanne.

**clock-read**: absolute_time;

Der aufrufende Prozeß liest die Uhrzeit.

```
prio-change   (p :   process_token;
               prio : integer);
```

Die Priorität des Prozesses p wird dynamisch geändert.

Durch Signale mit folgenden 4 Operationen wird ein einfacher Semaphor- Mechanismus ähnlich wie in EOS zur Verfügung gestellt.

```
create-signal   (s   : signal;
                 max : integer); ( maximaler Semaphor-Wert)

send-signal     (s   : signal) {<PR|NPR>};

wait-signal     (s   : signal);

delete-signal   (s   : signal);
```

Durch den Zusatz PR|NPR wird spezifiziert, ob eine Prozessorneuzuteilung erfolgen soll (vgl. MOBS /SLR 81/).

Zwei Systemdienste zur Anforderung bzw. Freigabe zusammenhängender Arbeitsspeicher-Blöcke werden zur Verfügung gestellt:

```
get-segment        ( anz  : integer;
                out  anf  : datensegment-adresse);

return-segment     (anz  :  integer;
                    anf  : datensegment_adresse);
```

Zur physikalischen Gerätekommunikation dienen folgende Funktionen send__dev, send__intr und rec__intr:

```
send-dev  (                                  Versorgen d. Gerätekontrollblocks
                g   : device__unit;          zu Gerät g mit Auftragsparametern pb
                pb  : device__request);      und Starten des Gerätes (rec__dev
                                             durch Controller-HW realisiert).

rec-intr  (
                it  : interrupt__vector;     Warten auf Geräte-Fertigmeldung
           out  st  : device__state);        unterbrechungsgesteuert, durch
                                             Treiber-Prozeß

send-intr (
                it  : interrupt__vector;     Senden der Geräte-Fertigmeldung
                it  : device__state);        an einen Treiber-Prozeß durch
                                             die Unterbrechungsroutine bei
                                             nichtmaskierter HW-Unterbrechung
```

Daneben existieren i.a. weitere Dienste zur Geräteinitialisierung, zum Anbinden einer Interrupt-Service-Routine an einen Interrupt-Vektor, sowie zum Anbinden einer Service-Routine an einen Treiber-Prozeß, die nur für die Rechnerinitialisierung gebraucht werden.

Eine **Erweiterung** dieser Schnittstelle für E' könnte z.B. wie folgt aussehen:

1. Der Dienst create__process wird erweitert zu (ersetzt durch)

```
create-process   ( ldo:                      load__module;
                   ptyp                      :  process__type;
                   ipar                      :  initialparameter;
                out p      :  process__token);
```

mit **type** process__type = (R, S, W);

Entsprechend dem Wert von ptyp wird eine R-, S- oder W-Prozeßinstanz erzeugt.

2. Die Dienste terminate, activate und delay__for werden für W-Prozesse übernommen, aber im Fall activate ohne die Attribute AT|AFTER|CYCLE|ON.
Auf den Dienst change__prio für W-Prozesse wird verzichtet, ebenso auf Signale.

3. Für Speichersegmente und E/A-Geräte stehen W-Prozessen dieselben Dienste zur Verfügung wie R-Prozessen.

4. Da die Dienste delay__for, clock__read unter 4.1.3 sich direkt aus der Übertragung der gleichnamigen Dienste von E in virtuelle-Zeit-Semantik ergeben, sind nur noch die Mailbox-Kommunikation (RW- und WW-Kommunikationsobjekte) und der Dienst WORK unter 4.1.3 neu aufzunehmen.

Die Zugriffsrechte der Prozesse sind in folgender Tabelle (4.3) zusammengefaßt.

| (Teil 1 von Tab 4.3) Beeinfl. Obj. / Prozeß | R- oder S-Prozesse | W-Prozesse | R-Segmente | W-Segmente |
|---|---|---|---|---|
| R od. S | create_process<br>terminate_process<br>activate_process<br>AT\|AFTER<br>CYCLE\|ON<br>delay_for<br>prio_change<br>clock_read<br>work (nur S-Prozeß) | -- | get_segment<br>return_segment | -- |
| W | -- | create_process<br>terminate_process<br>activate_process<br>delay_for | -- | get_segment<br>return_segment |

| (Teil 2 von Tab. 4.3) Beeinfl. Obj. / Prozeß | R-Gerät | W-Gerät | Signal | Komm. Objekt | RW-Komm. Objekt |
|---|---|---|---|---|---|
| R od. S | send_dev<br>rec_intr<br>send_intr | | create_signal<br>delete_signal<br>send_signal<br>wait_signal | | send_msg<br>rec_msg |
| W | -- | send_dev<br>rec_intr<br>send_intr | -- | create_mbx<br>delete_mbx<br>send_msg<br>rec_msg | create_mbx<br>delete_mbx<br>send_msg<br>rec_msg |

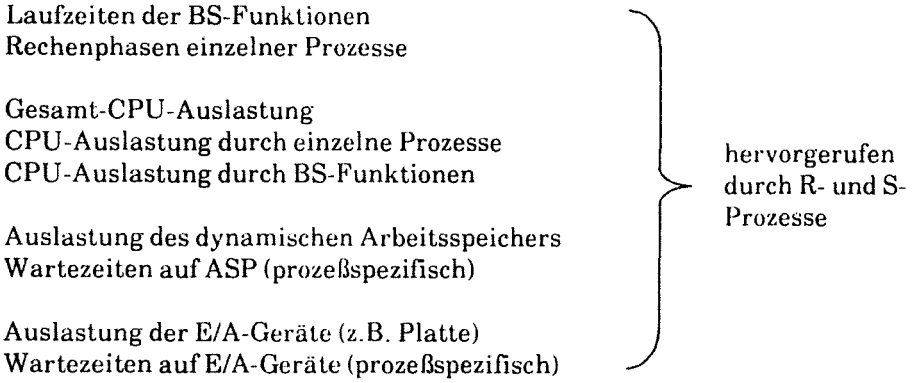
**Tab. 4.3**: Zugriffsrechte von Prozessen auf Objekte

### 4.1.5 Betriebssystem-Messungen

Von den Leistungsgrößen in 3.1.3 können nur wenige im BS-Kern bzw. auf der Kern-Schnittstelle ausgewertet werden. Die Führungsgrößen der technischen Prozesse und die Auslastungen logischer Betriebsmittel werden direkt über Dienste der betreffenden Modellkomponenten verfügbar gemacht, und die Antwortzeiten durch eine Instrumentierung der Datentypen der Modellierungs- und Zielsprachen.

Im BS-Kern bestimmbar sind lediglich Betriebsmittelauslastungen, denn der BS-Kern ist selbst ein logisches Betriebsmittel (gegeben durch die Menge seiner Dienste) und verwaltet physikalische BM. Deren Inanspruchnahme (Auslastung) durch das reale Testobjekt (R- und S-Prozesse) kann in Form folgender Größen ausgewertet werden.

Diese Meßgrößen werden durch zusätzliche Systemdienste zur Verfügung gestellt:




**laufzeit-statistik** (

p : process_token) : quadrupel;

**laufzeit-statistik-bs** (

b : bs_function) : quadrupel;

Das Ergebnis (quadrupel) enthält Minimum, Maximum, Mittelwert und Streuung der zeitlichen Dauer der seit Systeminitialisierung beobachteten Rechenphasen des ausgewählten Anwenderprozesses (laufzeit-statistik) bzw. der ausgewählten Betriebssystemfunktion (laufzeit-statistik-bs).

**auslastung** (

bm : resource_type;
p : process_token ∪ {L,*,BS} ): real;

liefert die durch verschiedene Benutzer verursachte Auslastung von bm seit dem letzten Aufruf von **auslastung** bzw. seit Systeminitialisierung (intervallbezogene Auslastung). Die Bedeutung der Parameterwerte für p ergibt sich aus folgender Tabelle:

| p (Nutzer) / resource_type | L | * | BS | process_token |
|---|---|---|---|---|
| CPU | Leerzeit | alle Prozesse | BS-overhead | Auslastung durch Prozeß p |
| ASP | freier dyn. Speicher | belegter dyn. Speicher | ( ) | ( ) |
| Gerät | Geräte-Leerzeit | Geräte-Belegtzeit | ( ) | ( ) |

Die für die CPU-bezogenen Größen notwendigen Rohdaten können aus einem einfachen Meßzyklus mit drei Meßpunkten

- $M_E$: Kerneintritt
- $M_A$: Kernaustritt
- $M_Z$: Prozeßwechsel

abgeleitet und die statistischen Parameter (Minimum, Maximum, Mittelwert, Streuung) ohne Speicherung von Zeitreihen 'on the fly' berechnet werden (vgl. z.B. /SCT 82/ für ähnliche BS-Messungen mit einem eigenen Meßrechner).

## 4.2 Ablaufsteuerung

Es wird i.f. beschrieben, wie ein korrekter Ablauf der R-,S- und W-Prozesse (vgl. 3.4.3.2, Anforderungen (AV), (AW4), (AR3), (AR4)) durch die BS-Erweiterung realisiert wird. Aus didaktischen Gründen gehen wir in mehreren Schritten vor:

- Zunächst werden nur die rechnerinternen Prozeßabläufe ohne die Kopplung mit E/A-Geräten und ohne Konkurrenz um einen endlichen Arbeitsspeicher betrachtet
- in 4.2.1 nur W-Prozesse (rein zeitdiskrete Simulation, wobei insbesondere Prozesse mit gleicher virtueller Zeit eine Rolle spielen)
- in 4.2.2, 4.2.3 die Kooperation mit R- und S-Prozessen über einer gemeinsamen virtuellen Zeitachse
- in 4.2.4 und 4.2.5 wird dann die Konkurrenz von R- und W-Prozessen um einen gemeinsamen, endlichen ASP sowie die Steuerung gemeinsamer E/A-Geräte diskutiert.

### 4.2.1 Reine prozeßorientierte Simulation

#### 4.2.1.1 Modell- und betriebsmittelbedingte Zustände von W-Prozessen

Es ist allgemein übliche Praxis, die Prozeßverwaltung und BM-Zuteilung in BS-Kernen durch Prozeßzustandsmodelle zu beschreiben, die die möglichen Zustände und Übergänge der Prozesse festlegen und um welche Rechnerbetriebsmittel sie sich bewerben, z.B.:

- wartend auf CPU-Zuteilung (bereit,b) oder rechnend (r)
- wartend auf Arbeitsspeicher (w__bm)
- wartend auf Fertigmeldung von E/A-Transfer (w__ea)
- zeitliche Wartebedingung (w__zeit)
- wartend auf ein Kommunikations- oder Synchronisationsereignis (w__co)
- inaktiv bzw. terminiert (Prozeß bewirbt sich nicht um Rechner-BM) (it)

Da W-Prozesse im Prinzip dieselben Arten von Operationen bzw. Systemdienste benutzen wie R-Prozesse, liegt ein solches Zustandsmodell auch den Wirtsprozessen zugrunde. Der Prozeßzustand (als Teil des Kontrollzustands c(p), vgl. 3.3.1) sei durch z(p) abgekürzt.

Auf Anwendungsebene gliedern sich die Aktionen eines W-Prozesses in zwei Gruppen auf

- solche, die grundsätzlich zeittransparent verlaufen (Klassen (K1),(K2) und (K5) in 3.3.1)
- solche, für die das i.a. nicht gilt (explizite zeitliche Wartebedingung oder Synchronisationsbedingung).

Es liegt nahe, diese Zweiteilung auf die Prozeßzustände auszudehnen, deren Übergänge ja eine Folge dieser Aktionen bzw. Systemdienste sind. Abb. 4.1 stellt die für einen W-Prozeß möglichen Übergänge des Prozeßzustands dar. Ein Prozeßzustand z sei **erreichbar** in einer Aktion <a>,

wenn ein Übergang x->z von einem beliebigen Zustand x als Folge der Aktion <a> oder einer ihrer Subaktionen möglich ist.

**Def. 4.1:** Ein Prozeßzustand z eines W-Prozesses heißt **betriebsmittelbedingt**, wenn er in einer zeittransparenten Aktion erreichbar ist. Andernfalls heißt er **modellbedingt**.

Sei <a0> eine Aktion, die einen W-Prozeß p aus einem beliebigen Zustand x in einen betriebsmittelbedingten Zustand $b \neq x$ überführt, und <a1> die erste nachfolgende Aktion, die p von b wieder nach $y \neq b$ überführt. Für die virtuelle Zeit tv(p) bzw. die in 3.4.3.3 eingeführte virtuelle Zeit der Prozeßaktionen muß dann gelten

$t_v(<a0>) = t_v(<a1>)$.

Da lokale Rechenoperationen eines Prozesses, die ohne BS-Unterstützung ablaufen, immer zeitlich transparent sind und ein W-Prozeß hierzu stets die CPU benötigt, sind rechnend (r) und bereit (b) in jedem Fall betriebsmittelbedingte Zustände. Ferner seien auch alle sonstigen Prozeßzustände, die mit der Inanspruchnahme von Betriebsmitteln des Experimentrechners zu tun haben, betriebsmittelbedingte *). Also

ZB := {b,r,w__bm,w__ea} Menge der betriebsmittelbedingten Zustände

w__bm und w__ea stehen für Warten auf beliebige Arten von Arbeitsspeicher oder E/A-Transfers.

Das Komplement, die modellbedingten Zustände, besteht aus

ZM := {it, w__zeit, w__co}.

Aus den Zuständen aller W-Prozesse ergeben sich folgende, dynamisch veränderliche, Prozeßmengen:

$M := M[R \cup S] \cup M[W]$ — Menge aller Prozesse
M[R ∪ S] : R- und S-Prozesse (in 4.2.1 ignoriert)
M[W] : W-Prozesse

$M[W] := FBB \cup FMB$

$FBB := \bigcup_{i=0}^{b} FW(i)$ — Menge der W-Prozesse in BM bedingten Zuständen.
Speziell
FBB := FB[W] ∪ FWB ∪FWEA mit
FB[W] - bereite oder rechnende W-Prozesse
FWB - auf passive BM (ASP) wartende W-Prozesse
FWEA - auf E/A wartende W-Prozesse

$FMB := \bigcup_{j=0}^{m} FM(i)$ — Menge der W-Prozesse in modellbedingten Wartezuständen.

*) Die modellbedingten Zustände im BS-Kern sind für die zeitliche Dynamik des Simulationsmodells relevant, die anderen dienen nur zur rechnergestützten Durchführung der Simulationsexperimente. Die Namensgebung bezieht sich auf die physikalischen Betriebsmittel des Experimentrechners.

Alle folgenden Ausführungen beziehen sich auf den Spezialfall, daß ein Prozeß sich immer in genau einem Prozeßzustand befindet und daher die obigen Mengen disjunkt sind. Sie lassen sich aber leicht auf den Fall verallgemeinern, daß ein Prozeß sich in mehreren Wartezuständen ($\neq$r,b) simultan befindet.

#### 4.2.1.2 Prozessorzuteilung und Zeitlistenverwaltung

Zunächst sind einige Vorbemerkungen über zeitliche Wartebedingungen notwendig.

**(1) Zeiteinträge, Zeitliste**

Alle W-Prozesse oder sonstigen Objekte, für die zeitliche (Warte)bedingungen spezifiziert sind (z.B. Nachrichten mit begrenzter Gültigkeitsdauer), sind durch einen **Zeiteintrag** ce in einer zeitlich geordneten **Zeitliste** zl repräsentiert:

ce = (t,x) (t$\in$ T Zeitstempel, x$\in$X weitere identifizierende Attribute)

zl$\in$ **F**(ZL) mit ZL: = T$\times$X

ce__min : **F**(ZL) -> T (kleinste zeitliche Wartebedingung).

$$ce\_min(zl) := \begin{cases} \min\{ce.t \mid ce \text{ in } zl\} & \text{falls } \neg leer(zl), \\ \infty & \text{sonst} \end{cases}$$

Es gebe ferner eine externe, monoton wachsende Zeitreferenz clock__**ack**:()->T, die die größte virtuelle Zeit liefert, zu der die nächste Aktion terminieren kann. Ihre Eigenschaften und Implementierung interessieren vorerst nicht, vgl. 4.2.2.3.

**(2) Ordnungskriterium**

zl ist in zeitlich aufsteigender Reihenfolge geordnet, wobei für **zeitgleiche** Zeiteinträge zwei Regime zur Auswahl stehen:

(a) SELECT

Zeitgleiche Zeiteinträge sind nach einer vorgegebenen Rangfolge der auf Zeit wartenden Objekte, z.B. W-Prozesse, geordnet.

Gegeben sei also eine Funktion

SELECT : $2^X \rightarrow X$, die aus einer beliebigen (endlichen) Menge von Objekten aus X das ranghöchste auswählt. Sei f: = $(ce_i)_{i=1..k}$ eine beliebige Teilfolge der Zeitliste zl mit $ce_1.t = ... = ce_k.t$. Dann muß stets gelten

erst(f).x = SELECT ($\{ce_1.x,...,ce_k.x\}$).

Die Reihenfolge gemäß SELECT-Funktion ist notwendig, um discrete-event-Netze in modularer Form korrekt simulieren zu können. Nach Def. 2.4 müssen zeitgleiche autonome Zustandsübergänge mehrerer DEVS-Komponenten in einer deterministischen, durch die SELECT-Ftn. festgelegten Reihenfolge ausgeführt werden.

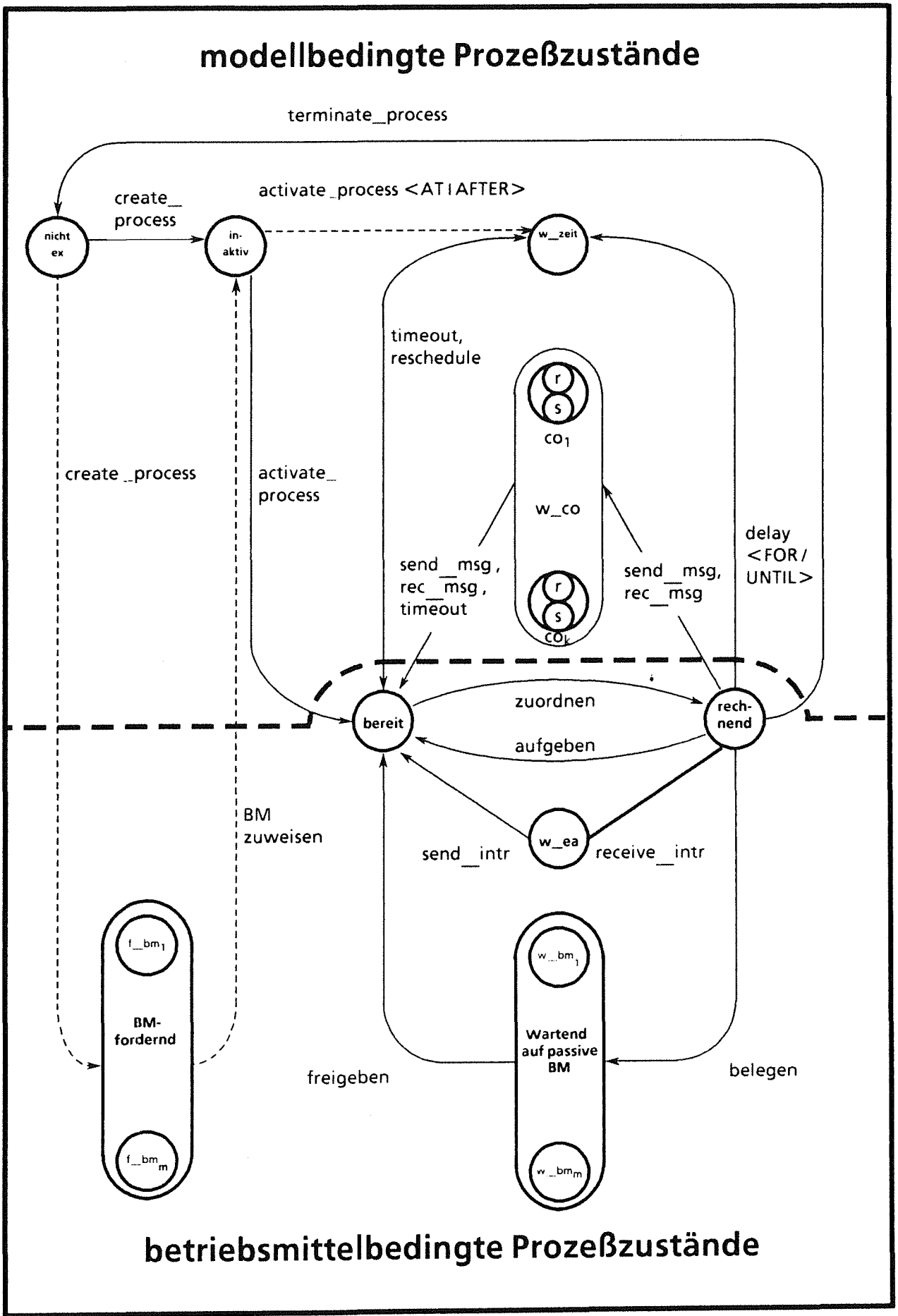



**Abb. 4.1** Prozeßzustandsdiagramm von W-Prozessen

(b) RANDOM

In manchen Fällen, besonders wenn W-Prozesse mit einem realen Testobjekt kooperieren (4.2.2), scheint allerdings eine **indeterministische** (zufällige) Reihenfolge zeitgleicher Zustandsübergänge vernünftiger. Diese RANDOM- Strategie wird wie folgt realisiert: bei jeder Einfügeoperation eines Zeiteintrags ce sei $ce_i,...,ce_{i+k-1}$ ($k \geq 0$) eine maximale Teilfolge von zl mit $ce_i.t = ... = ce_{i+k-1}.t$. Dann wird die neue Einfügeposition j für ce als Realisierung einer gleichverteilten Zufallszahl im Intervall [i, i+k] bestimmt.

**(3) Operationen auf Zeiteinträgen (zeitlichen Wartebedingungen)**

a) **Setzen** (Einfügen) eines Zeiteintrags ce, falls Bedingung ce.t≥clock__ack erfüllt; z.B. beim Übergang r->w__zeit eines W-Prozesses

b) **Aufheben** (Ausfügen) eines Zeiteintrags ce; z.B. beim Übergang w__zeit->b (reschedule) oder w__co->b (zeitüberwachte logische Bedingung wurde erfüllt, z.B. send__msg, rec__msg)

c) **Ablaufen** (Ausfügen) eines Zeiteintrags ce, falls folgende Bedingungen erfüllt sind:

- ce=erst(zl)
- clock__ack≥ce__min (Zeitpunkt ce__min "freigegeben")
- alle W-Prozesse sind modell-blockiert.

Wenn der Zeiteintrag ce mit ce.t=ce__min eines Prozesses abgelaufen ist, wird der Prozeß in jedem Fall wieder bereit und die virtuelle Zeit tv(p)=ce__min. Mit dem Bereitwerden (bm-bedingter Zustand) ist die dritte Bedingung für das Ablaufen weiterer, auch zeitgleicher, Zeiteinträge solange nicht mehr erfüllt, bis der Prozeß wieder modellblockiert wird. Eine zeitliche Wartebedingung kann dadurch immer erst dann ablaufen, wenn sie nicht durch einen Prozeß mit kleinerer virtueller Zeit oder ranghöheren zeitgleichen Prozeß mehr aufhebbar ist.

Die Prozessorzuteilung wird diktiert durch die möglichen Konstellationen von Prozeßzuständen aller W-Prozesse, genannt Prozessorzustand (Tab. 4.4), dessen Übergänge im **Prozessorzustandsdiagramm**, Abb 4.2, dargestellt sind. Der Übersichtlichkeit halber werden für die einzelnen W-Prozesse nur Elementarübergänge (blockieren, deblockieren) zwischen {b,r} und {w__bm, w__ea} bzw. {b,r} und {w__co,w__zeit} zugelassen, und keine direkten Übergänge zwischen {w__bm, w__ea} und {w__co,w__zeit}. Die resultierenden Änderungen des Diagramms bei solchen Übergängen sind jedoch trivial.

**Bedeutung der Prozessorzustände**

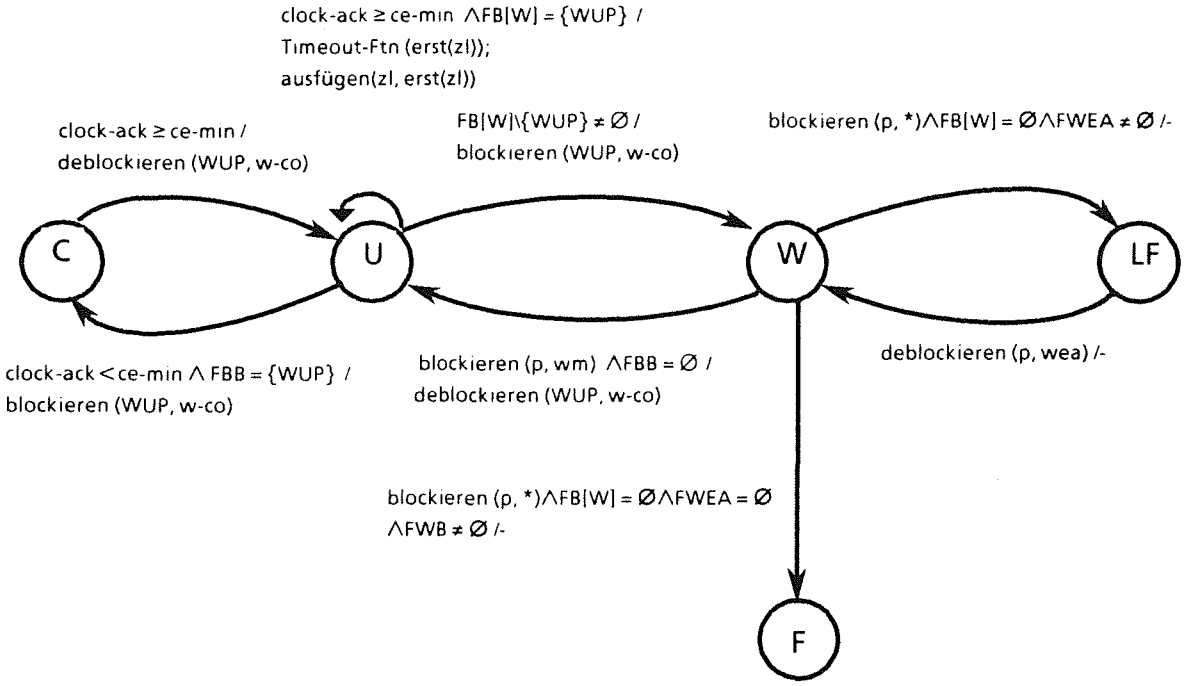
Die Zustände W, LF, und F sind durch die gemeinsame Obervoraussetzung FBB ≠ ∅ gekennzeichnet (es gibt W-Prozesse in betriebsmittelbedingten Zuständen); wegen (3c) dürfen keine zeitlichen Wartebedingungen ablaufen. Im einzelnen haben die Zustände folgende Bedeutung:

W Mindestens ein W-Prozeß ist bereit. Bereite W-Prozesse werden in einer für das Modellverhalten beliebigen Reihenfolge zugeteilt.

LF Es gibt keine zuteilbaren W-Prozesse, aber auf Fertigmeldungen von E/A-Aufträgen wartende W-Prozesse. Es wird ein Leerlaufprozeß zugeteilt, der untätig wartet. Falsch wäre es, wenn Zeiteinträge ce mit ce.t≤clock__ack abliefen und dadurch weitere Prozesse vorzeitig aktiviert (bereit) würden,

| Prozessor-zustand | Bedingung (Invariante) | rechnender W-Prozeß | Ablauf zeitlicher Wartebe-dingungen |
|---|---|---|---|
| W | FB [W] ≠ ∅ | erst(FB [W] ) | N |
| LF | FB [W] ≠ ∅ ∧ FWEA ≠ ∅ | Leerproz. | N |
| F | FB [W] ≠ ∅ ∧ FWEA = ∅ ∧ FWB ≠ ∅ | - - | N |
| U | FB[W] = {WUP} ∧ FWEA = ∅ ∧ FWB = ∅ ∧ce__min ≤ clock__ack | WUP | J |
| C | FBB = ∅ ∧ ce__min > clock__ack | Leerproz. | N |

**Tab.** 4.4: Prozessorzustände für reine prozeßorientierte Simulation




| [ de]blockieren (p, <* \|wm>) | Überführen eines W-Prozesses p in [aus] beliebigem (*) bzw. modellbedingtem (wm) Blockierungszustand |
|---|---|
| <ereignis> ∧ <p> / <aktion> | falls nach <ereignis> Bedingung <p> gilt, führe Zustandsübergang und <aktion> durch |

**Abb.** 4.2: Prozessorzustandsdiagramm für reine prozeßorientierte Simulation

z.B. die Zeitüberwachung eines auf eine Botschaft wartenden Prozesses p1, obwohl ein vorrangiger, momentan auf E/A wartender W-Prozeß p2 zum gegenwärtigen Zeitpunkt ce.t noch eine Botschaft an p1 senden könnte.

F Fehlerzustand ("Vorzeitiger Stop der Simulation mangels Betriebsmitteln") Folgende Verklemmungssituation liegt vor:

- Alle Prozesse in FBB sind blockiert wegen Betriebsmittelmangel (Menge FWB). Die Blockierung kann nicht durch externe Ereignisse (wie E/A-Fertigmeldungen), sondern nur durch betriebsmittel-belegende Prozesse aus FMB aufgelöst werden.
- Alle Prozesse aus FMB warten ihrerseits auf das Ablaufen zeitlicher Wartebedingungen oder auf Ereignisse oder Botschaften, aber beides setzt voraus, daß die zeitlich früheren oder zeitgleichen vorrangigen Prozesse in FBB zuerst zugeteilt werden.
  Die Verklemmungssituation ließe sich nur unter Verletzung der chronologischen Reihenfolge oder der Rangfolge zeitgleicher Prozesse auflösen.

U Alle Prozesse sind modellblockiert. Es gibt aber noch zeitliche Wartebedingungen, die ablaufen können (ce__min≤clock__ack). Es wird ein dedizierter W-Prozeß, der **W-Uhrprozeß** (WUP abgekürzt), mit der Aufgabe zugeteilt, abgelaufene Zeiteinträge aus zl auszufügen und die timeout-Aktionen für die betroffenen Objekte und Prozesse durchzuführen. Dies wird solange wiederholt, bis ein W-Prozeß bereit geworden ist, oder bis ce__min>clock__ack. Im ersten Fall wird sofort ein W-Prozeß zugeteilt (Rückkehr in Prozessorzustand W), im zweiten Fall wird auf die Bedingung ce__min≤clock__ack gewartet (Übergang nach C).

C Alle Prozesse einschließlich WUP sind modell-blockiert, aber die Bedingung ce__min≤clock__ack ist abzuwarten, bevor der Uhrprozeß wieder aktiviert und in den Zustand U zurückgekehrt wird.

Die Differenzierung zwischen den bm-bedingten Zuständen b, w__ea, w__bm ist notwendig, weil die Prozessorzuteilung direkt auf der Hardware der Experimentmaschine aufsetzt und entschieden werden muß, ob und welche Prozesse zuteilbar sind. In Koroutinen-Simulatoren, z.B. SIMULA, erfolgt diese Differenzierung nicht, weil ASP- und E/A-Zuteilung vom unterliegenden Wirts-BS übernommen werden und aus Sicht des Simulators immer nur eine Koroutine aktiv ist.

#### 4.2.1.3 Beispiel: Simulation von discrete-event-Netzen

In diesem Abschnitt sei die Aufgabe gestellt, ein discrete-event-Netz (DEVN) nach Def. 2.4 durch ein System kooperierender W-Prozesse unter dem hybriden BS-Kern zu realisieren. Mit diesem Anwendungsbeispiel

- wird konstruktiv gezeigt, daß der BS-Kern jedes zeitdiskrete Simulationsmodell realisieren kann (Allgemeingültigkeit, Def. 3.1 in 3.3.1)
- werden die Abläufe im BS-Kern, insbesondere die Prozessorzuteilung von 4.2.1.2 anhand eines konkreten Systems von W-Prozessen veranschaulicht.

Abb. 4.3 zeigt ein Programmfragment für eine einzelne DEVS-Komponente K(i) (i aus Indexmenge **K**); jedes K(i) wird durch einen W-Prozeß P(i) und eine zugeordnete Mailbox M(i) realisiert. Eine Ausgabe der DEVS-Komponente K(i) an K(j) ist gleichbedeutend mit Senden einer Nachricht λ(s,i) von P(i) an M(j). Die externe Zustandsübergangsfunktion δ__ex von K(j) wird beim Empfang einer Nachricht durch P(j) von M(j) aktiviert. Die M(i) haben die Kapazität 0; es handelt sich also um synchrone Kommunikation.

Die Zeigler'sche SELECT-Funktion $2^{\mathbf{K}} \rightarrow \mathbf{K}$ auf Teilmengen von Komponenten wird vereinfachend durch eine Prioritätsfunktion prio: $\mathbf{K} \rightarrow \mathbf{N}$ auf Komponenten nachgebildet, so daß die Komponente mit der höchsten Priorität zu jeder Teilmenge M mit SELECT(M) identisch sei.

```
type  range__devs is 1..no__devs;
type  result__par is (ok, timeout);


generic type interface__msg, devs__state is private;
        with function δ__ex(s:devs__state, e:time__duration, x:interface__msg)
                        return devs__state;
        with function δ__∅(s:devs__state)   return devs__state;
        with function ta (s:devs__state)    return time__duration;
        with function λ(s:devs__state, k:range__devs) return interface__msg;
        with function infl  return set of range__devs;
        with function own__id return range__devs;

package devs__component is

    host process p is priority (--entsprechend SELECT-Ftn.--) end;

end devs__component;


package body devs__component is

    host process body p is      --W-Prozeß--

        t__last : absolute__time := clock__read;
        s       : devs__state  := s0;
        x       : interface__msg;
        res     : result__par;
        i       : range__devs;

        begin
            loop
                rec__msg (M(own__id), x, ta(s), res);

                case res is
                    when ok          ⇒  s:= δ__ex(s, clock__read-t__last, x);

                    when timeout     ⇒  begin
                                           s:= δ__∅ (s);
                                           for i member of infl
                                               send__msg (M(i), λ(s,i), ∞, res);
                                       end;
                end case;

                t__last:= clock__read;
            end loop;
        end p;

    create__mbx (w, 0, b, FIFO, M(own__id));   -- W-Mailbox, Kapazität 0
                                               -- Senden blockierend

end devs-component;

-- Initialisierung --
i       : range__devs;
K       : array (range__devs) of devs__component;
M       : array (range__devs) of mailbox;

for i in range__devs
    package K(i) is new devs__component
                    (--aktuelle Parameter für generische
                    --Datentypen interface__msg, devs__state
                    --und Funktionen δ__ex,δ__∅,ta,λ, infl, own__id);
```

**Abb.** 4.3: Pseudocode-Formulierung einer DEVS-Komponente eines DEVN

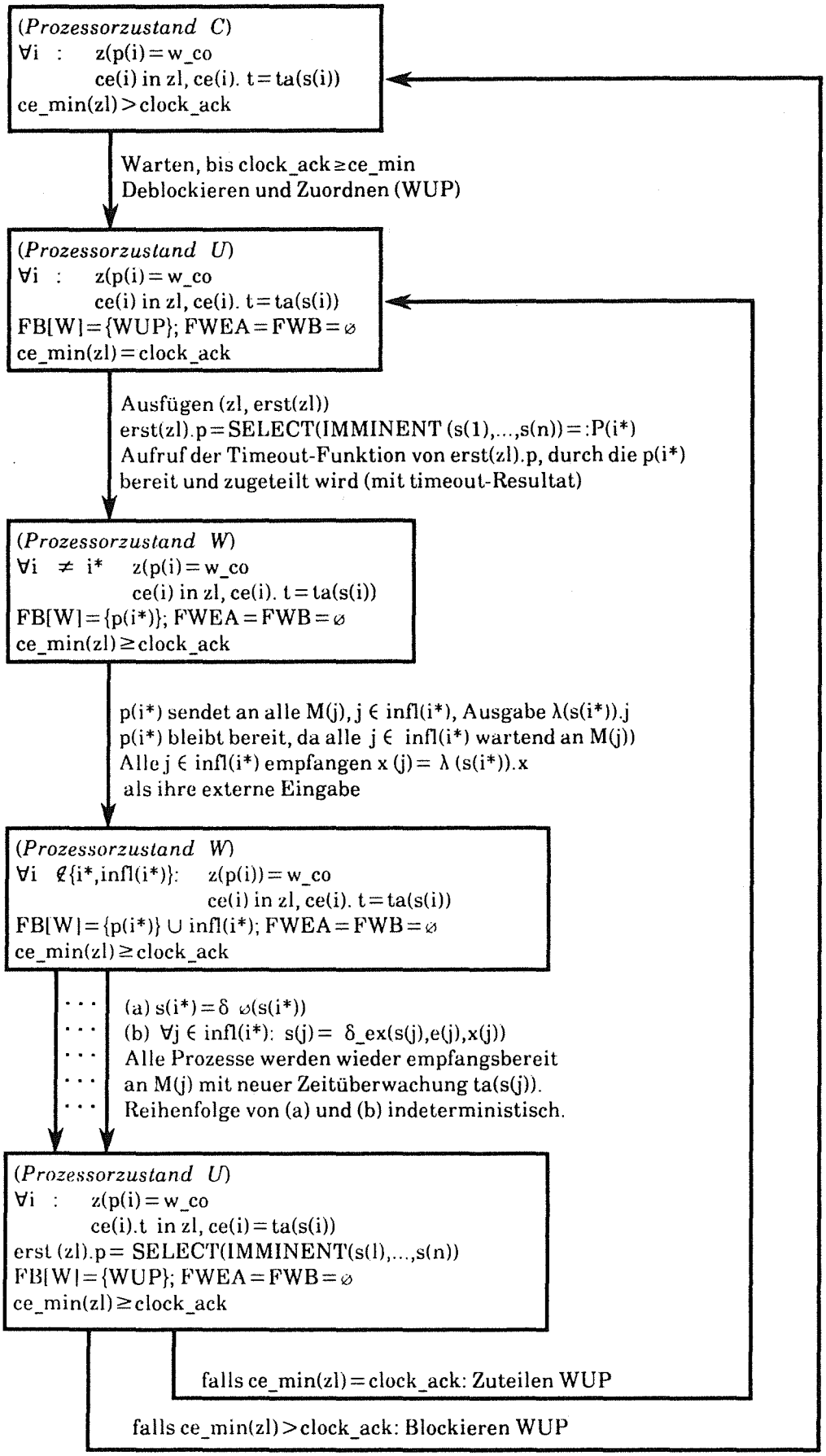



**Abb.** 4.4: Durchlaufene Prozessorzustände bei der Simulation des DEVN nach Abb. 4.4

Die W-Prozesse P(i) sind eingebettet in einen Datentyp (package) devs__component, der für jede Anwendung neu zu spezifizierende generische Datentypen und Funktionen enthält (interface__msg bzw. devs__state, δ__ex, δ__∅, ta, λ, influencee, vgl. Def. 2.3).
Trotz der Anlehnung des Pseudocodes in Abb 4.3 an ADA wurden hier die Nebenläufigkeiten explizit durch BS-Konstrukte (W-Prozesse, Kommunikationsobjekte) ausgedrückt, um deren Funktionsweise zu zeigen, und nicht durch das ADA-Rendezvous-Konzept. Abb. 4.4 zeigt die beim Ablauf dieses konkreten Prozeßsystems im BS-Kern durchlaufenen Prozessorzustände, die Prozeßmengen und den Zustand der Zeitliste nach 4.2.1.1, 4.2.1.2. Das Diagramm liefert einen informellen Nachweis dafür, daß nur solche Prozeßabläufe unter dem BS-Kern möglich sind, in denen alle autonomen und externen Zustandsübergänge der K(i) in der durch die Spezifikation des DEVN (Def. 2.4) vorgeschriebenen Reihenfolge stattfinden. Um einen formalen Nachweis zu führen, müßte der BS-Kern zusammen mit den W-Prozessen selbst als ein zweites discrete-event-Netz DEVN' spezifiziert und ein DEVN-Homomorphismus ('system coupling morphism' /ZEI 84a/) von DEVN' auf das Original-DEVN konstruiert werden.

Auf den ersten Blick nicht selbstverständlich ist, daß die Simulation trotz der synchronen Kommunikation verklemmungsfrei arbeitet, d.h. terminiert. Wannimmer aber ein Prozeß P(i) an eine Mailbox M(j) zu senden wünscht, ist der zugehörige Empfängerprozeß P(j) empfangsbereit und versucht nicht selbst gerade zu senden. Dies gilt aufgrund folgender Eigenschaften des DEVS-Mechanismus und seiner Realisierung:

- Jede Sendeoperation, also Ausgabe von P(i), ist das Ergebnis eines unmittelbar vorausgegangenen autonomen Zustandsübergangs von P(i).
- Jeder autonome Zustandsübergang ist das Ergebnis einer unmittelbar vorausgegangenen durch Zeitüberwachung beendeten Empfangsoperation. Da immer nur **eine** zeitliche Wartebedingung auf einmal abläuft, sind alle anderen P(j) noch empfangsbereit (w__co), auch wenn sie zeitgleiche Zeiteinträge besitzen.

Könnte eine Komponente als Folge eines externen Zustandsübergangs δ__ex(s,e,x) direkt eine Ausgabe erzeugen, ohne zwischendurch empfangsbereit zu werden, oder würden mehrere zeitgleiche Wartebedingungen auf einmal ablaufen und alle W-Prozesse deblockiert, so würden im obigen Prozeßsystem leicht Verklemmungen entstehen.

### 4.2.2 Prozessorzuteilung und Zeitführung für R- und W-Prozesse

#### 4.2.2.1 Prozessorzuteilungsregel

Das Ablaufmodell von 4.2.1 wird nun so erweitert, daß zusätzlich R-Prozesse mit den W-Prozessen über einer gemeinsamen virtuellen Zeitachse kooperieren, bzgl. der alle Aktionen chronologisch geordnet sind (vgl. 3.4.3.2). Der Verlauf dieser virtuellen Zeit wurde grob in 3.4.3.3 durch die Zeitmodi (R, F, D) charakterisiert; ihre Realisierung wird in 4.2.2.3 in Angriff genommen. I.f. geht

es zunächst um die Steuerung des Zeitmodus und die Zuteilung der R- und W-Prozesse. Beide sind i.w. durch eine einzige Regel miteinander verknüpft.

(R1) Sobald ein W-Prozeß in einen bm-bedingten Zustand eintritt, d.h. ein Übergang stattfindet:

FBB = ∅ -> FBB ≠ ∅ **(Aktion E0)**

so erfolgt ein Prozessorentzug des gerade rechnenden R-Prozesses, und der F-Zeitmodus wird eingeschaltet. Es können solange keine R-Prozesse zugeteilt werden, bis der letzte W-Prozeß einenBM-bedingten Zustand wieder verläßt. Jeder solche Übergang:

FBB ≠ ∅ -> FBB = ∅ **(Aktion E1)**

führt zur Wiederaufnahme eines R-Prozesses und des R-Zeitmodus.

**Aktion E0**

Aufgrund der vorläufigen Einschränkungen des Ablaufmodells zu Beginn von 4.2 gibt es genau zwei mögliche Situationen:

(a) Die früheste zeitliche Wartebedingung für einen W-Prozeß läuft im R-Modus der Virtuellen Uhr ab (Übergang C->U im Diagramm (4.3) bei erfüllter Bedingung clock_ack ≥ ce_min). Dadurch wird der W-Uhrprozeß rechenbereit, folglich ist FBB ≠ ∅.

(b) Ein R-Prozeß aktiviert durch eine Sende- oder Empfangsoperation an einem RW-Kommunikationsobjekt einen wartenden W-Prozeß (dies würde einem im Diagramm 4.3 nicht vorgesehenen Übergang C->W entsprechen).

**Aktion E1**

Der letzte W-Prozeß wird modell-blockiert, und es gilt clock_ack < ce_min (Übergang U->C im Diagramm 4.2).

Wir zeigen i.f., daß die restriktive Zuteilungsregel (R1) **notwendig** ist, um die Anforderungen (AW4),(AR3) und (AV) von 3.4.3.2 zu erfüllen. Dazu betrachten wir den Verlauf der Virtuellen Uhr als Funktion der Echtzeit (Funktion ts in 3.4.3.3), sowie zwei Aktionen E0 und E1.

(1) Zunächst muß im gesamten offenen (Echtzeit)Intervall

$I_e := (t_e(E0), t_e(E1))$ der F-Zeitmodus gelten, bzw. (gleichbedeutend)

$\forall t \in I_e$: $ts(t) = t_v(E0) = t_v(E1) = const.$

Da

$$I_e \subset \bigcup_{p \in P} (t_e(E0_p), t_e(E1_p)),$$

($E0_p$: Eintrittsaktion f. Prozeß p in einen Zustand ∈ ZB
$E1_p$: Austrittsaktion f. Prozeß p aus einem Zustand ∈ ZB)

also $I_e$ von Zeitintervallen bm-bedingter Zustände einzelner W-Prozesse überdeckt wird, in denen tv bzw. ts nach Def. 4.1 konstant sein muß, gilt dies trivialerweise auch für das Gesamtintervall $I_e$.

(2) Gäbe es ein nichtleeres Intervall (T1,T2) $\subseteq I_e$ ($t_e(E0) \leq T1 < T2 \leq t_e(E1)$), in dem ausschließlich R-Prozesse rechnen, so folgt aus (AR3) (Bewertung der R-Prozesse in Echtzeit), daß $t_s(T2)-t_s(T1) = T2 - T1$ gilt.
Selbst wenn man in der Praxis eine gewisse Abweichung (Drift) im R-Modus relativ zur "idealen" externen Echtzeit in Rechnung stellt, gilt
$1-\varepsilon \leq |t_s(T2)-t_s(T1)|/|T2-T1| \leq 1+\varepsilon$ mit $\varepsilon << 1$,
also in jedem Fall $t_s(T1) < t_s(T2)$ (und nicht lediglich "$\leq$").

3) Aus
$t_e(E0) \leq T1 < T2 \leq t_e(E1)$
folgt wegen (2) und der **Monotonie der virtuellen Zeit** (AV)
$t_v(E0) \leq ts(T1) < ts(T2) \leq t_v(E1)$ im Widerspruch zu (1) : $t_v(E0) = t_v(E1)$.

**Bemerkungen**:

- Es ist möglich, daß $I_e$ längere Leerlaufphasen enthält, in denen gar keine W-Prozesse bereit sind (Zustand LF in Tab. 4.3, Warten auf E/A-Geräte). Die Aussage von (R1) ist: diese Wartezeiten dürfen nicht durch bereite, bei E0 unterbrochene R-Prozesse ausgenutzt werden. (R1) ist also nicht mit einer Prioritätsfestlegung oder Strategiefunktion zu verwechseln ("W-Prozesse haben höhere relative Dringlichkeit als R-Prozesse"), sondern R- und W-Prozesse besitzen strukturell unterschiedliche Zuteilungseigenschaften.
- In (R1) erkennt man das prinzipielle Verfahren zur Echtteilesimulation von 3.3.2 wieder. Das Verlassen der Experimentphase E in Abb. 3.8a entspricht der Aktion E0. Das Verlassen der Simulationsphase S in Abb. 3.8 korrespondiert mit der Aktion E1, wobei anstelle von tv__su > tv die Bedingung ce__min > clock__ack steht.
  Die Beobachtungs- und Restaurationsphasen (B) und (R) in Abb. 3.8 werden durch das Unterbrechen (Anhalten) der R-Prozesse bei E0 bzw. Fortsetzen bei E1 realisiert.

#### 4.2.2.2 Leerlauf und D-Übergang

Falls die Menge FBB leer ist, d.h. insbesondere keine bereiten W-Prozesse in FB[W] existieren, rechnen nach (R1) bereite R-Prozesse. Falls auch FB[R] leer ist, liegt die Situation vor, in der in Betriebssystemen ein Leerlaufprozeß zugeteilt wird (nicht zu verwechseln mit dem in Tab. 4.3 eingeführten speziellen Leerprozeß im Prozessorzustand LF!). Falls keine E/A-Geräte oder externe Signalquellen existieren, welche Aufträge senden und den Leerlauf beenden können, könnte der Leerprozeß nichts anderes tun, als abzuwarten, bis die nächste **zeitliche** Wartebedingung abläuft. Statt in Echtzeit zu warten, stellt er im erweiterten BS-Kern die Uhr direkt auf diesen Zeitpunkt und ermöglicht damit den schnellstmöglichen Ablauf des nächsten Zeiteintrags.

```
process LD is loop          kern-interner Prozeß
        D-Übergang (..);    Zeitpunkt des frühesten Zeiteintrags für
    end loop;               R- oder W-Prozesse (vgl. 3.4.3.3)
```

Sobald der Zeitsprung ausgeführt wurde, werden die abgelaufenen Zeiteinträge ausgefügt und die zugehörigen Timeout-Aktionen durchgeführt. Falls noch immer kein Prozeß bereit ist, erhält erneut der Leerprozeß die Kontrolle, und der nächste Zeiteintrag wird abgerufen u.s.f. Der Prozeß LD ist die einzige Instanz im hybriden Kern, in der ein Zeitsprung (D-Übergang) stattfindet.
Erlaubt ist ein Zeitsprung immer dann, wenn keine rechnerexterne **Echtzeitumgebung** im Experiment aktiv ist (vgl. 4.2.5.3 und 5.1.1). Für reine Simulationsanwendungen ohne R-Prozesse ist das Setzen der Uhr in definierten Inkrementen (unabhängig von den Eigenschaften, z.B. Auflösungsvermögen, der Echtzeituhr) sogar die einzig sinnvolle Art der Zeitführung.

#### 4.2.2.3 Virtuelle Uhr

In diesem Abschnitt wird die Virtuelle Uhr nach 3.4.3.2, 3.4.3.3 im Detail entworfen. Sie bietet zugleich

(i) den W-Prozessen die Schnittstellen und Eigenschaften einer Simulationsuhr, d.h. sie ist unter der Kontrolle der Modellprozesse und schreitet nur auf Anforderung und nur in definierten Inkrementen fort;
(ii) den R-Prozessen die Schnittstellen und Eigenschaften einer "normalen" Echtzeituhr.

(ii) ist deshalb bemerkenswert, weil die Virtuelle Uhr kein absolutes Maß für die verstrichene Echtzeit liefert. Aber Zeit als Einflußgröße eines DV-Systems beruht ohnehin **immer** auf einer **Abstraktion** der Zeit (z.B. durch den Wert eines Uhrregisters, durch Weckersignale oder durch Operationen eines abstrakten Datentyps (ADT) bereitgestellt). Diese Abstraktion ist manipulierbar; es kommt nur darauf an, daß alle Programme **dieselbe** zeitliche Abstraktion referenzieren, und daß die **Zuordnung**

Zeit <-> dynamischer Zustand der Rechenprozesse

in simulierter Umgebung **dieselbe** ist wie in Echtzeitumgebung. Wir werden dieselben Geschwindigkeitstransformationen (lokales Dehnen bzw. Strecken um Faktor 0,1, oder $\infty$) auf die Virtuelle Uhr relativ zur externen Echtzeit anwenden, wie auf die Dynamik der Zustandsänderungen der R-Prozesse in simulierter Umgebung relativ zu ihren Pendents in Echtzeitumgebung.

Gegeben sei die **Echtzeituhr** des zu erweiternden BS-Kerns E, die o.E. folgende Operationen zur Zeitreferenzierung anbietet:

(1) **clock-read:** absolute__time;

Lesen der Uhrzeit, diese **Lesezeit** ist Referenz für zeitabhängige Programmentscheidungen, dient zur Zeitstempelung von Programmvariablen etc.

(2) **clock-ack**: absolute__time;

Aktualisieren der **Weckzeit**, die entscheidet, welche zeitlichen Wartebedingungen als abgelaufen gelten sollen.

Die Aktualisierung geschieht in regelmäßigen Zeitabständen durch eine **Uhrinterrupt-Service-Routine** (UISR), die ihrerseits durch Unterbrechungssignale einer Weckuhr (**Intervall-Timer**) aktiviert wird.

Wir nehmen i.f. an, daß der Intervall-Timer auf äquidistante physikalische Weckintervalle $\Delta$ (Ticks) eingestellt ist, und daß die Inkrementierung clock__ack dies nachvollzieht. Dies trifft auf fast alle bekannten BS-Kerne zu. Die Spezifikation der Virtuellen Uhr kann aber leicht auf variable Inkremente übertragen werden, falls der Echtzeit-BS-Kern dies unterstützt.

(3) **clock-init** (t__init: absolute__time, $\Delta$: time__duration);

Initialisieren (Stellen) der Uhr bei Rechneranlauf, mit t__init: Lesezeit und $\Delta$: Inkrement für Weckzeit.

Beim ersten Aufruf von clock__ack nach clock__init sind alle Zeiteinträge $\leq$ t__init + $\Delta$, nach dem k-ten alle Zeiteinträge im Intervall (t__init + (k-1)$\Delta$, t__init + k$\Delta$] abgelaufen (Weckzeitraster). Lesezeit und Weckzeit können identisch sein; dennoch ist es sinnvoll sie zu unterscheiden. Während die Prüfung auf abgelaufene Zeiteinträge immer in regelmäßigen, im voraus bestimmten Zeitabständen erfolgen muß, kann die momentane Uhrzeit i.a. abgelesen werden, wann immer sie benötigt wird.

Die Virtuelle Uhr ist eine **Erweiterung** dieses ADT um 3 Operationen (vgl. 3.4.3.3):

(4) **clock-freeze**;

Anhalten der Virtuellen Uhr, Übergang in den F-Modus

(5) **clock-go** (tw: absolute__time);

Fortsetzen der Virtuellen Uhr, Übergang in den Echtzeit(R)-Modus. tw ist der Zeitpunkt des frühesten Zeiteintrags der simulierten Umgebung (W-Prozesse), möglicherweise $\infty$.

(6) **clock-jump** (tr,tw: absolute__time);

Diskreter Sprung der Virtuellen Uhr (D-Übergang) zu dem durch die frühesten Zeiteinträge tr,tw des RTO bzw. der SU bestimmten Zeitpunkt.

Die Wirkung der neuen Operationen wird i.f. beschrieben. Abb. 4.5 zeigt den Verlauf der virtuellen Zeit als Funktion der Echtzeit.

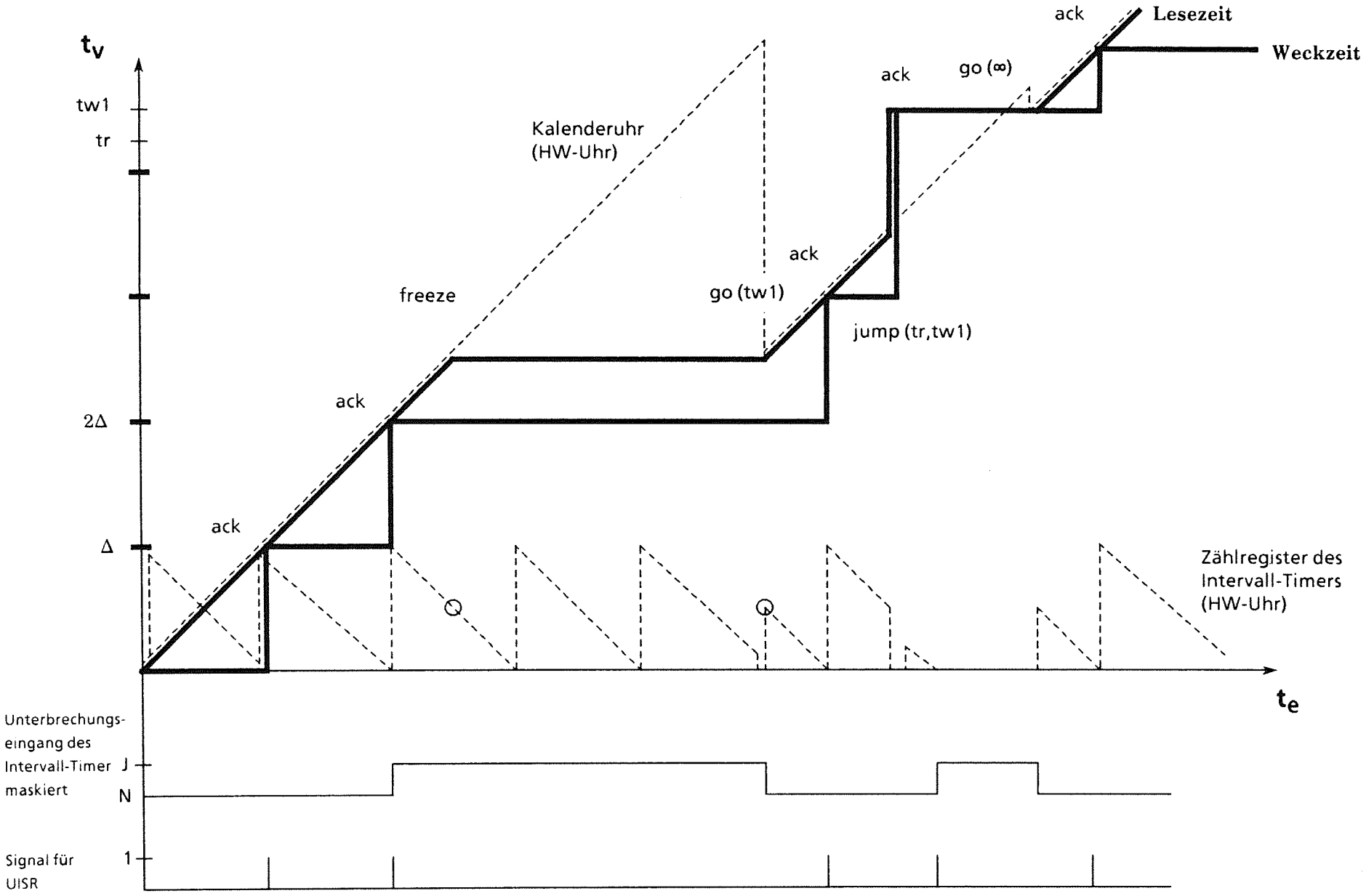



**Abb.** 4.5 Zur Wirkung der Operationen der virtuellen Uhr und ihrer möglichen Realisierung

Die Virtuelle Uhr verhält sich wie eine gewöhnliche Echtzeituhr (1)-(3), solange keine der neuen Operationen freeze, go, jump verwendet werden.

**Wirkung der Operationen unter Einbeziehung von freeze, go:**

a) read angewandt nach freeze (beliebig oft) liefert stets denselben Wert. read ändert den internen Zustand des ADT nicht.

b) ack angewandt nach freeze (im F-Modus) liefert denselben Zeitwert wie die letzte ack-Operation und verändert im F-Modus ebenfalls nicht den internen Zustand der Virtuellen Uhr. (Im F-Modus laufen keine Zeiteinträge ab!)

c) Der F-Modus ist **lesezeit-transparent**, d.h. read unmittelbar **nach** einer Sequenz freeze;go(tw) liefert denselben Wert wie unmittelbar **vorher**.

d) ack angewandt nach clock_go(tw) liefert als Zeitwert das Minimum folgender Größen
   - den auf den Zeitwert der **letzten** ack-Operation folgenden Rasterpunkt im Weckzeitraster des RTO
   - den kleinsten noch nicht abgelaufenen Zeiteintrag der simulierten Umgebung (Parameter tw einer früheren go- oder jump-Operation).

   Die Weck-Zeit respektiert also stets die chronologische Reihenfolge aller Zeiteinträge im RTO oder SU. Darüber hinaus bleibt das Weckzeitraster des RTO erhalten, unabhängig von der Verteilung und der zeitlichen Dauer der zwischengeschobenen F-Zeitmodi auf der Echtzeitachse (**Weckzeit-Transparenz** des F-Modus). Lese- und Weckzeittransparenz des F-Modus für das reale Testobjekt sind ein Gebot der **Interferenzfreiheit** (AR4 in 3.4.3.2), denn die Verteilung der F-Modi auf der Echtzeitachse (Anfangszeiten und reale Dauer) ist ein Zufallsprodukt der Implementierung der simulierten Umgebung und darf keine sichtbaren Auswirkungen auf das Leistungsverhalten des RTO haben.

e) freeze und go sind idempotent (da der F- bzw. R-Modus nach einer freeeze- bzw. go-Operation bereits gilt).

**Einbeziehung der jump-Operation:**

f) Nach jump ist jede weitere Operation go, freeze oder jump wirkungslos, bis eine ack-Operation die Weckzeit tatsächlich zum Sprungzeitpunkt inkrementiert hat.

g) Der Sprungzeitpunkt von jump(tr,tw) ist das Minimum von tw und dem frühesten Rasterzeitpunkt $\geq$ tr (vgl. d)), zu dem der Zeiteintrag tr des realen Testobjektes ablaufen kann. ack angewandt nach jump liefert diesen Sprungzeitpunkt als Weckzeit, und ist in der Wirkung so, als folgte gleich darauf noch eine freeze-Operation. D.h. die virtuelle Uhr befindet sich nach einem D-Übergang im F-Modus und muß durch eine go- oder eine jump-Operation **explizit** fortgesetzt werden.

h) read angewandt nach jump (vor oder nach ack) liefert denselben Zeitwert wie ack; Lese- und Weckzeit sind für eine reine Simulationsuhr, die nur die Operationen read, ack und jump kennt, identisch.

Die Form der Beschreibung legt eine algebraische Spezifikation /KLA 83/ nahe; eine solche ist in Abb. 4.6 dargestellt.

Der **Vorteil** einer solchen Beschreibung ist, daß sie die **funktionellen** Eigenschaften der Erweiterung wiedergibt, ohne ihre Implementierung vorwegzunehmen. Ein **Nachteil** ist, daß sich nicht alle Anforderungen an eine Implementierung **vollständig** beschreiben lassen, insbesondere die Zusammenhänge zwischen den **Ergebnissen** der Operationen (ack, read) und den Zeitpunkten ihres Aufrufes. Es läßt sich z.B. nicht formulieren

- daß die Virtuelle Uhr im R-Modus tatsächlich "echtzeit-synchron" ist, d.h. daß die von clock__read zurückgelieferten Werte der real verstrichenen Zeit entsprechen.
- daß die Weckzeit-Transparenz des F-Modus nicht nur für die Zeitwerte der clock__ack-Operation, sondern auch für ihre Aktivierungszeitpunkte (Weckersignale) gelten soll, vgl. Abb. 4.5. D.h. alle Δ im R-Modus real verbrachte Zeiteinheiten - i.a. unterbrochen von F-Modi - wird ein Wecker-Unterbrechungssignal für das reale Testobjekt generiert.

Diese Echtzeit-Dynamik wird berücksichtigt, indem man die virtuelle Uhr einschließlich ihrer Hardware-Umgebung (Echtzeituhr, Unterbrechungswerk des Experimentrechners) spezifiziert z.B. als discrete-event-System.

Auf ihre **Implementierung** wird hier nicht detailliert eingegangen, man vgl. die Diplomarbeit /STU 88/. Dagegen sollen i.f. noch die Hardware-**Voraussetzungen** genannt werden, die für eine korrekte Implementierung bzw. Portierung mit vertretbarem Aufwand erforderlich sind:

(1) Es muß ein **programmierbarer Intervall-Timer** vorhanden sein, d.h. die physikalischen Zeitabstände der Weckersignale sind dynamisch veränderbar (man vgl. hierzu /FAE 79/ oder HW-Unterlagen bekannter Hersteller, z.B. INTEL 8253 /INT 87/ oder KWV 11A /DEC 80/). Dies ist notwendig, um ein gutes Auflösungsvermögen für Zeiteinträge der simulierten Umgebung zu erzielen, welche im R-Modus ablaufen (Richtwert: Δ/100), und um die Weckzeittransparenz des F-Modus hinreichend gut zu approximieren.
Eine Netzuhr ('line time clock'), die Unterbrechungssignale zu festen, durch die Phasendurchgänge der externen Netzspannung gegeben Zeitpunkten erzeugt, reicht **nicht** aus. Je nach (Anfangszeit, Dauer)-Verteilung der F-Modi können hier große relative Fehler der Weckzeit auftreten. So bleiben z.B. F-Zeitmodi, die zwischen benachbarten Phasendurchgängen liegen, gänzlich unerkannt.

**Abb.** 4.6: Algebraische Spezifikation der virtuellen Uhr
**Datentypen**: U (Virtuelle Uhr)
T (Zeitbereich)

**Operationen:**

| | | |
|---|---|---|
| init | :T×T | -> U |
| freeze | :U | -> U |
| go | :U×T | -> U |
| jump | :U×T×T | -> U |
| ack | :U | -> U×T |
| read | :U | -> U×T |

**Hilfsoperationen:**

| | | |
|---|---|---|
| tw | : U | -> T |
| next_tr | : U×T | -> T |
| next_tr* | : U×T | -> T |
| last_ack | : U | -> T |

**Notation:** für x = (u,t) ∈ (U×T) sei
x.s: = u (state-Komponente)
x.t: = t (time-Komponente)

**Gleichungen:**

| | | |
|---|---|---|
| tw(go(freeze(u),T)) | = T | Kleinster Zeiteintrag der |
| tw(go(ack(jump(u,T1,T2)).s,T)) | = T | simulierten Umgebung |
| tw(freeze(u)) | = tw(u) | |
| tw(ack(u).s) | = tw(u) | |
| tw(jump(u,T1,T2)) | = T2 | |

| | | |
|---|---|---|
| next_tr{*}(freeze(u),T) | = next_tr{*}(u, T) | Frühester Zeitpunkt {>T} |
| next_tr{*}(go(u,tw),T) | = next_tr{*}(u, T) | zu dem Zeiteintrag T |
| next_tr{*}(ack(u).s,T) | = next_tr{*}(u, T) | des realen Testobjektes |
| next_tr{*}(jump(u,T1,T2),T) | = next_tr{*}(u, T) | ablaufen kann |
| next_tr (init(T1,Δ),T) | = min{t ≥ T\|(t-T1) mod Δ = 0} | |
| next_tr*(init(T1,Δ),T) | = min{t > T\|(t-T1) mod Δ = 0} | |

| | | |
|---|---|---|
| last_ack(init(T,Δ)) | = T | Zeitwert der letzten |
| last_ack(ack(u).s) | = ack(u).t | ack-Operation |
| last_ack(jump(u,T1,T2) | = min(next_tr(u,T1)-Δ,T2) | |
| last_ack(freeze(u)) | = last_ack(u) | |
| last_ack(go(u,T)) | = last_ack(u) | |

(a)
read(freeze(u)).s = freeze(u)
( = > read(read(freeze(u)).s).t = read(freeze(u)).t)
(b)
ack(freeze(u)).s = freeze(u)
ack(freeze(ack(u).s)).t = ack(u).t
(c)
read(go(freeze(u),tw)).t = read(u).t
(d)
ack(u).t = min (next_tr*(u,last_ack(u)), max(last_ack(u),tw(u)))
(e)
go(go(u,T),T') = go(u,T')
freeze(freeze(u)) = freeze(u)
(f)
freeze(jump(u,T1,T2)) = jump(u,T1,T2)
go (jump(u,T1,T2),tw) = jump(u,T1,T2)
jump (jump(u,T1,T2),T3,T4) = jump(u,T1,T2)
(g)
ack(jump(u,tr,tw)).s = freeze(ack(jump(u,tr,tw)).s)
(h)
read(jump(u,tr,tw)).t = min (next_tr(u,tr),tw)
read(ack(jump(u,tr,tw)).s).t = min (next_tr(u,tr),tw)

**Abgeleitete Gleichungen** (Forts. von Abb. 4.6)

ack(jump(u,tr,tw)).t = min(next_tr(u,tr),tw)

ack(ack(u).s).t = min(next_tr*(u,ack(u).t),max(ack(u).t,tw(u))

ack(go(init(T,Δ),tw)).t = min(T+Δ, max(T,tw))

ack(go(freeze(ack(u).s),tw)).t = min(next_tr*(u,ack(u).t), max(ack(u).t,tw))

Falls tw(u),tw nicht im Intervall [ack(u).t, next_tr*(u,ack(u).t]:

ack(go(freeze(ack(u).s),tw)).t = ack(ack(u).s).t = ack(u).t+Δ

Gleichungen und Ungleichungen, die von einer Echtzeituhr erwartet werden:

| | | |
|---|---|---|
| read(read(u).s).t | > read(u).t | (Monotonie der read-Ftn) |
| ack(ack(u).s).t | = ack(u).t + Δ | (Periodische Inkremente) |

---

(2) Es sollte zusätzlich ein **Absolutzeitgeber** existieren. Falls das Zählregister des Intervall-Timers ein störungsfreies Auslesen im laufenden Betrieb gestattet, kann die Absolutzeit per SW geführt werden; besser ist eine separate Kalenderuhr ('wall clock') mit gutem Auflösungsvermögen (Δ/100 bis Δ/1000).

(3) Der **Unterbrechungseingang** des Intervall-Timers sollte **maskierbar** sein, ohne daß bereits anstehende Unterbrechungssignale dadurch verlorengehen. Ist dies nicht gewährleistet, so sollte der Unterbrechungsvektor der UISR zusätzlich durch einen SW-Interrupt ansprechbar sein.

(4) Nicht notwendig ist das Anhalten des Zählregisters des Intervall-Timers oder der Kalenderuhr; dadurch vereinfacht sich allerdings die Realisierung. Eine solche Möglichkeit besteht bei KWV 11A oder bei PIT-8253, indem der GATE-Eingang des Bausteins rechnerseitig als digitales Ausgabesignal angesprochen wird.

Wenn die Voraussetzungen (1)-(3) erfüllt sind, lassen sich freeze und go im Prinzip z.B. so realisieren (Abb. 4.5):

**clock-freeze**:
- Kalenderuhr lesen (der gelesene Wert ist von nun an gültige Uhrzeit für read)
- Restintervall des Intervall-Timers bestimmen
- Unterbrechungseingang des Intervall-Timers maskieren (der Timer selbst kann entweder weiterlaufen oder abgeschaltet werden)

**clock-go**:
- Kalenderuhr auf die gültige Uhrzeit der letzten freeze-Operation stellen
- Intervall-Timer mit dem Restintervall des letzten clock_freeze neu starten (nach dem nächsten Uhrinterrupt ist es u.U. notwendig, das Standardintervall δ im laufenden Betrieb zu restaurieren!)
- Timer-Unterbrechungseingang freigeben.

Die Virtuelle Uhr bildet zusammen mit den Prozeß- und Prozessorzustandsübergängen die unterste Schicht des hybriden BS-Kerns.

### 4.2.3 Simulierte Zielprozesse (S-Prozesse)

Ein S-Prozeß ist ein Platzhalter für einen R-Prozeß, der dieselben Dienste benutzt, dieselben Zugriffsrechte besitzt und denselben Zuteilungsregeln unterliegt wie ein R-Prozeß. Der Hauptunterschied zwischen einem R- und einem S-Prozeß liegt in der Bewertung seines Verbrauchs an CPU-Zeit als Primär-Betriebsmittel: nicht der reale (implizite) Verbrauch ist wesentlich für die Experimentergebnisse, sondern der explizit zu simulierende. Konkret bedeutet das:

- Die tatsächliche Dauer der CPU-Belegung eines S-Prozesses ist **transparent**. Während ein S-Prozeß rechnet, wird (im Gegensatz zum R-Prozeß) der F-Modus eingeschaltet.
- Zur Kompensation simuliert der S-Prozeß durch WORK (d) einen Verbrauch von d CPU-Zeiteinheiten der Experimentmaschine.

Semantik von WORK (d) (d sei bereits konvertiert in eine Zeitdauer):
Mit den Aktionen
EW0 : Aufruf des WORK-Systemdienstes
EW1 : Beendigung des WORK-Systemdienstes
gilt

(A) Der S-Prozeß ist zwischen EW0 und EW1 ständig rechenbereit auf dem Experimentrechner, wobei beliebig viele Übergänge

ER $_i$- von 'rechnend' nach 'bereit' (Prozessorentzug)

ER+$_i$ - von 'bereit' nach 'rechnend' (Neuzuteilung)

möglich sind ($i = 1,\ldots,k$ und $k \geq 0$).

(B)

$$\sum_{i=0}^{k} (t_v(ERi^-_{i+1}) - t_v(ER^+_i)) = d$$

$$(ERi^-_{k+1} := EW1,\ ER^+_0 := EWO)$$

Wegen der Monotonie der virtuellen

Die WORK-Rechenphase erstreckt sich also über mindestens d virtuelle Zeiteinheiten, und exakt d Zeiteinheiten, falls der S-Prozeß ununterbrochen rechnet.

Das Konzept zur **Realisierung** von WORK-Rechenphasen ist denkbar einfach: ein spezieller Zeiteintrag (WORK-Eintrag) repräsentiert die zu simulierende Restrechenzeit und wird durch folgende Operationen manipuliert:

(a) Setzen des WORK-Zeiteintrags mit (clock-read + Restrechenzeit) als Weckzeit:
bei Erst- oder Neuzuteilung eines S-Prozesses innerhalb von WORK ($ER^+_i$)

(b) Ausfügen des WORK-Eintrags und Dekrementieren der Restrechenzeit um die virtuell abgelaufene Zeit:
bei Prozessorentzug des S-Prozesses oder Ablauf des WORK-Eintrags ($ER^-_i$).

Dieser Mechanismus ist entkoppelt von den konkreten Bedingungen und Strategien, die zur Zuteilung (a) bzw. zum Prozessorentzug (b) führen; es müssen nur **dieselben** Zuteilungs- und Präemptionsstrategien wie für **R-Prozesse** verwendet werden. Die spezifizierte CPU-Zeit d wird nicht real (in Echtzeit) verbraucht. Stattdessen wird, wannimmer der S-Prozeß rechnend wird, in Wirklichkeit der Leerprozeß LD aus 4.2.2.2 zugeteilt, der einen D-Übergang veranlaßt.
Die Inkonsistenz, daß jetzt sowohl LD als auch der S-Prozeß rechnend sind, wird durch eine Erweiterung des Prozeßzustandsmodells für S-Prozesse um einen Zustand si (simulierend, Unterzustand des Zustands 'rechnend') beseitigt.

**Bemerkungen**

- Das Ablaufmodell für S-Prozesse läßt sich verfeinern, so daß simulierte Rechenphasen auch mit **realen** (Echtzeit-)Rechenphasen abwechseln können. Um dem BS-Kern den Wechsel zwischen echtzeitbewertetem und transparentem CPU-Verbrauch innerhalb desselben S-Prozesses mitzuteilen, müssen weitere Systemdienste bereitgestellt werden.
- Das Ablaufmodell läßt sich auch auf ASP und E/A-Geräte als weitere physikalische Betriebsmittel übertragen (transparente/simulierte/reale Inanspruchnahme von ASP oder E/A-Transfers).

### 4.2.4 Konkurrenz um passive Betriebsmittel (Arbeitsspeicher)

#### 4.2.4.1 Problemstellung

Unter passiven Betriebsmitteln werden alle Arten von Arbeitsspeicher verstanden, die dynamisch belegt und freigegeben werden, z.B. Datenträger für Prozeßleitblöcke, Zeiteinträge, Seitentabellen, oder dynamische Datenstrukturen in Anwenderprogrammen (Halde). Wie bei jedem BM, entsteht eine Konkurrenzsituation und damit Wartezeiten, wenn die Summe der Anforderungen den verfügbaren Vorrat übersteigt. Speziell um die Wartezeiten des realen Testobjekts (R-Prozesse), die zu den Gesamtantwortzeiten beitragen, geht es i.f., weniger um die Auslastung des ASP oder den ASP-Verwaltungsaufwand.

Jede auf Messung, Simulation oder analytischen Modellen beruhende Methode zur Bestimmung dieser Größe erfordert die Kenntnis

(V1) **der Verfahren und Strategien der BM-Verwaltung** (/WET 84/, Kap. 3)
- Auswahl wartender Prozesse bei der BM-Zuteilung(Kandidatenauswahl)
- Organisation des BM-Vorrats und Auswahl freier BM (Speicherstücke),
- Wiedereingliederung freigewordener BM

(V2) **des Lastprofils der Anwendung**

z.B. Verteilungsfunktionen für
- Ankunftszeiten der BM-Anforderungen der Prozesse
- Dauer der BM-Belegungen
- Umfang (Länge) der Anforderungen ggf. weitere relevante Kenngrößen, z.B. Prozeßprioritäten.

Im zielorientierten Simulationsansatz werden im Experiment und realen Einsatz dieselben Verfahren verwendet. Ebenso wird davon ausgegangen, daß die R-und S-Prozesse in Umfang, Anzahl und Verhalten das Lastprofil ihres Zielrechners im Einsatz repräsentieren (V2). Das Problem liegt dann in den W-Prozessen, die u.U. zusätzlich zu R-Prozessen um den ASP eines Experimentrechners konkurrieren, aber nicht zur realen Last gehören (Interferenzproblematik (AR4), 3.4.3.2).

Der Experimentablauf ist nach 4.2.2.1 eine alternierende Sequenz von Rechenphasen der R-Prozesse und Aktionen der W-Prozesse, wobei die Wechsel durch die Aktionen

E0: $(FBB = \varnothing) \rightarrow (FBB \neq \varnothing)$

E1: $(FBB \neq \varnothing) \rightarrow (FBB = \varnothing)$

zustandekommen. Die Rechenphasen der R-Prozesse laufen nur dann unter identischen Bedingungen wie in realer Umgebung ab, wenn alle Aktionen der W-Prozesse zwischen E0 und E1 außer Ein-/Ausgaben an RW-Kommunikationsobjekten, die die Signalübertragung zu einem einzigen diskreten Zeitpunkt $t_v$ (E0) in der Realität modellieren, **unsichtbar** für die R-Prozesse sind. Alle relevanten Zustandsgrößen, insbesondere die Verwaltung der passiven Betriebsmittel betreffenden, müssen - aus Sicht der R-Prozesse - **unverändert** bleiben, so wie die Zustandsgrößen, die die Zeit repräsentieren, dank der virtuellen Uhr unverändert bleiben.

### 4.2.4.2 Lösungsansatz

Nun wäre es sicher unrealistisch zu erwarten, daß W-Prozesse alle Teile des ASP, die sie nach E0 belegen, vor E1 wieder freigeben, denn ein W-Prozeß hat i.a. eine virtuelle Lebensdauer $t > 0$ und belegt während dieser gewisse Betriebsmittel ständig (z.B. Kellersegment, Prozeßleitblock), vgl. Abb. 4.7. Deshalb bleibt nur folgender Ausweg:

(R2) Es muß disjunkte BM-Verwaltungsobjekte und - über die gesamte Laufzeit - disjunkte BM-Vorräte (Pools) für die R- bzw. S-Prozesse und die W-Prozesse geben. Wenn eine Anforderung aus ihrem eigenen Pool nicht sofort zu befriedigen ist, muß sie warten, auch wenn sie aus dem jeweils anderen Pool befriedigt werden könnte.

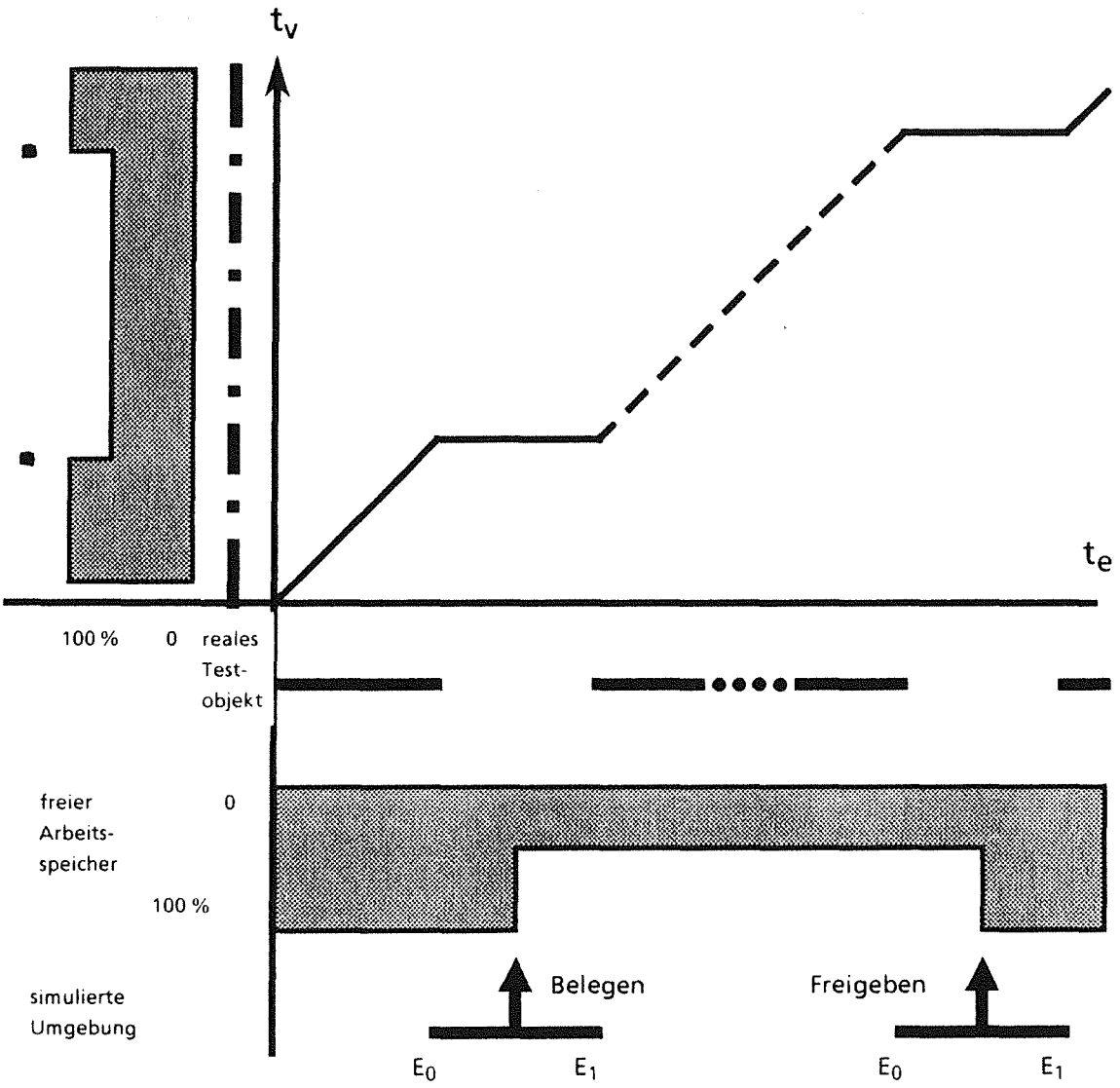



**Abb.** 4.7: Interferenz zwischen R- und W-Prozessen durch gemeinsamen Arbeitsspeicher (100 % = verfügbarer ASP für RTO in Echtzeitumgebung)

Dies soll anhand einer einfachen Architektur zur passiven BM-Verwaltung in Anlehnung an /WET 84/, Kap. 3.2-3.6, verdeutlicht werden (vgl. Abb. 4.8).

Die **unterste** Ebene der Architektur bilden Einzelobjekte zur Verwaltung zusammenhängender Speicherpools bzw. Adreßräume nach einem bestimmten Verfahren, z.B. Tabellenverfahren oder buddy-System. Die zugrundeliegenden Datentypen sind Zustandsbeschreibungen der Speicherpools und ihrer Belegungssituation (**type** pool__description).

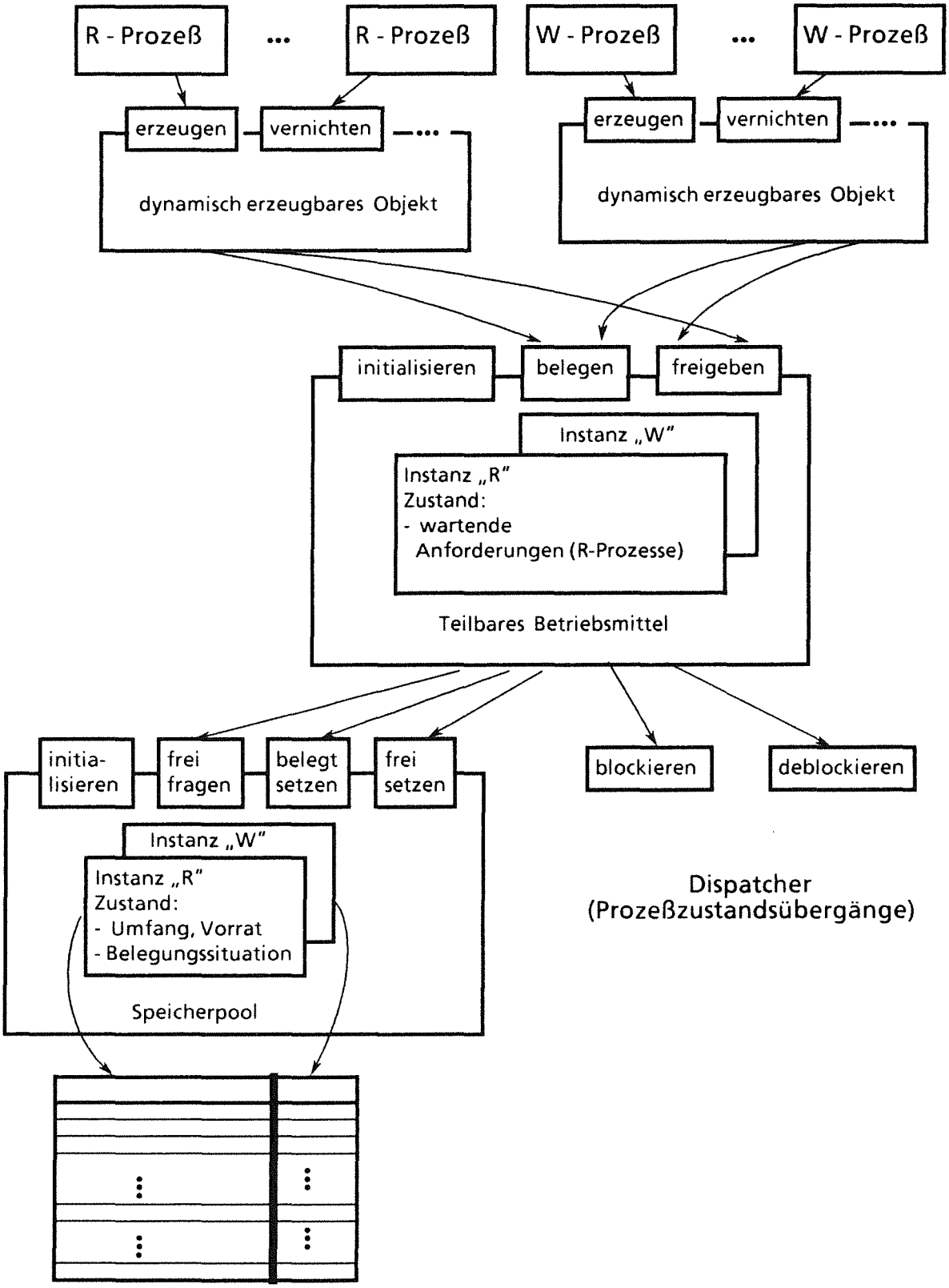



**Abb. 4.8** Verwaltung passiver Betriebsmittel für R- und W-Prozesse

(R2.1) Für jedes Verwaltungsobjekt sind mindestens zwei Instanzen ("Variablen" des Typs pool__description) vorzusehen, eine für R- und S-, und eine für W-Prozesse. Im einfachsten Fall also

**var** pool: **array** (user__class) **of** pool__description

mit

**type** user__class = (R,W);

(R2.2) Die Operationen (initialisieren, freifragen, belegtsetzen, freisetzen) enthalten die zu referenzierende Objektinstanz als Parameter (Formulierung als type manager /INT 81/).

An der Kern-Schnittstelle als **oberster** Ebene sind die Objekte sichtbar, die dynamisch erzeugt und vernichtet werden und auf Dienste der tieferliegenden Speicherverwaltung zurückgreifen. Nach 4.1.1 gilt

(R2.3) Alle Objekte an der Kern-Schnittstelle außer RW-Kommunikationsobjekten sind eindeutig der Klasse R oder W zugeordnet und dürfen nur durch Prozesse derselben Klasse erzeugt, manipuliert, und vernichtet werden (dies wird durch einen capability-Mechanismus o.ä.überprüft).

Für die BM-Verwaltungsobjekte der **mittleren** Ebenen sind die Regeln (R2.1),(R2.2) wie für die unteren Verwaltungsobjekte anzuwenden. Zusätzlich

- referenzieren ihre Operationen (belegen, freigeben) selbst tieferliegende Speicherverwaltungsobjekte
- blockieren/deblockieren sie die Betriebsmittel beanspruchenden Prozesse.

Daher folgende Ergänzungen

(R2.4) Referenziert wird immer die untergeordnete Objektinstanz derselben Klasse wie die aufrufende Objektinstanz bzw. wie der aufrufende Prozeß plauf, also z.B.

belegtsetzen (plauf.user__class,...);

Dies ist wegen (R2.3) korrekt und ausreichend, egal ob der aufrufende Prozeß BM für sich selbst oder für einen anderen, z.B. neu zu erzeugenden Prozeß anfordert (Fremdbelegung).

(R2.5) Alle Blockier-/Deblockieroperationen benennen außer dem Prozeß und der zuständigen Warteschlange auch den **Grund des Wartens**, z.B.

blockieren (plauf, bm.fwp, w__bm)

| | | |
|---|---|---|
| bm | : | BM-Verwaltungsobjekt |
| fwp | : | Folge wartender Prozesse |
| w__bm | : | Prozeßzustand (vgl. 4.2.1.1) |

Es ist die Aufgabe des Dispatchers, speziell für W-Prozesse den Grund des Wartens als modell- oder betriebsmittelbedingt zu interpretieren; die BM-Verwaltungsobjekte sind aber davon nicht betroffen und die Blockier-/Deblockieroperationen sind für R- und W-Prozesse identisch.

Für die Rechnerinitialisierung ergeben sich aus (R2.2) und (R2.3) folgende Konsequenzen:

(R2.6) Die Initialisierungsoperationen des Echtzeit-BS-Kerns für die Verwaltungsobjekte der unteren und mittleren Ebenen sind für R- und W-Instanzen zu duplizieren.

Für die dynamisch erzeugten Objekte der oberen Ebene stellt sich das Problem, wie überhaupt erstmals Objekte unterschiedlicher Klassen erzeugt werden. Normalerweise gibt es einen Ursprungsprozeß, der aus der Initialisierungsroutine des BS-Kerns an einem bestimmten Punkt des Ablaufes entsteht und von dem aus alle weiteren Objekte und Prozesse erzeugt werden.

(R2.7) Für die BS-Kern-Erweiterung sind zwei Ursprungsprozesse (ein R- und W-Prozeß) auf dieselbe Weise zu erzeugen, wie bisher ein einziger Ursprungsprozeß erzeugt wurde. Falls dies nicht praktikabel ist, muß der **eine** Ursprungsprozeß Zugriffsrechte auf **alle** Prozeßklassen erhalten.

Weil sowohl die ASP-Bereiche als auch die zugehörigen Verwaltungsstrukturen (wartende Anforderungen, Belegungssituationen) disjunkt sind, beeinflussen sich R- und W-Prozesse durch ASP-Belegung und -freigabe in keiner Weise. Wenn die einfachen Implementierungsregeln (R2.1)-(R2.7) eingehalten werden, können ferner identische Programme zur BM-Verwaltung eines Echtzeit-BS-Kerns, eines Simulations-Kerns oder eines hybriden BS-Kerns eingesetzt werden.

Die Größe der Pools für R- und W-Prozesse ist dagegen i.a. keineswegs identisch. Die Poolgröße für R-Prozesse richtet sich nach dem realen Einsatz (vgl. (V1) in 4.2.4.1). Falls damit der maximale Hauptspeicherausbau des Zielrechners bereits nahezu ausgeschöpft ist, müssen W-Prozesse auf andere Rechner ausgelagert, oder ihr ASP-Bedarf muß durch Hintergrundspeicher gedeckt werden. Generell sollten auf einem Experimentrechner, der Zielprozesse beheimatet, nur die unbedingt benötigten W-Prozesse existieren (lokale Debugger-Prozesse, Gerätetreiber für die Schnittstelle zur simulierten Umgebung).

#### 4.2.4.3 ASP-Bedarf zur RW-Kommunikation

RW-Kommunikationsobjekte sind die einzigen Objekte, die von R- und W-Prozessen gemeinsam referenziert werden. Da Nachrichtenkommunikation immer auch mit Pufferverwaltung, also ASP-Bedarf, einhergeht, ergibt sich folgende Forderung:

(R3) Die Verwaltung der RW-Kommunikationsobjekte muß sicherstellen, daß die **Interaktion** zwischen R- und W-Prozessen nicht von **Interferenzproblemen** (Verfügbarkeit von ASP) überlagert wird.

Hierbei sind mehrere Aspekte zu beachten.

**Nachrichtenaustausch über gemeinsame Adreßräume**

Um häufige Kopiervorgänge von Nachrichten zwischen Sender- und Empfängeradreßraum zu vermeiden, werden größere Datenbestände, z.B. Datenobjekte in Kommunikationsprotokollen, die zwischen Protokollschichten desselben Rechnerknotens übergeben werden, oft in einem gemeinsamen Speicher gelagert und etwa über folgende Dienste referenziert:

cm__send (mb: cm__mailbox; bt: buffer__token,..);
cm__rec (mb: cm__mailbox; **out** bt: buffer__token,..);

Kopiert werden beim Nachrichtenaustausch nur die Puffer-Identifikatoren bt, nicht die Information selbst. Leere Nachrichtenpuffer werden mit Hilfe einer ASP-Verwaltung vom Sender beschafft und vom Empfänger nach Auswertung der Nachricht wieder zurückgegeben.
Wenn Regel (R2) der disjunkten Adreßräume aufrechterhalten werden soll, folgt

(R3.1) Kommunikationsobjekte auf der Basis also eines gemeinsamen Speichers sind als RW-Kommunikationsobjekte, also klassenübergreifend, nicht zugelassen, sondern nur innerhalb jeder Prozeßklasse. Beim Senden/Empfangen an/von RW-Kommunikationsobjekten wird die Information stets kopiert.

**Bemerkungen**

- (R3.1) bedeutet keine Einschränkung der implementierbaren und bewertbaren Zielsysteme, da RW-Kommunikationsoperationen selbst nicht in Echtzeit bewertet werden, sondern nur ein Hilfsmittel zum Datenaustausch eines realen Testobjektes mit einer simulierten Umgebung darstellen.
- Trotz (R3.1) kann natürlich ein R-Prozeß eine Pufferreferenz bt auf einen Speicherbereich des R-Pools als Inhalt einer Nachricht über ein RW-Kommunikationsobjekt an einen W-Prozeß verschicken, doch hat dieser keinerlei Zugriffsrechte: er kann bt weder als Speicheradresse verwenden (wegen der disjunkten Adreßräume), noch kann er bt an den R-Pool zurückgeben (da unzulässiger Aufrufer) noch an den W-Pool (da unzulässige Pufferreferenz).

Als nächstes stellt sich die Frage nach der Verwaltung des Speichers zur **internen** Zwischenpufferung von Nachrichten im BS-Kern.

**Interne Speicherverwaltung von Kommunikationsobjekten**

(R3.2) Der Speicherbereich für RW-Kommunikationsobjekte ist ein eigener,sowohl zum R- als auch zum W-Pool disjunkter und in seiner Größe ebenfalls statisch festgelegter Bereich. Es wird sogar jedem RW-Kommunikationsobjekt ein eigener Bereich statisch bei seiner Erzeugung zugewiesen. Dies ist gleichbedeutend mit der für RW-Kommunikationsobjekte nach 4.1.3 zwingend vorgeschriebenen **endlichen Kapazität**.

(R3.3) Wenn die Kapazität eines RW-Kommunikationsobjektes erschöpft ist, aber ein sendewilliger W-Prozeß warten will, bis freie Kapazität vorhanden ist, so gelangt er in

einen **modellbedingten, nicht** in einen **bm-bedingten** Wartezustand, obwohl die Ursache der Blockierung eigentlich Speichermangel ist.

Begründung zu (R3.2), vgl. Abb. 4.9:

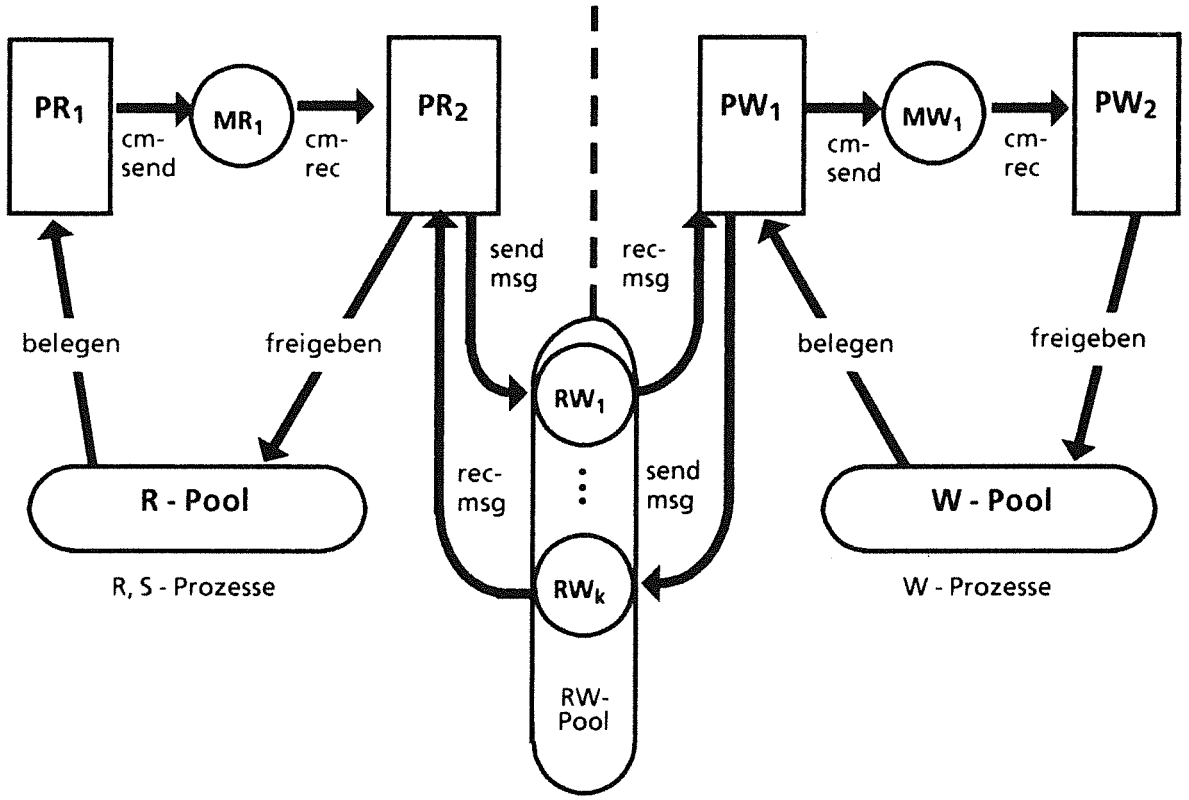



**Abb.** 4.9: Speicherverwaltung zur Prozeßkommunikation

Ein gemeinsamer Speicherbereich für alle RW-Kommunikationsobjekte, und erst recht dessen Zusammenlegung mit dem R- oder W-Pool würde die Interferenzfreiheit (AR4) verletzen. Das Synchronisationsverhalten bei der Kommunikation von R- und W-Prozessen hinge damit von der Verfügbarkeit geteilter Betriebsmittel ab, also letztlich von der **Implementierung** der simulierten Umgebung und dem ASP-Bedarf der W-Prozesse auf dem Experimentrechner. Genau das soll vermieden werden.

Zu (R3.3):

Ein betriebsmittelbedingter Wartezustand könnte zu folgender **Verklemmung** führen:

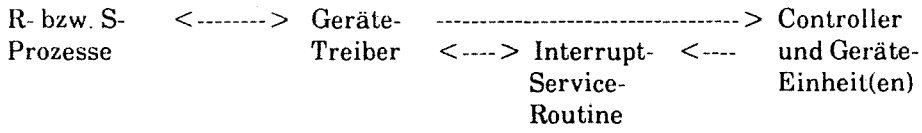
- ein sendewilliger W-Prozeß wartet bm-bedingt an einem RW-Kommunikationsobjekt
- ein empfangsbereiter R-Prozeß, der als einziger durch Nachrichtenkonsum diesen Wartezustand beenden könnte, darf wegen Regel (R1) in 4.2.2.1 nicht zugeteilt werden (FBB$\neq\varnothing$!).

### 4.2.5 Integration von E/A-Geräten

#### 4.2.5.1 Randbedingungen und Modellannahmen

Bei der Anwendung der integrierten Simulation auf Experimentrechner mit rechnerlokaler Geräteperipherie stellt sich folgendes Problem:

- Die E/A-Geräte sind durch die HW-Konfiguration des Experimentrechners vorgegeben und müssen zumindest teilweise **gemeinsam** von R-, S- und W-Prozessen benutzt werden. Die Möglichkeit, die Gerätekonfiguration räumlich zu verdoppeln und eine Hälfte jeweils exklusiv durch R- oder W-Prozesse zu nutzen wie bei ASP, besteht nicht.
- Es sollen auch E/A-Geräte mit den darauf ablaufenden E/A-"Programmen" (Aufträgen) als reale Testobjekte nach 3.3.2 einbezogen werden, soweit diese für R-Prozesse tätig sind. D.h. das gesamte Subsystem

  soll reales, echtzeitbewertetes Testobjekt sein.
- Wie bei allen realen Testobjekten wird gefordert, daß das dynamische Verhalten der E/A-Geräte - soweit es für die R-Prozesse relevant ist - nicht sichtbar von der konkurrierenden Nutzung durch W-Prozesse abhängt (Interferenzfreiheit (AR4)).
  Es wird sich allerdings zeigen, daß an diesem Idealziel Abstriche zu machen sind.

Dabei sollen folgende Randbedingungen/vereinfachenden Annahmen gelten:

(1) Es geht um **rechnerlokale**, zentral gesteuerte Peripherie (Hintergrundspeicher, Meldungs- und Ausgabegeräte, Prozeßperipherie). Dezentral von mehreren Rechnern benutzte "Geräte" wie z.B. ein Ethernet-Übertragungsmedium gehören nicht hierher; ihre Benutzung muß in jedem Fall über die Netzwerksynchronisation erfolgen (Kap.5).

(2) Zunächst werden **synchron** betriebene Geräte behandelt; das sind solche, die nur durch **Aufträge** des Rechners aktiv werden. Asynchrone Geräte (z.B. Bedienkonsole) folgen in 4.2.5.7.

(3) Bei der Kooperation Gerätetreiber<->Gerät wird der einfachste, aber wichtigste Standardfall diskutiert:

   a) Die Kommunikation besteht aus einem synchronen Anforderungs-Rückmeldungsmechanismus, realisiert durch die Funktionen send_dev, send_intr, rec_intr (vgl. Bsp. 4.1.4). Insbesondere erfolgt die Rückmeldung des Gerätes unterbrechungsgesteuert (kein polling, kein 'busy waiting'). Kompliziertere, asynchrone Interaktionen zwischen Gerätetreiber und -Controller wie z.B. Zeitüberwachung, werden nicht betrachtet.

b) Der Gerätetreiber sei durch einen eigenen Prozeß realisiert. Er führt nur Standardfunktionen wie Zusammenstellen der Geräteaufträge und Analyse der Auftragsergebnisse (Gerätestatus) durch.

c) Ein Gerätetreiber betreibt nur eine E/A-Geräteeinheit, insbesondere gibt es für einen Treiberprozeß zu jeder Zeit höchstens einen aktiven Geräteauftrag.

Auf Verallgemeinerungen wird an einigen Stellen hingewiesen.

Um das für die Leistungsbewertung wesentliche dynamische Geräteverhalten zu präzisieren, wird das Gerät grob als discrete-event-Komponente dargestellt (Abb. 4.10).

I.f. ist nur noch die reine Bearbeitungszeit tb(t,p,z) für einen Auftrag p wichtig. Dabei sind drei Fälle zu unterscheiden:

(a) tb hängt nur vom aktuellen E/A-Auftrag p ab und weder vom internen Gerätezustand z noch vom Zeitpunkt t, zu dem der Auftrag begonnen wurde (**zustandsinvariantes** Gerät)

(b) tb hängt von p und vom internen Zustand z, aber nicht vom Zeitpunkt t ab (**zeitinvariantes** Gerät)

(c) tb hängt von allen drei Parametern t,p,z echt ab (**zeitabhängiges** Gerät).

**Beispiele** für (a),(b),(c):

(1) Beispiele für (a) sind Prozeßperipheriebausteine (A/D- und D/A-Wandler): die Bearbeitungszeit hängt nur von den Auftragscharakteristika (z.B. Typ des Meßumformers, Wandlergenauigkeit), aber weder davon ab, welche Aufträge zuvor bearbeitet wurden, noch vom Zeitpunkt des Übertragungsbeginns.

(2) Bei einem Magnetbandgerät (b) mit wahlfreiem Zugriff hängt die Bearbeitungszeit außer von den Auftragsparametern (Band-, Spur-, Blocknr., Anzahl der zu übertragenden Blöcke) von der aktuellen Kopfposition bzw. im Extremfall davon ab, ob das gewünschte Band montiert ist (Zustand z).

(3) Eine Winchester-Platte ist ein Beispiel für (c) wegen der Latenzzeit des rotierenden Datenträgers. Wir geben hier ein einfaches Modell für die Bearbeitungszeit tb an.

x := (c,b,l)$\in$X Auftragsparameter (nur Lesen/Schreiben)
c: Zylinder- bzw. Spur-Adresse
b: Sektor- bzw. Blockadresse ($0 \leq b \leq B-1$)
l: Länge der zu übertragenden Information (Anzahl Sektoren bzw. Blöcke)

z : aktuelle Zylinderposition des Schreib/Lesekopfes

B : Anzahl Sektoren pro Zylinder
$\omega$ : Umdrehungsgeschwindigkeit in Sektoren/sec
p : Positioniergeschwindigkeit des Kopfes (Spuren/sec)
T0: konstanter Zeitbedarf zum Anfahren und Abbremsen bzw. Abheben und -senken des Kopfes.

Die physikalische Speicherung der Blöcke einer Spur erfolge gegenüber der Blocknummern-Reihenfolge versetzt um einen 'Interleave-Faktor' k gemäß einer Permutationsabbildung

$G := (T,X,S,Y,\delta\_\_ex,ta,\delta\_\_\varnothing,\lambda)$ mit

$X$ Menge der E/A-Aufträge (Auftragsparameter, Eingabedaten)

$S := \{'f','b','a'\} \times T \times \{X \cup \varnothing\} \times Z$

Menge von Gerätezuständen (st,t,p,z) mit

- st: Bearbeitungszustand ('f' = frei, 'b' = belegt, 'a' = Fehlbedienung)
- $t \in T$: Zeitpunkt der letzten Änderung von st
- $p \in X$: aktueller Geräteauftrag oder Leerauftrag ($\varnothing$)
- $z \in Z$: interner Gerätezustand ("Auftrags-Vorgeschichte", vgl. Beispiele (1)-(3) i.f.)

$Y$ Menge der Resultate der E/A-Aufträge (Statusmeldungen, Ausgabedaten)

$\delta\_\_ex: S \times T \times X \to S$ externe Zustandsübergangsftn.

$$\delta\_\_ex((st,t,p,z),e,p') := \begin{cases} ('b',t+e,p',z) & \text{falls } st = 'f' \\ ('a',t+e,\varnothing,z) & \text{sonst (Gerät belegt)} \end{cases}$$

$ta: S \to T$ Timeout-Funktion ('time advance function')

$$ta((st,t,p,z)) := \begin{cases} tb(t,p,z) & \text{falls } st = 'b' \\ \infty & \text{sonst} \end{cases}$$

wobei $tb: T \times X \times Z \to T$ Bearbeitungszeit des Auftrags

$\delta\_\_\varnothing: S \to S$ autonome Zustandsübergangsfunktion

$$\delta\_\_\varnothing((st,t,p,z)) := \begin{cases} ('f',t+tb(t,p,z),\varnothing,g(p,z)) & \text{falls } st = 'b' \\ (st,t,p,z) & \text{sonst} \end{cases}$$

wobei $g: X \times Z \to Z$ eine interne Gerätefunktion

$\lambda: S \times T \to Y$ Ausgabefunktion

$$\lambda((st,t,p,z),e) := \begin{cases} l(p,z) & \text{falls } st = 'b' \wedge tb(t,p,z) = e \\ \varnothing & \text{sonst} \end{cases}$$

wobei $l: X \times Z \to Y$ eine interne Gerätefunktion (liefert Gerätestatus und Resultatdaten)

**Notation** (Gerätezustand z explizit als Funktion z(H) der Auftragsvorgeschichte $H := (p_1,..p_n)$ dargestellt):

$$z(H) := \begin{cases} z_0 \text{ (Anfangszustand)}, & \text{falls leer}(H) \\ g(p_n, z(H \backslash p_n)) & \text{sonst} \end{cases}$$

**Abb.** 4.10: E/A-Gerät als DEVS-Komponente

$sp : (0,B-1) -> (0,B-1)$ mit $B := r*k-1, r \in \mathbf{N}$

$sp(s) := ks \bmod B$

$sp^{-1}(s) := s \operatorname{div} k + r(s \bmod k)$,

damit längere zusammenhängende Datenabschnitte ohne Zeitverlust (ohne eine weitere Umdrehung abzuwarten) übertragen werden können.
Dann ergibt sich

$tb(t,(c,b,l),z) := T_p + T_L + T_T$ mit

| | |
|---|---|
| $T_p := T_0 + \lvert c-z \rvert / p$ | Positionierzeit |
| $T_L := (((t+T_p)\omega - k*b) \bmod B)/\omega$ | Latenzzeit |
| $T_T := k*l/\omega$ | Transportzeit |

sowie für die interne Zustandstransformation g:

$g((c,b,l),z) := c$.

I.f. werden mehrere Methoden zur Integration solcher E/A-Geräte in eine Simulationsumgebung diskutiert, wobei die erste nur dem besseren Problemverständnis dient.

#### 4.2.5.2 Naive Integration (Methode 1)

'Naive Integration' bedeutet, daß die Gerätesteuerung des Echtzeit-BS-Kerns (Gerätetreiber, Unterbrechungsroutine) unverändert übernommen wird, und daß nur das Warten auf die Geräterückmeldung (rec_intr) für W-Prozesse als Auftraggeber zeitlich transparent ist (Zustand w_ea). Bei dieser Lösung sind jedoch sowohl die Forderung (AR4) nach Interferenzfreiheit als auch Forderung (AR3) nach realer Bewertung verletzt (vgl. Abb. 4.11).
ES bezeichne den Gerätestart, EF die Fertigmeldung.

(AR4) **Interferenz**

Die reale Bearbeitungszeit des Gerätes im Experiment

$tb = tb(t_e(ES),p,z(H_{R,W}))$ ($H_{R,W}$ Historie aller E/A-Aufträge bis p)

entspricht nicht der des realen Einsatzes ohne W-Prozesse. Die korrekte Bearbeitungszeit ist

$tbsoll := tb(t_v(ES),p,z(H_R))$ ($H_R$ Teilfolge der Aufträge für R- und S-Prozesse).

Dieser Fehler tritt bei zustands- oder zeitabhängigen Geräten (Typ (b) oder (c)) auf, nicht bei zustandsinvarianten (Typ (a)).

(AR3) **Reale Bearbeitungszeit**

Die Bearbeitungszeit des Gerätes wird vom Rechner obendrein falsch interpretiert. Statt nach dem Echtzeitbedarf

$t_e(EF)-t_e(ES) = tb(t_e(ES),p,z(H))$

wird die Bearbeitungszeit nach der rechner-internen virtuellen Uhr $t_v$ bemessen:

$t_v(EF)-t_v(ES)$, und diese ist ja nicht echtzeit-synchron.

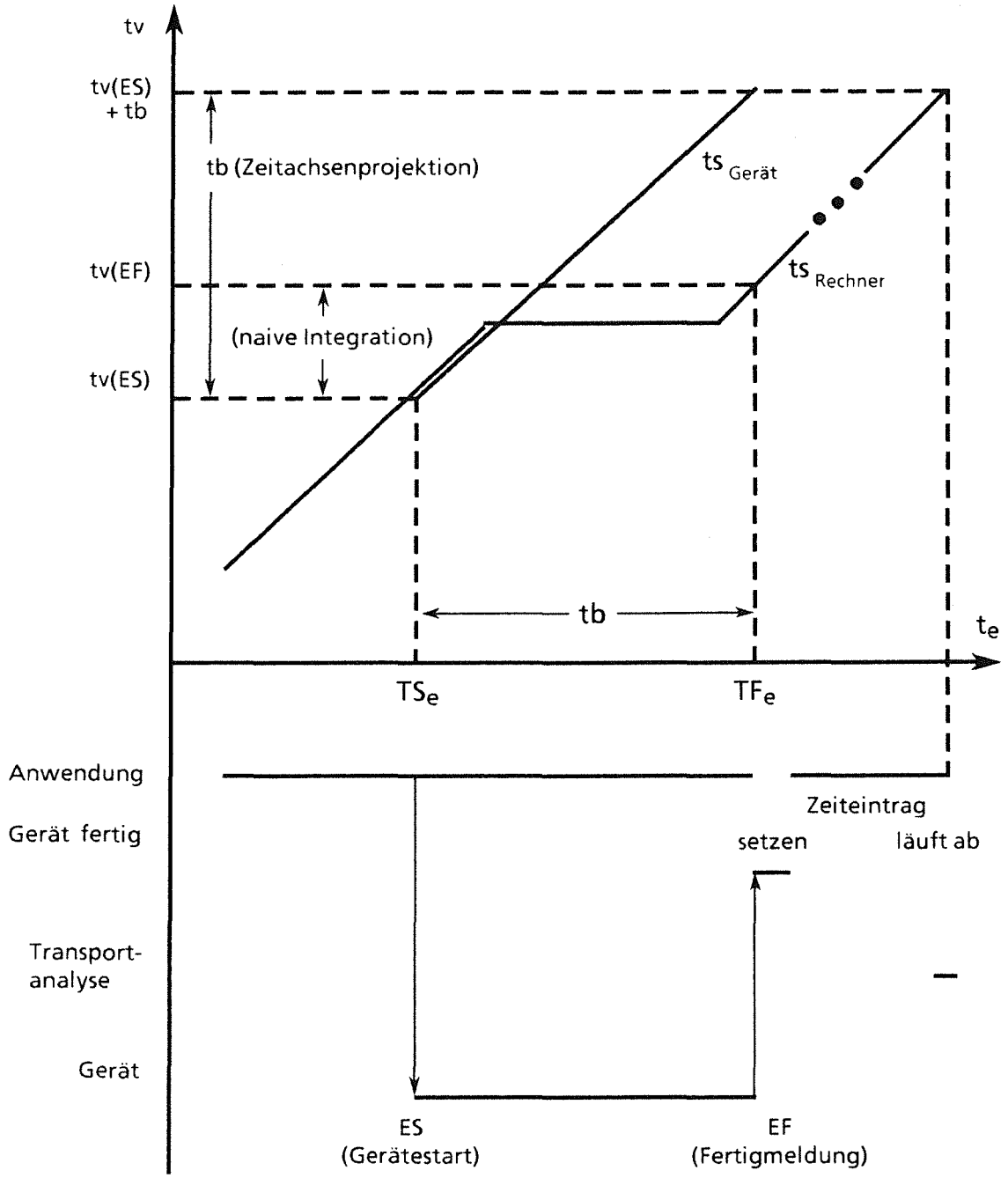



**Abb.** 4.11: Auswertung realer Gerätebearbeitungszeiten (naive Integration, Zeitachsenprojektion)

**Diskussion**

Zu (AR4)

Der **zeitabhängige** Anteil von tb dürfte in vielen Anwendungen vernachlässigbar sein. Bei der Winchester-Platte (Bsp. (3)) liefern Experiment und Einsatzsituation bei hinreichend langen Meßreihen wenigstens beide im Mittel die halbe Umdrehungszeit $T_L := B/2\omega$ als Latenzzeit. Aber selbst diese (schwache) Aussage gilt z.B. nur dann, wenn die Ankunftszeiten der physikalischen

Plattenaufträge nicht mit der Umlaufzeit des Datenträgers korreliert sind. Kompensieren lassen sich die zeitabhängigen Fehler jedenfalls nicht. Dazu müßte die tatsächliche Dauer der F- und D-Zeitmodi im Experiment auf die Umlaufzeit des Datenträgers abgestimmt werden:
$t_e(ES) \bmod (B/\omega) = t_v(ES) \bmod (B/\omega)$.
Die **zustands- (auftrags-) abhängigen** Fehler sind zwar theoretisch nach der in 3.3.2 beschriebenen Methode kompensierbar, indem nach E/A-Aufträgen für W-Prozesse der alte Gerätezustand z durch "Korrekturaufträge" restauriert wird (z.B. Zurückpositionieren des Schreib-Lese-Kopfes). Offensichtlich steht der Nutzen einer solchen Maßnahme in keinem vernünftigen Verhältnis zum Aufwand, und es läßt sich auch kein allgemeines Verfahren für beliebige E/A-Geräte angeben.

Zu (AR3)
Offenbar werden unabhängige Uhren für Rechner und E/A-Gerät benötigt: die Virtuelle Uhr zur Bewertung der Rechneraktivitäten, und eine eigene Echtzeituhr zur Vermessung der Gerätetätigkeit. Dies führt auf die nächste Methode.

#### 4.2.5.3 Zeitachsenprojektion (Methode 2)

Die Geräteaktionen (ES, EF) werden vom Rechner aus mit Hilfe einer Echtzeituhr vermessen, also nicht über den Zeitdienst clock_read der virtuellen Uhr von 4.2.2.3. Sobald eine Geräte-Fertigmeldung vorliegt, werden die ermittelten Werte $tb = TFe - TSe = t_e(EF) - t_e(ES)$ benutzt, um einen Zeiteintrag $t_v(ES) + tb$ zu setzen. Die reale Gerätebearbeitungszeit wird also auf die **virtuelle** Zeitachse projiziert. Nicht die physikalische Fertigmeldung des E/A-Gerätes, sondern der Ablauf dieses Zeiteintrags aktiviert den Treiber zur Transportanalyse (vgl. Abb. 4.12).
Im einzelnen ist zu beachten:

(1) Der Zeiteintrag darf bei der Fertigmeldung EF nicht bereits überholt sein, d.h. es darf nicht $t_v(EF) > t_v(ES) + tb$, d.h.
$t_v(EF) - t_v(ES) > t_e(EF) - t_e(ES)$ gelten.
Der Experimentrechner darf also, während Geräte aktiv sind, nicht schneller als in Echtzeit simulieren (keine D-Übergänge!). Dies kann durch fiktive, während der Gerätetätigkeit "bereite" R-Prozesse leicht erreicht werden.

(2) Falls $t_v(EF) = t_v(ES) + tb$, so kann das Setzen eines Zeiteintrags entfallen (Rechner und Gerät waren im Intervall (ES,EF) zeitlich synchron); die Unterbrechungsroutine aktiviert bei EF direkt den Gerätetreiber (send_intr).

(3) Falls $t_v(EF) < t_v(ES) + tb$, so besteht die Aktivität der Unterbrechungsroutine nur im Setzen des Zeiteintrags.

Auch die Zeitachsenprojektion besitzt einige Nachteile:

- Die Interferenzprobleme von 4.2.5.2 bestehen nach wie vor, es gilt ja immer noch $tb \neq tb$ soll !

- Methodisch korrekt ist der Ansatz nur für den Fall einer strikten Rendezvous-Kommunikation zwischen Treiber und Gerät, die keine weitere Interaktion vom Gerätestart bis zur Geräte-Rückmeldung zuläßt. Im Grunde handelt es sich um eine verteilte "Simulation", in der das E/A-Gerät irreversibel einen Auftrag im Zeitintervall $[t_V(ES), t_V(ES)+tb]$ zu Ende führt, während der Rechner mit $t_V(EF) < t_V(ES)+tb$ zeitlich zurückliegt, und seine zukünftigen, u.U. den aktuellen Auftrag betreffenden Aktionen im Zeitintervall $[t_V(EF), t_V(ES)+tb]$ noch gar nicht berechnet sind. In der Praxis kommt es durchaus vor, daß ein laufender E/A-Auftrag z.B. vom Rechner aus vorzeitig abgebrochen oder nachträglich modifiziert wird, falls die Programmier-Schnittstelle des Controllers dies erlaubt.

**Beispiel**

Über eine serielle Datenleitung werden Daten im Blockübertragungsmodus (Datenrate 9600 baud) in einen type-ahead-Puffer des Rechners (z.B. der Länge 256 byte) eingelesen. Ein hochpriorer Anwenderprozeß stelle einen Leseauftrag für 20 byte an den Leitungstreiber zu einem Zeitpunkt T, zu dem 10 byte Information im Puffer vorhanden sind (der vom DMA-Controller geführte Zähler sei rechnerseitig lesbar). Statt zu warten, bis 256 byte übertragen sind (Mindestdauer (256-10)/1200 ≃ 205 ms), könnte der Leitungstreiber die Blocklänge des laufenden Auftrags auf 20 byte vermindern, sodaß der Controller bereits nach ca. 8ms eine Fertigmeldung sendet, und so den Anwender schneller zufriedenstellen. Eine solche Interaktion ist mit der Zeitachsenprojektion nicht korrekt wiederzugeben, obwohl sie für die Leistungsbewertung nicht unwesentlich ist.

#### 4.2.5.4 Hardware-Lösung(Methode 3)

Der Vollständigkeit halber sei die Möglichkeit erwähnt, alle laufenden E/A-Aufträge während einer Aktivität der simulierten Umgebung im F-Modus physikalisch anzuhalten, oder nach der Methode von 3.3.2 die aktuellen Gerätezustände zu konservieren und bei Wiederaufnahme des R-Modus zu restaurieren. D-Übergänge des Rechners, während Geräte aktiv sind, sind wie bei Methode 2 unzulässig.

Prinzipiell realisierbar ist dies nur für rein digital aufgebaute, keinerlei mechanische Teile enthaltende Geräte-Controller (z.B. serielle Rechnerkopplungen mit DMA-Zugriff). Falls das Gerät während des F-Modus für weitere Aufträge aus der simulierten Umgebung zur Verfügung stehen muß, wird sogar ein noch komplizierterer Präemptionsmechanismus benötigt.

Die genannten Fähigkeiten erfordern einen Hardware- und Firmware-Entwicklungsaufwand, der nur in dedizierten Testbettsystemen wie z.B. /PAJ 84/ gerechtfertigt ist und der in dieser Arbeit verfolgten Zielsetzung widerspricht (vgl. Kriterien von 2.4).

#### 4.2.5.5 Ankoppelung über Vorschaltgerät (Methode 4)

Die flexibelste und allgemeinste Lösung besteht darin, die E/A-Geräte der simulierten Umgebung zuzuschlagen. Die reale Gerätefunktion wird zwar vom realen Testobjekt benutzt, aber auf die Auswertung der realen Gerätebearbeitungszeiten wird verzichtet.

(R4) Gerätebearbeitungszeiten werden, sofern sie zur Bewertung des RTO wichtig sind, grundsätzlich **simuliert**. Dazu wird für jedes Gerät G, das im Experiment von R- und W-Prozessen gemeinsam benutzt wird, eine Komponente $VG_G$ (**Vorschaltgerät** zu G) als Schnittstelle zwischen dem Gerätetreiber und dem physikalischen Gerät G eingeführt.

Die prinzipielle Funktionsweise dieser Komponente ist wie folgt (vgl. Abb. 4.12 zur HW-/SW-Konfiguration):

(1) $VG_G$ ist ein W-Prozeß

(2) Die Schnittstelle zwischen $VG_G$ und dem Treiber $T_R$ entspricht der zwischen G und $T_R$ im Einsatz, bis auf die Parameterversorgung und den Gerätestart: statt der Funktion send__dev wird eine Version send__dev__tr benutzt, die den Geräte-Kontrollblock mit Auftragsparametern versorgt, aber - im Unterschied zu send__dev - das Gerät nicht physikalisch startet (durch Setzen eines 'go-bit'), sondern stattdessen die Geräteparameter p zusätzlich an ein RW-Kommunikationsobjekt mds sendet, an dem $VG_G$ wartet.

(3) $VG_G$ bestimmt im Falle von R- oder S-Aufrufern die der Einsatzkonfiguration (nur R-Prozesse) entsprechende Bearbeitungsdauer $tb(t_v(ES),p,z(H_R))$ - vgl. 4.2.5.2 - und aktualisiert den Gerätezustand $z := g(p,z)$. Für W-Prozesse ist nichts zu tun.

(4) Nach tb virtuellen Zeiteinheiten (delay__for (tb)) startet $VG_G$ das Gerät physikalisch und wartet auf die Fertigmeldung mittels rec__intr. Als W-Prozeß befindet sich hierbei $VG_G$ im Zustand w__ea; die reale Dauer der Gerätetätigkeit ist also zeitlich transparent.

(5) Der wartende Treiberprozeß $T_R$ wird durch $VG_G$ mittels send__msg an $mdf_R$ fortgesetzt (Übergabe des Gerätestatus). $VG_G$ übernimmt für $T_R$ die Rolle der Unterbrechungsroutine. Die Reaktionszeit der eigentlichen Unterbrechungsroutine DISR in Abb. 4.12 wird bei dieser Methode nicht erfaßt, da DISR im F-Modus läuft wie $VG_G$. Falls eine pauschale Berücksichtigung ihrer Antwortzeit durch tb (Schritt (3)) zu ungenau ist, kann $VG_G$ auch einen SW-Interrupt auslösen, dessen Unterbrechungsroutine RISR im R-Modus abläuft und $T_R$ durch send__intr aktiviert.

Da Gerätetreiber selbst logische, durch Prozesse realisierte Betriebsmittel für ihre Auftraggeber sind, wird festgelegt:

(R5) Für jedes Gerät, das über ein Vorschaltgerät angeschlossen ist, werden **zwei** Treiber-Instanzen $T_R$ (für R- und S-Aufrufer) und $T_W$ (für W-Aufrufer) vorgesehen. $T_R$ ist ein R-Prozeß, $T_W$ ein W-Prozeß.

Der gegenseitige Ausschluß beim Zugriff von $T_R$ und $T_W$ auf G wird durch das Vorschaltgerät $VG_G$ gewährleistet.

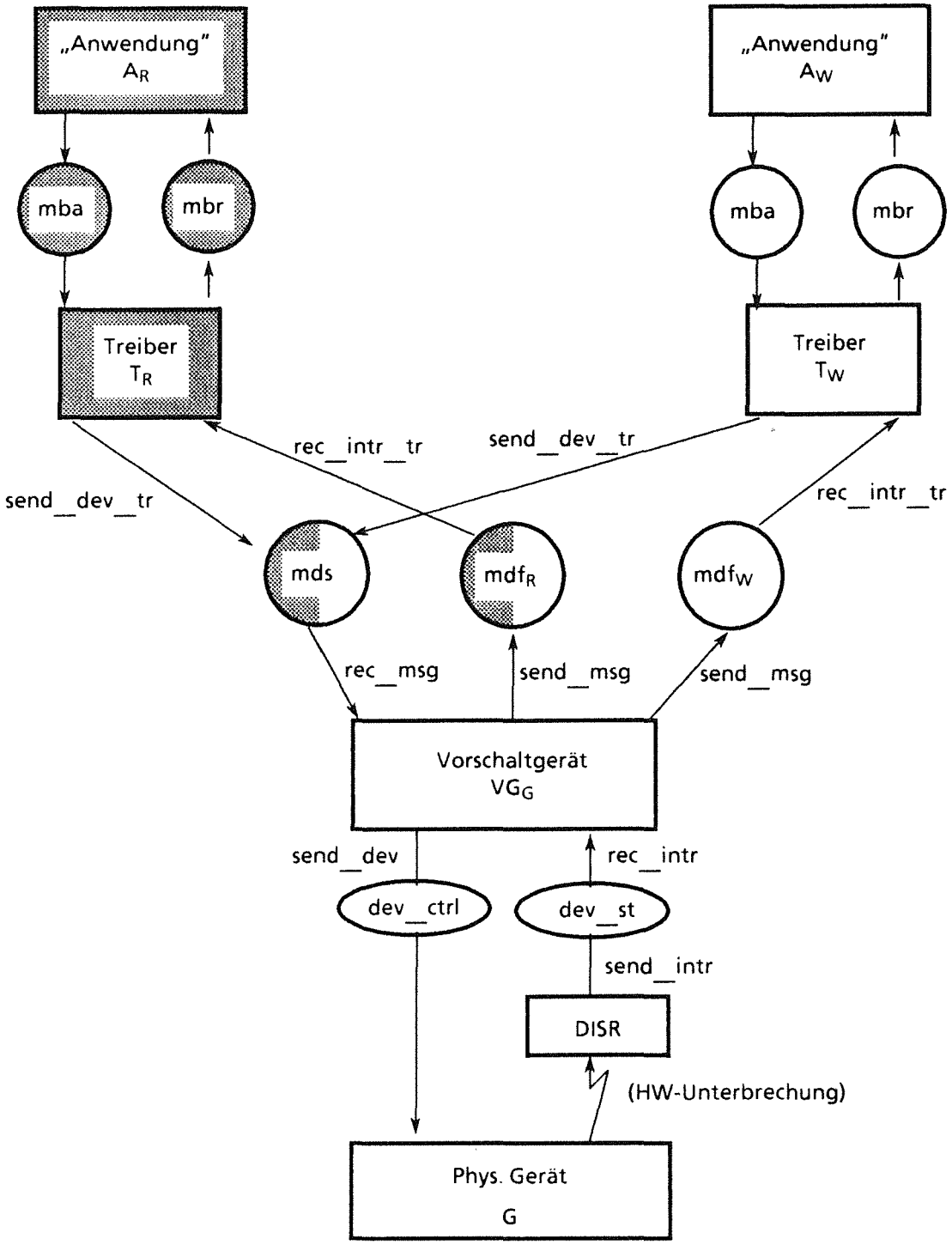



**Abb.** 4.12: Gerätetreiber Konfiguration für gemeinsamen Zugriff über Vorschaltgerät (dev_ctrl, dev_st): Kontroll- bzw. Statusregister des E/A-Gerätes)

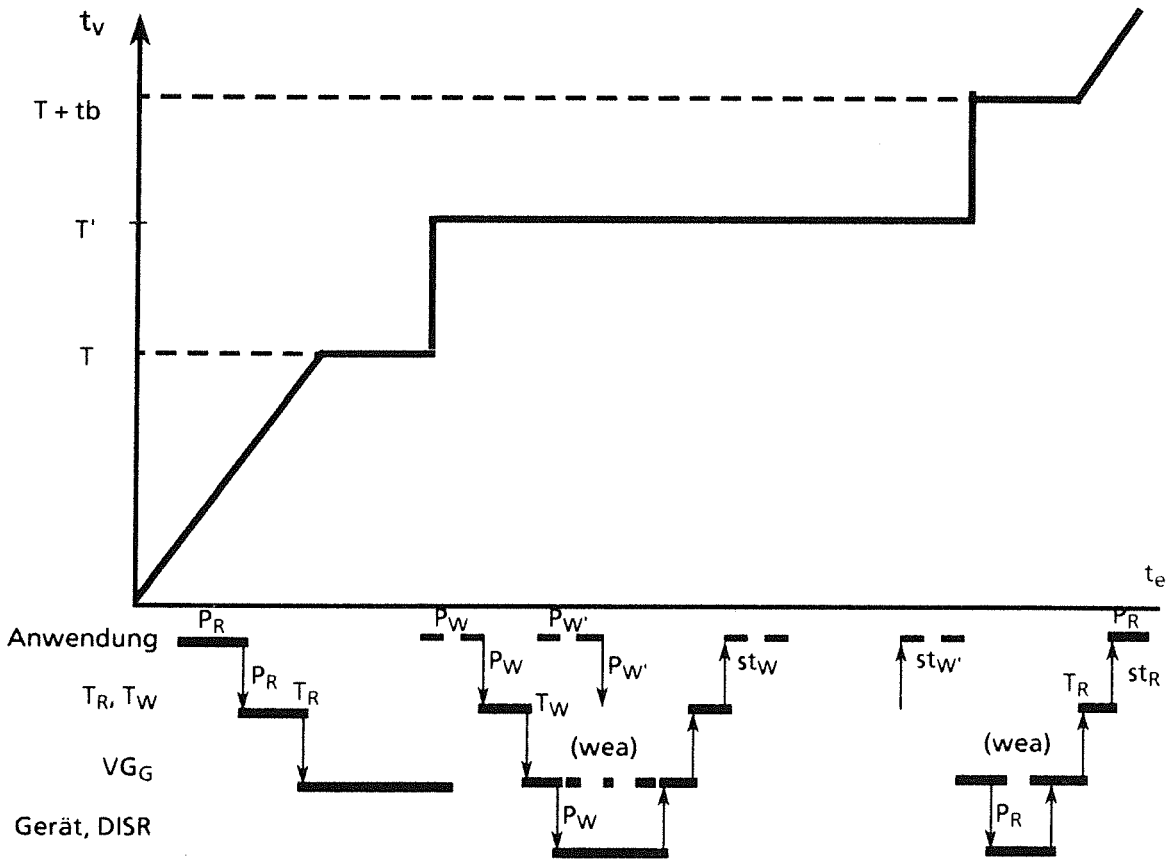



**Abb**. 4.13: Gantt-Diagramm für konkurrierenden Zugriff von R- und W-Prozessen

Abb. 4.13 zeigt ein Zeit/Ablaufdiagramm mit konkurrierenden R- und W-Aufrufern. Man sieht, daß der gegenseitige Ausschluß der Gerätebenutzer für die reale, nicht die virtuelle Zeitachse zutrifft: während aus Sicht des realen Testobjektes das Gerät im virtuellen Zeitintervall (T,T + tb) belegt ist, kann dennoch zu jedem virtuellen Zeitpunkt T', $T \leq T' \leq T + tb$, ein W-Prozeß dasselbe Gerät in Anspruch nehmen. $VG_G$ ist während der Zeitvergrößerung tb in Schritt (4) empfangsbereit für weitere Aufträge an mds (zeitlich befristete Empfangsoperation!).

Die geschilderte Methode ist leicht auf kompliziertere Kommunikationsprotokolle zwischen Treibern und E/A-Geräten übertragbar wie in dem Bsp. von 4.2.5.3, obwohl hier nur der einfachste Fall skizziert wurde.

Ein Anwenderprozeß kann die Konfiguration ($VG_G$,G) funktionell von Gerät G in Echtzeitumgebung, nicht unterscheiden. Dennoch gehört ($VG_G$, G) in seinem zeitlich-dynamischen Verhalten zur simulierten Umgebung und kann daher keine Quelle von Interferenzproblemen bilden. Die Methode demonstriert die Fähigkeit der integrierten Simulation, auf der Ebene einzelner Rechenprozesse oder E/A-Geräte flexibel zwischen Simulationsmodell oder realem Testobjekt abzuwägen, jenachdem ob die Probleme der Konstruktion und Validierung eines Simulationsmodells für das Gerät, oder die Interferenzprobleme bei seiner Integration als reales

Testobjekt schwerer wiegen. Erwähnenswert ist auch, daß keine neuen BS-Konzepte entwickelt werden mußten, sondern die zuvor eingeführten Konzepte (W-Prozesse, Zustand w__ea bei rec__intr, RW-Kommunikationsobjekte) nutzbringend angewandt werden konnten.

#### 4.2.5.6 Anwendungskriterien für die Verfahren

In der folgenden Tabelle 4.5 werden einige Kriterien der zu bewertenden Anwendung und der vorhandenen Experimentrechnerumgebung zusammengestellt und daraus Empfehlungen abgeleitet, welche der in 4.2.5.3-4.2.5.5 beschriebenen Verfahren zum Einsatz kommen.

Die Methode (3), durch HW-Entwicklung E/A-Geräte für einen Test in Nicht-Echtzeit-Umgebung tauglich zu machen, ist nur in sehr E/A-intensiven Anwendungen gerechtfertigt, in denen das Leistungsverhalten der Geräte-HW extrem wichtig ist (Bsp.: niedere Transportprotokolle in Rechnernetzen, Bilddatenerfassung). Sie ist nicht anwendbar auf Geräte mit mechanisch bewegten Teilen (Trägheitsmoment). Methode (2) setzt eine einfache, synchrone Schnittstelle zwischen Gerät und Treiber-SW voraus und empfiehlt sich dann, wenn der Aufwand zur Entwicklung von Modellen des E/A-Gerätes zu hoch ist, und andererseits Interferenzprobleme zwischen realem Testobjekt und simulierter Umgebung für die Experimentziele vernachlässigbar sind. Dies gilt z.B. bei zeit- oder zustandsunabhängigen Geräten, oder wenn ein Gerät ausschließlich vom realen Testobjekt benutzt wird.

In allen anderen Fällen sollte Methode (4) eingesetzt werden.

#### 4.2.5.7 Asynchrone Signalquellen

Bei asynchronen E/A-Geräten geht die Initiative zur Datenübertragung nicht (nur) vom Rechner, sondern von dem Gerät selbst zu unvorhersehbaren Zeitpunkten aus, z.B.:

a) Systemkonsole (Eingabetastatur)
b) interruptfähige digitale Eingabe-Schnittstelle ("Alarm")
c) Empfangs-Schnittstelle einer Rechnerkopplung.

Bei der Integration asynchroner E/A-Geräte in eine Simulationsumgebung ergeben sich folgende Probleme:

(1) Wie wird der zugehörige, empfangende Treiber-Prozeß (W-Prozeß) dynamisch an den Unterbrechungseingang eines solchen Gerätes gekoppelt?

(2) Welche Zeit wird den Signalen der Umgebung zugeordnet?

(3) Welche Auswirkungen haben asynchrone Ereignisse auf die bisherige Ablaufsteuerung des Experimentrechners (4.2.2)?

Zu (1)

Offenbar ist die Funktion rec__intr, die einen W-Prozeß in den Zustand w__ea versetzt, **nicht**

| Charakterisierung der Gerätekonfiguration bzw. Anwendungsmerkmale | | Vorschalt-gerät (4) | Zeitach-senproj. (2) | HW-Entwick-lung (3) | Empfeh-lung |
|---|---|---|---|---|---|
| enthält E/A-Gerät mechanisch bewegte Teile, bzw. ist tb zustands-/zeitunabhängig? | J | + | - | = | 4 |
| | N | - | + | o | 2 |
| Anwendung E/A-intensiv, E/A-Leistungsverhalten sehr wichtig? | J | + | - | + | 3,4 |
| | N | - | + | - | 2 |
| Entwicklung eines Bearbeitungs-zeitmodells für Controller/ Geräteeinheit nach Herstellerangaben | einfach | + | o | o | 4 |
| | komplex | o | + | o | 2 |
| Interaktion zwischen Treiber und Controller | synchron, request-reply | o | o | o | 4,2 |
| | asynchron, komplex | + | = | - | 4 |
| Betriebsumgebung, Benutzung des E/A-Gerätes | Echtzeit-Einsatz | = | + | o | 2 |
| | Benutzung nur durch R-Prozesse | o | + | o | 2,4 |
| | gemeinsam benutzt | + | - | - | 4 |

Legende:

Geräte- bzw. Anwendungsmerkmal { begünstigt (+), neutral bzgl. (o), benachteiligt (-), verbietet (=) } Anwendung einer bestimmten Methode

**Tab.** 4.5: Wegweiser für die Verfahren zur E/A-Gerätesteuerung

anwendbar: diese würde die Simulation von dem Zeitpunkt an, zu dem der Treiber empfangsbereit wird, zeitlich bis zum Eintreffen eines Signals, also potentiell unbegrenzt lang, blockieren. Nicht der Zeitpunkt der Empfangsbereitschaft des Treibers ist wesentlich, sondern die Ankunftszeit des Signals. Aufgrund der Entwurfsentscheidung (E6) in 4.1.2 sollte auf ein asynchrones Signal wie auf eine Botschaft (rec_msg) im **modellbedingten** Zustand gewartet werden, nur daß der "Sender", der der Empfangsoperation eine virtuelle Zeit zuweist, zur rechnerexternen Umgebung gehört und kein Prozeß dieses Rechners ist.

Zu (2)

Die virtuelle Zeit der rechnerexternen Signale kann prinzipiell auf drei Arten geführt werden:

(2.1) Sie ist per definitionem identisch mit der virtuellen Zeit des Rechners. Der Zeitpunkt wird also durch die Lesezeit clock__read beim Eintreffen des Signals festgelegt. Trifft das Signal "während" eines diskreten Zeitsprungs ein (nach clock__jump, aber vor clock__ack), so liefert clock__read bereits den Sprungzeitpunkt als Resultat (vgl. 4.2.2.3). Dies ist der einfachste, aber auch uninteressanteste Fall, da bei dieser Zeitführung die Ergebnisse von der zufälligen Simulationsgeschwindigkeit (virtuelle Zeiteinheiten pro Echtzeit-Einheit) des Experimentrechners abhängen.

(2.2) Die Signalquelle, z.B. ein Lastgenerator, operiert als reales Testobjekt **in Echtzeit,** unabhängig von der virtuellen Uhr des Rechners. Die gemessene Echtzeitdynamik der Ereignisse kann dann mit Hilfe der **Zeitachsenprojektion** (4.2.5.3) auf die virtuelle Zeitachse des Rechners projiziert werden.

Da keine untere Schranke für den Zeitpunkt des nächsten Ereignisses bekannt ist, aber die Monotonie der virtuellen Uhr erhalten bleiben soll, darf diese **global** der Echtzeit nicht vorauseilen. Ein D-Übergang ist also nur dann zulässig, wenn der Sprungzeitpunkt noch unterhalb der momentanen Echtzeit liegt. Andererseits darf die Virtuelle Uhr im Mittel auch nicht der Echtzeit nachhinken, weil die Anzahl der zu verarbeitenden Ereignisse sonst unbegrenzt wächst. Die externe Signalquelle muß in diesem Fall zeitweise abgeschaltet werden. Dieses Problem tritt bei synchronen, vom Rechner kontrollierten E/A-Geräten (4.2.5.3) natürlich nicht auf.

(2.3) Die Signalquelle ist Teil der simulierten Umgebung und zeitlich unabhängig von der virtuellen Uhr des Rechners; die Signale/Meldungen sind durch Zeitstempel o.ä. zeitlich selbstidentifizierend. Dies ist denkbar z.B. für Meldungen von der Systemkonsole oder Datenpakete vom Rechnernetz.Dieser Fall wird immer als Spezialfall der **Netzwerksynchronisation** behandelt (Kap. 5).

Zu (3)

Wesentlich ist, welcher Prozeß (R- oder W-Prozeß) durch das Signal unmittelbar aktiviert wird, und in welchen Zuständen sich die übrigen Prozesse befinden. Dies wird durch das **Prozessorzustandsdiagramm** geregelt, das bereits in 4.2.1 eingeführt und im nächsten Abschnitt aktualisiert wird, um die hinsichtlich Zuteilung und Zeitführung wichtigen neuen Regeln von 4.2.2-4.2.5 einzubringen und zusammenzufassen. Durch asynchrone Signalquellen ergeben sich dabei einige neue Zustandsübergänge, aber keine grundlegend neuen Zuteilungsregeln.

### 4.2.6 Prozessorzustandsdiagramm (Zusammenfassung der Ablaufsteuerung)

Die Tabelle 4.6 der Prozessorzustände und das Übergangsdiagramm 4.15 ähneln Tab. 4.4 bzw. Diagramm 4.2 von 4.2.1, mit folgenden Unterschieden bzw. Erweiterungen:

- die Prozessorzustände W und U in Abb. 4.2 werden zu einem einzigen Zustand W vereinigt (der Zustand U in 4.2 wird i.w. durch die interne Logik des W-Uhrproesses WUP realisiert, der in 4.2.1 bereits diskutiert wurde).
- der Zustand C in 4.2 (Warten auf Zeitfortschritt) wird in zwei Zustände unterteilt:
  R : R- oder S-Prozesse rechnen
  LD : der Leerprozeß LD rechnet, weil entweder keine R- oder S-Prozesse rechenbereit sind, oder der eigentlich rechnende (S-)Prozeß sich in einer Phase simulierten CPU-Verbrauchs befindet.

| Prozessor-zustand | Bedingung (Invariante) | rechnender Prozeß | Zeitmodus |
|---|---|---|---|
| W | FB[W] $\neq \varnothing$ | erst(FB[W]) | F |
| LF | FB[W] $= \varnothing$ $\wedge$ FWEA $\neq \varnothing$ | LF | F |
| F | FB[W] $= \varnothing$ $\wedge$ FWEA $= \varnothing \wedge$ FWB $\neq \varnothing$ | LF | F |
| R | FBB $= \varnothing \wedge$ FB[R] $\neq \varnothing$ $\wedge$ FS $= \varnothing$ | erst(FB[R]) | R,F |
| LD | FBB $= \varnothing$ $\wedge$ (FB[R] $= \varnothing \vee$ FS $\neq \varnothing$) | LD | D |

**Tab.** 4.6: Prozessorzustände im erweiterten BS-Kern

Übergänge des Prozessorzustands ergeben sich bei Zustandsänderungen einzelner Prozesse (hauptsächlich Prozeßblockierung und -deblockierung), wenn sich die globalen Konstellationen (Bedingungen in Tab. 4.6) dadurch ändern. Aufgrund der Entwurfsentscheidungen von 4.2.3-4.2.5 (z.B. disjunkte ASP-Verwaltung für R- und W-Prozesse, Ankoppelung von E/A-Geräten über Vorschaltgerät) bleiben nur sehr wenige Systemdienste oder externe Ereignisse, die zu Übergängen führen.

W -> LF, W -> F : Systemdienst, der den letzten W-Prozeß in einen beliebigen Wartezustand versetzt (Botschaft,Zeit, ASP oder E/A-Transfer), falls noch W-Prozesse auf E/A-Transfer (LF) oder ASP (F) warten

LF -> W :
- Fertigmeldung eines synchronen E/A-Gerätes (4.2.5.5)
- Signal eines asynchronen E/A-Gerätes (4.2.5.7) in der virtuellen Gegenwart, das einen W-Prozeß aktiviert

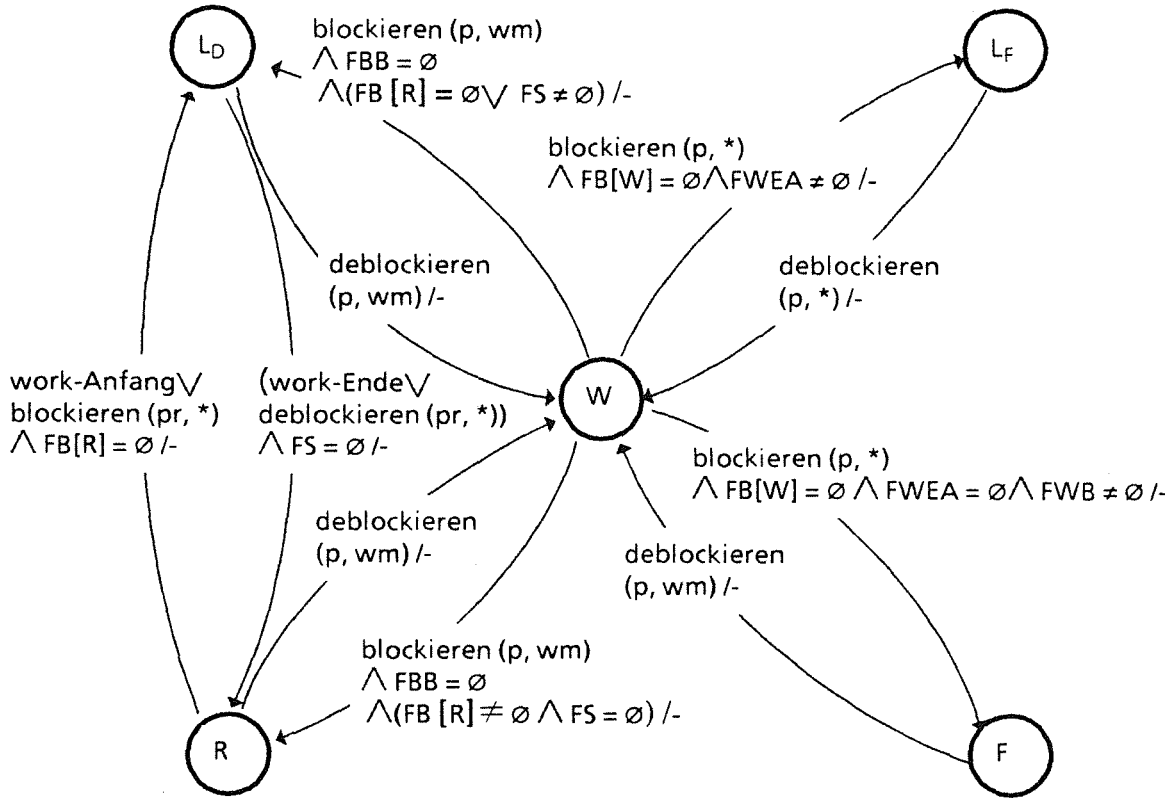



**Abb.** 4.14: Prozessorzustandsdiagramm (Bezeichnungen wie in Abb. 4.2)
FS: Menge von S-Prozessen in WORK-Phase (Zustand 'simulierend')
pr: Beliebiger R- oder S-Prozeß

F -> W nur durch Signal eines asynchronen E/A-Gerätes (4.2.5.7), das einen W-Prozeß aktiviert. Dieser kann durch Betriebsmittelfreigabe weiteren W-Prozessen das Fortfahren ermöglichen.

W -> LD, W -> R Systemdienst ,der einen W-Prozeß in **modellbedingten** Wartezustand versetzt (Botschaft, Zeit), wenn alle anderen W-Prozesse bereits modellblockiert sind. Übergang nach LD, falls Leerlaufsituation vorliegt oder falls der rechnende Prozeß ein S-Prozeß in WORK-Phase ist, nach R sonst.

R -> W
a) Ablauf eines Zeiteintragsfür W-Prozesse (deblockiert wird WUP)
b) Deblockierung eines W-Prozesses an RW-Kommunikationsobjekt
c) Signal eines asynchronen E/A-Gerätes (4.2.5.7) (**ausgeschlossen** ist: Deblockieren eines W-Prozesses, der auf ASP oder Ende eines E/A-Transfers wartet, denn während E/A-Transfers gilt stets FBB≠∅, also wird Prozessorzustand R nicht erreicht!)

LD-> W wie bei Übergang R -> W (Fälle (a)-(c))

R -> LD
a) Blockieren des letzten R- oder S-Prozesses
b) Beginn oder Fortsetzung der WORK-Phase eines S-Prozesses LD-> R

LD-> R
a) Deblockieren eines R- oder S-Prozesses
b) Ende der WORK-Phase eines S-Prozesses.

## 4.3 Architektur und Implementierung

### 4.3.1 Objekthierarchie

#### 4.3.1.1 Überblick

Nachdem im letzten Kapitel die wesentlichen neuen Verfahren zur Ablaufsteuerung des erweiterten BS-Kerns entwickelt wurden, soll der Entwurf nun architektonisch zu einem Gesamtbild gefügt werden (Blockdiagramm Abb. 4.15). Der an einer konkreten Implementierung interessierte Leser sei auf die Diplomarbeit /STU 88/ verwiesen, in der eine Teilmenge des Funktionsumfangs von 4.2 (ohne S-Prozesse und ohne die Verfahren zur E/A-Gerätesteuerung) auf einem Cadmus Mikrorechner 9200 als Erweiterung des BS-Kerns EOS /LIE 86/ implementiert wurde. Die Grobarchitektur von /STU 88/ lehnt sich an die hier beschriebene an; in Details (insbesondere Zeitverwaltung) bestehen jedoch Unterschiede. Der BS-Kern ist in vier Hauptebenen gegliedert:

Ebene 4 **Prozesse**, die Dienste des Kerns benutzen, und die in kern-interne, zum BS-Adreßraum gehörende Systemprozesse wie Leerprozesse oder Uhrprozesse, und kern-externe System- und Anwenderprozesse unterteilt werden können.

Ebene 3 **Systemkern-Schnittstelle**, über die die Dienste in Anspruch genommen werden, verbunden mit dem Übergang in den privilegierten Adressierungsmodus bzw. in den ununterbrechbaren Ablaufmodus. Der Eintritt in den Systemkern erfolgt entweder

- asynchron, durch Aktivierung einer zum Systemkern gehörenden Unterbrechungsroutine, oder
- synchron, durch Systemdienst-Aufruf.

Ebene 2 **Verwaltungsobjekte**, die den Hauptteil der Realisierung bilden und aufgrund ihrer Benutzungshierarchie in Unterebenen gegliedert werden können. Die Operationen der Objekte auf der obersten Ebene lassen sich in folgendes Schema einteilen:

- Operation zum Kreieren (bei Typ-Managern) oder Initialisieren eines Objektes (bei Einzelobjekten)
- Durch Systemdienstaufruf aktivierte Operationen (z.B. Starten, Blockieren, zeitliches Warten, Fortsetzen von Prozessen).
- Durch Ablauf einer zeitlichen Wartebedingung, also asynchron aktivierte Operationen.
- Informationsdienste zum Erfragen des Zustands der Objekte

Ebene 1 **Basis-Datenobjekte** (z.B. verkettete Listen, Bäume etc.) sowie **Hardware-Elementarfunktionen** (z.B. Operationen auf Prozeßkontext, Adreßräumen, Unterbrechungswerk).

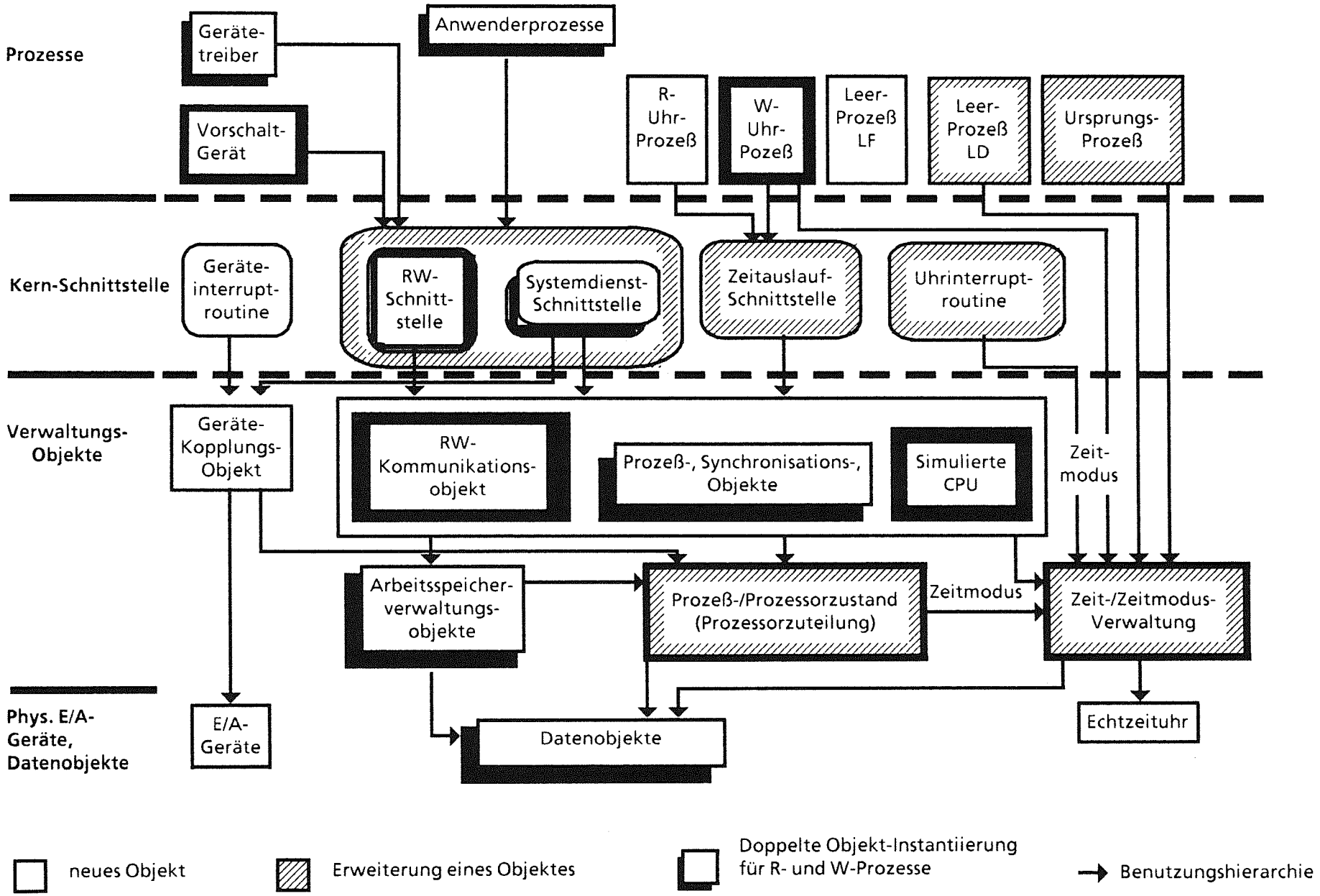



Abb. 4.15: Blockdiagramm des erweiterten Monoprozessor-BS-Kerns

Die Schraffierung der Objekte in Abb. 4.15 deutet die Art der funktionellen Erweiterungen an:

(1) Unschraffierte Objekte bleiben unverändert gegenüber einem Echtzeit-BS-Kern ohne Simulationsfunktionen, wie z.B. Gerätetreiber und Geräte-Kopplungsobjekte, welche die Basis-funktionen send_dev, rec_intr zur Verfügung stellen, die ASP-Verwaltung, die Prozeßverwaltung, -kommunikation und -synchronisation für R-Prozesse und der größte Teil der Basis-Datenobjekte und HW-Elementarfunktionen.

(2) Dunkel schraffierte Objekte und Prozesse stellen neue, also nur der Simulation bzw. Experimentführung dienende Komponenten dar. Teilweise handelt es sich dabei um **doppelte Instantiierung** von Objekten, die in reinen Echtzeit-Kernen auch existieren. Dann sind nur die Datenstrukturen doppelt vorhanden, die Verfahren dagegen nur einmal zu entwickeln. Z.B.

- Gerättreiberinstanzen $T_R$, $T_W$,
- ASP-Verwaltungsobjekte (vgl. Abb. 4.8)
- Prozeßverwaltung, -kommunikation und -synchronisation, die für R- und W-Prozesse weitgehend identisch sind
- Datenobjekte.

Teilweise handelt es sich auch um **Neuentwicklungen**, z.B.

a) W-Uhrprozeß

b) Vorschaltgeräte zur Bearbeitungszeit-Simulation von E/A-Geräten (4.2.5.5)

c) RW-Kommunikationsobjekt

d) Objekt zur Simulation des CPU-Verbrauchs für S-Prozesse (WORK).

Im Code-Umfang fallen hier i.w. nur die von der E/A-Gerätekonfiguration abhängigen Erweiterungen b), und die RW-Kommunikationsobjekte c) ins Gewicht. Natürlich kann durch Verwendung gemeinsamer Typen von Kommunikationsobjekten für alle Prozesse der Entwicklungsaufwand minimiert werden. Allerdings widerspricht dies strenggenommen der Erweiterungseigenschaft, nach der die Kommunikationsobjekte für R-Prozesse durch den Echtzeit-BS-Kern bereits vorgegeben sind, während die von W-Prozessen benutzten Kommunikationsobjekte nach dem Gesichtspunkt einer allgemeingültigen Modellierung zu gestalten sind.

(3) Die hell schraffierten Objekte existieren bereits in einem reinen Echtzeit-BS-Kern, sind aber für die integrierte Simulation neu zu implementieren und stellen z.T. auch neue Operationen zur Verfügung. Diese Objekte sind nur einmal vorhanden und werden von R- und W-Prozessen gemeinsam benutzt:

a) Systemdienst-Schnittstelle

b) Uhrinterrupt-Routine

c) Dispatcher (Prozeß-/Prozessorzustand)

d) Zeit-/Zeitmodusverwaltung.

Bei a) sind zwar die benötigten Datenstrukturen und Tabellen (z.B. zur Repräsentation der Zugriffsrechte) zu erweitern, aber keine neuen Verfahren zu entwickeln.

Die Hauptunterschiede zu einem konventionellen Echtzeit-BS-Kern liegen im Dispatcher c) und der Zeitverwaltung d). Diese Module werden i.f. ausschnittartig verfeinert.

#### 4.3.1.2 Zeit-/Zeitmodusverwaltung

Die Objekte der Zeitverwaltung sind auf 3 Ebenen angeordnet (Abb. 4.16):

**Ebene 3: Uhrinterrupt-Routine (UISR), W-Uhrprozeß**

UISR wird, wie in einem reinen Echtzeit-BS-Kern, durch das Unterbrechungssignal des Intervall-Timers aktiviert und aktualisiert die Weckzeit mit Hilfe des untergeordneten Dienstes clock_ack (Virtuelle Uhr, 4.2.2.3). Falls Zeiteinträge abgelaufen sind, werden die objektspezifischen Timeout-Operationen aktiviert. Dabei wird zwischen W-Zeiteinträgen der simulierten Umgebung und R-Zeiteinträgen des realen Testobjektes nach folgender Zuordnung unterschieden:

| | |
|---|---|
| Zeiteinträge für W-Prozesse, W-Objekte, RW-Kommunikationsobjekte: | W |
| Zeiteinträge für R- und S-Prozesse und sonstige R-Objekte außer WORK-Zeiteinträgen: | R |
| WORK-Zeiteinträge (4.2.3) | W |

Beim Ablauf von W-Zeiteinträgen aktiviert UISR in jedem Fall den W-Uhrprozeß (vgl. 4.2.1.2). Beim Ablauf von R-Zeiteinträgen wird wie im reinen Echtzeit-BS-Kern verfahren; z.B. könnten hier die objektspezifischen Timeout-Operationen auch direkt, ohne Zwischenschaltung eines R-Uhrprozesses, aktiviert werden.

**Ebene 2: Zeit-/Zeitmodusverwaltung**

Diese enthält Dienste zum Setzen und Löschen zeitlicher Wartebedingungen (R- und W-Zeiteinträge), und zur Festlegung des Zeitmodus.

**clockentry-set** ( ce : clock_entry; ( Zeiteintrag, vgl. 4.2.1.1)
**out** exc: error_code);

Der Zeiteintrag ce wird in die zugehörige Zeitliste (vgl. Ebene 1) eingetragen, falls die durch ce spezifizierte Weckzeit noch nicht abgelaufen ist.

**clockentry-cancel** ( ce : clock_entry;
**out** exc: error_code);

Der Zeiteintrag ce wird aus seiner Zeitliste entfernt.

**clockentry-net** (ce_type: clockentry_type) : absolute_time;

mit

**type** clockentry_type = (R,W)

liefert den Zeitpunkt, zu dem der nächste R- oder W-Zeiteintrag abläuft (bei leerer Zeitliste: ∞). Der Dienst

**zeitmodus** (zielmodus: time_mode)

mit

**type** time_mode = (R,F,D)

legt den gültigen Zeitmodus direkt mit Hilfe der untergeordneten Dienste freeze, go und jump der Virtuellen Uhr fest (Ebene 1, 4.2.2.3). Die Dienste clock_read, clock_init und clock_ack der Virtuellen Uhr werden exportiert, da sie von höheren Schichten benötigt werden.

**Ebene 1: Zeitlisten, Virtuelle Uhr**

Die Zeitliste ist das eigentliche Datenobjekt zur Speicherung der Zeiteinträge. Für R- und W-Zeiteinträge werden getrennte Instanzen vorgesehen: R- und W-Zeitliste. Die Implementierung des Datentyps zeitliste sollte Einfügen, Ausfügen beliebiger Einträge mit Aufwand O (n·log(n)) unterstützen, da insbesondere die W-Zeitliste eines Knotens einige hundert Zeiteinträge enthalten kann. In Frage kommen z.B. binäre Suchbäume (heap) oder AVL-Bäume. In der Tat wurden diese Verfahren zur Zeitlistenorganisation unabhängig voneinander sowohl in discrete-event-Simulatoren (z.B. /MCS 81/), als auch in Echtzeit-BSen (z.B. EOS, /LIE 86/) erfolgreich eingesetzt.

Ferner müssen, zumindest für die W-Zeitliste, verschiedene Unterkriterien für die Reihenfolge zeitgleicher Zeiteinträge vorgesehen werden (vgl. 4.2.1.2).

Neu aus Benutzersicht ist nur die Zeitmodusumschaltung der Ebene 2, die die "Geschwindigkeit" der virtuellen Zeit steuert. Sie wird von vier Objekten aus in Anspruch genommen, wie in Abb. 4.16 erkennbar:

- **Dispatcher** (Kontextwechseloperation ZUORDNEN, vgl. 4.3.1.3).
  Alle Zeitmoduswechsel zwischen R und F, die Folge von Änderungen des Prozessorzustands (Tab. 4.6) sind, fallen hierunter
- **Leerprozeß** LD (einziger Benutzer des D-Übergangs)
- **RW-Kommunikationsobjekt**: Operationen auf RW-Kommunikationsobjekten sind - wie bereits festgestellt - selbst nicht Teil des realen Testobjektes, obwohl sie von R-Prozessen aufgerufen werden. Sie werden daher ganz im F-Zeitmodus (zeitlich transparent) ausgeführt.

- **Uhrinterrupt-Routine**
  In UISR wird der R-, bzw. der F-Modus sofort eingeschaltet, sobald der Typ des abgelaufenen Zeiteintrags erkannt wurde, ohne auf eine mögliche Prozessorneuzuteilung (ZUORDNEN) zu warten.

#### 4.3.1.3 Prozeß-/Prozessorzustandsübergänge (Dispatcher)

Der Dispatcher ist in drei Funktionsebenen gegliedert (Abb. 4.17):

**Ebene 3: Kombinierte Prozeßoperationen**

Die Funktionen

| | | | |
|---|---|---|---|
| **eigen-blockieren** | (p | : process; | |
| | fp__nach | : sequence of process; | |
| | wfl | : wait__flag); | |
| **fremd-blockieren** | (p | : process; | |
| | fp__von, | | |
| | fp__nach | : sequence of process; | |
| | wfl | : wait__flag); | |
| **fortsetzen-pr** | (p | : process; | mit Neuzuteilung |
| | fp | : sequence of process; | |
| | wfl | : wait__flag); | |
| **fortsetzen-npr** | (p | : process; | ohne Neuzuteilung |
| | fp | : sequence of process; | |
| | wfl | : wait__flag); | |

mit

**type** wait__flag = (w__zeit, w__co, inaktiv, w__bm, w__ea)

blockieren/deblockieren den rechnenden oder einen fremden Prozeß und führen, falls notwendig, eine Prozessorneuzuteilung durch. In einigen Fällen, z.B. bei einem Prozessorzustandsübergang R->W, ist die Neuzuteilung zwingend vorgeschrieben. Zwei weitere parameterlose Operationen

**simulieren**;

**verdrängen**;

werden zur Implementierung der Prozessorzuteilung und -verdrängung von S-Prozessen in WORK-Phasen (4.2.3) benötigt.

**Ebene 2: Elementare Prozeßzustandsübergänge**

Diese Funktionen entsprechen syntaktisch und semantisch den Dispatcher-Funktionen vieler Betriebssysteme, z.B. /WET 84/, Kap. 2.3.4. Während

**blockieren** (p : process;
fp__von,
fp__nach : sequence of process;
wfl : wait__flag;
**out** res : result);

**deblockieren** (p : process;
fp__von : sequence of process;
wfl : wait__flag;
**out** res : result);

(mögliche Ergebnisse:
- unzulässiger Übergang
- Prozessorzustand F (beim Blockieren, vgl. Abb. 4.14)
- erzwungene Neuzuteilung
- ok)

die Mengen wartender Prozesse verändern und daher eine Auswertung des Prozessorzustands erfordern, bewirken die übrigen vier lediglich Prozessorzuteilung bzw. -entzug:

**aufgeben** (p : process); - Übergang rechnend->bereit

**zuordnen** (p : process); - Übergang bereit->rechnend
- physikalischer Kontextwechsel
- Zeitmoduswechsel

**sim-setzen** (p : process); - Übergang rechnend->simulierend
für S-Prozesse,
p behält noch den Prozessor

**sim-aufgeben** (pr : process; - Übergang rechnend->bereit
**out** ps : process); für pr
- Übergang simulierend->bereit für ps

**Ebene 1: Prozessorzustand**

Die Operation

**prozessorzustand** (p : process;
z__von,
z__nach : process__state;
**out** z : processor__state);

implementiert das Zustandsdiagramm 4.14 in 4.2.6, d.h. bestimmt aus der Klasse des Prozesses p, seinem Ausgangs- und Zielzustand und dem bisherigen Prozessorzustand den neuen. Falls der bisherige Prozessorzustand R und der aktuelle Prozeß ein R-Prozeß ist, ist die Operation leer.

**current** : process;

liefert den entsprechend dem Prozessorzustand zuzuteilenden Prozeß.

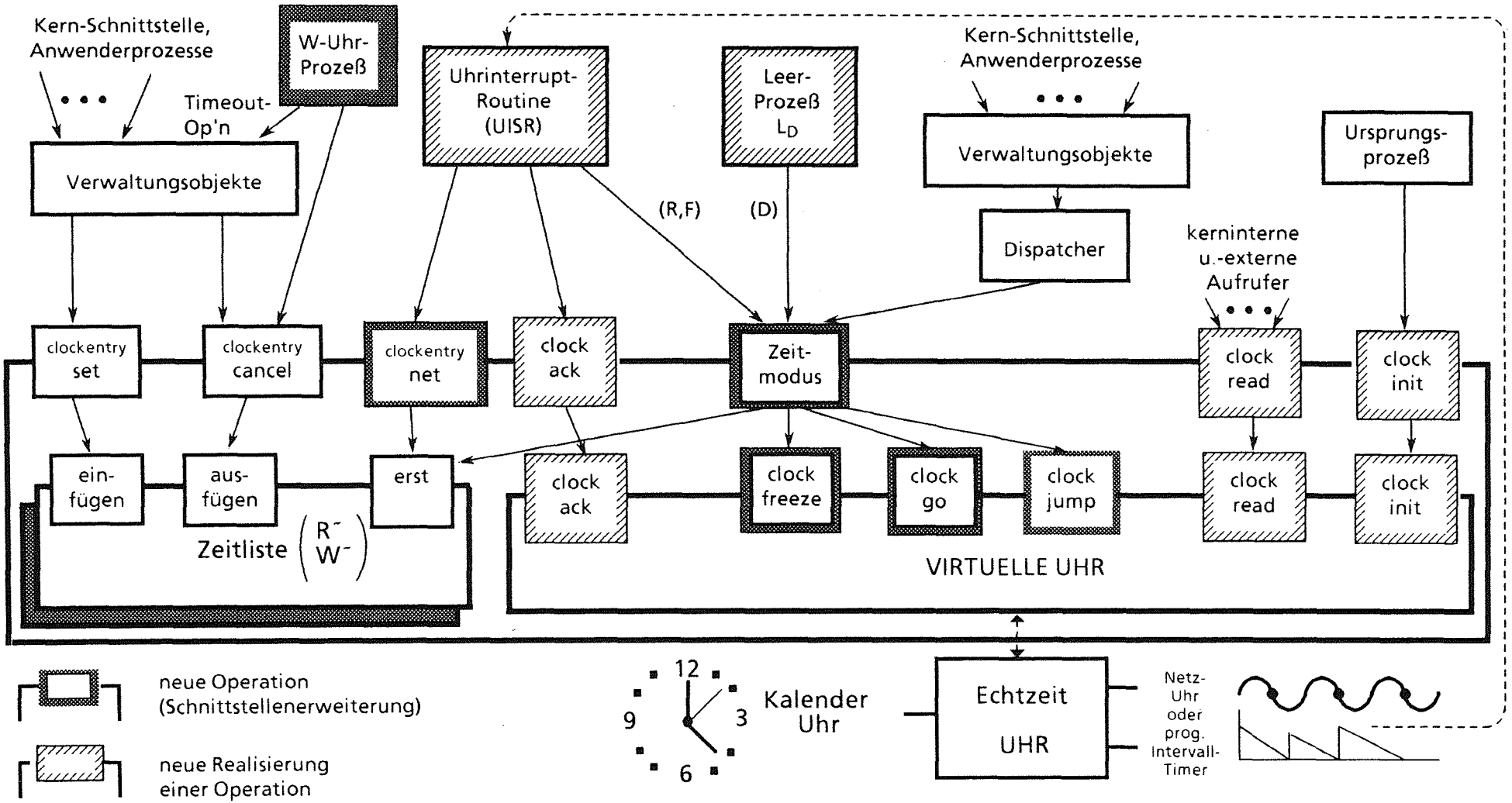



**Abb.** 4.16 Ausschnitt: Zeit- / Zeitmodusverwaltung

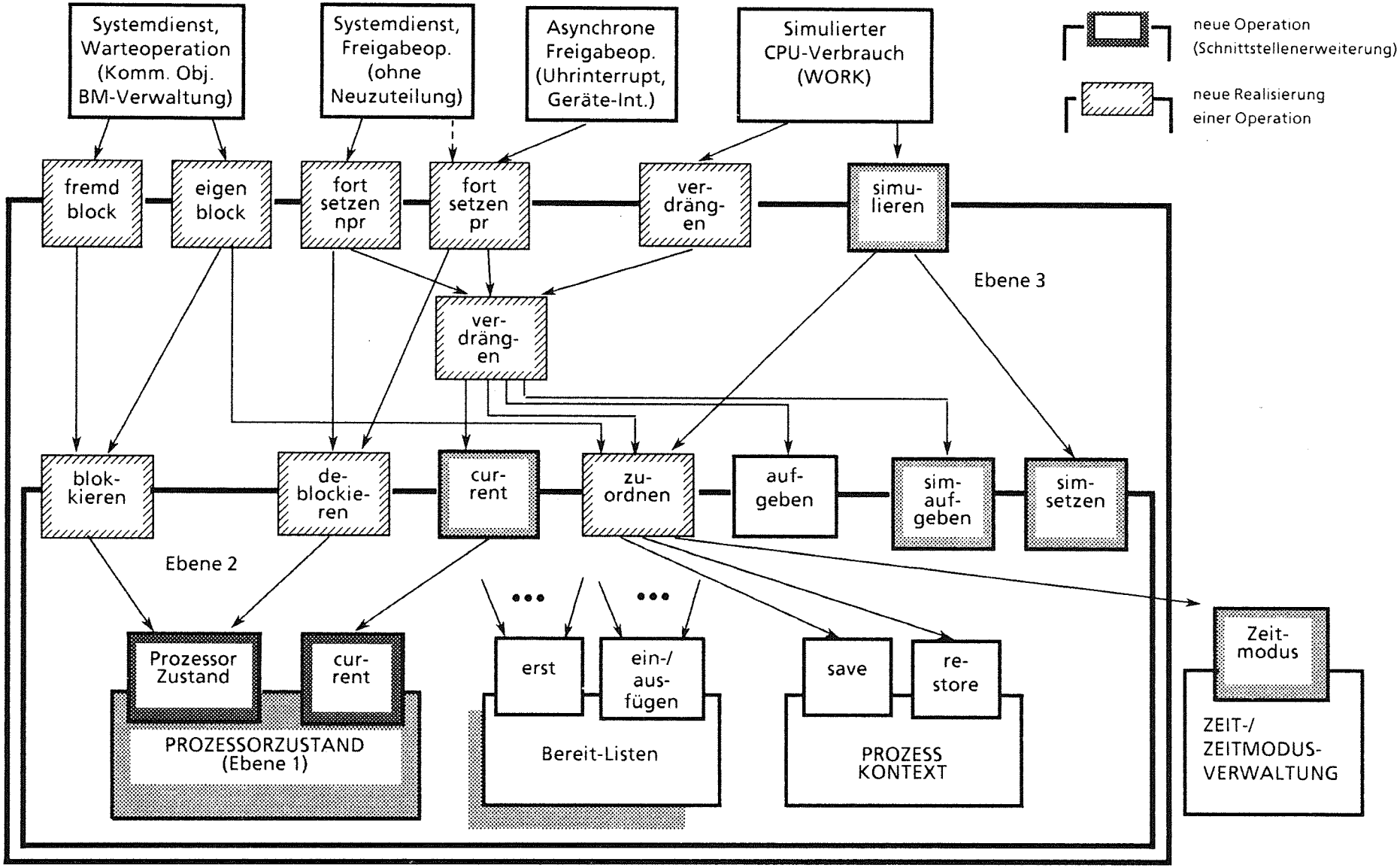



**Abb.** 4.17 Ausschnitt: Dispatcher

#### 4.3.1.4 Dualität der Objektinstanzen für R- und W-Prozesse

Beim Entwurf des BS-Kerns wurden prinzipiell disjunkte Instanzen aller dynamischen Datenstrukturen (z.B. Warteschlangen) für R- und W-Prozesse vorgesehen: disjunkte Prozeßwarteschlangen und Zustandsbeschreibungen für Prozeß- und ASP-Verwaltung, disjunkte Bereit-Listen FB[R], FB[W], disjunkte Zeitlisten zl[R], zl[W] etc. Die Hauptvorteile sollen noch einmal aufgezeigt werden:

- auf diesen Datenstrukturen operierenden Verfahren sind für R-Prozesse und oft auch für W-Prozesse dieselben wie in einem Echtzeit-BS-Kern.
- Es gelingt auf einfache Weise die Trennung der vom BS-Kern zur Laufzeit erbrachten Dienstleistungen in simulationsbezogene und zielsystembezogene. Der BS-Kern selbst ist als logisches BM anzusehen, dessen Verwaltungsaufwand (z.B. zum Durchsuchen von Listen) wie der seines jeweiligen Auftraggebers bewertet wird. Durch disjunkte Datenobjekte wird gewährleistet, daß dieser Aufwand derselbe ist wie im Echtzeiteinsatz, vorausgesetzt die Last der R-Prozesse ist dieselbe.

### 4.3.2 Konfigurationen

#### 4.3.2.1 BS-Kern im Einsatz

Es wird i.f. noch auf einige Punkte hingewiesen, die beim Übergang vom Simulationsexperiment zum Echtzeiteinsatz zu beachten sind (zunächst für den Fall eines Monoprozessors). Der Echtzeiteinsatz ist durch zwei Voraussetzungen charakterisiert:

a) Die **rechnerinterne** Anwendungs-SW besteht nur aus R-Prozessen (keine Simulationsmodelle).
b) Die gesamte über die E/A-Schnittstellen des Rechners erreichbare **rechnerexterne** Umgebung operiert in Echtzeit (insbesondere ist ein realer technischer Prozeß angeschlossen).

Um den BS-Kern zur Prozeßführung einsetzen zu können, sind nur zwei Maßnahmen notwendig

(1) Ersetzen des "zeitfortschaltenden" Leerprozesses LD durch einen normalen Leerprozeß;
(2) Ersetzen der Treiber $T_R$ aller E/A-Geräte, die über ein Vorschaltgerät angeschlossen sind (4.2.5.5), durch eine Version $T_R'$:
   - $T_R'$ verwendet anstelle von send__dev__tr die Standardfunktion send__dev, die das Gerät direkt startet
   - $T_R'$ wartet mittels rec__intr direkt auf die Gerätefertigmeldung der gerätespezifischen Unterbrechungsroutine (DISR), statt auf eine Rückmeldung von VG. Umgekehrt aktiviert die Operation send__intr in DISR direkt den R-Prozeß $T_R'$, statt den W-Prozeß VG.

Die Komponenten $T_W$, VG und die Kommunikationsobjekte mds, $mdf_R$, $mdf_W$ in Abb. 4.12 entfallen damit.

Die Umstellung (2) ist notwendig, weil das Vorschaltgerät nicht in eine Echtzeitumgebung paßt. Es hat seinen Platz dort, wo ohnehin eine simulierte Umgebung im Experiment existiert, und vergrößert diese als W-Prozeß noch. Eine wesentliche Voraussetzung zur Steuerung eines in Echtzeit operierenden technischen Prozesses ist eine echtzeit-synchrone Uhr. Sobald W-Prozesse existieren, ist die Rechner-Uhr aber nicht mehr echtzeit-synchron.

Die Umstellungen (1) und (2) sind dagegen **nicht** notwendig, wenn im Experiment alle E/A-Geräte nach Methode 4.2.5.3 (Zeitachsenprojektion) angeschlossen sind. Ein solches System paßt sich dem Echtzeitbetrieb automatisch an. Durch die Zeitachsenprojektion werden keine zusätzlichen W-Prozesse eingeführt. Falls auch sonst keine existieren (Annahme a)), wird der F-Zeitmodus niemals angefordert, d.h. die Virtuelle Uhr ist **mindestens** so schnell wie eine Echtzeituhr. Deshalb wird auch niemals eine Gerätefertigmeldung projiziert, sondern stets sofort bearbeitet (Fall (2) von 4.2.5.3). Wegen der rechnerexternen Echtzeitumgebung (4.2.5.7, Fall (2.2)) darf andererseits die Virtuelle Uhr nicht schneller als die Echtzeit sein, also ist jeder D-Übergang im Leerprozeß wirkungslos. Die Virtuelle Uhr ist also in diesem Fall automatisch echtzeit-synchron.

Da der W-Uhrprozeß und der Leerprozeß LF im Echtzeiteinsatz nie aktiviert werden, kann auf ihre Erzeugung natürlich verzichtet werden, und letztlich auch auf die Erzeugung eines ersten W-Prozesses bei der Systeminitialisierung .

#### 4.3.2.2 BS-Kern mit reduzierten Simulationsfunktionen

Nach 4.2.4 ist es in verteilten Simulationsexperimenten sinnvoll, einige der Rechner als Zielrechner für Prozeßführungs-SW und andere nur als Simulationsrechner einzusetzen. Dadurch steht dem realen Testobjekt der physikalische Speicherausbau der Zielrechner bereits im Experiment zum größten Teil zur Verfügung. Für die BS-Kerne der reinen Zielrechner ergeben sich leicht modifizierte Anforderungen:

a) Auf die Systemdienste zur Modellierung kann weitgehend verzichtet werden
b) Die Fähigkeit zur Kooperation mit einer Nicht-Echtzeit-Umgebung muß jedoch, anders als im reinen Echtzeitbetrieb, erhalten bleiben. D.h. W-Prozesse, RW-Kommunikationsobjekte, Zeiteinträge für W-Prozesse und der W-Uhrprozeß, unterschiedliche Prozessorzustände und Zeitmodi werden weiterhin benötigt (hauptsächlich von der Netzwerksynchronisation in Kap. 5).

Weil diese Funktionen dann i.w. nur noch kern-intern, aber nicht für den Modellierer relevant sind, kann eine "abgemagerte" BS-Konfiguration eingesetzt werden:

- Einsatz der Zeitachsenprojektion für rechnerlokale E/A-Geräte, damit Wegfall der Treiber $T_W$ und der Vorschaltgeräte VG

- Wegfall aller Erweiterungen der Systemdienst-Schnittstelle für Simulationsanwendungen (W-Prozesse). Alle benötigten W-Prozesse, RW-Kommunikationsobjekte etc. werden kernintern bei der Systeminitialisierung erzeugt.
  Die Systemdienst-Schnittstelle ist identisch mit der eines reinen Echtzeit-BS-Kerns.
- Wegfall der Erweiterungen für S-Prozesse
- Denkbar ist schließlich auch eine Einschränkung der Funktionalität der RW-Kommunikationsobjekte.

#### 4.3.2.3 BS-Kern als reine Simulationsmaschine

Umgekehrt kann ein Teil der Rechner ausschließlich zu Simulationszwecken dienen, also gar nicht als mögliche Zielrechner für PFS-Anwendungen in Betracht gezogen werden. Daraus ergeben sich natürlich auch Vereinfachungen des Dispatchers und der Zeitverwaltung. Die Virtuelle Uhr als reine Simulationsuhr braucht z.B. nur init, read, ack und jump zu unterstützen. Selbst wenn man Dispatcher und Virtuelle Uhr des hybriden Kerns wegen ihrer starken inneren Kohärenz unverändert beläßt, bleiben umfangreiche Einsparungen der höheren Schichten des BS-Kerns:

- Wegfall der Treiber $T_R$ und Vorschaltgeräte VG für alle E/A-Geräte
- Vereinfachte, standardisierte Systemdienst-Schnittstelle, die nur das zur allgemeingültigen Simulation unbedingt Nötige enthält (Erzeugen u. Vernichten von W-Prozessen, Zeitverzögerung und allgemeiner Kommunikationsdienst)
- Wegfall der gesamten Prozeßverwaltung und der Kommunikations- und Synchronisationsobjekte für Echtzeitanwendungen (R-Prozesse)
- Wegfall der Erweiterungen für S-Prozesse
- Wegfall des Leerprozesses LD
- Beschränkung auf einen Ursprungsprozeß (W-Prozeß).

## 5. Netzwerksynchronisation zur verteilten integrierten Simulation

Verfahren zur verteilten Simulation (VVS) sind erstmals Ende der 70er Jahre mit dem Aufkommen preisgünstiger, dezentraler Multi-Mikrorechnersysteme entstanden. Hauptziel dabei war es, die Laufzeiten bei der Simulation komplexer Systeme, die in der Realität oft aus vielen hundert parallelen Komponenten bestehen, durch Ausnutzung eben dieser modellbedingten Parallelität auch in der Simulation und Experimentauswertung zu verkürzen.
Wie in 3.4.4 festgestellt wurde, ergibt sich die Verteilung von Modellen im integrierten Simulationsansatz zwangsläufig, sobald verteilte Zielsysteme im Implementierungsstadium untersucht werden, unabhängig von Effizienzerwägungen. Andererseits passen die Zielsystemmodelle selten in das Schema der meist den Warteschlangennetzen entnommenen Demonstrationsbeispiele der VVS-Literatur. Reale Testobjekte schließlich stellen eine gänzlich neue Anwendung für VVS dar. Daher fehlten der vorliegenden Arbeit brauchbare Anhaltspunkte, welcher Ansatz am ehesten zum Ziel führen würde (vgl. die Anforderungen von 3.4.4.2). Aus diesem Grund wurden die beiden Hauptlinien der VVS - die zeitlich eng gekoppelten (Kap. 5.1) und die zeitlich lose gekoppelten Verfahren (Kap. 5.2) - **parallel** verfolgt und dann aus jeder Klasse ein neuer Algorithmus entwickelt (genannt FRED und OLGA). Diese Verfahren weisen ein stark unterschiedliches Eigenschaftsprofil auf, wie der zusammenfassende Vergleich in (5.3) zeigt, und können daher nicht als "redundant" gelten, obwohl letztlich beide denselben Zweck erfüllen.
Ein wesentlicher Gesichtspunkt der VVS ist ihre **Realisierbarkeit** durch die Netzwerksynchronisation innerhalb des Systemkerns, ihre Schnittstellendienste und ihre Kooperation mit den Knotenrechner-BSen (Zeitführung, Prozessorzuteilung). Ein Punkt wird dabei allerdings in dieser Arbeit vernachlässigt, der üblicherweise in Kommunikationsprotokollen eine zentrale Rolle spielt (um solche handelt es sich bei der NWS): die Behandlung von Ausfällen wie Knotenausfall, Ausfall einzelner Verbindungen oder Netzpartitionierung. Dies müßte vor einem praktischen Einsatz der Verfahren nachgeholt werden. Fehlertoleranz in verteilten Simulationen ist allein schon der oft hohen Kosten wegen, um einen kompletten Simulationslauf zu wiederholen, sinnvoll /REY 88/.

Zur Diskussion von VVS werden einige Begriffe und Bezeichnungen eingeführt. Ein verteiltes Simulationsmodell liege als eine endliche Menge von Modellkomponenten $K_i$, $i \in \mathbf{K}$, vor, die über zeitgestempelte Ereignisnachrichten kommunizieren und deren Kommunikationsbeziehungen durch eine Menge $\mathbf{E} \subseteq \mathbf{K} \times \mathbf{K}$ von gerichteten logischen Verbindungen (auch **link** genannt) gegeben seien ($(i,j) \in E \Leftrightarrow K_i$ sendet Nachrichten an $K_j$). Jede Komponente $K_i$ - ob reales Testobjekt oder Simulationsmodell - kann in ihrem Verhalten nach 3.4.4.1 durch ein **I/O-System** oder **discrete-event-System** beschrieben werden mit

$S_i$ - interner Zustandsbereich

$X_{ji}$ - Eingabemengen für $K_i$ (für alle j mit $(j,i)\in E$)

$X_{ik}$ - Ausgabemengen für $K_i$ (für alle k mit $(i,k)\in E$)

Alle Ein-/Ausgabetrajektorien $x_{ij}$ seien zeitdiskrete Signalverläufe, also für $i,j\in \mathbf{K}$ ist $x_{ij}\in(X_{ij},T)^e$. Die Eingaben von $K_i$ von allen Vorgängern und Ausgaben an alle Nachfolger bis Zeitpunkt $T\in\mathbf{T}$ werden gebündelt zu Zeitfunktionen $xi_i\in(XI_i,T)$, $xo_i\in(XO_i,T)$.

$(XI_i,T) := (\prod_{(j,i)\in E} X_{ji}, T)$ Menge der Eingabe-Zeitfunktionen von $K_i$

$(XO_i,T) := (\prod_{(i,k)\in E} X_{ik}, T)$ Menge der Ausgabe-Zeitfunktionen von $K_i$

$(xi_i(t))_j := x_{ji}(t)$ für $(j,i)\in E$, $t\leq T$.

Hält man die verteilte Simulation in einem beliebigen Zwischenstadium der Historien $x_{ij}\in(X_{ij},T_{ij})$ und der internen Komponentenzustände $(s_i, T_i)$ an, so bezeichnet man $T_{ij}$ als **Kopplungszeit (link time)** der logischen Verbindung (i,j), und $T_i$ als **lokale Simulationszeit** von $K_i$. $T_{ij}$ ist z.B. der maximale Zeitstempel der bisher über (i,j) übertragenen Nachrichten, und $T_i$ z.B. der dem letzten Zustandsübergang von $K_i$ nach $s_i$ entsprechende Zeitpunkt. Die Zeiten $T_{ij}$, $T_i$ können in verschiedenen Relationen zueinander stehen.

**Def. 5.1**

In einer **zentralen Simulation** (ZS) existiert eine globale Simulationszeit TIME für alle Komponentenzustände $s_i$, und für alle Zeitfunktionen gilt:

$x_{ij}\in(X_{ij},TIME)$.

In einer verteilten Simulation mit zeitlich enger Kopplung (VSE) besitzt jedes $K_i$ eine eigene lokale Simulationszeit $T_i$. $K_i$ sei ein **discrete-event-System** mit

ta: $S_i$->$\mathbf{T}$ Zeitspanne bis zum nächsten autonomen Zustandsübergang (Def. 2.3),

und es gelte

(5.1) $\max\{T_i, i\in N\} \leq \min\{T_i + ta(s_i), i\in N\}$,

$T_{ik}=T_i$ für alle Ausgabeverbindungen $(i,k)\in E$ von $K_i$.

In einer verteilten Simulation mit zeitlich loser Kopplung (VSL) besitzt jede Komponente eine eigene Simulationszeit $T_i$, wobei lediglich

$T_{ik}\leq T_i$ für die Ausgabeverbindungen $(i,k)\in E$ von $K_i$

gelte, aber die $T_i$, $T_{ij}$ ansonsten beliebig sind.

### 5.1 Zeitlich eng gekoppelte verteilte Systeme

Die **Netzwerksynchronisation** hat bei der zeitlich engen Kopplung (VSE) die Beziehung (5.1) sicherzustellen. Es werden stets die Komponenten mit dem frühesten Zustandsübergang (min

$\{T_i + ta(s_i)\}$) netzweit ermittelt. Deren Ausgabe wird von den betroffenen Komponenten $K_k,(i,k)\in E$ (i.a. parallel) verarbeitet. Dies führt zu Zustandsübergängen und neuen Werten für $T_k$ und $ta(s_k)$, aufgrund derer eine neue nächste Komponente ausgewählt wird. Solche Verfahren sind z.B. in /PEA 80/ oder (als hierarchisches Verfahren) in /CON 85/ beschrieben. Auch manche in der Literatur unter zeitlich lose gekoppelt aufgeführten Verfahren greifen auf dieses Prinzip zurück, z. B. Timelock /BEZ 87/.

Zwei Probleme ergeben sich unmittelbar:

- Falls die $K_i$ reale Testobjekte enthalten, existiert keine 'next-event-time'-Funktion ta, wie bereits in 3.4.4.2 (AN2) festgestellt. Das Verfahren muß also zunächst in geeigneter Weise verallgemeinert werden, um einsetzbar zu sein (5.1.1).
- Alle Zustandsübergänge finden, wie bei der zentralen Simulation, netzweit in strikt chronologischer Reihenfolge statt, nur geographisch verteilt. Eine Laufzeitreduktion gegenüber der zentralen Simulation ist selbst theoretisch nur dann möglich, wenn mehrere Komponenten mit exakt zeitgleichen Zustandsübergängen existieren, unter denen keine Rangfolge vorgeschrieben sein darf.

  Hinsichtlich Laufzeiteffizienz galten zeitlich eng gekoppelte Systeme daher bislang als "hoffnungsloser Fall". Dennoch gelingt es, durch eine größere Flexibilität in der Spezifikation der Modellkomponenten und ein neuartiges Ablaufmodell eine wesentliche Effizienzsteigerung bei der Leistungsvorhersage zu erzielen (5.1.2).

### 5.1.1 Netzwerksynchronisation FRED

#### 5.1.1.1 Prinzipielle Funktionsweise

Gegeben sei ein gekoppeltes System von n Modellkomponenten (Knoten), von denen jeder über einen erweiterten BS-Kern HYBRIS, und damit insbesondere über eine lokale virtuelle Uhr verfüge. Diese befindet sich zu jedem Experimentzeitpunkt in genau einem von drei Zeitmodi F, R oder D, der im Einklang mit den lokalen Anforderungen des Knotens (Prozessorzustand, Tab. 4.6) festgelegt und daher **lokaler** Zeitmodus genannt wird.

Die **Aufgabe** der NWS für die zeitlich enge Kopplung besteht nun darin, die lokalen virtuellen Uhren so zu synchronisieren, daß eine **globale** Uhr realisiert wird, deren Zeitmodus stets dem "langsamsten" Knoten entspricht. D.h. wenn

mindestens ein Knoten den F-Zeitmodus benötigt, ist der **globale** Zeitmodus **F** (alle virtuellen Uhren sind angehalten)

andernfalls, aber mindestens ein Knoten den R-Zeitmodus benötigt, sind alle Virtuellen Uhren echtzeitsynchron (globaler Zeitmodus **R**)

andernfalls, wenn alle Knoten einen Zeitsprung wünschen, werden alle Uhren zur systemweit frühesten zeitlichen Wartebedingung (Zeiteintrag) vorgestellt (globaler **D**-Übergang)

D.h. mit

$N_F$ Menge der Knoten im lokalen F-Modus
$N_R$ " " " " " R-Modus
$N_D$ " " " " " D-Modus

soll für den globalen Zeitmodus des Systems jederzeit gelten

(5.2)

Zeitmodus F $\Leftrightarrow N_F \neq \varnothing$

Zeitmodus R $\Leftrightarrow N_F = \varnothing \wedge N_R \neq \varnothing$

D-Übergang zu min $\{TN_i | i \in N_D\}$ $\Leftrightarrow N_F = \varnothing \wedge N_R = \varnothing$

($TN_i$: Sprungzeitpunkt des lokalen D-Übergangs in Knoten i)

Die Geschwindigkeit der Uhren und die Dynamik der Prozeßabläufe sind, wie in Kap. 4 auf Knotenebene, nun netzweit miteinander gekoppelt, d.h. die lokalen Simulations- oder Echtzeitanwendungen sind mit den virtuellen Uhren zu synchronisieren. Für jeden Knoten ergeben sich zwei Möglichkeiten.

- entweder der globale Zeitmodus stimmt mit seinem lokalen Zeitmodus überein (**selbstbestimmter** Zeitmodus), dann werden die lokalen R- oder W-Prozesse ausgeführt wie im Einrechnerbetrieb,
- oder der globale Zeitmodus ist "langsamer" (restriktiver) als der lokale (**erzwungener** Zeitmodus), dann ist der Knoten blockiert, d.h. der rechnende Prozeß wird durch einen Leerprozeß verdrängt. Andernfalls würden Operationen, die in der virtuellen Zukunft liegen, vorweggenommen und dadurch die netzweit chronologische Ereignis-Reihenfolge möglicherweise verletzt.

  Insbesondere muß z.B. das reale Testobjekt auf einem Knoten i angehalten (verdrängt) werden, wenn auf einem Knoten j ein Übergang vom R- in den F-Modus erforderlich wird.

Die NWS eines Knotens bietet folgende **Schnittstellendienste** an:

**zm-master** (zielmodus: time_mode);
Der eigene Knoten teilt eine Änderung des lokalen Zeitmodus mit und wünscht diesen als globalen Zeitmodus zu etablieren.

Die Operation entspricht der Operation zeitmodus der lokalen Zeit-/Zeitmodusverwaltung in der Einprozessorumgebung.

**zm-slave;**
Der eigene Knoten empfängt und verarbeitet Änderungen des lokalen Zeitmodus anderer Knoten.

Durch beide Dienste werden die Knotenmengen $N_F$, $N_R$, $N_D$ aktualisiert und der globale Zeitmodus neu festgesetzt, und in beiden Fällen kann dieser entweder konform mit dem lokalen Zeitmodus des eigenen Knotens (selbstbestimmt) oder erzwungen sein, also eine Blockierung bewirken.

Bei der Aktualisierung der Knotenmengen $N_F$, $N_R$, $N_D$ ist im Fall des D-Übergangs zu beachten, daß dieser nur **transient** ist und ein Knoten **nach** erfolgtem D-Übergang automatisch in den F-Zeitmodus übergeht (vgl. Spezifikation der Virtuellen Uhr, 4.2.2.3). Wenn also die $k \geq 1$ Knoten mit der kleinsten zeitlichen Wartebedingung einen D-Übergang ausführen, macht das verteilte System einen transienten Übergang

$(|N_F|,|N_R|,|N_D|) = (0,0,n) \text{ -> } (k,0,n-k)$

ohne daß es hierzu noch einer expliziten Zeitmodusoperation bedarf.

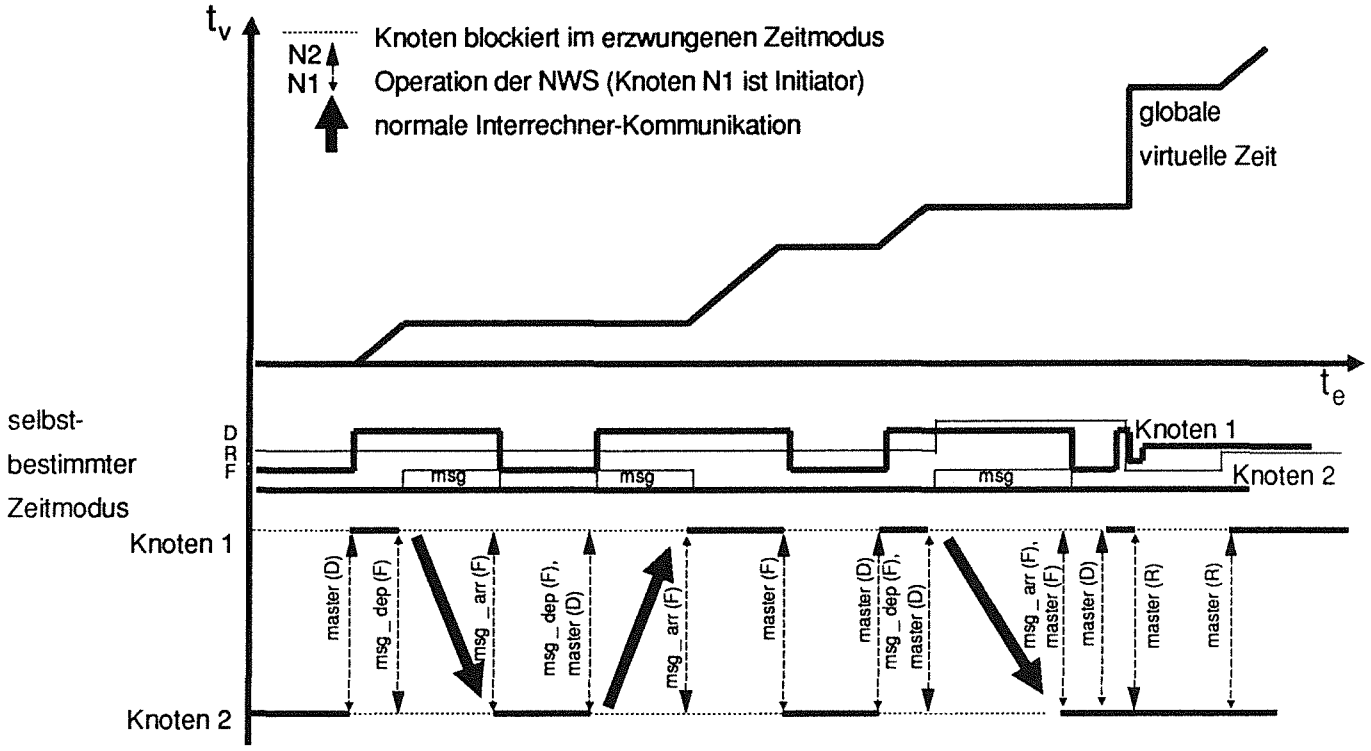



**Abb.** 5.1: Ablaufdiagramm einer zeitlich eng gekoppelten Simulation

Es genügt übrigens nicht, für den globalen Zeitmodus nur die lokalen Zeitmodi der **Rechner** heranzuziehen. I.a. sind auch die in Übertragung befindlichen **Nachrichten** im Rechnernetz zu berücksichtigen. Wenn die Übertragung z.B. zeittransparent ablaufen soll, aber der sendende und der empfangende Rechner beide untätig sind und einen D-Übergang anfordern, darf dieser dennoch nicht stattfinden, sonst wäre die zeitliche Transparenz der Nachrichtenübertragung verletzt (Gantt-Diagramm 5.1 veranschaulicht diesen Fall).

Dies ist leicht vermeidbar, indem der Sender die Nachricht(en) vor dem Absenden als zusätzliche Elemente in die Mengen $N_F$ oder $N_R$ entsprechend ihrer Anzahl und dem Zeitmodus der

Nachrichtenübertragung aufnimmt und der Empfänger sie nach dem Empfang wieder entfernt. Dies erfolgt durch zwei weitere Dienste der NWS:

**zm-msg-dep** (zielmodus: time__mode, [anz: integer]);

**zm-msg-arr** (zielmodus: time__mode, [anz: integer]);

In der Realisierung (5.1.1.2) sind diese Dienste Spezialfälle der master-Operation.

**Bemerkungen**

(1) **Spezialfall: verteilte Simulation von DEVNen**

Das Verfahren zur NWS enthält das bekannte Verfahren der zeitlich engen Kopplung verteilter diskreter Simulationsmodelle (z.B. /PEA 80/) als Spezialfall. Gegeben sei eine Realisierung von DEVS-Komponenten $K_i$ $(1 \leq i \leq n)$ durch W-Prozesse $P_i$ wie in 4.2.1.3, nun aber verteilt auf Knoten $N_i$, wobei die DEVS-Komponentenausgabe durch Interrechnerkommunikation realisiert sei. Beim Ablauf kommen nur die Zeitmodi D und F vor. Die Details, wie die korrekte netzweite Auswahl zeitgleicher Komponenten in der SELECT-Reihenfolge erfolgt, sollen hier nicht ausgeführt werden. Jedenfalls gilt

a) Wenn sich ein Knoten $N_i$ im lokalen D-Zeitmodus befindet, so gilt für den Sprungzeitpunkt $TN_i$:

   $TN_i := T_i + ta(s_i)$

   $T_i$: lokale Virtuelle Uhr von $N_i$

   $s_i$: interner Zustand der durch $N_i$ realisierten DEVS-Komponente $K_i$

b) Die zeitliche Restriktion einer zeitlich eng gekoppelten Simulation (5.1) wird immer eingehalten, d.h.:

   Wenn **alle** Knoten $N_i$ im lokalen D-Zeitmodus sind - dies ist Voraussetzung für einen globalen D-Übergang - und wenn **vor** diesem Übergang gilt

   $\max \{T_i\} \leq \min \{T_i + ta(s_i)\}$ $(1 \leq i \leq n)$

   so gilt **nach** dem globalen D-Übergang für die entsprechenden Werte $T'_i$, $ta(s'_i)$:

   $\forall i: T'_i \leq \min \{TN_i\} = \min \{T_i + ta(s_i)\}$ und

   $\forall_i: ta(s'_i) = ta(s_i) - (T'_i - T_i)$,

   also

   $\max \{T'_i\} \leq \min \{T_i + ta(s_i)\} = \min \{T'_i + ta'(s_i)\}$

(2) **Parallelarbeit der realen Testobjekte**

Wenn alle Knoten sich im lokalen R- oder D-Modus befinden, ist natürlich Parallelarbeit der R-Prozesse auf den Knoten möglich (verteilter Echtzeitablauf). Wenn dagegen auf nur einem Knoten ein W-Prozeß rechnet (lokaler F-Modus), befinden sich in der Regel alle anderen Knoten in einem Zustand untätigen Wartens.

**(3) Uhrensynchronisation im globalen R-Modus**

Von einem globalen Zeitmodus zu sprechen ist strengenommen nur dann korrekt, wenn im **globalen R-Modus** die lokalen Echtzeituhren der Knoten hinreichend gut synchronisiert sind, d.h. wenn die Uhren entweder periodisch - spätestens aber bei jedem Übergang in den F-Modus - synchronisiert werden (nach einschlägigen Algorithmen wie in /LAM 78/,/MAO 83/), oder wenn die Knoten eine **gemeinsame** Echtzeituhr referenzieren. Die Synchronie der Uhren im R-Modus sei i.f. als erfüllt angenommen.

**(4) Zeitmodus-Übergangsdiagramm eines Knotens**

Die möglichen Übergänge eines Knotens zwischen selbstbestimmten und erzwungenen Zeitmodi und ihre Auswirkungen werden anhand eines Zustandsübergangsdiagramms verdeutlicht (Abb. 5.2).

Die Zustände (Zeitmodi) sind

$S =$ {F, R, D, selbstbestimmte Zeitmodi
$F_D, R_D, F_R\}$ erzwungene "
(Index: zugehöriger lokaler Zeitmodus)

Übergänge in erzwungene Zeitmodi erfolgen immer durch master- oder slave-Operationen, die in einem restriktiveren als dem gewünschten Zeitmodus resultieren. Die Rückkehr in einen selbstbestimmten Zeitmodus kann z.B. durch eine "inverse" slave-Operation erfolgen, aber auch dadurch, daß ein Knoten später selbst den restriktiveren Zeitmodus wünscht, z.B. indem eine zeitliche Wartebedingung im erzwungenen R-Modus $R_D$ abläuft.

Abb. 5.3 zeigt ein Blockdiagramm der Netzwerksynchronisation FRED, deren Name aus den Zeitmodi (**f**reeze, **r**eal-time, **d**iscrete-event) abgeleitet ist.

### 5.1.1.2 Ein-/Ausgangssynchronisation der Zeitmodus-Operationen

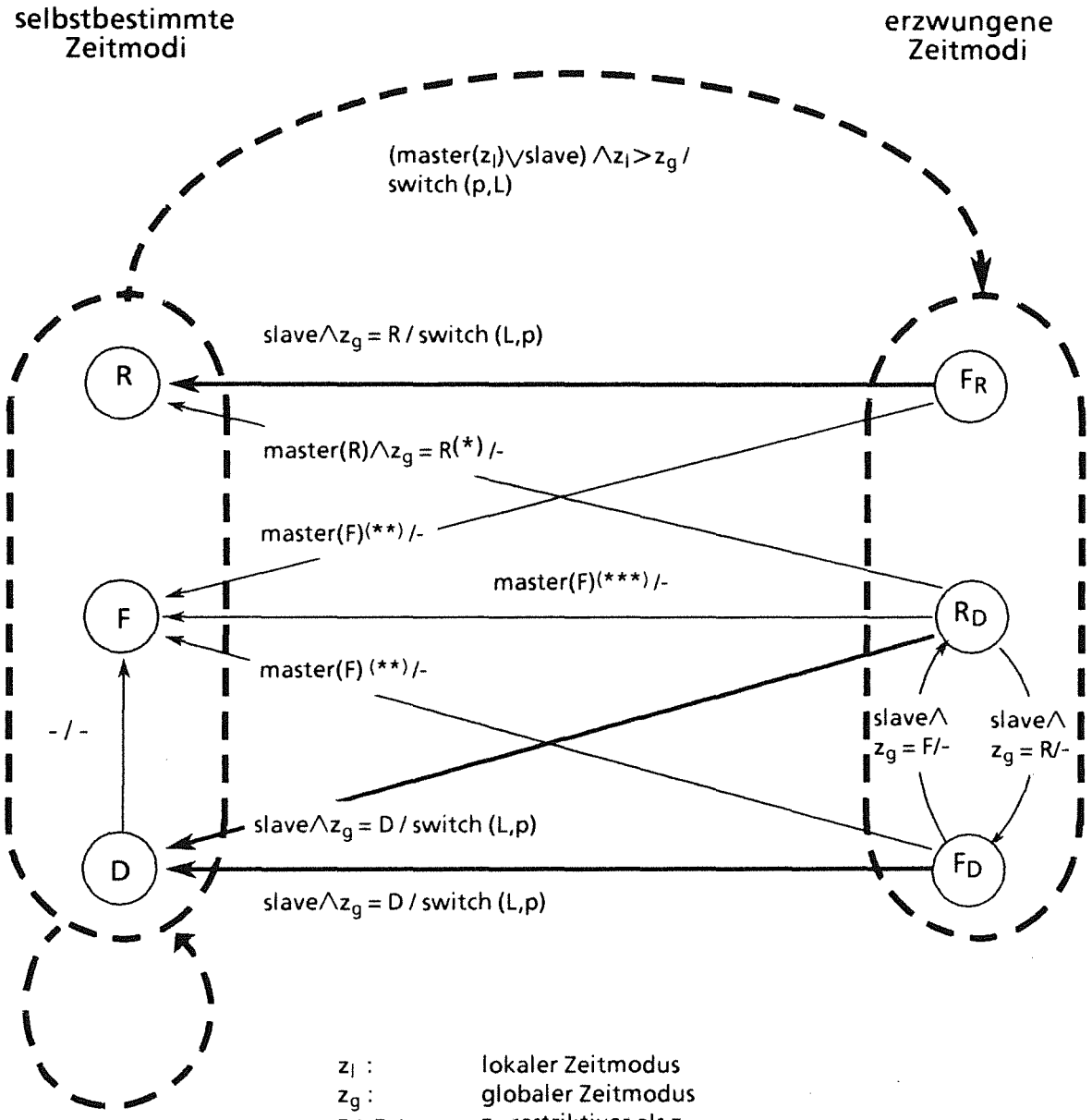
Es wird i.f. ein dezentrales, auf allen Rechnern identisches Protokoll zur Festlegung des globalen Zeitmodus vorgestellt, das keinen zentralen Koordinator oder Steuerungsrechner benötigt. Jeder Knoten agiert situationsabhängig selbst als master (initiierender Knoten) oder als slave (betroffener Knoten). Insbesondere können auch mehrere Knoten gleichzeitig eine master-Operation ausführen.

An die Realisierung werden folgende Anforderungen gestellt:

(a) Die Übergänge des globalen Zeitmodus, d.h. speziell die Übergänge vom bzw. in den Echtzeitmodus, sollen auf allen Knoten möglichst **gleichzeitig** erfolgen; zumindest müssen obere Schranken für den zeitlichen Abstand angebbar sein.

(b) Die eigentliche Durchführung von Zeitmodusoperationen soll zeitlich **transparent**, d.h. stets im F-Modus erfolgen (unabhängig von Ausgangs- und Ziel-Zeitmodus).

(c) Die Dauer der Zeitmodusoperationen soll aus Effizienzgründen möglichst **kurz** sein.




$z_l$ : lokaler Zeitmodus
$z_g$ : globaler Zeitmodus
$z_l > z_g$: $z_g$ restriktiver als $z_l$
switch(p,p'): Prozeß p' verdrängt rechenden Prozeß p
L : Leerprozeß
p : zuletzt rechnender Prozeß

(*) Asynchrones Gerätesignal (Echtzeitumgebung) oder Ablauf R-Zeiteintrag

(**) Asynchrones Gerätesignal (simulierte Umgebung)

(***)Ablauf W-Zeiteintrag

**Abb.** 5.2: Zeitmodus-Übergangsdiagramm

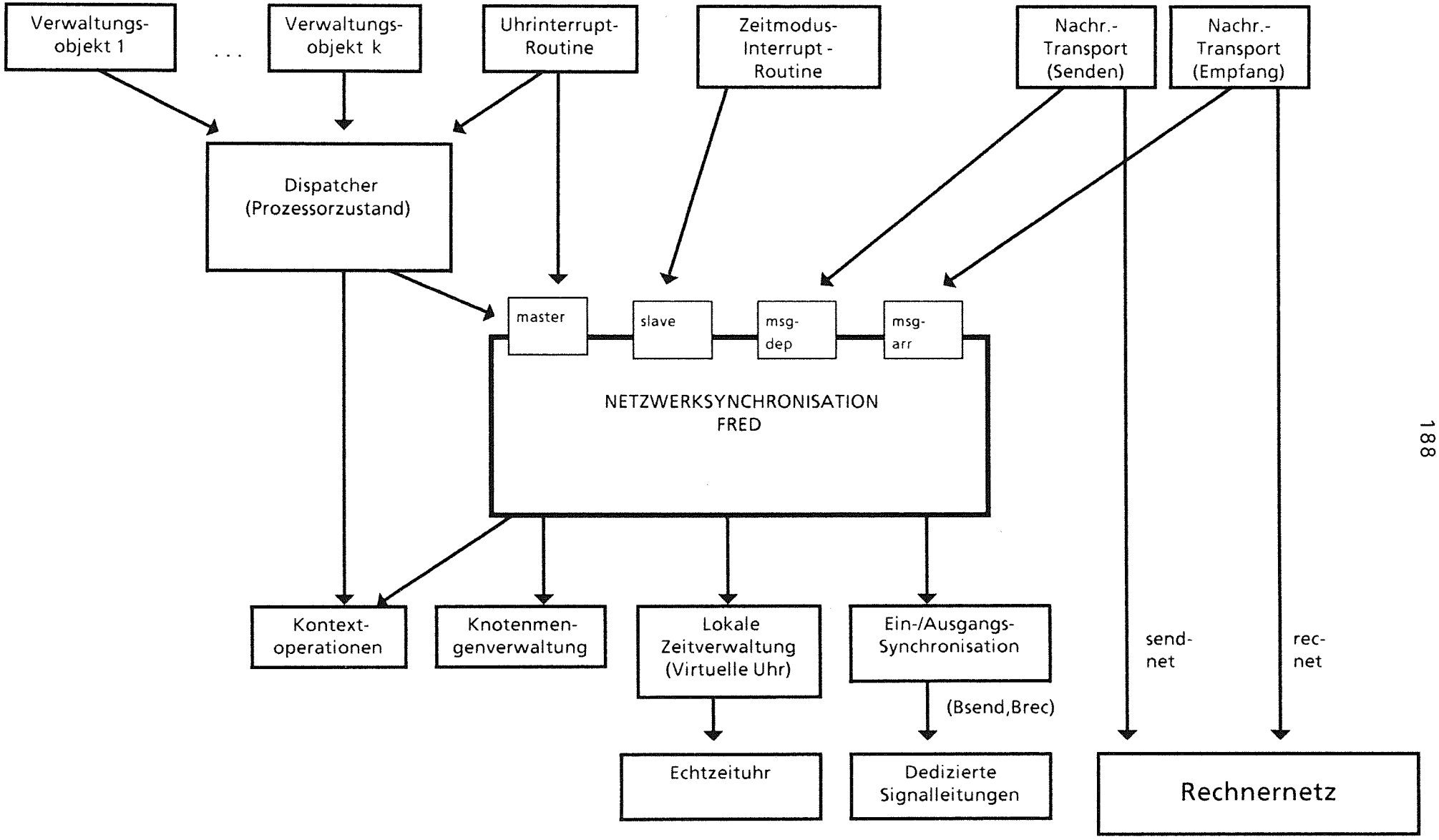



**Abb.** 5.3 Blockdiagramm der NWS für zeitlich enge Kopplung

Die für die Zeitmodusumschaltungen zuständigen Operationen zm__master, zm__slave sind als hochpriore, ununterbrechbare Unterbrechungsroutinen innerhalb des BS-Kerns konzipiert. Der Informationsaustausch der lokalen Zeitmodi erfolgt nach Abb. 5.4 in den Operationen **master__entry** und **slave__entry** (**Eingangssynchronisation**) bzw. **continue** (**Ausgangssynchronisation**) mit Hilfe eines broadcast-Protokolls. Als erstes wird in master__entry der F-Zeitmodus lokal eingeschaltet (Forderung b)). master__entry sendet an alle Knoten ein **request**-Signal, aus dem alter und neuer lokaler Zeitmodus des master-Knotens eindeutig hervorgehen, und erwartet von jedem Knoten genau eine Antwort: entweder ein **reply**-Signal, falls das request-Signal eine slave-Operation auf einem anderen Knoten aktiviert hat, oder ein request-Signal mit einem evtl. anderen Zeitmodus, falls dessen Sender selbst gleichzeitig eine master-Operation initiiert hat.

**slave__entry** wird durch die Ankunft des ersten Signals (request oder reply) eines beliebigen Knotens aktiviert und schaltet ebenfalls sofort den lokalen F-Modus ein, sendet an jeden Knoten ein reply-Signal, und erwartet, genau wie master__entry, insgesamt von jedem Knoten ein request- oder reply-Signal.

Wegen der Gleichartigkeit der Operationen und der Tatsache, daß jeder Knoten an jeden anderen genau ein Signal sendet und von jedem eines erwartet, spielt es keine Rolle, wieviele Knoten quasi-gleichzeitig eine master-Operation beginnen, und trotz der Ununterbrechbarkeit der Operationen sind Verklemmungen unmöglich. Alle Operationen terminieren, solange keine Knoten- oder Verbindungsausfälle auftreten.

Nachdem die Knotenmengen aktualisiert und gemäß (5.2) der globale Zeitmodus festgestellt wurde - im Normalfall müssen alle Knoten zum selben Ergebnis gelangen - wird dieser am Ende der master- bzw. slave-Operationen durch die Operation continue eingestellt. Im Fall des F-Modus bleibt nichts mehr zu tun. Im Fall des R-Modus synchronisieren die Knoten sich nochmals durch eine broadcast-Runde von cont (R)-Signalen, bevor jeder lokal den R-Modus einstellt. Im Fall des D-Übergangs tauschen alle Knoten ihre Sprungzeitpunkt $TN_i$ aus, setzen ihre Uhren zum globalen Minimum Tmin und gehen dann in den F-Modus über.

**HW-Annahmen (Instrumentierung)**

Für eine effiziente Übertragung der Zeitmodus-Signale (req, reply, cont, $TN_i$) werden folgende Voraussetzungen über die Instrumentierung gemacht:

- Jeder Knoten ist von jedem anderen direkt über dedizierte Datenleitungen oder einen dedizierten Bus mit hoher Bandbreite (>10Mbit/s) erreichbar, der exklusiv für die Zeitmodus-Kommunikation zur Verfügung steht.

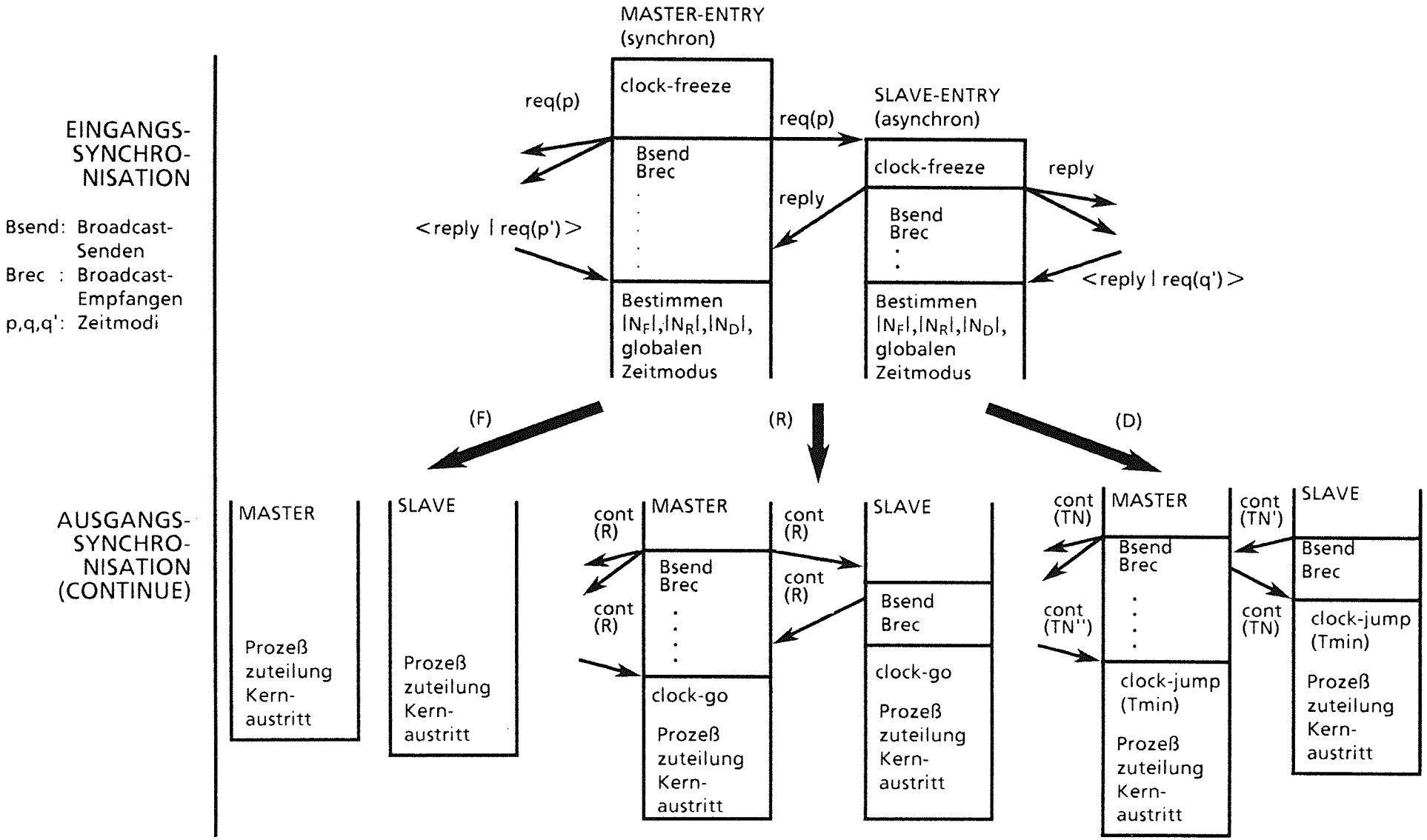



**Abb. 5.4** Ein-/Ausgangssynchronisation der Zeitmodus-Operationen

- Jeder Knoten verfüge über einen Satz von Empfangsregistern $RR_1,..,RR_n$ (für jeden Partnerknoten eines) und ein Senderegister SR. Jedem $RR_i$ ist ein separates Empfangs-Statusregister $I_i$ zugeordnet. Die $I_i$ sind über einen gemeinsamen Unterbrechungseingang (Empfangs-Interrupt) "geodert". Die zugeordnete Unterbrechungsroutine, über die die Operation zm__slave aktiviert wird, sei selbst ununterbrechbar.
- Es stehen zwei broadcast-Operationen zur Verfügung:

  broadcast__send - Simultanes Senden des Inhalts von SR an alle Partner-Knoten ($RR_j$)

  broadcast__rec - Warten, bis alle Knoten Eingabe an $RR_1,..,RR_n$ gesendet haben, d.h. bis alle $I_i$ gesetzt sind

Beispiele für Buskonzepte, welche broadcast-Übertragungen unterstützen, sind z.B. FUTUREBUS /IEE 86/ oder INTEL MULTIBUS II.

### 5.1.1.3 Grobabschätzung der zeitlichen Fehler bei Zeitmodusumschaltungen

Im Idealfall sollten Prozeßwechsel zwischen realem Testobjekt und simulierter Umgebung, und die korrespondierenden Zeitmodusumschaltungen zwischen R- und F-Modus auf allen Knoten gleichzeitig stattfinden. Implementierungsbedingt (wegen der endlichen Signalübertragungszeiten) treten in einem geographisch verteilten System natürlich zeitliche Diskrepanzen auf, die ein Maß für die Verfälschung der Experimentergebnisse (Interferenz) sind und i.f. grob abgeschätzt werden. Dazu werden folgende Bezeichnungen eingeführt:

$\Delta_I$ maximale Erkennungszeit für die Zeitmodus-Unterbrechung, d.h. Zeitspanne vom Anliegen des Unterbrechungssignals bis zum Beginn der Bearbeitung der Unterbrechungsroutine (slave__entry), vorausgesetzt, kein Knoten befindet sich bereits in einer master- oder slave-Operation, wenn das Unterbrechungssignal gesetzt wird (sonstige Ununterbrechbarkeitsphasen sind aber in $\Delta_i$ zu berücksichtigen)

$\Delta_T$ maximale Dauer einer broadcast__send- Operation, d.h. Zeitspanne vom Beginn der Operation im Sender-Knoten, bis Signal Ii in allen Empfängern gesetzt.

$\Delta_C$ maximaler Zeitbedarf der lokalen Uhroperationen clock__freeze, clock__go

$\Delta_K$ maximaler Zeitbedarf für den Austritt aus einer Unterbrechungsroutine, einschließlich Kontextwechsel.

**(a) Anhalten der Uhren**

Wenn zu einem Zeitpunkt $TE_i$ der erste Knoten $N_i$ im R-Modus in eine Zeitmodusoperation eintritt, und zwar zm__master (master__entry), so schalten alle Knoten innerhalb eines Intervalls $[TF_{min}, TF_{max}]$ den F-Modus ein mit

(5.3) $|TF_{max}-TF_{min}| \leq \Delta_I + \Delta_T + 2\Delta_C$.

Wir nehmen hierbei allerdings an, daß keine Rechnerausfälle auftreten und alle Signale korrekt und in endlicher Zeit übertragen werden.
Wegen der broadcast-Operation in $N_i$ liegen spätestens zum Zeitpunkt $TE' := TE_i + \Delta_C + \Delta_T$ bei **allen** Knoten Unterbrechungsanforderungen vor. Weil sich zum Zeitpunkt $TE_i$ laut Annahme noch kein Knoten in einer master- oder slave-Operation befindet und die maximale Erkennungszeit für die Unterbechung $\Delta_I$ beträgt, treten **alle** Knoten zu einem Zeitpunkt $TE_j \in [TE_i, TE' + \Delta_I]$ in eine Zeitmodus-Operation zm__slave oder zm__master ein (beides ist möglich). Wegen $TF_j \geq TE_i$ und $TF_j \leq TE_j + \Delta_C$ folgt die Behauptung (5.3).

**(b) Fortsetzen der Uhren**
Wenn sich alle Knoten im F-Modus befinden und der erste Knoten zum Zeitpunkt $TG_{min}$ den R-Modus einschaltet, so schalten alle Knoten im Intervall $[TG_{min}, TG_{max}]$ den R-Modus ein mit

(5.4) $|TG_{max}-TG_{min}| \leq \Delta_T + \Delta_C$,

und alle verlassen die Unterbrechungsroutine zm__master oder zm__slave zu Zeitpunkten $TA_j$, für die gilt

(5.5) $|TA_{max}-TA_{min}| \leq \Delta_T + \Delta_C + \Delta_K$.

Zunächst kann ein Knoten nur den R-Modus einschalten, wenn er zuvor an alle Knoten continue-Signale gesandt und umgekehrt von jedem eines empfangen hat (Operation continue). Daher muß auf **jedem** Knoten eine broadcast-Sendeoperation vorangegangen sein. Bei zuverlässiger Signalübertragung empfängt jeder Knoten nach endlicher Zeit von jedem anderen (genau) ein continue-Signal. Deshalb sind folgende Größen wohldefiniert:

$TS_i$ spätester Zeitpunkt $< TG_{min}$, zu dem eine Broadcast-Operation auf einem Knoten i begonnen wird

$TR_j$ spätester Zeitpunkt eines Signalempfangs in Knoten j zu einer Broadcast-Operation

Dann gilt

$TG_{max} \leq TR_j + \Delta_C \leq TS_i + \Delta_T + \Delta_C$

(zum Zeitpunkt $TR_j$ hat jeder Knoten von **allen** anderen ein Signal empfangen). Für die Austrittszeitpunkte aus einer Operation zm__master, zm__slave, welche im R-Modus endet, gilt offenbar

$TA_j \geq TG_{min}$,

$TA_{max} \leq TG_{max} + \Delta_K$.

Daraus folgen (5.4) und (5.5). Zum Zeitpunkt $TA_{max}$ ist die Voraussetzung für (a) wieder erfüllt (R-Modus, alle Knoten haben ihre zm__master- oder zm__slave-Operation verlassen).

Aus der obigen Abschätzung ergibt sich insbesondere, daß für die maximal zu verkraftende "Umschaltrate" $\lambda$ in den F-Modus gilt

(5.6) $\lambda << 1/(\Delta_I + 2\Delta_T + 3\Delta_C + \Delta_K)$,

bzw. daß immer dann, wenn ein reales Testobjekt im Experiment vorhanden ist, die zeitliche Granularität der Ereignisse der simulierten Umgebung wesentlich gröber als $1/\lambda = (\Delta_I + 2\Delta_T + 3\Delta_C + \Delta_K)$ sein muß.

**(c) Zeitsprung**

Wichtig in diesen Fall ist nur folgende Tatsache: **alle** vom D-Übergang betroffenen Knoten müssen ihre lokalen Uhren auf denselben Wert T inkrementiert haben, bevor auf **irgendeinem** Knoten eine weitere Uhroperation (R-Modus oder D-Übergang) erfolgen kann, die zu einem Fortschreiten einer der Virtuellen Uhren führt. Daher sind D-Übergänge nicht echtzeit-kritisch; sie müssen nur in **endlicher** Zeit stattfinden.

**Diskussion**

Die Abschätzungen (5.3)-(5.5) setzen abgesehen davon, daß Rechner- und Leitungsausfälle nicht betrachtet wurden, offenbar die **Ununterbrechbarkeit** zumindest der Operationen master__entry, slave__entry, continue zwingend voraus. Nur dann lassen sich deterministische obere Schranken für die absolute Dauer von Operationen überhaupt angeben.

Die Zeiten $\Delta_C$, $\Delta_T$, $\Delta_K$ sind anwendungs- und datenunabhängige Konstanten. Die Zeit $\Delta_T$ liegt im Bereich des n-fachen der reinen Transportzeit einer einzelnen Broadcast-Operation (für 4 bit Nutzdaten), falls das Transportmedium exklusiv für das Zeitmodusprotokoll zur Verfügung steht (NB: aufgrund der Protokollstruktur ist niemals mehr als **eine** Übertragung **pro Knoten** gleichzeitig aktiv!). Unter diesen Voraussetzungen ist $\Delta_T$ mehr oder weniger deterministisch. Für Anwendungen mit hohem Genauigkeitsbedarf, bei denen dedizierter Entwicklungsaufwand zu rechtfertigen ist, lassen sich $\Delta_C$, $\Delta_T$, $\Delta_K$ alle durch HW-Unterstützung in die Größenordnung weniger (10-20) Prozessortakte bringen, ohne Änderungen am Synchronisationsverfahren zu erfordern, z.B.

- HW-Realisierung der Uhr-Funktionen clock__freeze, clock__go
- Verschmelzung von Uhrfunktionen mit dem Eintritt/Austritt in/aus Unterbrechungsroutinen zu einem Maschinenbefehl /KOH 82/
- HW-unterstützter Kontextwechsel (es würde schon Entlastung bringen, wenn 2 getrennte Registersätze für R- und W-Prozesse zur Verfügung stünden)
- Hochgeschwindigkeits-Bussysteme für die Zeitmodusübertragung.

Der eigentlich **kritische** Summand ist die **Interrupt-Erkennungszeit** $\Delta_I$ in (5.3), die $\Delta_C$ bzw. $\Delta_T$ um ein Vielfaches übersteigen kann. Daher sollte ein Experimentrechner jederzeit (auch innerhalb einer BS-Kern-Funktion) auf solche Signale reagieren, die eine Änderung des globalen Zeitmodus bewirken können (Zeitmodus- sowie Uhrunterbrechungen, vgl. 5.1.3).
Das Unterbrechungskonzept als solches scheint zur Realisierung von Zeitmodusumschaltungen dagegen unverzichtbar, im Gegensatz etwa zu der bei Wartungsrechnern oder in-circuit-Emulatoren angewandten Methode, einen Rechner durch Suspendieren des Befehlsausführungs-Taktes anzuhalten. (vgl. 3.6.3).

### 5.1.2 Intervallgesteuertes Ablaufmodell für W-Prozesse

#### 5.1.2.1 Grundidee, Anwendbarkeit

In allen bekannten prozeßorientierten Simulationssprachen entspricht jeder Aktion eines Modellprozesses ein diskreter Zeitpunkt auf der virtuellen Zeitachse. Wenn ein solcher Modellprozeß z.B. die Bearbeitung von Aufträgen auf einer Bedienstation simuliert, stellt sich dies typischerweise (vereinfacht) so dar:

```
host process server is
...
loop
    rec__msg (mb__auf, request);          Warten auf Aufträge
    {tv(p)=T0}
    res.comp1 := F1 (request);            Simulierte Auftragsbearbeitung,
      ...                                 Leistungsgrößen, etc.
    res.compn := Fn (request);
    d:= service__time (request);          Berechnen und "Verbrauchen"
    delay__for (d);                       der Auftragsbearbeitungszeit
    {tv(p)=T0+d}
    send__msg (mb__res, res);             Rückmeldung bzw. Weiterleitung
end loop;                                 der Ergebnisse
```

Jede der Aktionen $F_1,..,F_n$ findet entweder bei T0 oder (falls die Zeitverzögerung delay und die $F_i$ vertauscht werden), bei T0+d, in jedem Fall zu einem definierten Zeitpunkt statt.
Offensichtlich ist dieser Ablauf zeitlich überspezifiziert. Erforderlich ist nur, exakt zum Zeitpunkt T0 Eingabedaten (request) entgegenzunehmen und entsprechend der simulierten Bedienzeit d die Ergebnisse exakt zum Zeitpunkt T0+d zur Verfügung zu stellen, wobei von weiteren möglichen Interaktionen mit der Außenwelt innerhalb [T0,T0+d], wie Simulation einer präemptiven Bearbeitung oder Abbrechen eines Auftrags, vorerst abgesehen werde, s.u..

Das Simulations-Laufzeitsystem braucht die notwendigen Berechnungen (beginnend bei $T0_e$ und endend bei $T1_e$ in Echtzeit) nur so in dem virtuellen Zeitintervall [T0,T0+d] "unterzubringen", daß der zeitliche Verlauf ts:T->T (vgl. 3.4.3.3) durch ($T0_e$,T0),($T1_e$,T0+d) geht und monoton

wachsend ist (Abb. 5.5).

Zur Klammerung solcher Aktionsfolgen eines sequentiellen W-Prozesses durch virtuelle Zeitintervalle werden zwei neue Systemdienste vorgeschlagen:

**set-interval** spezifiziert ein Zeitintervall bzw. seinen Endpunkt für die nachfolgenden Operationen, und **check-interval** schließt die Operations-Folge ab. Damit erhält man folgende Formulierung des obigen Beispiels (man könnte sich sicherlich sprachlich elegantere Konstruktionen vorstellen):

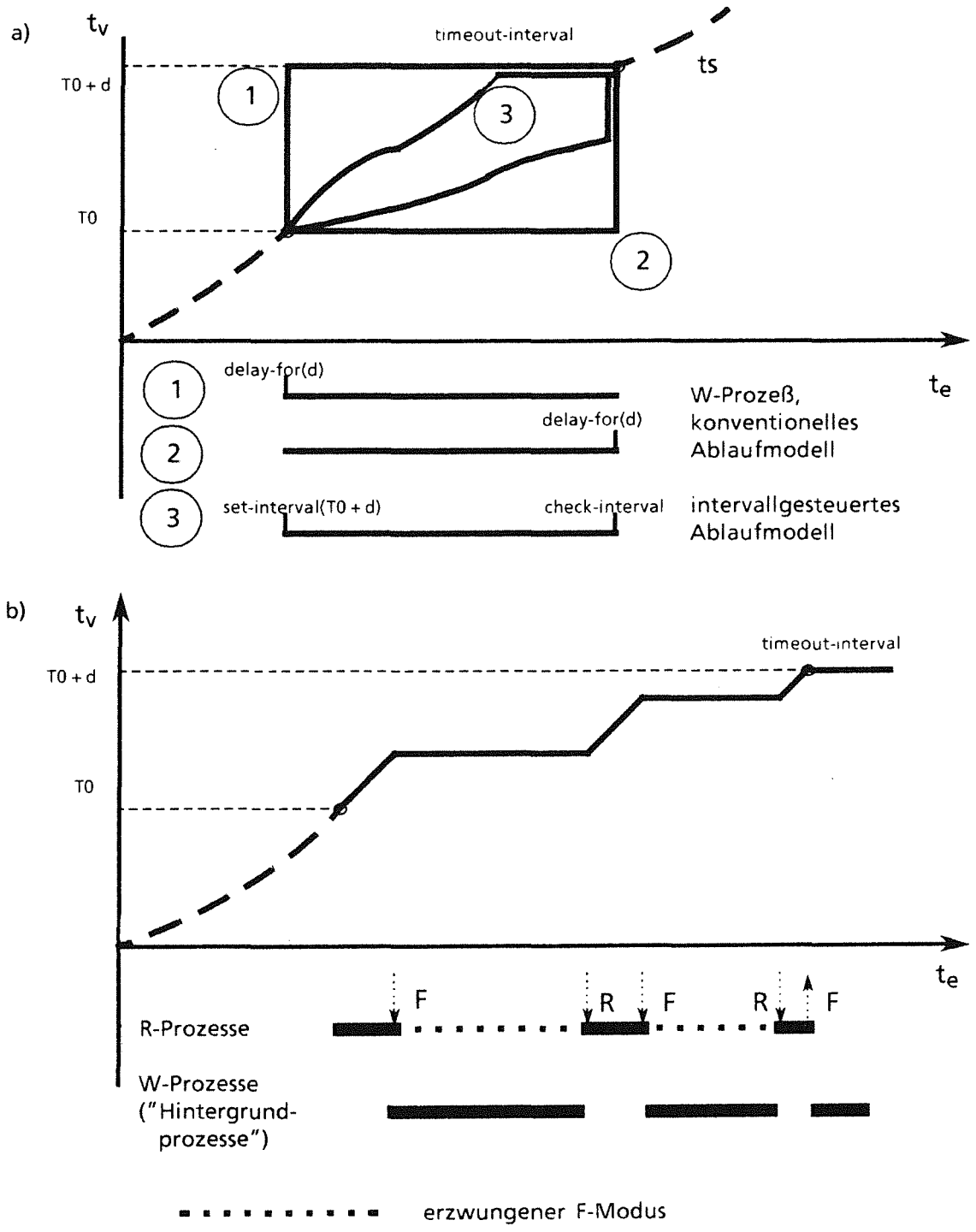



**Abb.** 5.5 Zeitführung (a) und Hintergrundbetrieb (b) im intervallgesteuerten Ablaufmodell

```
host process server is
...
loop
    rec__msg (mb__auf, request);

    {tv(p)=T0}
    set__interval (clock__read + service__time(request));
    res.comp1 := F1 (request);
    ...
    res.compn := Fn (request);
    check__interval;
    {tv(p)=T0+d}

    send__msg (mb__res, res);
end loop
```

Ein W-Prozeß p habe ein **offenes Intervall**, wenn gilt:

- p hat set__interval durchlaufen (d.h. ein virtuelles Zeitintervall mit Endpunkt T1 ist für p spezifiziert)
- p hat das korrespondierende check__interval noch nicht durchlaufen
- der obere Intervall-Endpunkt T1 ist noch nicht abgelaufen (clock__ack $\leq$ T1).

Die Ablaufsemantik für einen W-Prozeß mit offenem Intervall ist wie folgt: wenn die virtuelle Uhr T1 erreicht (**timeout-interval**), aber check__interval noch nicht durchlaufen wurde, wird der F-Zeitmodus eingeschaltet (die virtuelle Uhr wird angehalten). Wenn dagegen die Endeklammer check__interval zuerst erreicht wird, wird p zeit-blockiert bis T1.

Dieses Ablaufmodell bringt zwar in einer Einprozessorumgebung noch keine Effizienzvorteile, wohl aber in einer Mehrrechnerumgebung. Solange alle rechenbereiten W-Prozesse eines Knotens ein offenes Intervall besitzen, wird bereiten R-Prozessen - falls vorhanden - sogar der Vorrang gegeben. Aber jedesmal, wenn der Knoten extern in den erzwungenen F-Modus überführt wird, die R-Prozesse also unterbrochen werden müssen, werden diese W-Prozesse als Hintergrundaufgabe bearbeitet (diese Möglichkeit bestand vorher nicht), vgl. Abb. 5.5 (b).

Das **intervallgesteuerte Ablaufmodell** eignet sich zur Simulation von Warteschlangensystemen (insbesondere nichtpräemptiver Bedienung) sehr gut. Um seine Eignung für **allgemeine**, diskrete Simulationen zu prüfen, soll noch einmal die DEVS-Komponente (4.2.1.3, Abb. 4.3) als Beispiel herangezogen werden.

Der Rechenaufwand zur Simulation einer DEVS-Komponente entfällt vor allem auf die wiederholte Anwendung der Funktionen

(1) ta(s) - Bestimmung der verbleibenden Zeit im Zustand s bis zum nächsten autonomen Zustandsübergang

(2) $\delta\_\varnothing(s)$ - autonomer Zustandsübergang

(3) $\lambda(s)$ - Ausgabe nach autonomem Zustandsübergang

(4) $\delta\_ex(s,e,x)$ - externer Zustandsübergang

Um die Monotonie der virtuellen Zeit zu wahren, darf bei (1) oder (4) die Intervallänge die kleinstmögliche Zeit bis zum nächsten Zustandsübergang nicht überschreiten. Da die Zeit ta(s) in (1) erst berechnet wird und auch keine feste untere Schranke > 0 hierfür a priori bekannt ist, hat das Intervall für (1) und (4) die Länge 0. Dagegen kann in jedem Zustand s der nächste autonome Übergang $\delta\_\varnothing(s)$ in (2) und evtl auch die Ausgabe $\lambda(s)$ (3) in einem Intervall der - nunmehr bekannten - Länge ta(s) vorausberechnet werden. Die Berechnung wird aber abgebrochen, wenn in [T, T+ta(s)) eine externe Eingabe einer anderen Komponente eintrifft. Der aktuelle Zustand s darf erst dann durch $\delta\_\varnothing(s)$ ersetzt werden, wenn das Intervall abgelaufen ist.

Dies zeigt

- Manche Beispiele enthalten wenig Potential zur Ausnutzung des intervallgesteuerten Ablaufmodells.
- Intervalle und ihre Aktionen sollten vorzeitig abgebrochen werden können.

#### 5.1.2.2 Systemdienste, Abgrenzung

Die wichtigsten Systemdienste für ein intervallgesteuertes Ablaufmodell werden i.f. beschrieben.

**set-interval** (intv: interval, T1: absolute_time);

Der aufrufende W-Prozeß eröffnet ein Intervall intv mit Endpunkt T1 (o.E. sei der Intervall-Anfangspunkt T0 stets die Gegenwart) und bleibt rechnend. Der Dienst kann auch verwendet werden, um den Endpunkt eines bereits eröffneten Intervalls dynamisch zu verändern, die Schachtelung mehrerer Intervalle für denselben Prozeß ist jedoch unzulässig.

**check-interval** (intv: interval)

schließt die durch ein Intervall geklammerte Aktionsfolge des Aufrufers ab und zeit-blockiert diesen bis zum Endpunkt des offenen Intervalls. Falls intv nicht eröffnet oder bereits abgelaufen ist, ist der Dienst wirkungslos. Zum Abbruch eines Intervalls können Asynchronisations-Elemente nach /WET 84/, Kap. 2.7, vorteilhaft verwendet werden. Es wird hier nur eine einfache Version mit zwei Diensten benötigt.

**connect-interval** (intv: interval, adr: address);

**abort-interval** (intv: interval);

Durch connect__interval spezifiziert der aufrufende W-Prozeß einen "Notausgang" (Sprungadresse bzw. Marke) für den möglichen späteren Abbruch eines Intervalls (abort__interval) durch einen fremden Prozeß. Die Adresse kann leer sein (nil); in diesem Fall wird nur das Intervall geschlossen, aber der Prozeß läuft normal weiter.
Die Wirkung von abort__interval ist wie folgt: wenn intv offen oder der beeinflußte W-Prozeß p zeit-blockiert in check__interval ist, wird

- p in jedem Fall bereit
- der Kontrollfluß von p zu der durch connect__interval spezifizierten Adresse adr umgelenkt (er bleibt unverändert, falls adr = nil)
- das Intervall intv geschlossen, falls es noch offen ist
- die virtuelle Zeit von p nach dessen Wiederaufnahme (bei adr) vom aufrufenden Prozesses q übernommen. q sollte sich nicht selbst in einem offenen Intervall befinden, sonst wäre diese Zeit zufällig.

Wenn intv nicht offen oder unbekannt ist, ist der Aufruf wirkungslos.

In Abb. 5.6 ist das DEVS-Beispiel aus Abschnitt 4.2.1.3 unter Benutzung dieser Dienste neu formuliert. Wenn eine DEVS-Komponente ihr Intervall für $\delta\_\_\varnothing$ (vgl. 5.1.2.1, (2)) erfolgreich durchlaufen hat und die Frist ta(s) abgelaufen ist, wird der vorausberechnete autonome Zustandsübergang $\delta\_\_\varnothing$ durchgeführt, die Ausgabe $\lambda$(s) produziert und zugleich die Intervalle ihrer Empfängerkomponenten abgebrochen. Die Empfänger werden nach ext__trans__label "umdirigiert", um dort ihre Eingabe entgegenzunehmen und einen externen Zustandsübergang durchzuführen.

**Randbedingungen für Intervalle**

Durch Intervalle werden vor allem lokale, interaktionsfreie Rechenphasen von W-Prozessen geklammert. Mit Einschränkungen sind aber innerhalb eines offenen Intervalls auch BS-Dienste möglich:

(a) E/A-Gerätesteuerung (send__dev, rec__dev)
(b) Anforderung/Freigabe von Arbeitsspeicher.

Mit Hilfe von a) läßt sich z.B. die Aktionsfolge eines Vorschaltgerätes (4.2.5.5)

- Simulation des Gerätezustands und der Bearbeitungszeit
- Physikalischer Gerätebetrieb (send__dev, rec__intr)

durch ein Intervall der zu simulierenden Gerätebearbeitungszeit tb(t,p,z) klammern.

**Nicht erlaubt** für einen W-Prozeß mit offenem Intervall sind hingegen

(c) Senden und Empfangen von Botschaften
(d) Erzeugen, Starten und Beenden von W-Prozessen
(e) Warten auf Zeit (delay), Ändern zeitlicher Wartebedingungen (reschedule, **auch** nicht: abort__interval).

**Abb.** 5.6 Formulierung einer discrete-event-Komponente im intervallgesteuerten Ablaufmodell

```
-- Globale Deklarationen etc. wie in Abb. 4.3
intv : array (range_devs) of interval;

host process body p is

t_last     :   absolute_time := clock_read;
s_next,
s          :   devs_state  := s0;
out_msg    :   array (range_devs) of interface_msg;
x          :   interface_msg;
res        :   result_par;
i          :   range_devs;

begin
    connect_interval (intv(own_id), ext_trans_label);

    loop
        set_interval (intv(own_id), clock_read + ta(s));
        s_next := δ_∅ (s);
        for i member of infl
            out_msg(i) := λ (s,i);
        check_interval (intv(own_id));

        for i member of infl
        loop
            abort_interval (intv(i));
            send_msg (M(i), out_msg(i),∞ , res);
        end loop;
        s:= s_next;

        if false then
        begin
            <<ext_trans_label>>
            rec_msg (M(own_id), x, 0, res); -- Sendewunsch liegt bereits an
            s:=δ_ex(s, clock_read-t_last, x);
        end;

        t_last:= clock_read;
    end loop;
end p;
```

Solche Dienste zuzulassen, würde die zeitliche Semantik der prozeßorientierten Simulation prinzipiell in Frage stellen, insbesondere im Fall der Kommunikationsoperationen (c) und ähnlich für (d) und (e):

(1) Konstruktionen der Form

```
set_interval (T);
... rec_msg (mb, msg, ∞, res);
...
check_interval;
```

würden zusätzliche Festlegungen erfordern, sollen sie nicht in sich widersprüchlich sein: falls bis zum Zeitpunkt T keine Botschaft eintrifft, wird dann die Empfangsoperation zwangsweise mit timeout-Resultat beendet (obwohl timeout ∞ spezifiziert ist), oder wird das Intervall mit Fehler abgebrochen, obwohl kein Fehlerausgang angegeben ist ?

(2) Die Intervallspezifikation vermeidet unnötig restriktive zeitliche Festlegungen für lokale, interaktionsfreie Rechenphasen von W-Prozessen. Kommunikationsoperationen sind dagegen Interaktionen mit der Außenwelt, die zu definierten, durch das Modell spezifizierten Zeiten stattfinden. Wären diese auch im offenen Intervall zulässig, so bliebe die Wahl der Interaktionszeitpunkte selbst der unterliegenden Ablaufmaschine überlassen, abhängig z.B. von der zufälligen realen Ausführungsdauer der Simulationsmodelle und ihrer Verteilung auf die Experimentrechner.

### 5.1.2.3 Prozessorzustandsdiagramm

Die Implementierung der zusätzlichen Systemdienste für das intervallgesteuerte Ablaufmodell als solche ist unproblematisch und soll hier nicht diskutiert werden. Interessant ist jedoch die Prozessorzuteilung (dispatching), insbesondere der in 5.1.2.1 erwähnte Hintergrundbetrieb von W-Prozessen, der ja der eigentliche Grund für die Einführung dieses Ablaufmodells war. Alle hierzu nötigen Zuteilungsregeln lassen sich durch einfache Modifikationen aus dem in 4.2.6 vorgestellten Prozessorzustandsdiagramm für das konventionelle Ablaufmodell gewinnen (Tab. 5.1 und Diagramm 5.7).

Die Bedeutung der Prozeßmengen FB[R], FB[W], FWEA, FWB ist dieselbe wie in 4.2.6. Sei ferner

FI: Menge der W-Prozesse mit offenem Intervall (5.1.2.1).

Es gilt offenbar

$FI \subset (FB[W] \cup FWB \cup FWEA)$

wegen der Einschränkungen 5.1.2.2 (c)-(e) der zulässigen Systemdienste für W-Prozesse mit offenen Intervallen.

Aus Gründen der Übersichtlichkeit werden S-Prozesse i.f. nicht betrachtet.

| Zust. | Bedingung | lok. Zeit-modus | glob. Zeit-modus | rechnender Prozeß |
|---|---|---|---|---|
| WI | $(FB[W] \cup FWEA \cup FWB) \setminus FI \neq \emptyset \wedge FB[W] \neq \emptyset$ | F | F | erst $(FB[W] \setminus FI)$ falls $\neq \emptyset$<br>erst $(FB[W]$ sonst |
| LFS | $(FWEA \cup FWB) \setminus FI \neq \emptyset \wedge FWEA \neq \emptyset \wedge FB[W] = \emptyset$ | F | F | LF |
| LFA | $FWB \setminus FI \neq \emptyset \wedge FWEA = \emptyset \wedge FB[W] = \emptyset$ | F | F | LF |
| RI | $(FB[W] \cup FWEA \cup FWB) \subset FI \wedge FB[R] \neq \emptyset$ | R | R<br>$F_R$ | erst $(FB[R])$<br>erst $(FB[W])$ falls $\neq \emptyset$<br>LF sonst |
| LDI | $(FB[W] \cup FWEA \cup FWB) \subset FI \wedge FB[R] = \emptyset$ | D | D<br>$R_D, F_D$ | LD<br>erst $(FB[W])$ falls $\neq \emptyset$<br>LF sonst |

**Tab.** 5.1 Prozessorzustände im intervallgesteuerten Ablaufmodell

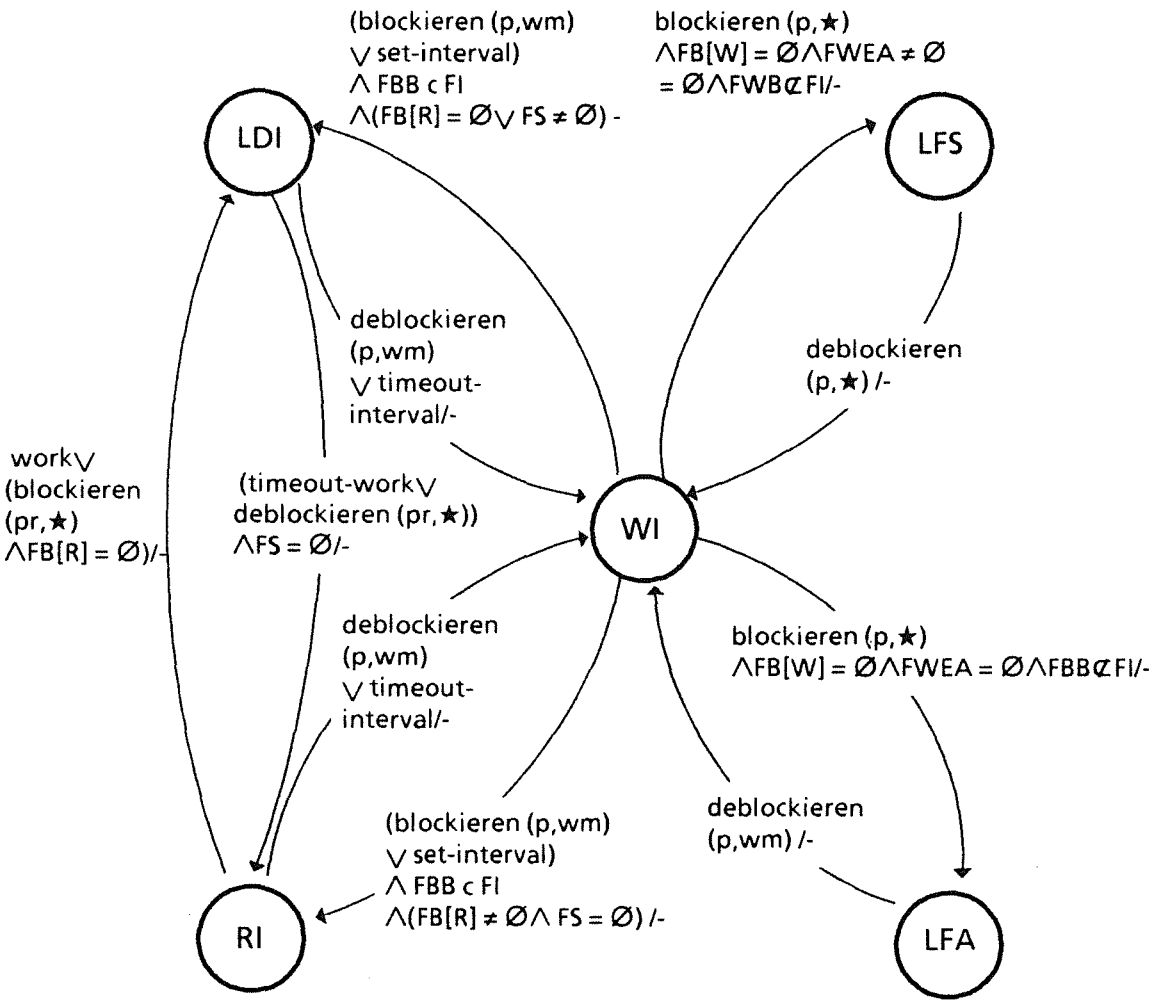



**Abb.** 5.7: Prozessorzustandsdiagramm im intervallgesteuerten Ablaufmodell (Bezeichnungen wie in Abb. 4.14)

**Bedeutung der Zustände**

Die ersten drei Zustände WI, LFS und LFA sind durch die gemeinsame Obervoraussetzung (FB[W]∪FWEA∪FWB)\FI≠∅ gekennzeichnet. Danach gibt es Aktionen von W-Prozessen, die zum gegenwärtigen Zeitpunkt bearbeitet werden müssen und daher (lokal und global) den F-Zeitmodus verlangen.

Im einzelnen

WI Mindestens ein W-Prozeß ist entweder bereit oder wartet auf ASP oder E/A, für den **kein** offenes Intervall existiert. Also muß der F-Modus herrschen,und nur bereite W-Prozesse sind zuteilbar.

Wenn möglich, werden natürlich solche W-Prozesse zugeteilt, die zum gegenwärtigen Zeitpunkt fällig sind, also kein offenes Intervall haben. Falls alle diese W-Prozesse E/A-blockiert sind, können auch bereite W-Prozesse ∈ FI rechnen.

LFS Mindestens für einen auf passive BM oder E/A wartenden W-Prozeß existiert kein offenes Intervall (daher F-Modus). Es dürfen keine Prozesse zugeteilt werden (auch wenn FB[R]≠∅ gelten sollte). Diese Blockade wird aber durch **erwartete** Fertigmeldungen von E/A-Geräten (E/A-Aufträge) beendet werden.

LFA Es existiert mindestens ein auf ASP wartender W-Prozeß ohne offenes Intervall. Weil weder bereite noch auf E/A wartende W-Prozesse existieren, besteht eine Chance, die geforderten BM ohne Fortschreiten der Zeit zu erlangen, nur dann noch, wenn ein ASP belegender Prozeß durch ein **unerwartetes** (asynchrones) externes Ereignis aktiviert wird, und dieses muß auch noch zeitlich in der Gegenwart liegen.

RI Die zu WI, LFS, LFA komplementäre Obervoraussetzung - (FB[W] ∪FWEA∪FWB) ⊆ FI - besagt, daß keine W-Prozesse existieren, die ein Anhalten der virtuellen Zeit **erzwingen**, möglicherweise existieren aber bereite bzw. auf Betriebsmittel wartende W-Prozesse.

Der lokale Zeitmodus ist R, daher sollten R-Prozesse jetzt vorzugsweise zum Zuge kommen. Dies ist nur möglich, wenn auch global der Zeitmodus R herrscht. Falls jedoch aufgrund externer Anforderungen der F-Modus $F_R$ erzwungen wird, werden bereite W-Prozesse als "Hintergrundprozesse" zugeteilt, die ja alle ein offenes Intervall besitzen. Nur wenn FB[W] leer ist, muß noch der Leerprozeß LF zugeteilt werden wie im konventionellen Ablaufmodell.

LDI Die Obervoraussetzung ist die des Zustands RI, aber es existieren keine zuteilbaren R-Prozesse. Vorzugsweise wird die Leerphase durch einen D-Übergang überbrückt (LD wird zugeteilt). Bei erzwungenem Zeitmodus $R_D$ oder $F_D$ rechnen Hintergrundprozesse wie im Zustand RI.

Das zugehörige Übergangsdiagramm Abb. 5.7 ist fast identisch mit dem Diagramm 4.14 im konventionellen Ablaufmodell. Indem man die möglichen Kombinationen von gegenwärtigem Prozessorzustand und nächsten Systemdienst bzw. asynchronem Ereignis durchspielt und die Einschränkungen c)-e) in 5.1.2.2 beachtet, kann man leicht nachweisen, daß die in Abb. 5.7

dargestellten Übergänge tatsächlich die einzig möglichen sind. Insbesondere sind die einzigen gegenüber Abb. 4.14 neuen Übergänge des Prozessorzustands: WI->LDI bzw. WI->RI (set_interval), bzw. LDI->WI bzw. RI->WI (timeout_interval).

### 5.1.3 Wiedereintrittsfähiger BS-Kern (interrupt-resume-Schema)

Zeitlich eng gekoppelte verteilte Experimentrechnerkonfigurationen beruhen auf einem Gleichlauf der Knoten nach einer einheitlichen, globalen virtuellen Zeit. Der Grad an Interferenz hängt entscheidend von den maximalen Reaktionszeiten der Experimentrechner auf Zeitmodus-Unterbrechungssignale (vgl. 5.1.2.3) und Uhr-Unterbrechungssignale (Ablauf von W-Zeiteinträgen im R-Modus) ab. Diese Reaktionszeiten gilt es zu minimieren. Notwendig ist zunächst, daß diese Signale und ihre zugeordneten Unterbrechungsroutinen:

- zm_master-Operation (vgl. 5.1.1.2)
- zm_slave-Operation (vgl. 5.1.1.2)
- Uhrinterrupt-Routine

eine höhere Unterbrechungsebene (UE) als alle anderen Systemaktivitäten, insbesondere auch BS-Kernfunktionen, besitzen. D.h. falls im Zielsystem die UEen 0,..,i belegt sind, belegen die o.g. Operationen exklusiv eine UE z > i.

Denkt man sich den Ablauf aller R-Prozesse eines Experimentrechners, in dem Anwendungs- und BS-Kern-Funktionen abwechseln, zu einem einzigen sequentiellen Prozeß zusammengefaßt, und den Ablauf der W-Prozesse zu einem zweiten, so kann man sich die Operationen der Zeitmodusumschaltung als Kontextwechseloperationen auf höherer Ebene vorstellen, welche zwischen diesen beiden "Prozessen" umschalten (vgl. Ablaufdiagramm Abb. 5.8). Die Realisierung eines solches Schemas setzt zunächst ein leistungsfähiges Unterbrechungssystem voraus, das eine präemptive Verarbeitung von Unterbrechungssignalen auf mehreren Prozessorprioritäts-Ebenen der zugeordneten Unterbrechungsroutinen garantiert, wie z.B. in /FAE 79/, Kap. 3.2.4 und 4.1.4. Daneben muß vor allem die SW-Architektur des BS-Kerns besondere Anforderungen erfüllen, die in der BS-Literatur, z.B. /LEV 81//PES 83//WET 84/ kaum behandelt werden. Das Studium der Implementierung existierender BS-Kerne wie z.B. RSX-11M oder UNIX hilft auch nicht weiter, weil diese nur eine sehr eingeschränkte Unterbrechbarkeit von Kernfunktionen vorsehen und vor allem keine methodischen, auf unterschiedliche Hardware portierbaren Ansätze liefern. Eine Ausnahme bildet die Arbeit /TEM 87/, welche prinzipielle Lösungsansätze zur Konstruktion unterbrechbarer (mikroprogrammierter Echtzeit-)BS-Kerne diskutiert, das sog. interrupt-abort und das interrupt-resume-Schema. Letzteres kommt im Grundsatz unserer Anwendung entgegen; aus Gründen des Umfangs wird jedoch nur das Lösungsprinzip angedeutet.

Welches sind die **Kernprobleme** bei der Realisierung eines unterbrechbaren BS-Kerns?

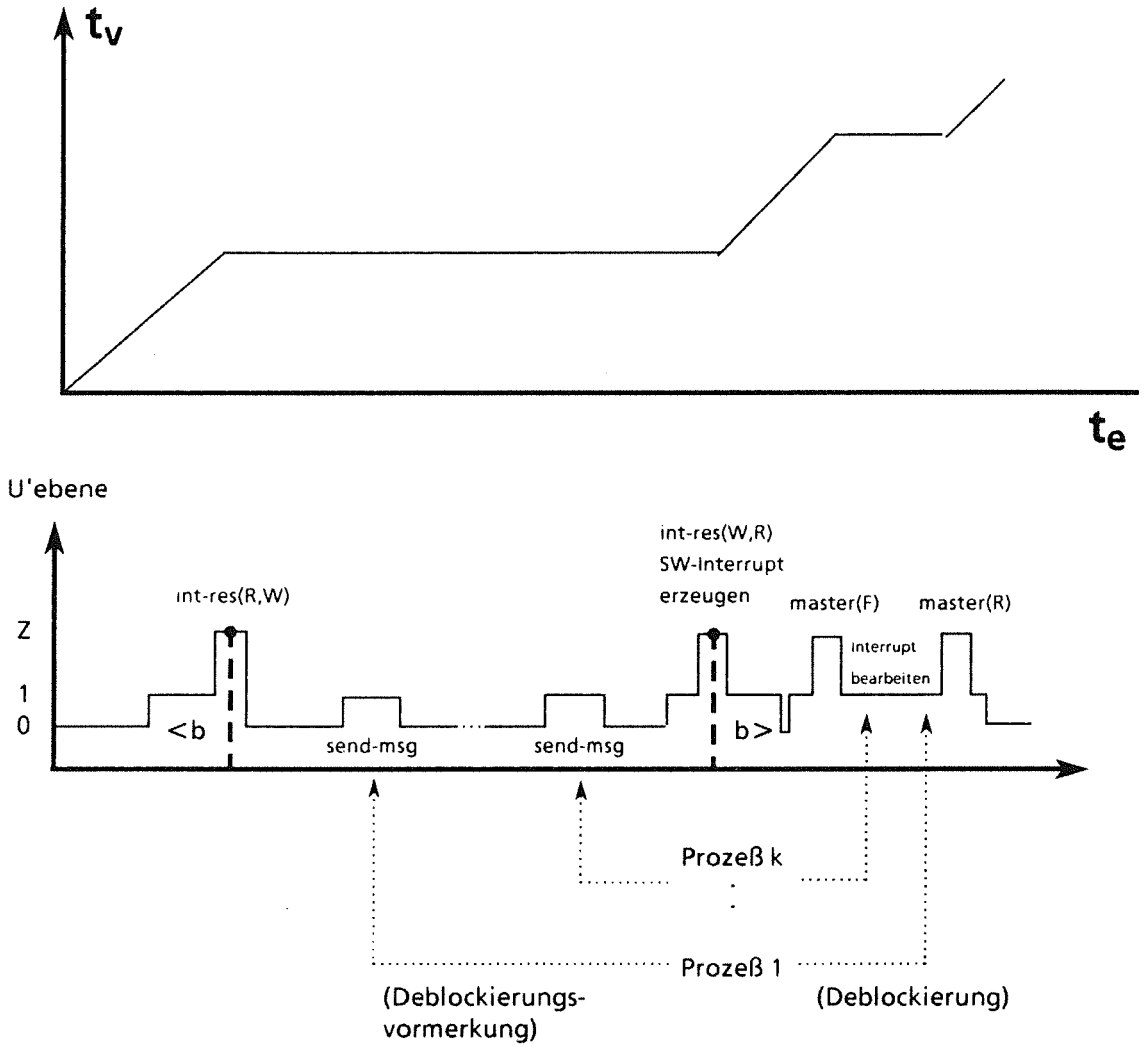



**Abb.** 5.8: Gantt-Diagramm zum wiedereintrittsfähigen BS_Kern.
(Deblockieren unterbrochener R-Prozesse)

(P1) Unter der Regie von R-Prozessen ablaufende BS-Kernfunktionen werden durch Operationen der Zeitmodusumschaltung unterbrochen und können später unter der Regie von W-Prozessen erneut betreten werden (symmetrisch für die W-Prozesse (Hintergrundprozesse)). Diese **Wiedereintrittsfähigkeit** von Kernfunktionen verlangt, daß gemeinsame Datenobjekte des BS-Kerns, die sich zum Zeitpunkt der Unterbrechung in einem inkonsistenten Zustand befinden können, nicht überschrieben, sondern beim Prozessorentzug von R-(W-)Prozessen zusammen mit dem Rechnerkontext (Registerinhalte etc.) gesichert und bei ihrer Wiederaufnahme restauriert werden. Hierzu dient eine spezielle Kontextwechseloperation

**int_res** (von, nach: exec_mode) mit

**type** exec_mode = (R,W).

(P2) Da Kommunikation und Synchronisation zwischen R- und W-Prozessen sich letztlich auch im Zustand gemeinsamer BS-Daten widerspiegelt, bleibt das Problem, wie jede Klasse von Prozessen eine **konsistente** und **aktuelle** Version dieser Daten unter Berücksich-

tigung der seit der letzten Unterbrechung von der jeweils anderen Klasse von Prozessen eingebrachten **Änderungen** vorfindet.

Während die Wiedereintrittsfähigkeit in beliebig strukturierten Betriebssystem-Kernen fast unlösbare Probleme aufwirft, wird das Problem im hybriden Kern durch folgende Eigenschaften und Ansätze entscheidend erleichtert:

(1) R- und W-Prozesse sind innerhalb des BS-Kerns nach 4.3 bereits zum größten Teil durch disjunkte Datenstrukturen vertreten: disjunkte Listen bereiter Prozesse, disjunkte ASP- und E/A-Verwaltungsobjekte und disjunkte Zeitlisten. Dieses Prinzip läßt sich auf die gesamte Zeitverwaltung ausdehnen, wobei auch die virtuelle Uhr und die Uhrinterrupt-Routine (UISR) in disjunkte Module aufgetrennt werden. Es gibt dann zwei UISRen auf unterschiedlichen UEen, von denen die niederpriore nur für das reale Testobjekt zuständig und identisch mit der UISR eines reinen Echtzeit-BS-Kerns ist, und die höherpriore nur für die Zeiteinträge der simulierten Umgebung bzw. Zeitmodusumschaltungen.

Durch konsequente Trennung der Datenstrukturen gelingt es, die Menge der gemeinsamen, bei int_res zu sichernden bzw. zu restaurierenden BS-Daten auf ein Minimum zu reduzieren:

- Der aktuell rechnende, bzw. die Folge der rechnenden Prozesse (FRP)
- Der Verweis auf den aktuellen Prozeßkontrollblock (plauf).

Das Prinzip der Kontextwechseloperation int_res, ist in Abb. 5.9 grob skizziert.

(2) Die Interaktion zwischen R- und W-Prozessen wurde bereits auf das unbedingt Notwendige, die Prozeßkommunikation über RW-Kommunikationsobjekte beschränkt. Dies erleichtert die Lösung von Problem (P2). Ferner laufen die Operationen Senden, Empfangen, Ablauf von Wartezeiten bzw. Gültigkeitszeiten alle im **selbstbestimmten F-Modus** ab. Dieser schützt, da er nicht gegen den Willen eines Rechnerknotens aufgehoben werden kann, die Daten vor Zeitmodusumschaltungen, also den kritischen Kontextwechseln unter (P1), und zwar auch dann, wenn der Prozessorzustand unterbrechbar ist.

Die internen Datenstrukturen eines RW-Kommunikationsobjektes (Prozeß- und Nachrichtenwarteschlangen) sind daher stets konsistent und auf dem aktuellsten Stand. Allerdings referenzieren die RW-Kommunikationsoperationen über **untergeordnete** Objekte (Dispatcher, Zeitverwaltung) auch Datenstrukturen, die infolge anderer, ihrerseits unterbrechbarer BS-Operationen inkonsistent sein können, z.B. :

(a) Ein R-Prozeß wird während einer BS-Funktion <b> durch int_res (R,W) unterbrochen. Die W-Prozesse deblockieren später über RW-Kommunikationsoperationen R-Prozesse, aber die Folge der bereiten R-Prozesse FB[R] ist wegen der unterbrochenen Funktion <b> temporär inkonsistent.

(b) Ein W-Prozeß wird im Hintergrundbetrieb während einer BS-Funktion <b> unterbrochen, um das reale Testobjekt wiederaufzunehmen (Übergang in den selbstbestimmten R-Modus).

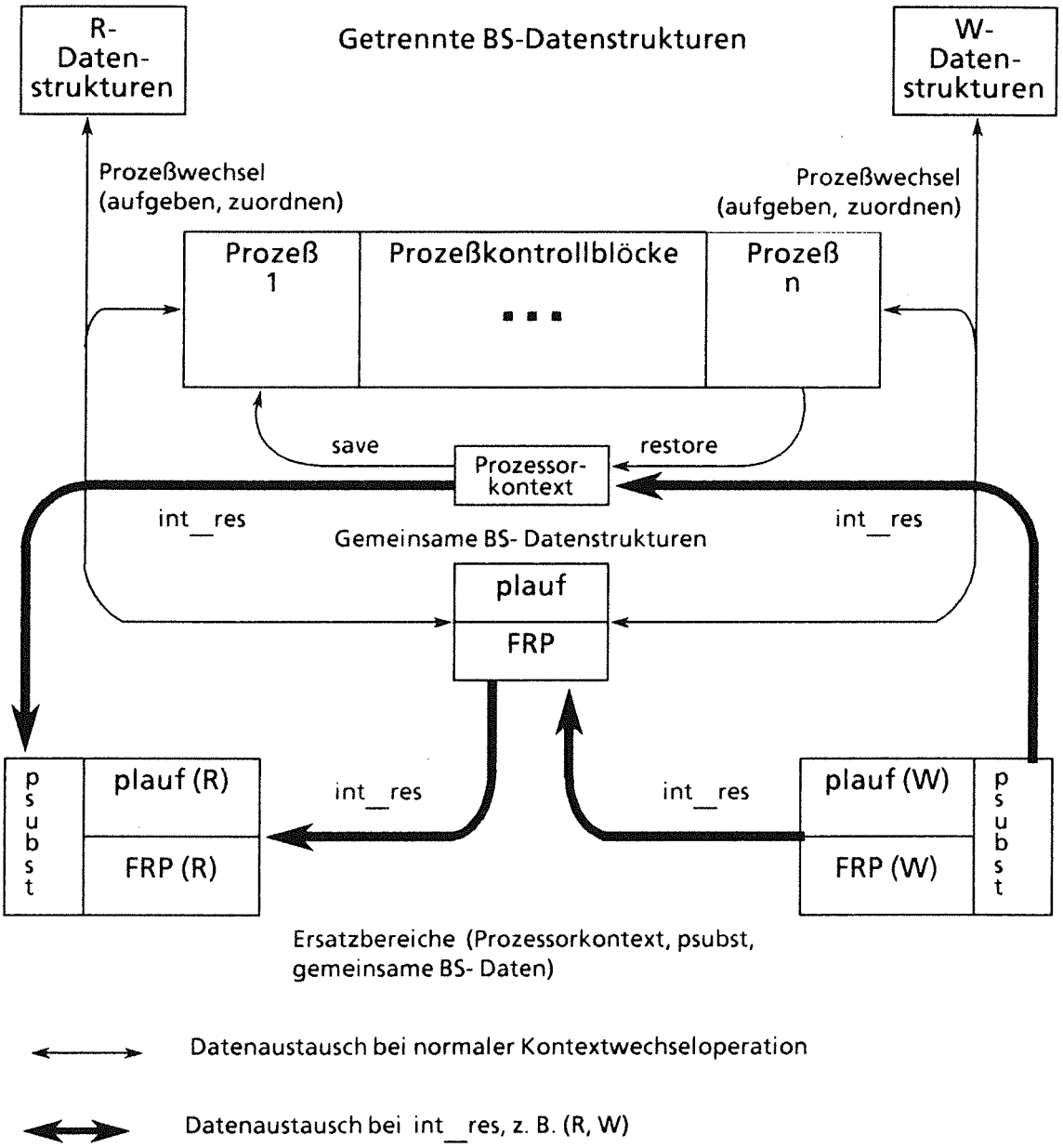



**Abb**. 5.9: Prinzip der interrupt-resume-Prozedur

Die R-Prozesse deblockieren später über RW-Kommunikationsoperationen W-Prozesse, aber FB[W] ist temporär inkonsistent.

Abb. 5.8 zeigt das Prinzip einer Lösung für (a) : die temporär undurchführbaren Operationen auf FB[R] unter send__msg werden aufgeschoben, in einer Liste o. ä. aufgesammelt und bei Wiederaufnahme der R-Prozesse (int__res (W,R)) vorgemerkt, indem ein SW-Interupt auf derselben UE wie die unterbrochene BS-Funktion <b> gesetzt wird. Die zugehörige Unterbrechungsroutine wird sofort nach Beendigung von <b> aktiv und arbeitet die vorgemerkten Operationen ab. Die Technik im Falle (b) ist ähnlich, aber - wegen der unterschiedlichen Zuteilungsregeln für R- und W- Prozesse - nicht ganz symmetrisch. Entscheidend ist in jedem Fall die korrekte zeitliche Reihenfolge und Bewertung ( im R- oder F-Modus) aller Operationen.

## 5.2 Zeitlich lose gekoppelte verteilte Systeme

Eine Klasse von VVS, die den Modellkomponenten eine größere zeitliche Autonomie und damit mehr Parallelarbeit gestatten, sind die zeitlich lose gekoppelten Verfahren (Def. 5.1). Auf sie hat sich daher das Hauptinteresse der VVS-Forschung konzentriert. Es wird zunächst ein Abriß und eine kritische Analyse zum Stand der Technik gegeben, wobei die zu Beginn von Kap. 5.1 eingeführte Notation verwendet wird.

### 5.2.1 Defensive Verfahren zur Netzwerksynchronisation

Diese Verfahren gehen von folgenden Grundannahmen aus:

(i) Jede Komponente K kann sicher sein, Nachrichten $(x_1,t_1),..,(x_n,t_n)$ über jede Eingabeverbindung (j,i) in chronologischer Reihenfolge der Zeitstempel zu erhalten. Jede solche Teilfolge repräsentiert stets den vollständigen und korrekten Eingabeverlauf von $K_j$ an $K_i$ bis zum Zeitpunkt $t_n$.

(ii) Jede Komponente ist ihrerseits dafür verantwortlich, die Bedingung (i) für alle ihre Ausgabe-Verbindungen sicherzustellen.

Eine Komponente darf also ihre Ausgabe zeitlich nur soweit fortschreiben, wie diese aufgrund der aktuellen Kenntnis der Eingabeinformation umumstößlich feststeht. Andernfalls muß auf zusätzliche Eingabe gewartet werden.

#### 5.2.1.1 Minimum-link-time-Algorithmus fur azyklische Komponenten-Topologien

Für zyklenfreie Komponenten-Topologien (z.B. Tandem-Netze oder kaskadenförmige Netze) existiert ein sehr einfaches defensives Verfahren, der minimum - link-time - Algorithmus. Aus ihm haben sich verschiedene Verfahrensvarianten entwickelt /CHM 79//MIS 86/ /KUM 86//PWM 79//PEA 80/.

Sei

(5.7) $MLT_i := \min \{T_{ji} | (j,i) \in E\}$ (**minimum link time**)

die minimale Eingabe-Verbindungszeit der Komponente $K_i$, also die Zeit, bis zu der die Eingabe für $K_i$ von allen Vorgänger-Komponenten bekannt ist. Bis zum Zeitpunkt $MLT_i$ steht folglich auch ihre Ausgabe (wegen der Kausalität) fest - aber i.a. nicht darüber hinaus. Jede Komponente $K_i$ führt iterativ folgende Schritte aus:

$K_i$ wartet in einer **Eingabephase** auf eine Zunahme von $MLT_i$, d.h. wartet disjunktiv an allen Eingabeverbindungen (j,i) mit $T_{ji} = MLT_i$ auf Nachrichten,

aktualisiert in einer **lokalen Simulationsphase** den Zustands- und Ausgabeverlauf um das Teilstück $[T_i, MLT_i]$ ($T_i$ bisherige Simulationszeit) und

sendet in einer **Ausgabephase** die neu erzeugte Ausgabe an die Nachfolgerkomponenten.
Falls im Intervall $[T_i, MLT_i]$ **keine** Ausgabe erzeugt wurde, teilt $K_i$ auch dies **explizit** den Nachfolgern durch sog. **link-time-Nachrichten** $(\varnothing, MLT_i)$ mit, sodaß diese ebenfalls ihre Eingabeverbindungszeit fortschreiben können. Die Ausgabeverbindungen mit kleinster Verbindungszeit werden immer zuerst durch normale oder durch link-time-Nachrichten aktualisiert.

Link-time-Nachrichten sind notwendig, um Verklemmungen zu vermeiden, die bei endlicher Kapazität der logischen Verbindungen sonst auftreten (vgl. Abb. 5.10). Falls z.B. im Zeitintervall [0,10] nur Ausgaben erzeugt werden, die den oberen Pfad in Abb. 5.10 nehmen, so sind schließlich alle Komponenten auf diesem Pfad (einschließlich $K_1$) **ausgabe-blockiert**, während alle Komponenten auf dem unteren Pfad **eingabe-blockiert** sind, weil nur Nachrichten von dort zu einer Erhöhung von MLT führen. Erst wenn die letzte Komponente $K_n$ ihre Eingabe von dem unteren Pfad [0,T] kennt, kann sie Nachrichten auf dem oberen Pfad mit Zeitstempel T verarbeiten. Diese Verklemmung ist eine Folge der verteilten virtuellen Zeitführung und würde in einem Abb. 5.10 topologisch äquivalenten **realen** Rechnernetz, in dem die Nachrichten-Zeitstempel **reale** Zeiten repräsentieren, nicht auftreten.
Aus Effizienzgründen wurden auch Varianten des Algorithmus ohne link-time-Nachrichten veröffentlicht (TBASIC /KUM 86/), deren Einsatz aber wegen der o.g. Deadlockgefahr problematisch ist.

**Reale Testobjekte**

Die interne Struktur der lokalen Simulation im MLT-Algorithmus ist beliebig. Insbesondere spricht nichts dagegen, die zeitgestempelten Eingabehistorien in Echtzeit zu überspielen, durch ein reales Testobjekt verarbeiten zu lassen und die in Echtzeit von $K_i$ produzierten Ausgaben wieder in zeitgestempelte Ausgabehistorien zu konvertieren. Allerdings muß immer dann, wenn die zeitliche Schranke MLT in Echtzeit erreicht wird, RTO angehalten werden. Sobald MLT inkrementiert wurde, d.h. neue Eingabe verfügbar ist, wird RTO fortgesetzt. Anhalten und Fortsetzen sind für R-Prozesse als reales Testobjekt leicht realisierbar.

**Variante**

Ein weiterer Ansatz, der auf dem MLT-Algorithmus aufbaut, ist der 'Shared Resource Algorithm for Distributed Simulation' (SRADS, /REY 82/,/REY 83/). Der logischen Verbindung (i,j) zwischen zwei Komponenten entspricht die "Shared Facility" (SF), an die bzw. von der $n \geq 2$ Komponenten senden bzw. empfangen können. Während bisher stets der Sender als aktiver Kommunikationspartner für die Fortschreibung der Verbindungszeiten (durch normale oder durch link-time-Nachrichten) verantwortlich war und der Empfänger passiv wartete, sind bei /REY 82/ beide Partner aktiv. Jeder lesende oder schreibende Zugriff einer Komponente k auf eine SF zur virtuellen Zeit $C_k$ muß solange warten, bis für die lokale Simulationszeit aller anderen an diese SF

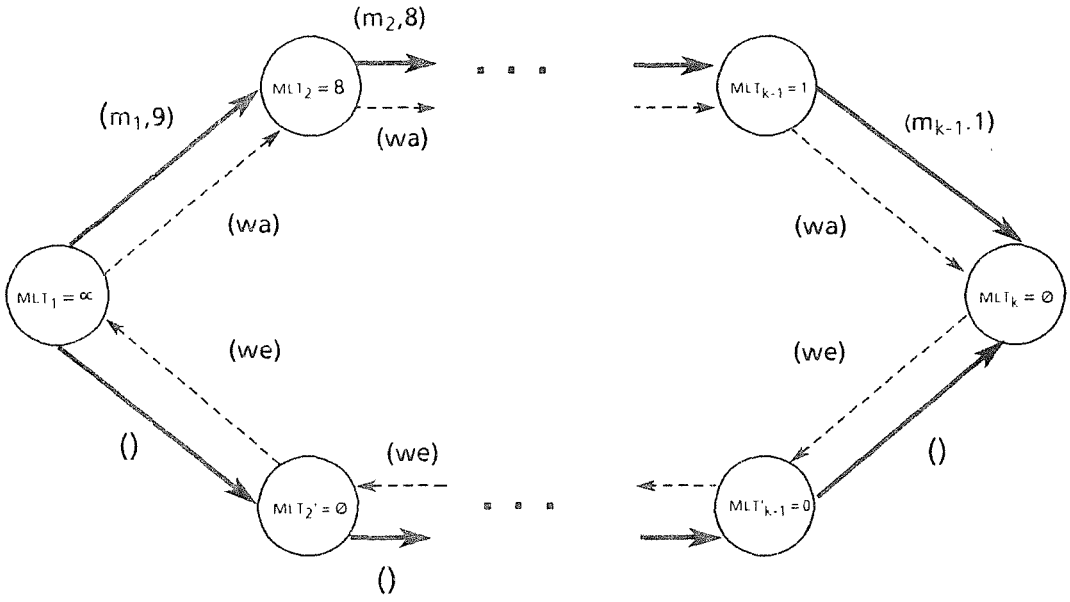



**Abb.** 5.10: Beispiel für die Verklemmung des MLT-Algorithmus bei endlicher Verbindungskapazität (nach /CHM 81/)

angeschlossenen Komponenten j (Leser oder Schreiber) gilt $C_j \geq C_k$.
Um dies sicherzustellen, werden in jedem $K_k$ zwei Dienste WAIT(i,SF,$C_i$), und SIGNAL(i,SF,$C_i$) installiert: durch WAIT zeigt eine Komponente $K_i$ an, daß sie Zugriff auf die SF zur Zeit $C_i$ wünscht; durch SIGNAL zeigt Komponente i eine neue Simulationszeit $C_i$ an, die der wartenden Komponente k den Zugriff ermöglichen kann.

#### 5.2.1.2 Verfahren für stark zusammenhängende Komponenten-Topologien

Azyklische Kommunikationstopologien kommen z.B. zwischen dem eigentlichen Testobjekt und den Komponenten des Leitstands vor; in diesem Kontext ist der MLT-Algorithmus auch sinnvoll einsetzbar (vgl. 5.2.4.2).

Das Testobjekt selbst dagegen ist ein **geschlossener Wirkungskreis** aus technischem Prozeß und Prozeßführungskomponenten. Er besitzt keine ausgezeichneten Nachrichtenquellen und -senken. Jede Komponente kann selbst aktiv werden und Nachrichten erzeugen, die potentiell jede andere Komponente beeinflussen. Die Kommunikationsstruktur des Wirkungskreises ist sogar **mehrfach stark zusammenhängend** (Abb. 5.11), enthält also viele Zyklen, z.B.

- Der Hauptregelkreis TechnischerProzeß<->PFS, der durch die Meß-, Stellsignale und Alarme gebildet wird

- Die interne Zerlegung des TP und des PFS in Einzelkomponenten und deren Informationsflüsse (z.B. Teilprozeßkopplungen und Koordinierung von PFS-Komponenten)
- Die **gegenseitige** Beeinflussung von PFS-Komponenten eines Rechners durch gemeinsame Betriebsmittel-Benutzung.

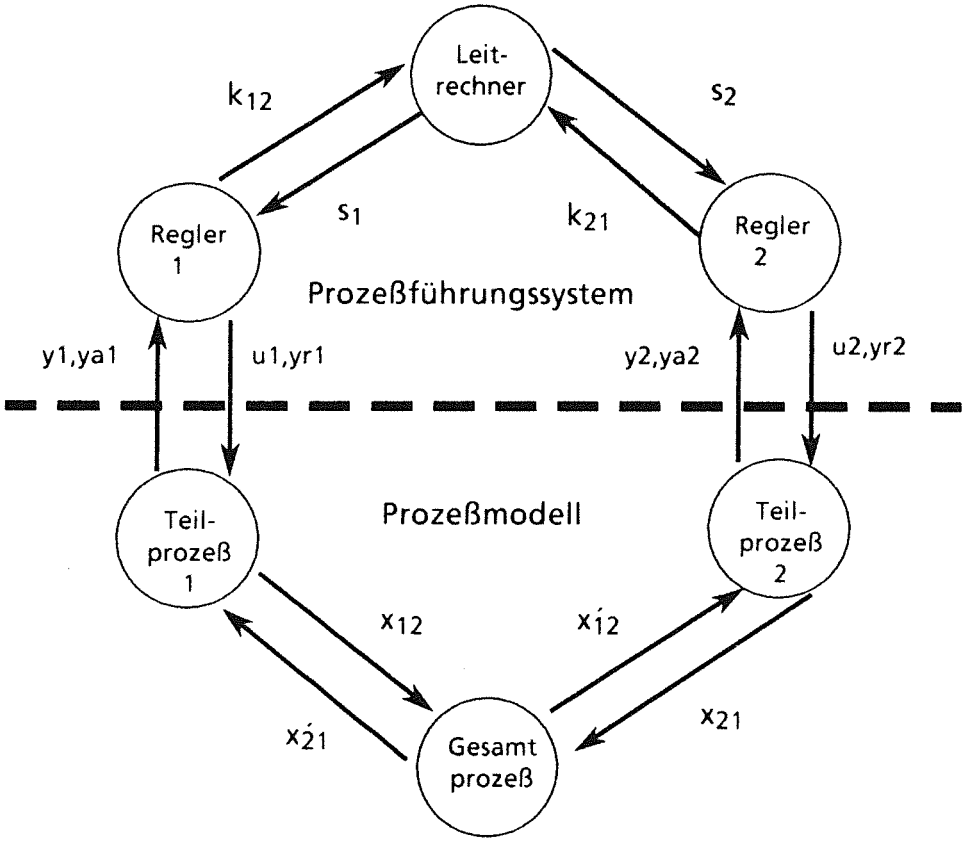



**Abb.** 5.11: Kommunikationstopologie geschlossener Wirkungskreise

Sei c ein Zyklus, der alle Komponenten eines stark zusammenhängenden Graphen in der Reihenfolge $K_1,..,K_n$ durchläuft, wobei einzelne Komponenten mehrmals auftreten können. Bei Anwendung des MLT-Algorithmus (5.2.1.1) würde dann zwangsläufig gelten

(5.8) $MLT_1(\geq T_1) \geq T_{12} \geq MLT_2 \geq ... \geq MLT_n(\geq T_n) \geq T_{n1} \geq MLT_1$

also

$MLT_1 = MLT_2 = .. = MLT_n = MLT$

und überdies sogar $T_{ij} = MLT$ für alle logischen Verbindungen (i,j). Kein Knoten könnte also jemals seine Simulationsuhr inkrementieren (kein Knoten produziert eine Ausgabe mit $t > 0$,

bevor er nicht selbst eine Eingabe mit $t>0$ erhalten hat!). In **jedem** zeitlich lose gekoppelten Verfahren **muß** es also Komponenten $K_i$ geben, für die zumindest zeitweise

(5.9) $MLT_i < T_{ik}$

für gewisse Ausgabeverbindungen (i,k) gilt. Eine solche Komponente befindet sich im **Lookahead-Modus**. Da $K_i$ ihre Eingabe im Lookahead- Intervall $[MLT_i, T_{ik}]$ nicht vollständig kennt, wird eine leere zukünftige Eingabe $\varnothing$ in $[MLT_i, T_{ik}]$ bei der Berechnung der Ausgabe zugrundegelegt. Dies wird durch folgende Definition präzisiert:

**Def. 5.2 Vorausschauende Ein-/Ausgabefunktion (Lookahead)**

Sei K ein I/O-System und

$F_T : S \times (XI,T) \rightarrow (XO,T)$ die Ein-/Ausgabefunktion für Ein-/Ausgabesegmente der Länge T.

Dann heißt

$F_{T,\delta} : S \times (XI,T) \rightarrow (XO,T+\delta)$ mit

$F_{T,\delta}(s, xi) := F_{T+\delta}(s, xi \circ \varnothing|_{[T,T+\delta]}) \; \forall s \in S, xi \in (XI,T)$

Lookahead-Funktion der Komponente K.

Entscheidend für **defensive** Verfahren ist nun, daß die durch Lookahead erzeugte Ausgabe xo im Intervall $[t,t+\delta]$ irreversibel ist und daher korrekt sein muß. Zunächst gilt ja nur die Kausalität von $K_i$. Alle defensiven Verfahren versuchen die weitergehende Korrektheit der Vorausschau **entweder** durch **lokale** Eigenschaften (zusätzliche Anforderungen an die Komponenten) zu gewährleisten, **oder** versuchen **global** (netzweit) durch Rückgriff auf "zentrale" (aber dezentral implementierte) Konzepte - nämlich Bestimmung des Zeitpunktes T' des netzweit nächsten Ereignisses - zu garantieren, daß die Eingabe für alle Komponenten im Intervall [T,T'] tatsächlich leer, d.h. die Annahme $xi|_{[T,T+\delta]} = \varnothing|_{[T,T+\delta]}$ bei der Berechnung des Lookahead in Def. 5.2 richtig ist.

### 5.2.1.3 Lokale Vorhersehbarkeit

Die folgende Systemeigenschaft ist notwendig und hinreichend dafür, daß die durch Lookahead vom Betrag $\varepsilon$ erzeugte Ausgabe einer Komponente immer korrekt ist, egal ob die Hypothese $xi|_{[T,T+\delta]} = \varnothing|_{[T,T+\delta]}$ zutrifft oder nicht:

**Def. 5.3 (Vorhersehbarkeit, predictability)**

Eine Komponente (I/O-System) heißt vorhersehbar, wenn ein $p>0$ existiert, so daß

$\forall T \in T, \forall s \in S, \forall x,x' \in (XI,T+p)$ gilt:

$x|_{[t0,T]} = x'|_{[t0,T]} \Rightarrow F_{T+p}(s,x) = F_{T+p}(s,x')$

**Bemerkungen**

- Aus der Definition folgt leicht: die Komposition von n vorhersehbaren I/O-Systemen $K_1,..,K_n$ ist vorhersehbar mit $p \geq \min(p_1,..,p_n)$;

- Die praktische Anwendung von Def. 5.3 besteht darin, die Lookahead-Funktion $F_{Tp}$, anstelle von $F_T$ in die lokale Simulationsphase des MLT-Algorithmus einzusetzen und dadurch die Simulationszeit voranzutreiben, um Bedingung (5.9) zu erfüllen. Der Algorithmus ist korrekt, wenn auf jedem gerichteten Kreis mindestens eine vorhersehbare Komponente liegt (Beweis /CHM 79/, Kap.6).

  Hervorzuheben ist, daß p in Def. 5.3 von der aktuellen Simulationszeit und dem internen Zustand s der Komponente **unabhängig** sein muß; nur dann ist garantiert, daß der Algorithmus terminiert.

Eine ähnliche Systemeigenschaft ist unter dem Namen **shortest delay** bekannt /PWM 79/, /PEA 80/ und /MUE 86/. Informell bedeutet shortest delay "den minimalen zeitlichen Abstand zwischen einem Eingabeereignis (z.B. Ankunft eines Kunden an einer Bedienstation) und einem daraus resultierenden Ausgabeereignis (Verlassen des Kunden nach Bedienung)". Der Versuch, dies systemtheoretisch zu präzisieren, führt auf folgende Definition, die für Bedienstationen als Spezialfall die intuitive Bedeutung "minimale Bedienzeit" wiedergibt.

**Def. 5.4 (Minimale Latenzzeit, shortest delay)**

Eine Komponente besitze minimale Antwortzeit $sd > 0$, wenn

$\forall\, t0, \forall\, T < t0 + sd,\ s \in S,\ e \in XI$:

$im(F_T(s, e_{[t0,T]})) \subseteq im(F_T(s, \varnothing_{[t0,T]}))$.

Dabei bedeute

$im(x) := \{x(t) \neq \varnothing \mid t \leq T\}$ für $x \in (X,T)$

die **Bildmenge** der echten Ereignisse bzw. Nachrichten im Zeitverlauf x (ohne Zeitinformation), ferner

$$e_{[t0,T]}: [t0,T] \Rightarrow XI \cup \{\omega\} \text{ eine "Sprungfunktion" mit}$$

$$e_{[t0,T]}(t) = \begin{cases} e \in XI & \text{für } t = t0; \\ \varnothing & \text{sonst} \end{cases}$$

D.h. wenn das System zum Zeitpunkt t0 in einem beliebigen Zustand mit einem einzelnen Ereignis e beaufschlagt wird, so erscheint frühestens nach sd Zeiteinheiten eine Ausgabe, die bei leerer Eingabe nicht aufgetreten wäre, also aus e "resultiert".

**Bemerkungen**

- Aus der Definition folgt unmittelbar:

  K vorhersehbar mit $p > 0 \Rightarrow$ minimale Latenzzeit sd von $K \geq p$.

  Die Umkehrung gilt nicht (!), siehe Beispiel (3) unten.
- Die minimale Latenzzeit wird in /PEA 80/ ähnlich eingesetzt wie die Vorhersehbarkeit in /CHM 79/; eine Erweiterung dieses Verfahrens wird in /MUE 86/ beschrieben. In einer empirischen Studie /FUJ 88/ wird der Quotient aus Latenzzeit und Vorhersehbarkeitszeit

('lookahead ratio') als Kriterium untersucht, wann bei einem Simulationsmodell ein spürbarer Effizienzpensum durch Verteilung zu erwarten ist.

In den folgenden Beispielen untersuchen wir die **Anwendbarkeit** dieser Konzepte zur Leistungsbewertung von Echtzeitsystemen.

**Beispiel** (1)
Eine Komponente ohne Vorgänger ist trivialerweise vorhersehbar mit $p=\infty$ und hat damit auch minimale Antwortzeit $sd=\infty$.

**Beispiel** (2)
Für eine Bedienstation mit beliebiger, nichtpräemptiver Bedienstrategie und einer minimalen Bedienzeit $\delta>0$ gilt: $p=\delta$ und $sd=\delta$. Denn für jeden Zeitpunkt T und Zustand s ist ihre Ausgabe im Intervall $[T,T+\delta]$ entweder leer oder enthält genau einen bearbeiteten Kunden zur Zeit $T+t$, falls im Zustand s ein Kunde mit Restbearbeitunszeit t bereits bedient wird. Dies gilt unabhängig von der Ankunft neuer Kunden in $[T,T+\delta]$; diese können frühestens zum Zeitpunkt $T+t$ bedient werden und nicht vor $T+\delta$ die Station verlassen.

Ist die Bedienzeit eine Zufallszahl mit kontinuierlicher Verteilungsfunktion (z.B. exponentialverteilt mit Erwartungswert $\lambda$) so treten für jedes $\delta>0$ Bedienzeiten $\leq\delta$ mit einer Wahrscheinlichkeit $1-\exp(-\lambda\delta)>0$ auf. Strenggenommen gilt also: $p=0$, $sd=0$.

**Beispiel** (3)
Wir betrachten nun
(a) eine Bedienstation mit minimaler Bearbeitungszeit $\delta$, aber präemptiver prioritätsgesteuerter Bedienstrategie.

(b) eine Bedienstation mit minimaler Bearbeitungszeit $\delta$, die eine Timeout- Reaktion veranlaßt, falls mindestens $\Delta$ Zeiteinheiten im Zustand 'frei' keine Aufträge eintreffen.

Für beide Komponenten gilt: $sd=\delta$, aber $p=0$. $sd=\delta$ folgt aus der Def. 5.4: ein bei T eintreffender Kunde (a) kann trotz präemptiver Bedienung frühestens bei $T+\delta$ die Station verlassen, und die Bedienung verdrängter Kunden kann sich allenfalls weiter verzögern. Auch in (b) kann ein bei T eintreffender Kunde keine **zusätzliche** Ausgabe vor $T+\delta$ bewirken, er kann allenfalls eine vorausgeplante Timeout-Ausgabe **verhindern**.
Dennoch sind beide Systeme nicht vorhersehbar mit einem $\delta>0$. Wie klein auch $\delta$: es ist eine Situation zur Zeit T konstruierbar, in der ein bedienter Kunde zur Zeit $T_1$'vor $T+\delta$ die Station verlassen wird, **falls kein** höherpriorer Kunde bis dahin eintrifft, aber **kein** Kunde die Station verläßt, **falls** ein höherpriorer Kunde K2 zuvorkommt (vgl. Abb. 5.12).
Eine sinngemäße Überlegung gilt für (b): eine Vorausschau ist im Zustand 'frei' niemals über den nächsten Timeout hinaus möglich, so nahe dieser Zeitpunkt auch liegen mag.
Auch wenn ein System minimale Latenzzeit sd besitzt, ist also die Vorhersage seines Ausgabeverlaufes über sd Zeiteinheiten i.a inkorrekt.
Ein weiterer Grund, dem Konzept der minimalen Latenzzeit mit Vorsicht zu begegnen, ist, daß es die Kompositionseigenschaft (vgl. Def. 2.4) nicht erfüllt. So hat z.B. die Serienschaltung einer präemptiven Bedienstation (a) und einer Timeout-Bedienstation (b), also zweier Systeme mit $sd>0$, als Gesamtsystem minimale Latenzzeit $sd=0$.

**Beispiel** (4)
Abschließend ein Beispiel eines einfachen physikalischen Prozesses, das zwar keine praktische Bedeutung besitzt, aber als symptomatisch für viele andere kontinuierlich-diskrete Prozeßmodelle (Kap 6.2.1) gelten kann. Zu modellieren sei die Bewegung von (Billard-)Kugeln innerhalb eines rechteckigen Bereiches $0\leq x\leq a$, $0\leq y\leq b$ (Abb. 5.13) unter dem Einfluß von
(a) Zusammenstößen (vollelastisch, reibungsfrei)
(b) Reflektion am Rande des Bereiches.

Die Startgeschwindigkeit jeder Kugel liege im Bereich $0\leq|v|\leq v\max$. Zur Vereinfachung der Rechnungen nehmen wir an

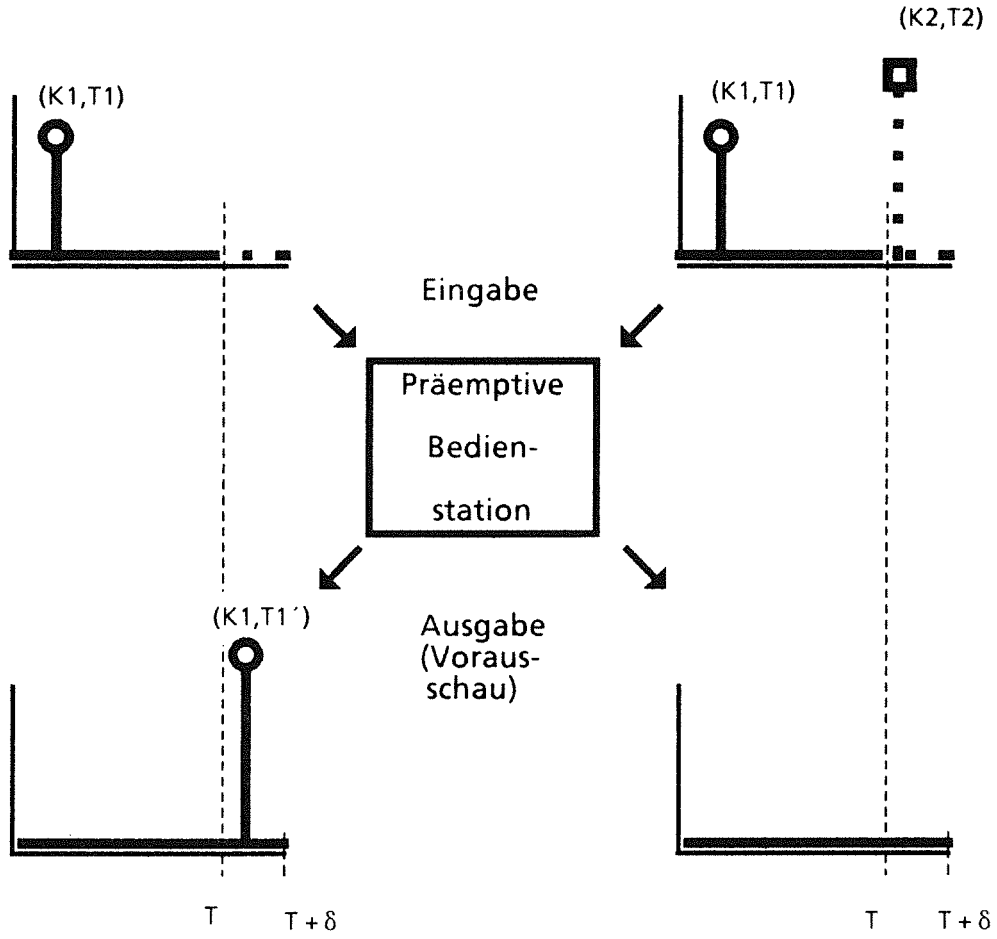



**Abb.** 5.12: Zur Vorhersehbarkeit präemptiver Bedienstationen

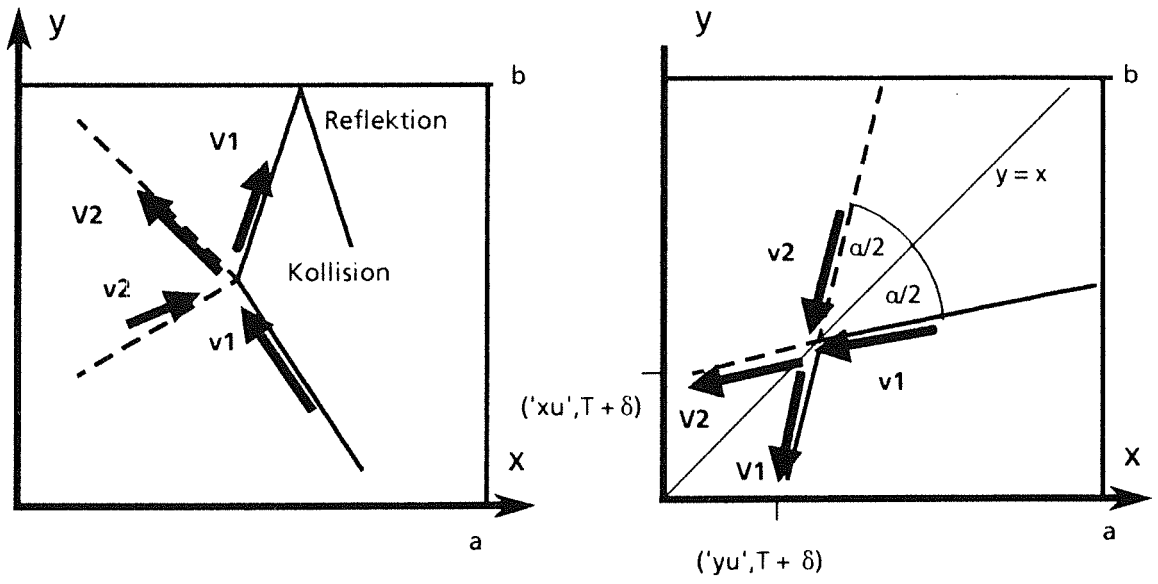



**Abb.** 5.13: Beispiel zur minimalen Latenzzeit (elastischer Stoß)

- die Massen der Kugeln sind gleich
- die Radien sind vernachlässigbar klein
- die Zusammenstöße erfolgen symmetrisch (die Stoßtangente ist Winkelhalbierende des Auftreffwinkels α der beiden Kugeln vor dem Stoß).

Seien **v1**, **v2** die Geschwindigkeiten der beiden Kugeln vor dem Stoß. Bei geeigneter Festlegung des Koordinatensystems gilt:

$\mathbf{v1} = |\mathbf{v1}| * |0, 1|^T$

$\mathbf{v2} = |\mathbf{v2}| * |-\sin\alpha, \cos\alpha|^T$.

Durch Anwendung des Energieerhaltungssatzes, des Impulssatzes sowie arithmetischer Umformungen erhält man für die Geschwindigkeiten **V1**,**V2** nach dem Stoß:

(5.10)

$\mathbf{V1} = (-\sin\alpha * (|\mathbf{v1}| + |\mathbf{v2}|)/2,\ |\mathbf{v1}|\cos^2(\alpha/2) - |\mathbf{v2}|\sin^2(\alpha/2))^T$

$\mathbf{V2} = (\sin\alpha * (|\mathbf{v1}| - |\mathbf{v2}|)/2,\ |\mathbf{v1}|\sin^2(\alpha/2) + |\mathbf{v2}|\cos^2(\alpha/2))^T$

Wegen

(5.11)

$|\mathbf{v1}|^2 = |\mathbf{v2}|^2\sin^2(\alpha/2) + |\mathbf{v1}|^2\cos^2(\alpha/2)$

$|\mathbf{v2}|^2 = |\mathbf{v1}|^2\sin^2(\alpha/2) + \mathbf{v2}|^2\cos^2(\alpha/2)$

und $|\mathbf{v1}|, |\mathbf{v2}| \leq \text{vmax}$ gilt auch $|\mathbf{V1}|, |\mathbf{V2}| \leq \text{vmax}$ nach jedem Stoß.

Die relevanten Eingabe- und Ausgabeereignisse (-meldungen) bei der Modellierung jeder Kugel sind durch folgende Komponenten beschreibbar:

(i, p, v, e)

mit

i Identifikation der Kugel

p = (x,y) neue Positionskoordinaten

v = (vx,vy) neuer Geschwindigkeitsvektor

e letztes Ereignis:

| | |
|---|---|
| xu | Reflektion am linken Rand |
| xo | Reflektion am rechten Rand |
| yu | Reflektion am unteren Rand |
| yo | Reflektion am oberen Rand |
| (koll,j) | Kollision mit Kugel Nr. j |

Jede Kugel besitzt minimale Latenzzeit sd = 0:

Dazu betrachte man eine Kugel K1 (Abb. 5.13), deren Geschwindigkeitsvektor **v1** = (vx,vy) mit der Geraden y = x den Winkel α/2 bilde. Eine Kugel K2 kollidiere mit K1 im Winkel α auf der Geraden y = x. Falls gleiche Geschwindigkeiten $|\mathbf{v1}| = |\mathbf{v2}| = v$ vor dem Stoß als Spezialfall angenommen werden, gilt wegen (5.11) auch $|\mathbf{V1}| = |\mathbf{V2}| = v$ nach dem Stoß, sowie
**V1** = **v2**, **V2** = **v1** (wegen (5.10) und cosα = cos**2(α/2)-sin**2(α/2),
d.h. K1 und K2 tauschen ihre Geschwindigkeiten aus.

Für jedes feste $\delta$ reagiert eine Kugel in Position (x,x) auf ein Eingabeereignis
(j, **p** = (x,x), **v2**, (koll,j)) zur Zeit T (Kollision) mit einem Ausgabeereignis
(i, (h,0), **v2**, 'yu') zu einer Zeit $\leq T+\delta$,
im ungestörten Fall dagegen mit
(i, (0,h), **v1**, 'xu') zu einer Zeit $\leq T+\delta$,
sofern nur (x,x) genügend nahe am Nullpunkt gewählt wird.

Aus sd = 0 folgt insbesondere auch p = 0.

**Zusammenfassung**

(1) Die Vorhersehbarkeitseigenschaft (Def. 5.3) ist gerade für typische Modellkomponenten von Echtzeitsystemen (präemptive Bedienstationen, Zeitüberwachungs-Instanzen, gemischt kontinuierlich-diskrete technische Prozesse) **nicht** erfüllt. Dasselbe gilt für die minimale Latenzzeit. Letztere ist auf allgemeine I/O-Systeme nicht sinnvoll übertragbar und leistet nicht immer das Erwartete, den zeitlichen Verlauf der Ausgabe eines Systems tatsächlich vorhersagen zu können.

(2) Die Anwendung dieser Konzepte erfordert die "Spezifikation desselben Systems auf zwei Arten" durch den Benutzer: einerseits die Modellspezifikation (Reaktion des Systems auf **konkrete** externe Eingabeverläufe und interne Zustände), andererseits die sehr viel schwierigere Bestimmung von Vorhersage-Intervallen unter allen **erdenklichen** externen Eingabeverläufen und Zuständen. Falls kein zustandsunabhängiges Intervall existiert, sondern die Vorhersageintervalle von Fall zu Fall gewählt werden müssen, besteht die Gefahr, daß die Intervallängen gegen 0 konvergieren, und die Simulationsläufe nicht terminieren.
Da keine automatischen Verfahren zur Generierung der Vorhersageintervalle bekannt sind, entsteht ein beträchtlicher Analyse-Aufwand für den Modellierer, der sich durch sämtliche Modellanpassungs- und -verfeinerungsschritte hindurchzieht, und ein erhebliches Fehlerrisiko.

D.h. lokale Vorhersehbarkeitseigenschaften widersprechen der Allgemeingültigkeit der verteilten Simulation und ihrer Transparenz für den Programmierer. Daher wird auf ihre Ausnutzung i.f. verzichtet.

### 5.2.1.4 Global kleinste Ereigniszeit

Die meisten defensiven Verfahren zur NWS verlassen sich nicht allein auf lokale Vorhersehbarkeitseigenschaften. Zumindest ab und zu wird in der Simulation konkret ermittelt, wann das netzweit nächste Ereignis stattfindet. Für diesen Rückgriff auf zeitlich eng gekoppelte Simulationstechniken gibt es zwei Gründe

(1) **Verklemmungsauflösung**

In manchen Ansätzen wird auf Vorausschau und sogar auf link-time-Nachrichten ganz verzichtet, z.B. /CHM 81/ /GRT 86/ /PRA 88/. Verklemmungen kommen daher bereits in

azyklischen Netzen der Bauart von Abb. 5.10 und erst recht in Netzen mit Zyklen vor (Abb. 5.11). Der Ablauf einer Simulation besteht - stark vereinfacht - aus drei periodisch wiederkehrenden Phasen

a) Anwendung des MLT-Algorithmus 5.2.1.1 ohne Lookahead und link-time-Nachrichten
b) Auftreten einer Verklemmung, die nach einem Algorithmus von Dijkstra u. Scholten erkannt wird
c) Inkrementieren der Simulationsuhren aller Komponenten zum netzweit nächsten Ereignis, wodurch die Verklemmung aufgelöst wird.

(2) **Effizienzsteigerung ("Beschleunigungsalgorithmen")**

Das Argument Effizienzsteigerung klingt insofern überraschend, als zeitlich lose gekoppelte Verfahren gerade zu größerer zeitlicher Autonomie und damit Effizienz als eng gekoppelte führen sollen.

Bei der Simulation von Bedienstationen (Beispiel (2) in 5.2.1.3 entspricht jedoch das Vorhersageintervall immer der kleinstmöglichen Bedienzeit $\delta$, auch wenn über lange Zeitspannen gar keine Aufträge bearbeitet werden. Infolge der Zeitfortschaltung in $\delta$-Inkrementen unabhängig von der realen Zwischenankunftszeit der Aufträge ('pseudo time driven behaviour' /PEA 80/*) wächst die Laufzeit - anders als bei zentralen discrete-event-Simulationen - proportional zur absoluten Simulationsdauer TS, statt zur Anzahl der Ereignisse und ihrer Berechnungskomplexität. Die "Time Acceleration"-Verfahren in /PEA 80/,/MUE 86/ vermeiden diesen unerwünschten Effekt, indem sie in regelmäßigen Abständen netzweit den Zeitpunkt des nächsten Ereignisses, z.B. Ankunft des nächsten Auftrags an einer Bedienstation, bestimmen. Diese Beschleunigungsalgorithmen sind dem MLT-Algorithmus 5.2.1.1 mit lokaler Vorhersage überlagert, die Modellkomponenten werden also nicht blockiert.

Wie bereits mehrfach erwähnt, ist die Bestimmung eines zukünftigen Ereignis-Zeitpunktes auf **reale Testobjekte** nicht anwendbar, bzw. man würde zwangsläufig wieder auf das Verfahren von 5.1.1 zurückkommen.

### 5.2.2 Spekulative Verfahren zur Netzwerksynchronisation

#### 5.2.2.1 Grundansatz

Das unter dem Namen Time Warp (TW) bekannte, von D.Jefferson entwickelte Verfahren /JEF 83//JES 83/ weicht in seinen Grundideen von den defensiven Verfahren radikal ab (vgl. Beginn von 5.2.1).

(i) Eine Komponente kann **nicht** damit rechnen, daß ihre Eingabehistorien, soweit sie vorliegen, vollständig und korrekt sind. Insbesondere muß z.B. mit Nachrichten in nicht-chronologischer

*) Das Problem tritt bei der Simulation von Echtzeitsystemen wegen der zeitlichen Hierarchie der Systemebenen (Bedienzeiten) verschärft auf:
- Interrupt-Ebene (direkte Alarmreaktion) : ca. 100µsec-1ms
- Regelungs-bzw. Steuerungsebene : ca 10ms-1sec
- Leit-Ebene : sec bis Min.

Wenn mehrere Ebenen in einem Modell repräsentiert sind, bildet natürlich die Ebene mit den kleinsten Zeitkonstanten (z.B. Interrupt- Ebene) einen Engpaß.

Reihenfolge ("Nachzüglern") gerechnet werden. Liegt ein solcher Fall vor, müssen fälschlich erfolgte Berechnungen **zurückgesetzt** werden.

(ii) Eine Komponente braucht ihrerseits auch nicht zu warten, bis sie einen vollständigen, korrekten Ausgabeverlauf erzeugen kann. Jede verfügbare Eingabe wird sofort verarbeitet - aber unter Vorbehalt.

Time Warp setzt darauf, daß irreguläre Ereignisse in (i) relativ selten auftreten und ein Nettogewinn an nützlicher Arbeit verbleibt (zeitliches Lokalitätsprinzip). Angenommen die Eingabehistorie $x \in (XI,T)$ einer Komponente K sei bis zum Zeitpunkt T verarbeitet, und nun tritt folgender Fall ein:

(C1) eine Nachricht mit Zeitstempel $T' < T$ trifft ein und wird nachträglich in die Historie aufgenommen

(C2) eine in x enthaltene Nachricht mit Zeitstempel $T' < T$ erweist sich als falsch und wird aus der Historie entfernt.

Ein solcher Fehler in der Eingabehistorie zum Zeitpunkt $T' < T$ hat folgende Auswirkungen:

(1) lokal: der von K berechnete Zustand $s(t) = \delta(s0, x|_{[t0,t]})$ ist für $t \geq T'$ i.a. falsch

(2) global: die von K produzierte und an andere Komponenten weitergegebene Ausgabe $F_t(s0, x|_{[t0,t]})$ ist für $t \geq T'$ i.a. falsch.

Der Kern des TW-Algorithmus besteht in einem Mechanismus, genau diese Fehler zu korrigieren.

(F1) **Lokale Fehlerbeseitigung (Rollback)**

Bei der Simulation jeder Komponente werden regelmäßig zeitgestempelte Sicherheitskopien des Zustands s angefertigt. Im Falle einer Eingabekorrektur zur Zeit T', die zu einer korrigierten Eingabehistorie $x' \in (XI,T)$ führt, wird der jüngste Zustand (s,T") mit $T" < T'$ restauriert (**Rollback**) und die Simulation mit $x'|_{[T",T]}$ als neuer Eingabehistorie wiederholt (**Restaurationsphase**).

(F2) **Globale Fehlerbeseitigung (Anti-Nachrichten)**

Jede unter x erzeugte, aber bei der korrigierten Eingabehistorie x' nicht mehr gültige Nachricht (m,t) mit $t > T'$ wird in ihrer Wirkung "ausradiert", indem ihr eine "Anti-Nachricht" (-m,t) nachgesandt wird (Fall (C2) für den Empfänger). Technisch wird dies durch zusätzliche Kontrollinformation (Vorzeichenbit) erreicht: eine Nachricht ist Anti-Nachricht einer anderen, wenn die beiden Nachrichten entgegengesetztes Vorzeichen besitzen und in allen anderen Attributen (Sender, Empfänger, Zeitstempel, Nutzdaten) übereinstimmen.

Der Vorgang der globalen Fehlerbeseitigung durch Antinachrichten pflanzt sich i.a. dominoartig durch das Netz fort und kann auf die initiierende Komponente während ihres Rücksetzens zurückschlagen ('secondary rollback'). Wenn eine **kausale** Wirkungskette solcher Rücksetz-

maßnahmen besteht, kann sie sich allerdings nicht in die Vergangenheit fortpflanzen, weil jede einzelne Komponente zeitlich kausal ist. Es gibt auch Optimierungsversuche von Time Warp /MWM 88/, um die Flut der Antinachrichten auf diejenigen Knoten zu begrenzen, die die ursprüngliche (falsche) Nachricht zum Zeitpunkt des Bekanntwerdens der Antinachricht bereits verarbeitet haben können ("Einflußsphäre").
Der Anlaß für das Zurücksetzen ist immer (C1) (verspätete Nachricht) oder C2 (inkorrekte Nachricht), und die Reaktion ebenfalls immer dieselbe, nämlich Durchführung der Schritte (F1) und (F2), egal ob die **Ursachen** für (C1) oder (C2) nur Geschwindigkeitsunterschiede der Komponenten oder ein primärer oder sekundärer Rollback einer Vorgängerkomponente sind.

Für eine korrekte Operation von TW werden zusätzlich zu (F1),(F2) zwei weitere Elemente benötigt.
Da TW viele Versionen derselben Datenstrukturen (Systemzustände, Ein-/Ausgabenachrichten) zu verschiedenen Zeitpunkten speichert, besteht Verklemmungsgefahr aufgrund Speichermangels. Daher ist eine leistungsfähige **dynamische Speicherverwaltung** besonders wichtig. Erwähnenswert ist ein Flußkontrollmechanismus auf Anwendungsebene, mit dessen Hilfe zu weit in der virtuellen Zukunft liegende, nicht mehr speicherbare Informationen an ihren Absender zurückgesendet werden, der daraufhin zeitlich zurücksetzen und wiederholen muß (diese Informationen dürfen ja nicht "vergessen" werden). Dadurch kommt es zu einer zeitlichen Angleichung der Komponenten /GAF 88/.
Es muß bekannt sein, wie weit in die Vergangenheit jede Komponente schlimmstenfalls noch zurückgesetzt werden kann. Andernfalls könnte niemals Speicher freigegeben und irreversible Ausgaben an Peripheriegeräte niemals durchgeführt werden. Dieser Zeitpunkt heißt **globale virtuelle Zeit** (GVT) und errechnet sich als das Minimum der Simulationszeiten aller Komponenten und der Zeitstempel aller umlaufender Nachrichten. GVT muß parallel zur normalen Simulation in regelmäßigen Zeitabständen bestimmt werden (broadcast-Verfahren).

#### 5.2.2.2 Bewertung, Schlußfolgerungen

**(AN1) Korrektheit,Allgemeingültigkeit** (vgl. Kriterien in 3.4.4.2)
Für den Einsatz in stark zusammenhängenden Komponenten-Topologien ist zu beachten, daß normalerweise unter TW für jede Komponente gilt

$$T_i \leq TI := \max\{T_{ji} | (j,i) \in E\}$$

(Eine Komponente blockiert dann, wenn ihre Eingabe-Warteschlange leer ist).
Da aber andererseits

$$T_i \geq T_{ik}$$

für alle $(i,k) \in E$ gilt ($K_i$ erzeugt Ausgabenachrichten mit Zeitstempeln T in der Gegenwart, d.h. mit $T = T_i$, vgl. Def. 5.1), muß wegen der Lookahead-Bedingung (5.9)

$T_i > TI$

zugelassen werden, um Verklemmungen in stark zshgd. Netzen auszuschließen. D.h. eine Komponente wird **nicht** blockiert, wenn ihre Eingabe-WS abgearbeitet ist, sondern die Simulation wird mit **leerer** Eingabehistorie weitergeführt (Anwendung der Lookahead-Funktion $F_{TI,\varepsilon}$ für ein gewisses $\varepsilon$).

Mit dieser kleinen technischen Modifikation ist der TW-Algorithmus korrekt für beliebige gekoppelte I/O-Systeme mit zeitdiskreter Ein-/Ausgabe, also allgemeingültig.

**(AN2) Integration realer Testobjekte**

Obwohl von seinen Erfindern nicht vorgesehen, ist das TW-Konzept prinzipiell auf einen R-Prozeß mit echtzeit-synchroner virtueller Uhr ebenso anwendbar wie auf einen W-Prozeß mit modellgesteuerter virtueller Uhr. Wenn eine für R-Prozeß p bestimmte Eingabe (m,T) im Experiment verspätet eintrifft, also tv(p) größer als die virtuelle Empfangszeit T ist oder (m,T) sich nachträglich als inkorrekt herausstellt, wird p auf einen Zustand (s'(p), tv'(p)≤T) zurückgesetzt und der Ablauf von p mit der korrigierten Eingabehistorie wiederholt. Im Unterschied etwa zum Zurücksetzen von Transaktionen in Datenbanksystemen oder von Recovery-Blöcken ist aber mit dem Prozeßzustand stets auch die virtuelle Uhr des Knotenrechners zurückzusetzen. Die Zuordnung Prozeßzustand <-> Zeit entspricht also stets dem Ablauf des realen Testobjektes in realer Umgebung.

**(AN3) Transparenz**

Die SW-Realisierung der Netzwerksynchronisation für TW, insbesondere das Sichern und Restaurieren (Zurücksetzen) der Komponentenzustände ist für die Simulationsanwendung vollständig transparent, ähnlich wie die Kontextwechsel eines Betriebssystems für die Anwenderprozesse. Die Modellprogrammierung (oder -generierung) erfolgt auf dieselbe Art wie für eine zentrale Simulation.

Diesen wichtigen Vorteilen stehen einige praktische Probleme gegenüber. Trotz seiner konzeptuellen Klarheit, Symmetrie und Eleganz ist TW ein sehr komplexer Algorithmus, sein Implementierungs- und Portierungsaufwand weit höher als für alle defensiven Verfahren. Die bisherigen Implementierungsvorhaben für TW /JBH 85//CLU/ /JEF 87/ haben gezeigt, daß TW eine **dedizierte** BS-Entwicklung verlangt, insbesondere wegen der Speicherverwaltung, was der Erweiterung konventioneller Echtzeit-BS-Kerne in gewisser Weise widerspricht.

Nachteilig ist v.a. der extrem hohe dynamische Speicherbedarf:

a) Es müssen **alle** dynamischen Daten einer Modellkomponente (Keller- und Datensegmente) in sq gespeichert werden. TW kann nicht erkennen, welche Daten tatsächlich modifiziert wurden. Würde man dem Programmierer die Verantwortung für eine **selektive** Zustandssicherung und -restaurierung auferlegen, ginge die Transparenz (AN3) verloren.

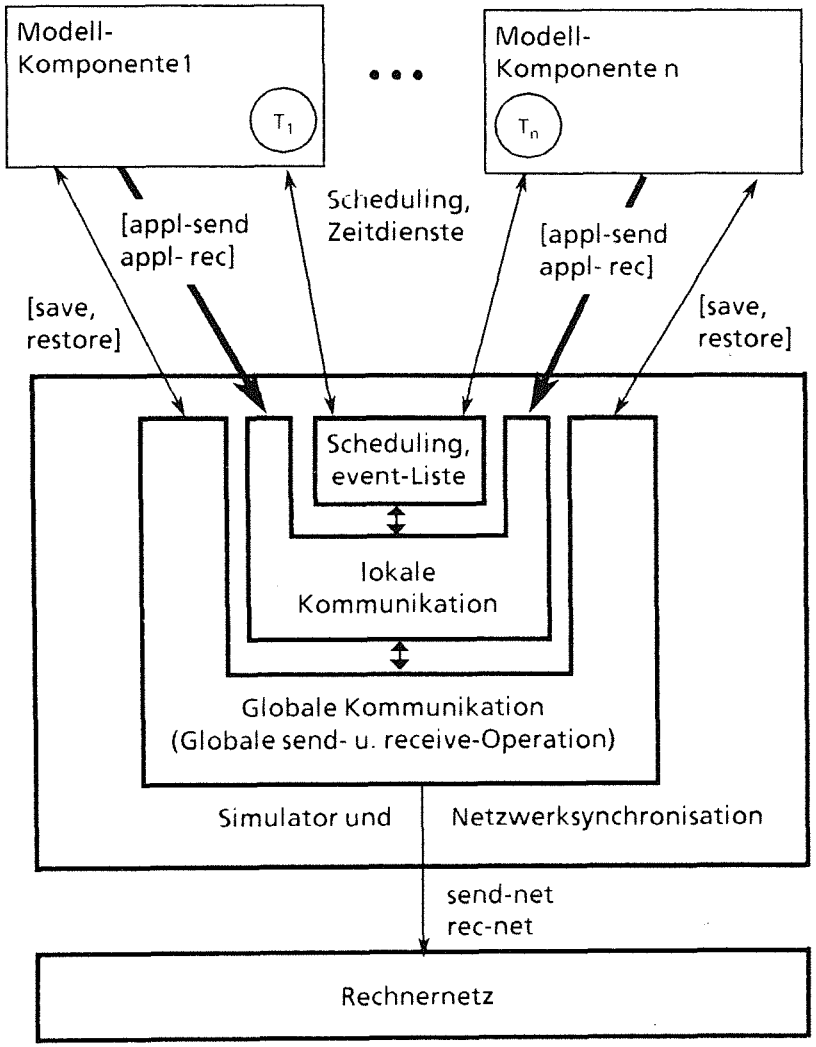
b) Der Speicherbedarf muß durch **realen**, nicht durch virtuellen Speicher gedeckt werden, sonst geht jeder Effizienzgewinn verloren.
c) Im Implementierungsstadium eines PFS besteht kaum ein Unterschied zwischen Simulationsmodell und **realer PFS-Anwendung** im dynamischen Speicherbedarf.

Sobald Echtzeitanwendungen von realistischer Größenordnung untersucht werden, ist die tatsächlich erreichbare zeitliche Autonomie der Komponenten dann nur gering, zumal TW die Sicherungskopien der dynamischen Daten in festen (virtuellen) Zeitabständen anlegt und über kein Konzept zur Wahl der Sicherungspunkte verfügt. Nur die Abstände größer zu wählen hilft wenig, da sich hierduch gleichzeitig die Kosten des Zurücksetzens erhöhen. Ein Großteil der aufwendigen Rücksetz-Operationen dient dann allein der Flußkontrolle wegen Speicherknappheit.
Aber selbst bei unbegrenztem Speicher bedeutet die potentiell unbegrenzte Vorausschau einzelner Komponenten nicht automatisch auch eine hohe globale Effizienz (einen steilen Gradienten von GVT).

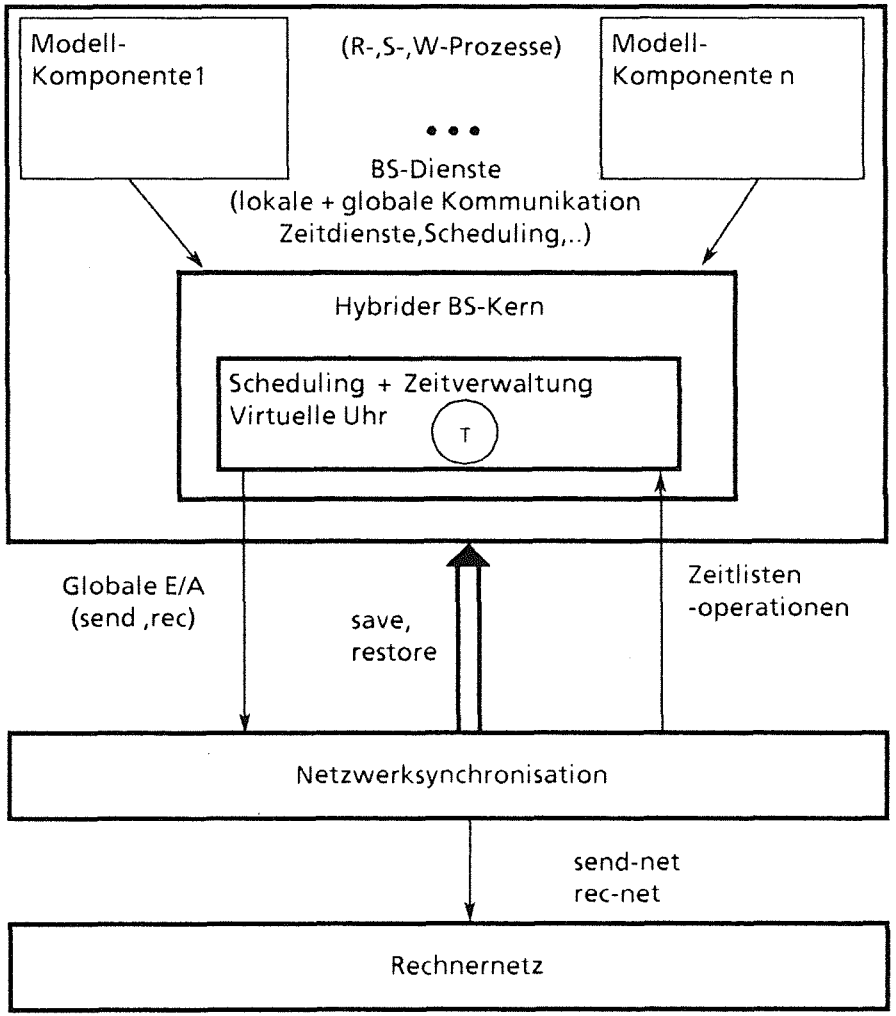
Die Zielrichtung besteht daher in der Entwicklung eines Algorithmus unter Beibehaltung des Rücksetz-Konzeptes und der Eigenschaften (AN1)-(AN3), aber mit
- durchgreifenden konzeptionellen Vereinfachungen, insbesondere Verzicht auf dynamische Elemente, um das Verfahren leichter als Zielmaschinen-Erweiterung einsetzen zu können
- dem Ziel der Minimierung von Sicherungskopien
- einem Konzept zur Steuerung bzw. Begrenzung der Vorausschau von Komponenten.

### 5.2.3 Zur Architektur verteilter Simulationssysteme

In diesem Abschnitt werden einige prinzipielle Unterschiede in der SW-Architektur zwischen dem vorliegenden Ansatz und anderen verteilten Simulationssystemen (gleich welche der Verfahren von 5.2.1 oder 5.2.2 diese anwenden) herausgestellt. Während die bekannten Systemen ausschließlich zur Simulation eingesetzt werden, ist der vorliegende Systemansatz als funktionelle Erweiterung von Echtzeit-BSen konzipiert und dient insbesondere zur Bewertung realer Testobjekte (verteilter Echtzeitsysteme).
Die Unterschiede liegen in der Schnittstelle und der Art der Kooperation zwischen Netzwerksynchronisation und Modellkomponenten. Die Architektur der bekannten verteilten Simulationssysteme läßt sich grob wie folgt charakterisieren (Abb. 5.14a):




a) Herkömmliche verteilte Simulationsmaschine




b) Zweistufige Architektur mit zentralem lokalen Simulator

**Abb.** 5.14: Architekturen verteilter Simulationssysteme

(1) Die Modellkomponenten interagieren ausschließlich über Botschaftendienste der NWS (in Abb. (5.14a) appl__send, appl__rec genannt). Abgesehen von zeitlicher Verzögerung und dem (optionalen) dynamischen Erzeugen/Vernichten von Komponenten sind das die einzigen zur Laufzeit möglichen Dienste.

(2) Die NWS ist nicht nur für die zeitliche Synchronisation der Modellkomponenten auf unterschiedlichen Rechnern zuständig, sondern auch für die korrekte Zuteilungsreihenfolge (scheduling) der Komponenten des eigenen Rechners. Ein solches "multitasking" mit mehreren Modellkomponenten pro Experimentrechner ist durchaus üblich /MUE 86/ /JES 85/ /CLU 85/.

(3) Auch Komponenten desselben Knotens werden nach **verteilter Zeitführung** (Def. 5.1) simuliert, so wie Komponenten auf unterschiedlichen Knoten. Insbesondere besitzt jede Komponente bzw. jeder Prozeß eines Knotens eine eigene Simulationsuhr.

(4) Falls die NWS spekulative Verfahren (5.2.2) einsetzt, werden immer einzelne Komponenten zurückgesetzt ("Prozeß-Rollback").

Die in dieser Arbeit vorgestellte SW-Architektur weicht hiervon in folgenden Punkten ab (Abb. 5.14b).

(1') Die Komponenten (Wirts- und Zielprozesse) interagieren über beliebige **Dienste** der **hybriden BS-Kerne**, bzw. darauf aufbauender höherer Kommunikationsschichten.

(2') Die NWS hat allein die Aufgabe, die korrekte zeitliche Synchronisation der Komponenten **eines Knotens** mit den übrigen **Knoten** bei der Interrechnerkommunikation zu leisten. Die Prozessorzuteilung der Komponenten eines Knotens bleibt Sache des BS-Kerns.

(3') Jeder hybride BS-Kern ist nach Def. 5.1 ein **zentraler** Simulator für die W-, S-, R-Prozesse seines Knotens mit zentraler virtueller Uhr.

(4') Beim Zurücksetzen wird nicht nur der direkt betroffene, z.B. auf Eingabe wartende Prozeß, sondern es werden **alle** Prozesse eines Knotens **einschließlich** des **Zustands** des **BS-Kerns** und der virtuellen Uhr zurückgesetzt ("Knoten-Rollback"). Rollback in diesem Fall hat man sich wie eine zweite Kontextwechsel-Operation vorzustellen: es wird der Aufrufkontext der korrespondierenden save-Operation restauriert.

Es handelt sich also um eine zweistufige Architektur:

- auf Knotenebene bildet der hybride Kern zusammen mit den lokalen Prozessen $K_{i1},..,K_{in}$ einen Simulator für ein gekoppeltes I/O-System $K_i := (K_{i1},..,K_{in})$
- auf Systemebene bilden die $K_i$ zusammen mit ihren NWSen $N_i$ einen (verteilten) Simulator für das Gesamtsystem, wobei jedes $K_i$ für $N_i$ **eine** zusammengesetzte Komponente mit unbekannter interner Struktur darstellt. Die NWS sieht also als Kommunikationstopologie die Interrechnerkommunikation, nicht die Interprozeßkommunikation.

Zur Begründung der Entwurfsentscheidungen (1'),(2'):

Der **Hauptvorteil** besteht darin, daß **R-Prozesse** in ihrer **realen Ablaufumgebung** belassen

und bewertet werden können. Nur so lassen sie sich sinnvoll als reales Testobjekt bewerten.

Für **W-Prozesse** bedeutet die Struktur nach Abb. 5.14b zumindest keine Nachteile, denn alle nötigen Dienste zur prozeßorientierten Simulation werden vom BS-Kern unterstützt.

Zu (3')

Die zeitlich lose Koppelung von Komponenten bringt unter den spekulativen Verfahren (TW) auf einer Einrechneranlage keine erkennbaren Effizienzvorteile, weil blockierte Modellkomponenten, anders als in Mehrrechnersystemen, nicht zu untätigem Warten ihres Prozessors führen, das durch Vorausschau ausgenutzt werden könnte. Die Vorausschau einzelner Komponenten zu Lasten anderer auf demselben Prozessor führt insgesamt zu keiner höheren Prozessorauslastung.

Zu (4')

Durch das Knoten-Rollback bleibt für den hybriden Kern die Illusion einer zentralen, monotonen Zeit gewahrt. Sie ginge beim Zurücksetzen auf Prozeßebene verloren, denn dann müßte der BS-Kern Prozesse verwalten, deren Zustände unterschiedlichen Zeitpunkten entsprechen. Mit (4') gelingt eine klarere Trennung zwischen lokalem BS-Kern und der verteilten Simulation, und der in Kap.4 entwickelte BS-Kern kann unverändert übernommen werden.

Als einen **Hauptnachteil** der Architektur würde man zunächst einen sehr viel höheren Aufwand des Knoten-Rollback erwarten. Unter der naheliegenden Annahme, daß Komponenten auf demselben Knoten häufiger interagieren als auf unterschiedlichen Knoten, trifft jedoch eher das Gegenteil zu.

**Beispiel**: Man betrachte z.B. die zeitliche Folge der Interaktionen von Prozessen P1,..P5 eines Knotens mit ihrem Prozessor bzw. Dispatcher D, dargestellt als Nachrichten. Einem simulierten Ablauf des Prozeßsystems im Intervall [100,150] entspricht aus Sicht des Dispatchers D z.B. eine Nachrichtenfolge:

```
('assign',    P5, 101);
('preempt',   P5, 104);
('assign',    P1, 105);
('block',     P1, 109);
('assign',    P2, 110);
....
('preempt',   P3, 149).
```

Trifft nun eine Nachricht mit Zeitstempel 105 verspätet ein, so wäre unmittelbar deren Empfangsprozeß, z.B. ein hochpriorer Prozeß P6, vom Zurücksetzen betroffen. Weil aber alle anderen Prozesse indirekt über den gemeinsamen Dispatcher (D) auch mit P6 verknüpft sind, würde nach und nach die gesamte Prozessorbelegung im Zeitraum [105,150] durch eine Serie von Antinachrichten

```
('un-assign', P1, 105);
('un-block',  P1, 109);
('un-assign', P2, 110);
....
('un-preempt',P3, 149);
```

"aufgerollt", wobei u.U. jeder Prozeß von mehreren Rollbacks betroffen ist. Offensichtlich ist es in solchen Fällen effizienter, die Prozessorbelegung als Ganzes auf den Stand t = 105 zurückzusetzen.

**Bemerkungen**

- Die Aufteilung der W-Prozesse auf die Experimentrechner sollte also so vorgenommen werden, daß intensiv kommunizierende Prozesse auf demselben, weniger häufig kommunizierende auf verschiedenen Rechnern liegen.
- Durch eine geringe Hardware-Unterstützung läßt sich die Effizienz des Knoten-Rollbacks noch verbessern (5.2.5.2).

### 5.2.4 Ein neues Verfahren: Ausgabebegrenzte Vorausschau (OLGA)

I.f. wird ein sehr einfacher Algorithmus für die zeitlich lose Kopplung entwickelt, der auf folgendes Anwendungsprofil zugeschnitten ist:

a) Die Modellkomponenten sind I/O-Systeme unbekannter, beliebig komplexer interner Struktur mit zeitdiskreten Ein-/Ausgabeverläufen (nicht notwendig discrete-event-Systeme). Sie können insbesondere auch reale Testobjekte (R-Prozesse) enthalten. Eine Modellkomponente besteht aus einem Experimentrechnerknoten einschließlich lokaler E/A-Peripherie und zugehöriger SW (hybrider BS-Kern, R- und W-Prozesse), vgl. 5.2.3
b) Die durch die Kommunikation der Komponenten definierte Netztopologie ist stark zusammenhängend; Ausgaben einer Komponente können sich in potentiell beliebig kurzer Zeit auf den internen Zustand oder die Ausgabe jeder anderen Komponente auswirken.

Das Verfahren (genannt OLGA) geht also von ungünstigsten Annahmen aus, die vor allem für Realzeitsysteme oft zutreffen, und erlaubt daher beliebige Simulationsanwendungen. Die Regeln der NWS bleiben für die Programmierung der Simulationsanwendung vollständig transparent (vgl. 3.4.4.2). Andererseits kann die NWS auch nicht die zeitliche Semantik spezieller Anwendungen, z.B. Warteschlangensysteme, effizienzsteigernd ausnutzen.

Ausgangspunkt von OLGA ist die Notwendigkeit von Komponenten-Vorausschau in stark zusammenhängenden Topologien nach Def. 5.2 in 5.2.1.2 bzw. Beziehung 5.9. In OLGA simuliert jede Komponente grundsätzlich soweit in die Zukunft - ab einer gewissen Zeitspanne mit leerer Eingabe - bis sie an eine Ausgabeoperation für eine andere Komponente gelangt. Dabei können zwischenzeitlich beliebig viele komponenteninterne Ereignisse stattfinden.
Lookahead unter diesen Bedingungen bedeutet per se, daß Nachrichten mit kleinerem Zeitstempel als die aktuelle Simulationszeit der Komponente eintreffen können. Die in der Vorausschau produzierten Zustände und Ausgaben sind also - zumindest was ihre Zeitpunkte betrifft - i.a. inkorrekt und müssen zurückgesetzt werden. Daher werden zeitgestempelte Sicherungskopien aller dynamischen Daten der Komponenten, d.h. der Keller- und Datensegmente der R- und W-Prozesse sowie des lokalen BS-Kerns selbst, angelegt.

Von Time Warp unterscheidet sich OLGA dadurch, daß die Auswirkungen von Präemptionen **komponenten-lokal** bleiben, daß also eine globale Fehlerkorrektur durch Anti-Nachrichten entfällt. Dies wird dadurch erreicht, daß der Nachrichtenaustausch der Komponenten direkt mit der globalen virtuellen Zeit (GVT) im Netz synchronisiert wird. GVT wird mit Hilfe eines **zirkulierenden marker** bestimmt, in den die zu einem Virtuellen Ring organisierten Komponenten ihre aktuellen Simulationszeiten eintragen. Eine sendebereite Komponente ist nur dann sendeberechtigt, wenn ihre Simulationszeit bei Empfang des Markers unterhalb allen Zeiteinträgen im Marker liegt, andernfalls wird sie blockiert. Im Vergleich zu Time Warp bedeutet dies einen weitgehenden Verzicht auf zeitliche Autonomie, andererseits wird die Anzahl der Sicherungskopien minimiert (es wird nur eine benötigt).

### 5.2.4.1 Grundregeln der Netzwerksynchronisation

Seien $N_1,..,N_n$ die zu synchronisierenden Modellkomponenten, $XI_i$ der Wertebereich aller Eingabenachrichten für $N_i$

| | |
|---|---|
| $TI := \max\{T_{ji} \mid (j,i) \in E\}$ | maximaler Zeitstempel der bisher von Komponente $N_i$ empfangenen Nachrichten |
| $xi_i \in (XI,TI)^e$ | zugehörige Eingabehistorie von $N_i$ |
| $T_i$ | lokale Simulationszeit von $N_i$ |
| $(sb,TS_i)$ | Sicherungskopie des Zustands von $N_i$, mit<br>sb : dynamische Daten<br>$TS_i$ : assoziierte Simulationszeit (**Sicherungszeit**) |

Folgende Hilfsfunktionen seien auf dem Sicherungszustand definiert:

| | |
|---|---|
| save | aktueller Zustand s von $N_i$ wird neuer |
| (**out** sb: zustand) | Sicherungszustand ($sb := s$, $TS_i := T_i$) |
| restore | Sicherungszustand (sb, $TS_i$) wird als aktueller |
| (sb: zustand) | Zustand restauriert ($s := sb$, $T_i := TS_i$) |

$N_i$ befinde sich im **Vorausschau-Modus**, falls $T_i \geq TI$, und im **Rücksetz-Modus**, falls $T_i < TI$.
Der zirkulierende Marker trägt folgende vorläufig wesentliche Information für jede Komponente $N_i$ (in 5.2.4.3 und 5.2.4.4 folgt eine vollständige Definition der Datenstruktur und der Operationen):

- $TM_i$ : Simulationszeit von $N_i$ beim letzten Besuch des Markers
- $a_i$ : Statusvariable, ob Komponente $N_i$ sendebereit ist, d.h. vor einer Ausgabe steht.

Es folgen die wichtigsten Regeln der NWS.

**(1) Lokale Simulation und Vorausschau**

Die Komponente $N_i$ simuliert ab $T_i := TI$ mit leerer Eingabe voraus, bis die nächste externe Ausgabe für eine andere Komponente erreicht wird oder die spezifizierte Simulationsdauer des Experimentes erreicht ist. Der Vorausschau-Modus wird abgebrochen, sobald eine echte Eingabenachricht aus der virtuellen Vergangenheit eintrifft (Empfangsoperation (3)).
Es wird also folgende Invariante eingehalten:

(5.12) $TS_i \leq T_i \leq TA_i := \min\{t \geq TI \mid F_{TI,t}(s0,xi) \neq \emptyset\}$

mit $xi \in (XI,TI)$. $F_{TI,t}$ ist die Lookahead-Funktion (Def 5.2) der Komponente $N_i$ mit Startzustand s0.

**(2) Sendeoperation (m,$T_i$)**

$N_i$ hat also im Vorausschau-Modus die nächste (zeitlich früheste) Ausgabe an eine andere Komponente $N_j$ erreicht, d.h. $T_i = TA_i$ in (1). Die Sendeoperation und die Simulation oder Echtzeitausführung von $N_i$ selbst werden solange blockiert, bis $N_i$ in den Besitz des Markers kommt und

(5.13) $t \leq \min\{TM_j \mid j \neq i\}$

feststellt (vgl. 5.2.4.3). Gilt (5-13) nicht, trägt $N_i$ sich lediglich als sendebereit mit aktueller Simulationszeit $TM_i := T_i$ ein und sendet den Marker an den Nachfolgerknoten im Virtuellen Ring weiter.

**(3) Empfangsoperation (m,t)**

Eine Nachricht (m,t) von einer anderen Komponente trifft ein. Normalerweise liegt ihr Zeitstempel t wegen der Bedingung (5.13) ($t \leq TM_i \leq T_i$) in der virtuellen **Vergangenheit** ihres Empfängers $N_i$. Eine Ankunft in der virtuellen Zukunft ist möglich, wenn mehrere Nachrichten mit gleichen (oder "im Rahmen der zeitlichen Ungenauigkeit gleichen", vgl. 5.2.4.4) Zeitstempeln an $N_i$ gesendet wurden, von denen eine bereits ein Zurücksetzen von $N_i$ bewirkt hat.
Eine Nachrichtenankunft in der Vergangenheit führt zur Restaurierung des Sicherungszustands:
restore (sb);
Unmittelbar danach gilt

(5.14) $TS_i = T_i \leq t \leq TI$.

Im Rücksetz-Modus werden alle empfangenen Nachrichten ($m_k$, $t_k$) zugestellt, sobald die Simulationszeit $T_i$ $t_k$ überschritten hat (Ablauf von Zeiteinträgen). Sobald $T_i = TI_i$, beginnt der Vorausschau-Modus von neuem.

**(4) Wahl der Sicherungspunkte**

Beim Anlegen der Sicherungskopien mittels save (sb) werden drei Ziele verfolgt.

(1) Es muß folgende **Konsistenzbedingung** für jede von $N_i$ noch zu erwartende Nachricht (m,t) gelten:

$t \geq TS_i$,

d.h. keine dieser Nachrichten darf zeitlich jenseits des (ältesten) Sicherungszeitpunktes liegen.

(2) Es soll nur **ein** Sicherungszustand gehalten werden.

(3) Dieser soll sich auf einem zeitlich möglichst aktuellen Zustand befinden, um den Rücksetz-Aufwand zu minimieren und externe Ein-/Ausgabenachrichten nur einmal verarbeiten zu müssen.

Die Zustandskopie wird an folgenden Stellen aktualisiert:

(a) am Ende des Rücksetz-Modus, nachdem alle empfangenen Nachrichten zugestellt wurden ($T_i = TI$)

(b) unmittelbar nach jeder Sendeoperation in (2)

(c) als Option immer dann, wenn ein Knoten im Marker-Besitz

$T_i \leq \min\{TM_j \mid j \neq i\}$ (5.13)

feststellt. Der zusätzliche Sicherungsaufwand ist gegen eine mögliche Aufwandsersparnis im Falle eines späteren Zurücksetzens abzuwägen.

**(5) Marker-Empfang und Steuerung des Marker-Transportes**

Wenn $N_j$ den Marker empfängt und selbst sendebereit ist, wird verfahren, wie unter (3) beschrieben. Andernfalls wird unterschieden:

Fall 1 Aufgrund der Information im Marker existiert bereits eine sendebereite Komponente $N_j$. Sei

(5.15) $TA\_min := \min\{TM_j \mid j \neq i \wedge a_j\}$

Das Ziel besteht darin, unter den als sendebereit bekannten Komponenten diejenige mit minimalem $TM_j$ möglichst rasch zum Zuge kommen zu lassen:

Falls $T_i \geq TA\_min$ gilt, sendet $N_i$ den Marker unverzüglich weiter.

Falls $T_i < TA\_min$, so kehrt $N_i$ in die lokale Simulation zurück und sendet den Marker erst weiter, wenn die Zeit TA_min abgelaufen oder $N_i$ selbst sendebereit ist, d.h. sobald

$T_i = \min(TA_i, TA\_min)$ (**virtueller Lookahead-Timeout**)

geworden ist. Zusätzlich muß jedoch noch eine **Echtzeitüberwachung** $\Delta R$ gesetzt werden: spätestens $\Delta R$ Zeiteinheiten seit Empfang des Markers ist dieser weiterzuleiten, auch wenn weder die Simulationszeit TA_min noch der nächste Ausgabezeitpunkt $TA_i$ erreicht wurde.

Fall 2 Keine Komponente war zum Zeitpunkt ihres letzten Eintrags in den Marker sendebereit. Ziel des Marker-Transportes in diesem Fall ist es, einen minimalen **virtuellen** Zeitfortschritt $\Delta$ pro Marker-Umlauf zu erzielen. Dieses Inkrement $\Delta$ kann wahlweise auf TA_min oder auf das eigene $TM_i$ beim letzten Marker-Empfang bezogen werden.

Sobald die lokale Simulation diese Zeit erreicht hat (Lookahead-Timeout), spätestens jedoch nach $\Delta R$ Echtzeiteinheiten, sendet $N_i$ den Marker weiter. In beiden Fällen wird die erreichte Simulationszeit $T_i$ als $TM_i$ in den Marker eingetragen.
Die Echtzeitüberwachung $\Delta R$ ist für die Terminierung (Verklemmungsfreiheit) des verteilten Simulationssystems notwendig (vgl. 5.2.4.5).
$\Delta$ und $\Delta R$ sind frei wählbare Strategieparameter des Verfahrens.

### 5.2.4.2 Eigenschaften, Erweiterungen

Obwohl die lokalen Virtuellen Uhren der Komponenten entkoppelt sind, gewährleistet die NWS die netzweit chronologische Reihenfolge der zwischen verschiedenen Komponenten ausgetauschten Nachrichten:

(5.16) $(m_i,t_i)$ gesendet vor $(m_j,t_j)$ (der Sender von $m_j$ wurde vor dem Sender von $m_j$ sendeberechtigt) $\Rightarrow t_i \leq t_j$

Vereinfacht gesagt, ist jedes $t_i$ zum Zeitpunkt des Absendens von $m_i$ das Minimum von n Komponenten $TM_j$ des Markers, die

- entweder monoton wachsen (als Simulationszeiten $T_j$ der Knoten $N_j$);
- oder, wenn sie zurückgesetzt werden, jedenfalls nicht mehr vor den Zeitpunkt der **zuletzt** gesendeten Nachricht.

Aus der zeitlichen Monotonie folgt weiter die **Konsistenz** des einfachen Zustandssicherungskonzeptes (4) in 5.2.4.1, das nur eine Sicherungskopie, und keinen dynamischen Keller von Sicherungskopien benötigt. Da die Sicherungskopie nach jeder E/A-Operation aktualisiert wird, wird auch keine dynamische Speicherverwaltung für Ein-/Ausgabenachrichten benötigt. Jede Nachricht wird nur einmal produziert bzw. verarbeitet.
Aus der Monotonie und der Konsistenz der Zustandssicherung läßt sich die **schwache Korrektheit** der NWS folgern: die unter OLGA produzierte Nachrichtenfolge $(m_i,t_i)$ $(i=0,1,2,...)$ ist identisch mit der des gekoppelten I/O-Systems $(N_1,..,N_n)$ mit zeitdiskreten Ein-/Ausgabesignalverläufen (vgl. 3.4.4.1, (3.6)).
Schließlich **terminiert** die verteilte Simulation unter OLGA, falls das Modell wohldefiniert ist, d.h. falls überhaupt eine Simulation für das gekoppelte I/O-System terminiert.

Der formale Nachweis wird hier aber nicht geführt, sondern getrennt veröffentlicht. Die chronologische Reihenfolge der Interkomponenten-Nachrichten und die Ähnlichkeit der Lookahead-Funktion (Def. 5.2) zur next-event-time- Funktion ta(s) der zeitdiskreten Simulation könnten zu dem Fehlschluß verleiten, daß die OLGA-NWS in Wirklichkeit eine zeitlich eng gekoppelte Simulation realisiert. Aber die Nachrichten $(m_i,t_i)$ sind nicht gleichzusetzen mit "Ereignissen" der zeitlich eng gekoppelten Simulation (VSE). I.a. stellt jede Komponente $N_i$ eine Verknüpfung mehrerer DEVS-Systeme $K_{ij}$, $j=1,..m(i)$ dar. Unter VSE müßten auch alle **internen**

Zustandsübergänge (Ereignisse) jedes $K_{ij}$ netzweit chronologisch stattfinden (der Zeitpunkt des nächsten Ereignisses ist stets

$$\min_{i=1}^{n} \min_{j=1}^{m(i)} ta_{ij}(s_{ij}).$$

Unter OLGA sind nur die **komponentenübergreifenden** Ausgabenachrichten zeitlich zu synchronisieren. Andererseits muß die zeitlich bereits vorausgelaufene Empfängerkomponente einer solchen Nachricht i.a. zurückgesetzt werden, auch falls diese ein R-Prozeß ist, während sich unter VSE ein Zurücksetzen erübrigt.

Wegen der Verwendung nur einer Sicherungskopie ist die zeitliche Autonomie im Vergleich zu TW relativ gering und OLGA daher i.a. nicht besonders laufzeiteffizient. Zum Teil wird dieser Nachteil dadurch ausgeglichen, daß OLGA außer dem Marker keine organisatorischen Nachrichten (Anti-Nachrichten, link-time-Nachrichten o.ä) benötigt.
In **nicht stark** zusammenhängenden Komponenten-Topologien jedoch ist die zeitliche Monotonie der Nachrichten (5.16) sicherlich unnötig restriktiv. Daher verwenden wir hier eine andere Vorgehensweise.
Wir zerlegen den Netzgraphen in seine starken Zusammenhangskomponenten (ZK)
$V_1,...,V_r$: der **reduzierte Graph** ( **V**, **E**) mit
$\mathbf{V} := \{V_1,...,V_r\}$,
$\mathbf{E} := \{(i,j) \mid \exists \text{ Knoten } N_k \in V_i, N_l \in V_j \wedge (k,l) \in E\}$
ist bekanntlich **zyklenfrei** /DOM 73/.

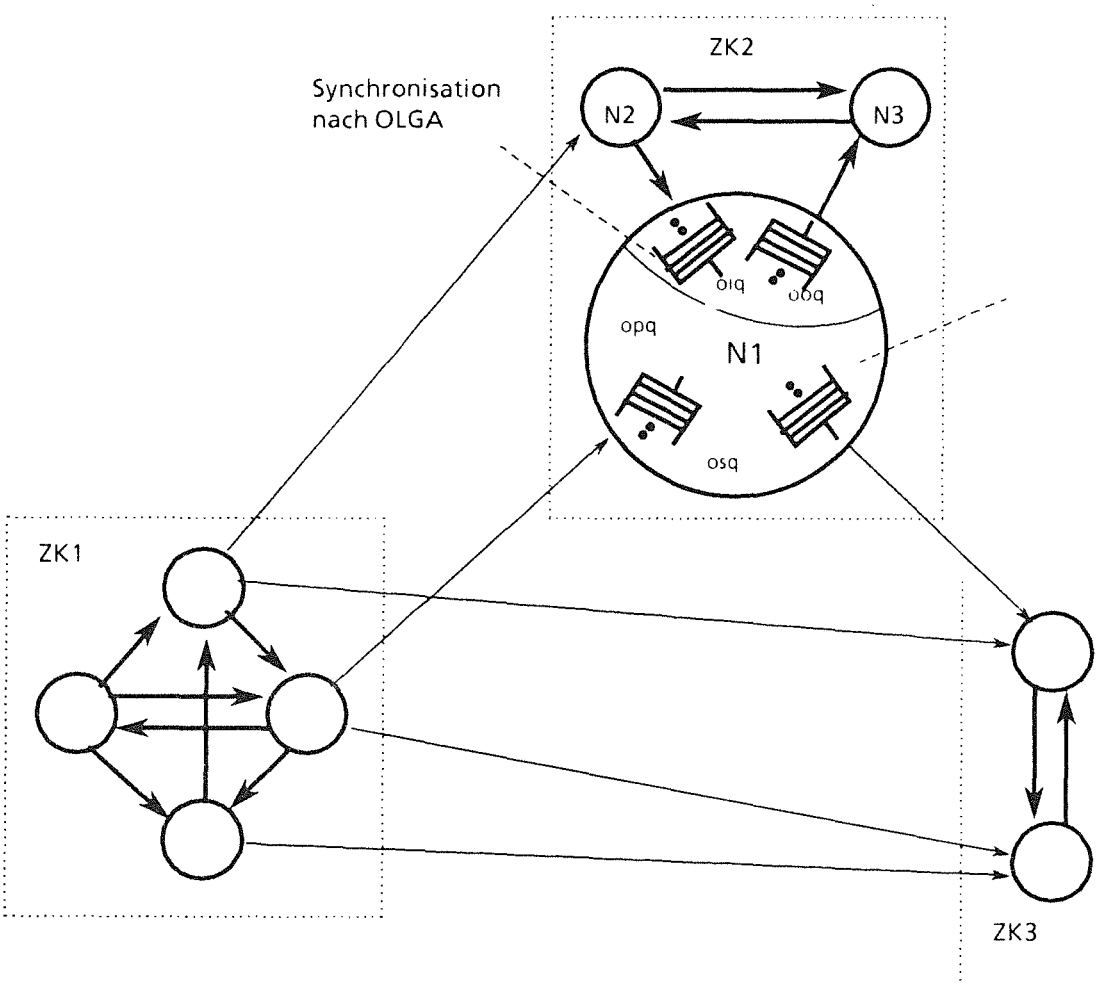
Die OLGA-NWS und der MLT-Algorithmus von 5.2.1.1 können so überlagert werden, daß jede Komponente

- mit Komponenten derselben (starken) Zusammenhangskomponente nach den Regeln der OLGA-NWS
- mit Komponenten unterschiedlicher ZK (Vorgänger- und Nachfolgerkomponenten) nach den Regeln der MLT-NWS

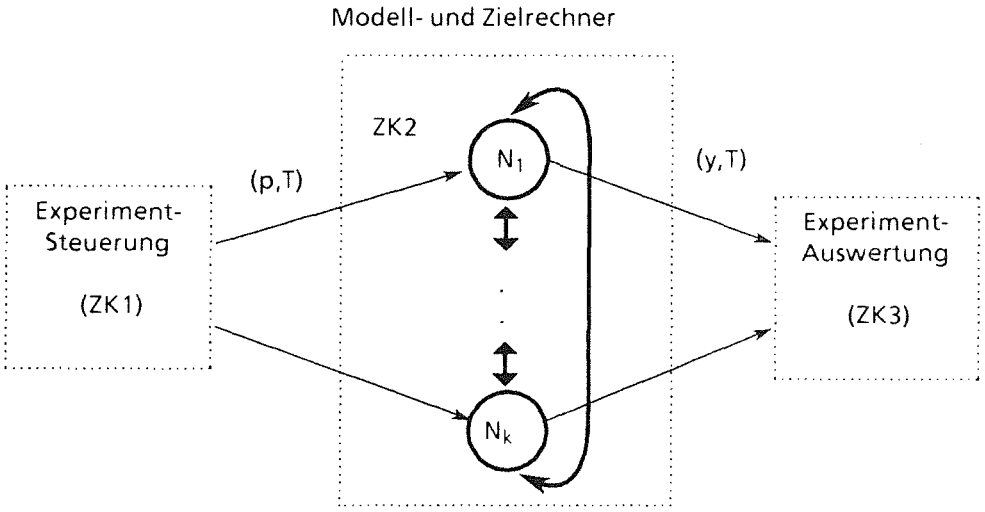
kommuniziert (vgl. Abb. 5.15).
Zur Kommunikation mit unterschiedlichen ZK dienen zwei zusätzliche Ein-/Ausgabe-Warteschlangen:

opq : zentrale Eingabe-WS für alle Komponenten in Vorgänger-ZK (**p**redecessors)
osq : zentrale Ausgabe-WS für alle Komponenten in Nachfolger-ZK (**s**uccessors).

Die virtuelle Zeit einer Komponente wird dabei auch im Vorausschau-Modus durch MLT(opq) nach oben begrenzt. Eingabenachrichten aus opq werden infolge von Rücksetz-Operationen u.U. der Modellkomponente mehrmals zugestellt und verarbeitet, müssen also aufbewahrt werden.




a) Zeugung einer Komponenten-Topologie in ihre starken Zusammenhangskomponenten




(p,T) Simulationsparameter, der zur Zeit T wirksam wird
(y,T) Leistungsdatum, das Testobjekt zur Zeit T repräsentiert

b) Anwendung: Einbettung der Experimentrechner in das Leitsystem

**Abb.** 5.15: Überlagerung von OLGA-NWS und MLT-Algorithmus

Das kombinierte Schema OLGA/MLT führt auch zu einer Vereinfachung des Kommunikationsprotokolls zum Entwicklungs- und Auswertungssystem (vgl. 3.5). Die Experimentsteuerung ist als eine reine Nachrichtenquelle, die Datenerfassungs- und Auswertungskomponente als reine Datensenke anzusehen (vgl. Abb. 5.15b). Da diese Komponenten auf kommerziellen BSen und Kommunikationssystemen basieren, soll hier nicht das spezielle Protokoll der OLGA-NWS mit Marker-Transport, Zustandssicherung und Zurücksetzen implementiert werden. Ferner kann die Experiment-Datenerfassung als ein System mit praktisch unbegrenztem Speicher angesehen werden, das zeitgestempelte Leistungsgrößen akzeptiert und ggf. zwischenspeichert, wie sie anfallen (nicht notwendig chronologisch). Also spielt die endliche Kanalkapazität keine Rolle mehr; auf den Austausch von link-time-Nachrichten zur Verklemmungs-Vermeidung kann daher verzichtet werden.

Die Realisierung der Überlagerung OLGA/MLT soll hier nicht vertieft werden. Die folgenden Abschnitte 5.2.4.3 bis 5.2.4.6 dienen der Präzisierung der in 5.2.4.1 nur grob dargestellten OLGA-NWS und der Lösung von Detailproblemen.

### 5.2.4.3 Marker-Operationen zur Simulation von discrete-event-Netzen

In Abb. 5.16 ist der Marker im Detail als Datentyp mit 5 Operationen a__send, a__save, tlt, m__update und ms__update dargestellt, mit folgender Bedeutung:

| | |
|---|---|
| a__send | Feststellung der Sendeberechtigung einer sendebereiten Komponente |
| a__save | Feststellung, ob aktueller Komponenten-Zustand sicherungsberechtigt (vgl. Fall (c) von (4) in 5.2.4.1) |
| tlt | virtuelle Zeitschranke für die Vorausschau (Lookahead-Timeout), nach deren Ablauf die Komponente den Marker weiterzuleiten hat (vgl (5) in 5.2.4.1) |
| m__update | Modifikation des Marker-Eintrags (durch eine nicht sendeberechtigte Komponente) |
| ms__update | Modifikation des Marker-Eintrags (durch eine sendeberechtigte Komponente). |

Gegenüber 5.2.4.1 werden zwei zusätzliche Komponenten der Marker-Datenstruktur, preempt und rang, benötigt, die folgende Funktionen erfüllen:

**preempt**

Bei der Auswahl einer sendeberechtigten Komponente sind nicht nur die lokalen Virtuellen Uhren $T_i$, sondern auch die Zeitstempel der in Übertragung befindlichen Nachrichten zu berücksichtigen. Insbesondere ist folgende Eigenschaft zu gewährleisten.

> Nachdem ein sendeberechtigter Knoten N1 eine Nachricht (m1,T1) an Knoten N2 (mit $TM_2 > TM_1$) gesendet hat, darf solange kein Knoten N3 mit $TM_1 < TM_3 < TM_2$ sendeberechtigt werden, bis $N_2$ m1 erhalten, zurückgesetzt und erneut eine Simulationszeit $\geq TM_3$ erreicht

**Abb.** 5.16: Marker-Datenstruktur und Operation

```
type marker is array (range__nodes) of
                record
                    TM      :  time;     /* Simulationszeit der Komponente
                    a       :  boolean;  /* Komponente sendewillig?
                    preempt :  integer;  /* noch zu empfangende Nachrichten
                    rang    :  integer;  /* Priorität der Komponente
                end;

Globale Größen :    k          - eigene Knoten-Nr.
                    T(k)       - lokale Simulationszeit
                    msg__rec   - Anzahl empfangener Nachrichten
                    Δ          - virtuelles Zeitinkrement (vgl. 5.2.4.1, (5))

Operationen:

function    a__send (mar: marker, rang: integer) return boolean;

            a__send ≡ mar(k).preempt - msg__rec = 0 ∧
                      ∀j≠k :  (mar(j).TM > T(k) ∨
                          mar(j).TM = T(k) ∧ mar(j).a ∧ mar(j).rang < rang)

function    a__save (mar: marker) return boolean;

            a__save ≡ ∀ j≠k : (mar(j).TM > T(k))

function    tlt (mar: marker) return time;

                    ⎧ min {mar(j).TM | j≠k ∧ mar(j).a}      falls ∃ j: mar(j).a
            tlt =   ⎨
                    ⎩ mar (k).TM + Δ                       sonst

procedure   m__update (out mar: marker; sw: boolean; rang: integer);

    mar(k).preempt: = mar(k).preempt - msg__rec;
    msg__rec : = 0;
    if  mar(k).preempt = 0 then
        mar(k).TM      : = T(k);
        mar(k).a       : = sw;      /* Sendebereitschaft der eigenen Komponente
        mar(k).rang    : = rang;    /* Rang der eigenen Komponente

procedure   ms__update (out mar: marker; recset: set of range__nodes);

    mar(k).preempt: = mar(k).preempt - msg__rec;
    msg__rec : = 0;
    mar(k).TM      : = T(k);
    mar(k).a       : = false;

  for j in recset:
      mar(j).preempt: = mar(j).preempt + 1;
      mar(j).TM : = T(k);
      mar(j).a : = false;
```

hat ($N_2$ könnte als Reaktion auf m1 selbst eine Nachricht (m2,T4) produzieren mit $TM_1 < TM_4 < TM_3$, die **vor** $N_3$ sendeberechtigt ist).

Da keine Aussagen über die Laufzeiten des Markers und der übrigen Nachrichten gemacht werden (insbesondere ist das Kommunikationsmedium nicht notwendig reihenfolge-erhaltend), könnte im nächsten Marker-Umlauf nach dem Senden von m1 gelten: $TM_1 \geq TM_3$, sodaß $N_3$ sich sendeberechtigt glaubt, obwohl N2 die Nachricht m1 noch immer nicht erhalten hat.
Dies wird durch folgende Maßnahmen verhindert:

- ein sendeberechtigter Knoten $N_i$ setzt beim Senden einer Nachricht (m,T) die $TM_j$ aller Empfängerkomponenten auf T zurück (ms__update)
- preempt(j) wird vom Sender $N_i$ inkrementiert und vom Empfänger $N_j$ entsprechend der Anzahl erhaltener Nachrichten dekrementiert, zeigt also an, ob noch Nachrichten an $N_j$ unterwegs sind. Falls ja, darf $N_j$ nicht, wie sonst, die Komponente $TM_j$ im Marker durch seine eigene Simulationszeit ersetzen bzw. aktualisieren.

**rang**

Diese Komponente ist notwendig für die verteilte Simulation von DEVS-Komponenten wie in 4.2.1.3, wenn für zeitgleiche Zustandsübergänge eine strikte Reihenfolge (Zeigler'sche SELECT-Funktion) vorgeschrieben ist.
Die Reihenfolge jeder Sendeoperation ergibt sich aus der Rangnummer des Senders; besteht eine Komponente aus mehreren Unterkomponenten, können diese unterschiedliche Rangnummern haben. D.h. die Rangnummer gilt immer nur für eine Sendeoperation. Unter mehreren Komponenten kann also erst dann eine als sendeberechtigt ausgewählt werden (vgl. a__send), wenn alle übrigen zeitgleichen Komponenten auch sendebereit sind, also eine definierte Rangnummer haben.

### 5.2.4.4 Marker-Operationen zur verteilten Simulation mit realen Testobjekten

**Problemstellung (Nichtreproduzierbarkeit der R-Prozesse im Rücksetz-Modus)**

Jede Komponente eines zeitdiskreten Simulationsmodells wird im Rücksetz-Modus vom Zustand (sb,TS) ausgehend mit derselben Eingabehistorie wie im Vorausschau-Modus auch denselben Zustandsverlauf und dieselbe Ausgabehistorie generieren. Dies trifft auch bei stochastischen Simulationen zu. Bei der Erzeugung von Zufallszahlenfolgen gehören deren "Saatkörner" auch zum restaurierten Zustand; es werden also im Rücksetz-Modus identische Replikationen bisheriger Zufallszahlenfolgen erzeugt.
Durch die Integration realer Komponenten (R-Prozesse) wird hingegen ein Element des Indeterminismus eingeführt. Selbst ein einzelner sequentieller R-Prozeß wird beim Übergang vom Zustand s1 in den Zustand s2 selten zweimal exakt dieselbe gemessene Zeit benötigen, auch wenn er beide Male ununterbrochen (!) rechnet und wenn neben dem Prozeßzustand s1 auch der Zustand seiner Umgebung (BS-Kern, Virtuelle Uhr, E/A-Geräte, übrige Prozesse) reproduziert wird, was bei der integrierten Simulation ja geschieht. Dafür gibt es mehrere Gründe:

- die Ungenauigkeit der Meßuhr (i.d.R. wohl vernachlässigbar)

- die unterschiedlichen Bus- und Speicherzugriffszeiten des R-Prozesses (wait states), abhängig von **externen** Zugreifern, welche **nicht** durch die Experimentführung kontrolliert werden, z.B. Speicher-refresh-Zyklen.
- bei Mehrprozeßbetrieb (R- und W-Prozesse): die implementierungsbedingte Ungenauigkeit bei der Prozeß- und Zeitmodusumschaltung im hybriden Kern.

Abläufe realer Testobjekte sind also nur eingeschränkt reproduzierbar. Grundsätzlich ist es möglich, daß ein R-Prozeß ausgehend vom Zustand sb als nächste Ausgabe (m,T) erzeugt, aber nach dem Zurücksetzen durch eine Nachricht (m',T'), $T'<T$, nun (m,T'') mit $T''<T'<T$ generiert. Dies würde die zeitliche Monotonie der Nachrichtenkommunikation verletzen. Als Folge könnte eine Komponente eine Nachricht erhalten, deren Zeitstempel **vor** dem ihrer Sicherungskopie liegt. Eine solche Konsistenzverletzung der Zustandssicherung ist zu verhindern.

**Ziele, Vorgehensweise**

Angenommen jede Komponente $N_k$ verfügt zumindest über eine absolute obere Schranke $\varepsilon_k$ der zeitlichen Ungenauigkeit ihrer Zustände/Ausgaben (eine grobe Vorschrift zur Bestimmung wird unter (f) i.f. angegeben). Wenn also $N_k$ eine Nachricht (m,T) erzeugt, so entspricht dieser in realer Umgebung eine Nachricht (m,T') innerhalb des **zeitlichen Unsicherheitsintervalls**
$T' \in I_k := [T-\varepsilon_k, T+\varepsilon_k]$.

(1) Die Zeitrechnungen der NWS mit lokalen Simulationszeiten, Nachrichten-Zeitstempeln und Marker-Einträgen sollen unter expliziter Berücksichtigung der zeitlichen Unsicherheit erfolgen. Solange die tatsächlich produzierten Zustände und Ausgaben zeitlich innerhalb dieser Unsicherheitsintervalle liegen, soll keine Komponente vor den Zeitpunkt TS ihrer Sicherungskopie zurücksetzen müssen.

(2) Wenn schon zeitliche Unsicherheit in Rechnung gestellt wird, so sollten sendebereite Komponenten mit "im Rahmen der Unsicherheit" gleicher Simulationszeit (5.17) auch gleichzeitig senden dürfen. Auch sollte eine Nachricht, deren Zeitstempel zwar in der virtuellen Vergangenheit ihrer Empfängerkomponente, aber noch innerhalb ihres Unsicherheitsintervalls liegt, kein Zurücksetzen erfordern.

Dies widerspricht zwar 5.2.4.3, aber eine Prioritätsreihenfolge zeitgleicher Zustandsübergänge wird sinnlos, sobald über die Kommunikations-Schnittstelle zu den R-Prozessen auch im zeitlichen Verhalten der W-Prozesse Unsicherheit entsteht. Eine Rangfolge dieser W-Prozesse (SELECT-Ftn.) kann keine deterministische Zuteilungsreihenfolge im Experiment erzwingen.

(3) Die Virtuellen Uhren der Komponenten sollen von Zeit zu Zeit auf einen Wert innerhalb ihres Unsicherheitsintervalls normiert und die Fehlerschranken $\varepsilon_k$ wieder auf 0 zurückgesetzt werden, weil sonst mit zunehmender Simulationsdauer die absoluten Unsicherheitsintervalle beliebig groß werden. Diese Normierung geschieht an den Punkten der Nachrichtenkommunikation bzw. Zustandssicherung.

Anders als bei der Synchronisation von Echtzeituhren /MAO 83/ kann aber die zeitliche

Genauigkeit als solche nicht verbessert werden, weil keine zeitliche Redundanz im System vorhanden ist (die Virtuellen Uhren der Komponenten $N_1,..,N_n$ repräsentieren unterschiedliche Teile des realen Wirkungskreises zu **verschiedenen** Zeiten!).

I.f. werden die Regeln der NWS auf Unsicherheitsintervalle erweitert; Abb. 5.17 zeigt den modifizierten Marker-Datentyp und seine Operationen.

a) **Marker-Einträge**

Jede Komponente $N_k$ trägt außer $TM_k$ ihre Fehlerschranke $\varepsilon_k$ für die Simulationszeit ein. Dadurch wird ein absolutes Unsicherheitsintervall $I_k$ definiert:

$I_k := [TM_k - \varepsilon_k, TM_k + \varepsilon_k]$

b) **Bedingung der Sendeberechtigung**

Eine sendebereite Komponente $N_k$ ist genau dann sendeberechtigt, wenn sie zu der wie folgt definierten Menge Nmin der **zeitminimalen Komponenten** gehört:

(5.17)

$\forall N_i \in Nmin, \forall j \notin Nmin : TM_i \leq TM_j$

$\forall N_i \in Nmin : TM_i \in \bigcap_{N_k \in Nmin} I_k$

Die Simulationszeit jeder Komponente von Nmin liegt also innerhalb des Unsicherheitsintervalls aller anderen Komponenten dieser Menge.

c) **Lookahead-Timeout**

Eine Komponente $N_k$, die zeitlich unterhalb der kleinsten Zeit TA__min des Markers liegt, zu der eine Komponente $N_j$ sendebereit ist (vgl. 5.15), behält diesen solange, bis auch $N_j$ zur Menge Nmin gehört und damit sendeberechtigt ist, bis also $T_k$ in das Unsicherheitsintervall von $N_j$ eintritt und umgekehrt das eigene Unsicherheitsintervall $I_k$ den Punkt TM(j) enthält:

(5.18) $T_k \geq TA\_\_min - \min\{\varepsilon_j, \varepsilon_k\}$

d) **Sendeoperation (m,t)**

Sei $N_k$ sendeberechtigt und Tmin : = min $\{TM_i\}$ kleinste Zeit im Marker.

- Statt (m,t) wird (m,Tmin) gesendet (zeitliche Normierung)
- Die lokale Simulationszeit $T_k$ wird zu Tmin normiert (clock__init)
- Es wird eine Sicherungskopie (sb, Tmin) erstellt
- Die Fehlerschranke $\varepsilon_k$ wird auf 0 zurückgesetzt.

Da $N_k$ sendeberechtigt ist, liegt die Normierung Tmin innerhalb des bisherigen Unsicherheitsintervalls $I_k$ von $N_k$.

**Abb.** 5.17: Marker- Datenstruktur und Operationen mit Unsicherheitsintervallen

```
type marker is array (range__nodes) of
                record
                    TM          : time;      /* Simulationszeit der Komponente
                    a           : boolean;   /* Komponente sendewillig?
                    preempt     : integer;   /* noch zu empfangende Nachrichten
                    e           : time;      /* zeitliche Unsicherheit
                end;
```

Bezeichnungen:

| | | | |
|---|---|---|---|
| TM(j) | := mar(j).TM | falls $j \neq k$ | |
| | T(k) | sonst (lokale Simulationszeit) | |
| e(j) | := mar(j).e | falls $j \neq k$ | |
| | e(k) | sonst (lokale zeitliche Unsicherheit) | |

**Operationen:**

**function** asend (mar: marker) **return** boolean; (vgl. 5.17)

$$a\_send \equiv \forall i,\forall j:\ TM(i) \leq TM(k) \wedge TM(j) \leq TM(k) \Rightarrow (TM(i)-e(i) \leq TM(j) \leq TM(i)+e(i) \wedge TM(j)-e(j) \leq TM(i) \leq TM(j)+e(j))$$

**function** a__save (mar: marker) **return** boolean;

(* identisch zu a__send *)

**function** tlt (mar: marker) **return** time; (vgl. 5.18)

$$tlt = \begin{cases} TM(j) - \min(e(j),e(k)), \text{ falls } mar(j).a \wedge\ TM(j) = \min\{TM(j) \mid j \neq k \wedge mar(j).a\} \\ TM(k) + \Delta \qquad \text{sonst} \end{cases}$$

**procedure** m__update (**out** mar: marker; sw: boolean);

```
mar(k).preempt:= mar(k).preempt - msg__rec;
msg__rec := 0;
if  mar(k).preempt=0 then
    mar(k).TM       := T(k);
    mar(k).a        := sw;
    mar(k).e        := e(k);
```

**procedure** ms__update (**out** mar: marker; recset: set of range__nodes);

```
mar(k).preempt:= mar(k).preempt - msg__rec;
msg__rec := 0;
if mar(k).preempt=0 then
    mar(k).TM       := min {(TM(i)};
    mar(k).a        := failse;
    mar(k).e        := e(k);

for j in recset:
    mar(j).preempt:= mar(j).preempt + 1;
    mar(j).TM:= min {TM(i)};
    mar(j).e := ∞;
```

e) **Empfangsoperation (m,t)**

**Fall** 1: Ankunft in der virtuellen Vergangenheit, d.h. $t < T_k - \varepsilon_k$

- Die Sicherungskopie (sb, TS) wird restauriert
- Die eigene Fehlerschranke $\varepsilon_k$ wird auf 0 zurückgesetzt.
- Die Zustellung der Nachricht wird durch einen Zeiteintrag vorgemerkt wie unter 5.2.4.1 (3).

**Fall** 2: Ankunft in der Gegenwart, d.h. $t \varepsilon I_k$

- Die Nachricht wird sofort zugestellt
- Es wird eine Sicherungskopie (sb, t) erstellt
- Die Simulationszeit $T_k$ wird ebenfalls zu t normiert (clock__init)
- Die Fehlerschranke $\varepsilon_k$ wird auf 0 zurückgesetzt.

**Fall** 3: Ankunft in der virtuellen Zukunft, d.h. $t > T_k + \varepsilon_k$ (nur im Rücksetz-Modus möglich)

- Die Zustellung der Nachricht wird durch einen Zeiteintrag vorgemerkt.

Die globale Monotonie der Nachrichtenkommunikation, und damit auch die Konsistenz der Zustandssicherung, bleiben erhalten, falls die $\varepsilon_k$ korrekte Schranken für die tatsächliche zeitliche Unsicherheit liefern. Entweder der Zeitstempel t der ankommenden Nachricht liegt noch innerhalb von $I_k$, so kann $N_k$ wegen der Normierung der Zeiten $T_k$, $TS_k$ zu t, bzw. $\varepsilon_k$ zu 0, keine Nachricht (m',t') mehr mit $t' < t$ erzeugen (Monotonie der lokalen Virtuellen Uhr). Oder aber t liegt in der Vergangenheit von $I_k$ - **nur** dann wird zurückgesetzt - so kann $N_k$ im Rücksetz-Modus ebenfalls kein (m',t') mit $t' < t$ generieren, denn $T_k - \varepsilon_k$ ist nach Wahl von $\varepsilon_k$ eine untere zeitliche Schranke für jede Ausgabe von $N_k$ ausgehend vom Zustand (sb, $TS_k$) und leerer Eingabe.

f) **Bestimmung der Schranke $\varepsilon_k$ für die zeitliche Unsicherhei t**

- $\varepsilon_k$ wird bei jedem save und restore der Sicherungskopie auf 0 gesetzt (siehe d),e), **Normierung**).
- Falls die Komponente $N_k$ keine R-Prozesse enthält, ist $\varepsilon_k = 0$ zu setzen (reine discrete-event-Modelle sind zeitlich stets exakt reproduzierbar). Andernfalls kann $\varepsilon_k$ z.B. nach folgender Näherungsformel bestimmt werden:

(5.19) $\varepsilon_k := nr * \varepsilon 1 + lr * \varepsilon 2$ mit

nr : Anzahl der Zeitmodusumschaltungen zwischen F- und R-Modus seit der letzten Normierung

$\varepsilon 1$ : maximale zeitliche Ungenauigkeit bei der Realisierung einer Zeitmodus-umschaltung

lr : Zeitdauer, die seit der letzten Normierung im R-Modus verstrichen ist

$\varepsilon 2$ : maximaler relativer Fehler für die Drift der Echtzeituhr und die Laufzeitunterschiede der R-Prozesse infolge der zu Beginn von 5.2.4.4 erwähnten Einflußfaktoren.

### 5.2.4.5 Terminierung und Fehlersituationen

Notwendig für die Terminierung eines Simulationsexperimentes ist die Verklemmungsfreiheit der NWS. Entscheidend hierfür ist der Marker, der von allen Komponenten benötigt wird, um Ausgabe senden zu können, aber von gewissen Komponenten auch festgehalten wird, bis eine gegebene Simulationszeit TA__min erreicht ist (vgl. 5.2.4.1 (3)). Es ist möglich, daß eine solche Komponente

**weder**

(a) eine Ausgabeoperation

**noch**

(b) die vorgegebene Simulationszeit TA__min erreicht,

sondern entweder

(c) ihre Simulationszeit bei $T1 < TA\_\_min$ stationär wird, z.B. weil eine unendliche Schleife oder ein Systemfehler in einem Zustandsübergang auftritt, oder

(d) in einer Folge $(s_i,t_i)$ von Zustandsübergängen die Folge der Simulationszeiten $t_i$ einem endlichen Grenzwert $T1 < TA\_\_min$ zustrebt. Handelt es sich z.B. um eine DEVS-Komponente, könnte mit
$s_i = \delta\_\_\varnothing^i(s0)$ gelten:

$$\lim_{i\to\infty} t_i = \lim_{i\to\infty} \sum_{j=0}^{i} ta(\delta\_\_\varnothing^j(s0) < TA\_min$$

Dies bedeutet nun nicht zwangsläufig, daß das gekoppelte Systemmodell als Ganzes inkonsistent bzw. nicht wohldefiniert ist. Es könnte daran liegen, daß eine Komponente $N_k$ im Vorausschau-Modus mit leerer Eingabe "zu weit gegangen" ist und die Situation (c) oder (d) bei $T1 < TA\_\_min$ nicht auftreten bzw. zurückgesetzt würde, wenn irgendwann $N_k$ eine nichtleere Eingabe (m,T) mit $T<T1$ von einer Komponente $N_j$ erhielte (dann würde $N_k$ ab Zeitpunkt T einen anderen Zustandsverlauf einschlagen, Abb. 5.18a).

Um zu senden, benötigt Nj den Marker. Um folgende **Verklemmungssituation** zu vermeiden (vgl. Abb. 5.18b):

- $N_k$ gibt in Situation c) oder d) wegen $T1 < TA\_\_min$ den Marker nicht frei ($N_j$ ist noch nicht als sendebereit eingetragen)
- $N_j$ wird mit $T_j = T < T1$ sendebereit und erwartet den Marker, um die Sendeoperation eintragen und ausführen zu können,

überwacht jede Komponente im Marker-Besitz zusätzlich die Bedingung lnt ("lookahead nicht terminierbar"):

lnt ≡ Marker mindestens $\Delta_R$ Zeiteinheiten (in **Echtzeit**) in Besitz,
oder ein Fehler ist aufgetreten.

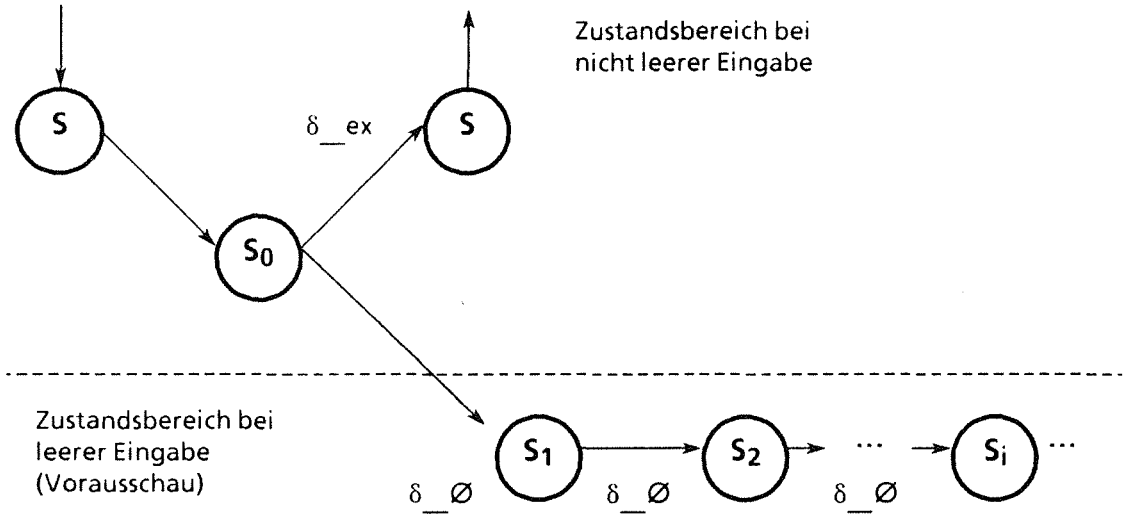



**Abb.** 5.18 a Beispiel für DEVS-Komponente mit nichtterminierender Vorausschau

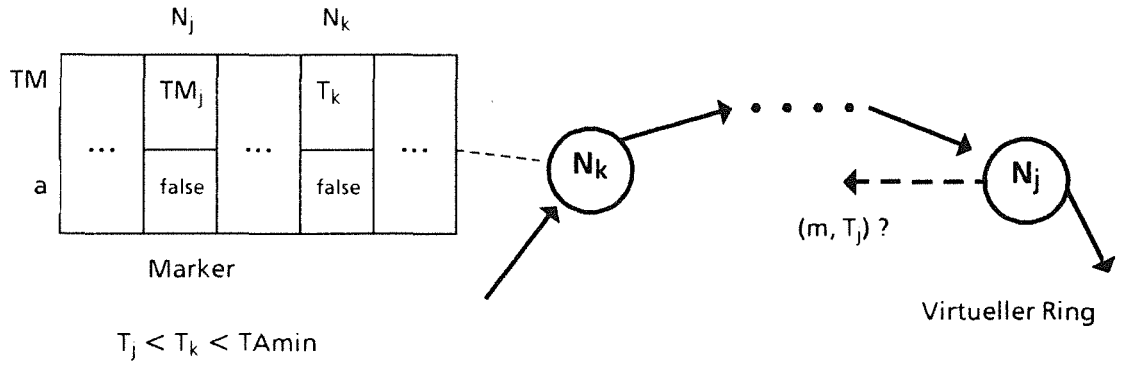



**Abb.** 5.18b: Verklemmungssituation aufgrund Lookahead -Timeout TAmin

Der Marker wird unabhängig von der virtuellen Zeit T1 weitergesandt, sobald die Bedingung Int wahr wird oder wenn Int beim Marker-Empfang bereits gilt.

Eine Fehlersituation (Fall (c)) während der Simulation ist dagegen **endgültig** und führt zum Abbruch des Experimentes, wenn sie

- im Rücksetz-Modus auftritt, oder
- im Vorausschau-Modus auftritt, aber die Komponente sicherungsberechtigt (5.2.4.1, (5.13)) und ihr Zustand damit irreversibel ist.

Falls schließlich im Vorausschau-Modus der Marker permanent im System kreist, aber keine Komponente je sendebereit wird und auch kein zeitliches Abbruchkriterium von allen Komponenten erreicht wird, so kann auch keine sequentielle Simulation terminieren. Das Modell ist dann nicht wohldefiniert.

**Bemerkungen**

- Eine temporärer Fehler im Vorausschau-Modus darf keine irreversiblen Folgen haben; insbesondere muß die Fähigkeit des Systems, Nachrichten empfangen und den Zustand

zurücksetzen zu können, in jedem Fall erhalten bleiben. Der BS-Kern muß also geschützte Bereiche bereitstellen, zu denen auch die OLGA-NWS zählt.

- Die Fehlerfälle (c) und (d) sind zwar theoretisch wichtig, treten allerdings bei der Modellierung von Prozeßführungssystemen in der Praxis nicht auf. Ein Simulationsmodell eines technischen Prozesses, das über beliebig lange Zeitspannen ohne externe Stellgrößen (Eingabenachrichten) operiert, erreicht entweder nach endlicher Zeit einen Alarmzustand (Grenzwertüberschreitung), d.h. eine **Ausgabeoperation** (Fall (a)), oder sein Zustand bleibt im Normalbereich und wird dann in festen Zeitinkrementen weiterintegriert, bis die Simulationszeit TA__min überschritten ist (Fall (b)).
  Ähnlich verhält sich eine PFS-Komponente beim Ausbleiben von Eingabedaten (Aufträgen): entweder sie produziert selbst eine Ausgabe (Fall (a)), oder erlaubt im modellbedingten Wartezustand das Fortschreiten der Simulationszeit (Fall (b)).

#### 5.2.4.6 Lokale E/A-Gerätesteuerung

Bei allen spekulativen Verfahren zur NWS stellt sich das Problem, daß der Zustand der Modellkomponenten (dynamische Daten der Rechenprozesse) zwar rücksetzbar, aus diesem Zustand heraus erfolgte physikalische E/A-Operationen (z.B. Drucker- und Terminal-Ausgabe, Hintergrundspeicher-E/A, Prozeßperipherie-E/A) dagegen **irreversibel** sind.

In Time Warp (5.2.2) wird dieses Problem im Prinzip so gelöst, daß jedem E/A-Gerät eine zusätzliche Auftragswarteschlange dev__q vorgeschaltet wird:

- E/A-Aufträge des Gerätetreibers an das Gerät werden zunächst nur zeitgestempelt in dev__q eingetragen; der physikalische Gerätestart selbst wird aufgeschoben.
- Bei jeder Erhöhung der globalen virtuellen Zeit GVT (5.2.2.1) werden alle Aufträge in dev__q mit $t < GVT$ physikalisch ausgeführt.
- Bei jeder Rücksetz-Operation infolge einer Nachricht (m,T) werden alle Aufträge in dev__q mit Zeitstempel $t > T$ verworfen.

Auf die Realisierung dieses Konzeptes wird in /JEF 83/,/JES 83/,/JBH 85/ nicht eingegangen; es wird nur bemerkt, daß der Mechanismus der E/A-Steuerung transparent für die Modellierungsanwendung sei und daß eine Symmetrie zwischen Geräteeingabe und -ausgabe bestehe. Eine genauere Betrachtung am Beispiel von Hintergrundspeicher-E/A wirft jedoch praktische Probleme auf.

**(1) Ausgabe (Schreiboperation)**

In dev__q müssen neben den Auftragsparametern auch die zu schreibenden Daten selbst (u.U. viele KByte pro Auftrag) über längere Zeitspannen zwischengepuffert werden.

**(2) Eingabe (Leseoperation)**

Ein Aufschieben der Leseoperationen reicht offenbar nicht aus: eine Leseoperation im Vorausschau-Modus muß zu jeder Zeit den aktuellen Hintergrundspeicher-Inhalt unter Berücksichtigung aller Schreiboperationen mit kleinerer virtueller Zeit (ob physikalisch bereits durchgeführt oder aufgeschoben) liefern. Jeder E/A-Auftrag in dev__q sei vereinfacht durch ein Tupel dargestellt:

rd= (T, s__adr, t__adr) mit
s__adr: Adreßintervall auf Hintergrundspeicher (zu lesende Daten)
t__adr: Adreßintervall im ASP (Zielbereich)

wr= (T, s__adr, t__adr) mit
s__adr: Adreßintervall im ASP (zu schreibende Daten)
t__adr: Adreßintervall auf Hintergrundspeicher (Zielbereich)

Für eine Leseoperation $rd_i$ ist die gesamte Historie der Schreiboperationen $wr_j$ in dev__q mit $wr_j.T < rd_i.T$ durchzuspielen, um die Daten im Hintergrundspeicherbereich

$$s\_\_adr_i \cap \bigcup_{wr_j.T < rd_i.T} t\_\_adr_j$$

zu ermitteln, und im Adreßbereich

$$s\_\_adr_i \setminus \bigcup_{wr_j.T < rd_i.T} t\_\_adr_j$$

sind die Daten physikalisch vom Hintergrundspeicher zu lesen. Die durchzuführenden Operationen sind also komplex und aufwendig; von einer Symmetrie zwischen Schreib- und Lese-Aufträgen kann keine Rede sein.

**Konzept-Alternative:**

**Schreiboperationen** auf E/A-Geräten werden wie Sendeoperationen an Komponenten behandelt und in das Protokoll der OLGA-NWS eingebunden. D.h. die Geräte-Ausgabe und die lokale Simulationsanwendung sind solange blockiert, bis Sende- bzw. Sicherungsberechtigung für den Zeitpunkt der Operation anhand des Markers festgestellt wird. Dann sind die Schreiboperation selbst nicht mehr rücksetzbar. Die Sicherungskopie wird ebenfalls aktualisiert wie nach einer Sendeoperation. Der einzige Unterschied zum Senden einer Nachricht besteht darin, daß Geräteoperationen rechner-lokal sind, d.h. kein Empfängerknoten davon betroffen ist.
**Leseoperationen** werden dagegen sofort physikalisch ausgeführt und liefern damit stets ein korrektes und aktuelles Ergebnis.

Aus dem Zurücksetzen des gesamten Knotenzustands unter OLGA resultiert ein weiteres Problem: den **gegenseitigen Ausschluß von NWS-Operationen und Geräteoperationen** sicherzustellen.
Die Hauptspeicherbereiche s__adr bei einer Schreiboperation und t__adr bei einer Leseoperation werden vom E/A-Gerät i.d.R. im DMA-Modus gelesen und beschrieben. Zugleich sind sie potentieller Zielbereich der Zustandssicherung und -restaurierung. Aus Konsistenzgründen folgt, daß zumindest save/restore und Gerätetätigkeit einander zeitlich ausschließen müssen. Dies gilt nicht nur wegen der Gefahr der Inkonsistenz von Daten, sondern auch der Prozeßzustände der beteiligten Treiberprozesse: würde z.B. eine physikalische Leseoperation von einer restore-Operation in die virtuelle Vergangenheit unterbrochen, so fände die Gerätefertigmeldung (Interrupt-Routine) ihren Auftraggeber nicht mehr in dem erwarteten Prozeßzustand zu Beginn der Geräteoperation vor.

Eine save- oder restore-Operation muß also solange aufgeschoben werden, bis keine physikalischen E/A-Aufträge mehr aktiv sind. Dies ist durch eine Buchführung aktiver E/A-Aufträge realisierbar.

### 5.2.5 Software-Architektur der OLGA-NWS

#### 5.2.5.1 Schnittstellendienste und Zustandsdiagramm

Nachdem in 5.2.4 das Verfahren zur NWS als solches vorgestellt wurde, sollen i.f. die SW-Architektur und Realisierung kurz diskutiert werden. Es werden die von OLGA exportierten und die von tieferen Schichten importierten Dienste beschrieben, und die interne Ablaufsteuerung der NWS wird durch einen Zustandsautomaten dargestellt. Das Blockdiagramm (Abb. 5.19) und das Zustandsdiagramm (Abb. 5.20) erläutern die Darstellung.

**(1) Anwendungsseitige Kommunikationsdienste**

Für die Anwendung stellt sich die Schnittstelle von OLGA als verbindungsloser Transportdienst dar, mit Hilfe dessen Nachrichten (Pakete) zwischen beliebigen Experimentrechnerknoten end-to-end übertragen werden.

**appl-send** (msg : message); Sendeoperation
**appl-rec** (**out** msg : message); Empfangsoperation

wobei der Datentyp message wie folgt strukturiert ist:

- Länge
- Zeitstempel
- Sender-Knoten-Id.
- Empfänger-Knoten-Id.
- Rangnummer (optional, für reine discrete-event-Simulation)
- Nutzdaten

Die Dienste appl_send, appl_rec werden von verteilten Simulationsanwendungen als Transportdienste entsprechend Schicht 4 im ISO-Schichtenmodell benutzt (i.a. indirekt über anwendungsnähere Kommunikationsdienste, z.B. netzweite Mailbox-Kommunikation). Sollen die unteren Kommunikationsschichten selbst als Teil des realen Testobjektes bewertet werden, so liegen appl_send , appl_rec auf der untersten Schicht des RTO, im Extremfall auf Schicht 1 (physikalisches Kommunikationsmedium), und bilden zugleich die Schnittstelle zur simulierten Umgebung (vgl. 6.1.4).

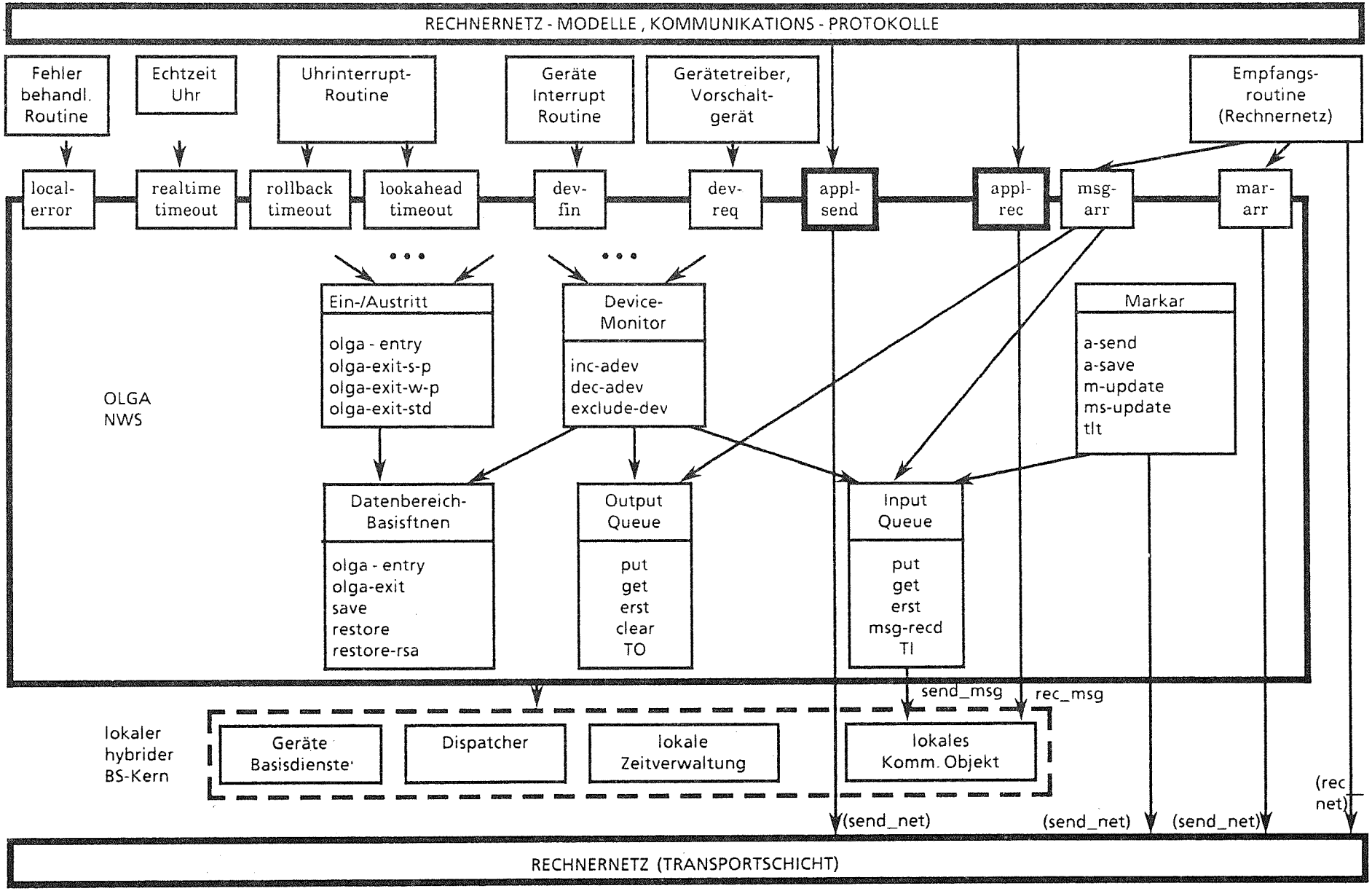



Abb. 5.19 Blockdiagramm der OLGA-Netzwerksynchronisation

**(2) Sonstige Schnittstellendienste von OLGA**

Mit Hilfe der folgenden Operationen werden alle Ereignisse mitgeteilt, die für die Synchronisationsfunktion der NWS wesentlich sind. Neben den Kommunikationsdiensten (1) sind dies

**a) Geräteoperationen**

| | | |
|---|---|---|
| **dev-req** | (g : device__unit;<br>pb : device__request); | Gerätestart |
| **dev-fin** | (g : device__unit); | GerätefertigmeldungDiese Operationen führen keine Gerätetätigkeit als solche durch, sondern erfüllen nach 5.2.4.6 ausschließlich Synchronisationsaufgaben: |

- gegenseitiger Ausschluß von aktiven E/A-Aufträgen und Operationen save und restore der NWS;
- Sicherstellen der Sendeberechtigung (a__send) bei einem **schreibenden** E/A-Auftrag.

**b) Nachrichten- und Marker-Ankunft**

| | | |
|---|---|---|
| **msg-arr** | (msg : message); | Ankunft einer echten Nachricht (vgl. 5.2.4.1, (3)) |
| **mar-arr** | (mar : marker); | Ankunft des Markers (vgl. 5.2.4.1, (5)) |

**c) Ablauf zeitlicher Wartebedingungen, Fehlermeldungen**

| | | |
|---|---|---|
| **rollback-timeout** | (rce : clock__entry); | Ablauf des Zeiteintrags im Rücksetz-Modus für eine zuzustellende Nachricht |
| **lookahead-timeout** | (lce : clock__entry); | Ablauf der virtuellen Frist, nach der der Marker freizugeben ist |
| **realtime-timeout**; | | Ablauf der Echtzeitüberwachung für den Marker (5.2.4.5) |
| **local-error**; | | Benachrichtigung über Systemfehler in Simulationsanwendung (5.2.4.5) |

Die Operationen realtime__timeout, local__error sind zur Auswertung der Bedingung Int notwendig (vgl. 5.2.4.5).

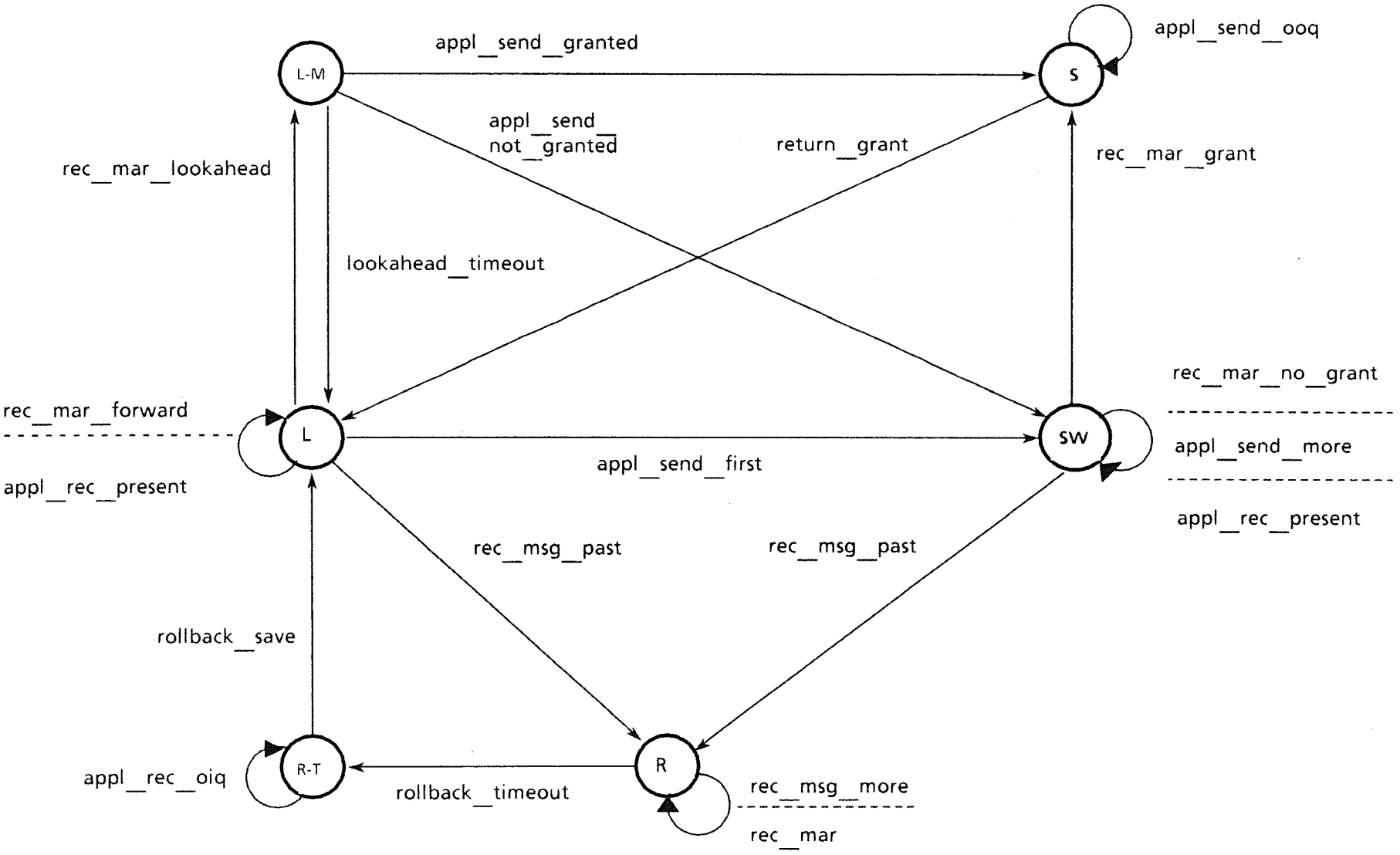



**Abb.** 5.20 Zustandsdiagramm der OLGA-NWS

**(3) Zustandsübergangsdiagramm**

Das Zustandsdiagramm (Abb. 5.20) faßt in kompakter Form die Regeln zur Ablaufsteuerung der NWS von 5.2.4.1-5.2.4.6 zusammen. Zur besseren Übersicht sind die Zustandsübergänge nur mnemotechnisch abgekürzt wiedergegeben; die zugehörigen Bedingungen und Aktionen sind im Anhang A.2 genauer spezifiziert. Ferner wurden einige Vereinfachungen vorgenommen: schreibende Geräteaufträge werden als Unterfall der Sendeoperation (appl__send) behandelt, und der gegenseitige Ausschluß von E/A-Operationen und save/restore wird nicht dargestellt. Nur Ereignisse, die im Bedingungsteil eines Zustandsüberganges aufgeführt sind, gelten in dem betreffenden Ausgangszustand als zulässig, die übrigen (nicht aufgeführten) als Fehlerfälle (z.B. appl__send im im Zustand R oder R__M).

**Der Zustandsbereich** ist

```
type olga__state = (
L,        Vorausschau-Modus
L__M,     Vorausschau-Modus, im Marker-Besitz
SW,       Knoten sendebereit
S,        Knoten sendeberechtigt (transienter Zustand, in dem Nachrichten gesendet werden)
R,        Rücksetz-Modus
R__T,     Rücksetz-Modus (Rollback-timeout abgelaufen)
          );
```

Der Startzustand des Automaten ist L__M für den Knoten im virtuellen Ring, der den Marker zuerst in Umlauf bringt, L für alle anderen Knoten.

Die konkreten Bedingungen und Aktionen der Zustandsübergänge für eine reine discrete-event-Simulation mit vorgeschriebener Reihenfolge zeitgleicher Komponenten (5.2.4.3) unterscheiden sich z.T. von denen einer Echtteilsimulation bei zeitlicher Unsicherheit (5.2.4.4); dies ist in A.2 berücksichtigt. Z.B. bewirkt im ersten Fall eine Nachrichtenankunft (m,T) mit $T = T_k$ im Zustand L oder SW , daß die Komponente $N_k$ zurücksetzen muß, während bei der Echtteilesimulation unter der entsprechenden Bedingung $T \in I_k$ die Nachricht (m,T) direkt zugestellt und $N_k$ nicht zurückgesetzt wird.

**(4) Von der NWS benötigte Hilfsobjekte und -operationen**

a) **Marker** (vgl. 5.2.4.3, 5.2.4.4)

b) **Ein-/Ausgabewarteschlangen ooq, oiq**

Diese Warteschlangen dienen der Zwischenspeicherung der von der Anwendung abgesetzten (zu sendenden) bzw. der vom Rechnernetz empfangenen (zuzustellenden) Nachrichten.

Die wichtigsten Dienste von ooq sind

```
ooq-put  (oq  : output__queue;     Einfügen einer Nachricht
          msg : message);          (bei Sendewunsch aus Anwendung)
```

| | | |
|---|---|---|
| **ooq-get** | (oq : output__queue;<br>msg : message); | Ausfügen einer Nachricht<br>(bei Sendeberechtigung) |

Einfügen in ooq bedeutet zugleich die Blockierung des zeitlichen Fortschritts der rechnerlokalen Simulationsanwendung, Ausfügen die Deblockierung (vgl. 5.2.4.1, (2)). Denn am Ende der Sendeoperation erfolgt eine Zustandssicherung, und sicherungsberechtigt ist der Zustand zum Zeitpunkt der gesendeten Nachricht, aber i.a. kein späterer Zustand.

Erreicht wird die zeitliche Blockierung wie folgt: bei Aufruf von ooq__put wird ein (fiktiver) W-Prozeß blockiert mit Zielzustand w__ea (vgl. 4.2.1.1), bei ooq__get wird einer deblockiert. Der appl__send aufrufende Prozeß oder ein anderer bereiter W-Prozeß kann zunächst fortfahren, aber die virtuelle Uhr befindet sich im F-Modus, und es können keine zeitlichen Wartebedingungen ablaufen.

Die Eingabe-WS oiq verfügt über die Dienste

| | | |
|---|---|---|
| **oiq-put** | (iq : input__queue;<br>msg : message); | Einfügen einer Nachricht<br>(bei Ankunft von Rechnernetz) |
| **oiq-get** | (iq : input__queue;<br>msg : message); | Ausfügen einer Nachricht<br>(bei Zustellung) |

Zwischen oiq und den auf Nachrichten wartenden Anwenderprozessen verbirgt sich ein NWS-eigenes RW-Kommunikationsobjekt olga__mbx. Eine Nachricht wird an olga__mbx gesendet, nachdem sie aus oiq ausgefügt wurde, und von olga__mbx durch appl__rec empfangen.
Weitere Operationen auf oiq, ooq sind z.B. das Löschen (ooq__clear) oder die Zählung empfangener Nachrichten (oiq__msg__recd) etc.

**c) Lokale BS-Funktionen**

Folgende Unterstützung durch den Monoprozessor-BS-Kern wird benötigt:

- Lokales **Kommunikationsobjekt** (send__msg, rec__msg),
  **Prozeßzustandsübergänge** (blockieren, deblockieren) (siehe unter b))
- **Zeiteinträge** (clockentry__set, clockentry__cancel)
  Setzen, Löschen und Ablaufen von Vorausschau- und Rücksetz-Zeiteinträgen
- **Zeit-/Zeitmodusoperationen** (zeitmodus, clock__init)
  Alle Operationen der NWS unter (1) und (2)(a)-(c) werden im F-Zeitmodus ausgeführt, da sie nicht Teil des realen Testobjektes sind. Daher wird zeitmodus (F) benötigt.
  Der Dienst clock__init der Virtuellen Uhr wird zur Restaurierung der Zeit des Sicherungszustands benötigt, ferner zur Normierung der Virtuellen Uhr im Fall zeitlicher Unsicherheitsintervalle (5.2.4.4).

**d) Datenbereich-Basisfunktionen**

Hierunter fällt z.B. das Aktualisieren und Restaurieren von Sicherungskopien (**save/restore**). Diese und weitere Funktionen werden im nächsten Abschnitt erläutert. In Diagramm 5.20 bzw.

A.2 wird eine zusammengesetzte Operation

**rollback**

für das Zurücksetzen (Übergänge rec__msg__past in L->R und SW->R) benötigt, die folgende Einzeloperationen zusammenfaßt

- Zustandsrestaurierung (restore)
- Rücksetzen der im Vorausschau-Modus aufgetretenen Fehler
- Löschen der Ausgabe-WS (ooq__clear).

**e) Netzwerkdienste**

Das zugrundeliegende Rechnernetz muß folgende zwei Dienste anbieten:

| | | |
|---|---|---|
| **send-net** | (msg : message); | Sendeoperation |
| **rec-net** | (**out** msg : message); | Empfangsoperation |

die den von der NWS angebotenen Diensten appl-send, appl__rec gleichen. Darin kommt die Tatsache zum Ausdruck, daß die NWS keine eigentlichen Protokollfunktionen im Sinne des ISO-Schichtenmodells erfüllt, sondern eine reine Synchronisationsfunktion. Als Nutzdaten werden Marker oder Anwender-Nachrichten transportiert.

Die **Anforderungen** an send__net, rec__net sind:

- verbindungsloser Dienst, Paketübertragung zwischen beliebigen physikalisch erreichbaren Experimentrechnern (end-to-end datagram service)
- zuverlässige Übertragung, d.h. das Protokoll muß die fehlerfreie Übertragung der Pakete mit einer hohen Wahrscheinlichkeit garantieren
- Beliebige Paketlänge.

Benötigt wird also etwa der Funktionsumfang der Schicht 4 (Transportschicht).

**Problem** (Randbedingungen der Realisierung)
Die Implementierung der Dienste send__net, rec__net muß ohne Verwendung von Prozessen und Betriebssystemdiensten auskommen. Als Prozesse organisiert, würden ihre Daten und Zustände selbst der Zustandssicherung/-restaurierung unterliegen, was offensichtlich inkorrekt ist. Ein selektiver Rücksetzmechanismus, der bestimmte Prozesse ausklammert, ist zwar für die Adreßräume der Prozesse leicht zu realisieren, nicht aber für die gemeinsamen BS-Kern-Daten (vgl. 5.2.5.2). Dies ist als ein Nachteil der Zustandssicherung auf Knotenebene anzusehen.

### 5.2.5.2 Zustandssicherung und -restauration auf Knotenebene

Im folgenden wird eine mögliche Realisierung der Zustandssicherung und -restauration grob skizziert, in der folgenden segmentorientierte Aufteilung der Adreßräume eines Knotens angenommen wird (Abb. 5.21).

- Konstanten- und Codesegmente C(i) für alle Anwender- und Systemprogramme einschließlich der OLGA-NWS.
  Der Programmcode sei nicht-selbstmodifizierend; d.h. die Segmente C(i) sind von den Operationen save und restore nicht betroffen.

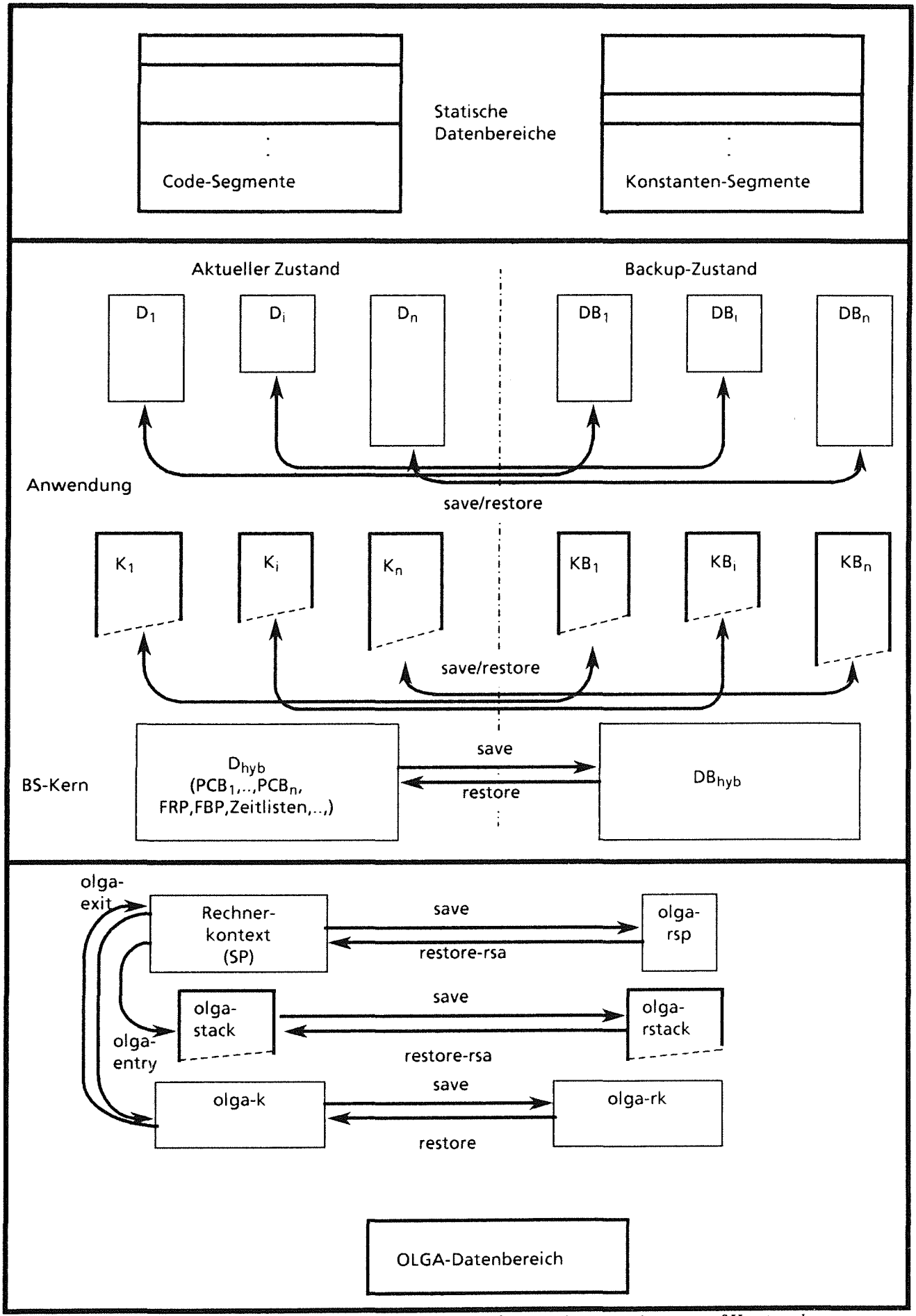



**Abb.** 5.21 Adreßraumverwaltung für Zustandssicherung / -restaurierung auf Knotenebene

- Adreßräume dynamischer Daten
- für jeden Prozeß p:
    - o Datensegment D(p)
    - o Kellersegment K(p) für temporäre Variablen von Anwenderfunktionen, BS-Funktionen und Unterbrechungsroutinen. Es können auch getrennte Kellersegmente für BS-Kern- und Anwendermodus existieren.
- Datenbereich D__hyb des Betriebssystemkerns für Prozeßleitblöcke, Prozeßlisten, Zeitlisten etc.

  Jedem dieser Adreßräume D(p), K(p), D__hyb wird genau ein Backup-Bereich DB(p), KB(p), DB__hyb zugeordnet.
- Adreßraum der dynamischen Daten der OLGA-NWS, z.B. für die Ein-/Ausgabewarteschlangen oiq und ooq. Dieses Datensegment ist nur einmal vorhanden.

I.f. sei vereinfachend ein hauptspeicherresidentes System angenommen.

**Datenbereich-Operationen**

(1) **olga-entry**
Wird eine Operation der NWS durch Aufruf des Schnittstellendienstes oder durch eine Unterbrechung aktiviert (vgl. 5.2.5.1 (1), (2)(a)-(c)), so wird die Prozessor-HW den Rechnerkontext (BZ, Register, Prozessorstatus) auf dem Benutzer- oder Kernkeller K(p) des aufrufenden oder unterbrochenen Prozesses p ablegen. Die bei Eintritt in OLGA als erstes aufgerufene Operation olga__entry
- rettet den aktuellen Rechnerkontext (insbesondere Kellerzeiger SP) in einem speziellen Bereich olga__k
- schaltet auf ein eigenes Kellersegment olga__stack um, d.h. lädt SP mit @(olga__stack)

Der Grund für diese Umschaltung: K(p) ist selbst Quelle oder Ziel der save- und restore-Operationen und soll daher nicht gleichzeitig zur Ablage temporärer Variablen während der Durchführung der Operationen von OLGA dienen.

(2) **save**
Diese Operation umfaßt folgende Schritte:

(a) Kopieren aller Daten- und Kellerbereiche
D(p)->DB(p), K(p)->KB(p), D__hyb->DB__hyb.
Insbesondere müssen auch alle Zustandsgrößen der Virtuellen Uhr (und der unterliegenden Echtzeituhr) gerettet werden, die ein späteres lesezeit- und weckzeittransparentes Wiederaufsetzen erlauben, z.B.
    - Uhrzeit (Wert von clock__read)
    - zeitliche Auflösung von R-Zeiteinträgen (Weckerintervall)
    - Zeitdistanz bis zum nächsten Weckersignal.

(b) Retten des Kontextes olga__k in einen Sicherungsbereich
olga__k -> olga__rk
Der bei Eintritt in OLGA gerettete Kontext olga__k muß beim Verlassen wieder geladen werden (Operation olga__exit (5)); im Falle einer restore-Operation ist aber der Aufrufkontext des restaurierten Sicherungspunktes, und nicht der momentane Aufrufkontext relevant. Daraus ergibt sich die Notwendigkeit von olga__rk.

(c) Retten der Rücksprungspur dieser save-Operation:
(olga__stack, SP) -> (olga__rstack, olga__rsp)
Save-Operationen können an mehreren Stellen aufgerufen werden, die eine unterschiedliche Endebehandlung und Rückkehr in die Anwendung erfordern.Beim Restaurieren eines Sicherungspunktes muß auch die Rückkehr in die Anwendung auf dieselbe Weise erfolgen wie

beim erstmaligen Passieren des Sicherungspunktes. Textuell an die Aufrufstelle der save-Operation innerhalb OLGA zurückzukehren, erscheint als die einfachste Lösung. Genau dies wird mit Hilfe von olga__rsp ermöglicht.

(3) **restore**
Zur Restauration eines Sicherungspunktes sind i.w. die inversen Operationen zu (2) auszuführen.
(a) Rückkopieren der Datenbereiche DB(p)->D(p), KB(p)->K(p), DB__hyb->D__hyb.
Restaurieren, d.h. Initialisieren (clock__init) der Virtuellen Uhr mit Hilfe der unter (2)(a) abgelegten Daten.

(b) Restaurieren des Kontextes
olga__rk-> olga__k.

Der letzte Schritt, den Aufrufkontext des Sicherungspunktes selbst zu restaurieren, erfolgt in einer separaten Operation:

(4) **restore-rsa**
Überschreiben des aktuellen Kellers olga__stack und Laden von SP:
(olga__rstack, olga__rsp) -> (olga__stack, SP)
Der abschließende Unterprogramm-Rücksprung aus restore__rsa führt dann unmittelbar hinter die Aufrufstelle der korrespondierenden save-Operation. Der Grund für die Trennung in zwei Operationen restore und restore__rsa ist, daß BS-Funktionen auf den restaurierten Betriebssystemdaten durchgeführt werden müssen. Z.B. muß ein Rücksetz-Zeiteintrag für die Zustellung empfangener Nachrichten gesetzt werden. Also muß die Operation restore zuerst aufgerufen werden. Andererseits müssen die BS-Dienste **vor** dem Umlenken des Kontrollflusses ("Sprung zur letzten save-Operation"), also textuell **vor** restore__rsa stehen.

(5) **olga-exit**
Die zu (1) inverse Operation wird bei Verlassen der NWS ausgeführt. Der in olga__k abgelegte Kellerzeiger desjenigen Prozesses, unter dessen Regie OLGA aktiviert wurde, wird restauriert. Danach erfolgt - ggf. nach einem Kontextwechsel - die eigentliche Rückkehr in einen Anwenderprozeß.

**Effizienzsteigerung bei seitenorientierter Adressierung**

Die umfangreichen Kopieroperationen in (2)(a) und (3)(a) lassen sich erheblich beschleunigen, wenn die Segmente in Seiten unterteilt sind und jeder Seite p ein Modifikator-Bit mod(p) zugeordnet ist, das vom Prozessor beim Beschreiben der Seite gesetzt und per SW gelesen und rückgesetzt werden kann. Sei va(s) Anfangsadresse, bzw. p(va(s)) erste Seite eines Segmentes s, und ve(s) Endadresse, bzw. p(ve(s)) letzte Seite von s.

Bei **Systeminitialisierung** sind zunächst aktueller Zustand und Sicherungs-Zustand auf denselben Stand zu bringen. D.h.
Für jedes Segment s des aktuellen Zustands, s' des Sicherungszustands:
Für jede Seite p ∈ [p(va(s)),..,p(ve(s))], p'∈ [p(va(s')),..,p(ve(s'))]
Kopieren Seite p->p'
Zurücksetzen mod(p)
Bei der **Save-Operation** (Schritt (2)(a)):
Für jedes Segment s des aktuellen Zustands, s' des Sicherungszustands:
Für jede Seite p∈ [p(va(s)),..,p(ve(s))]:
Falls mod(p): Kopieren Seite p->p'
Zurücksetzen mod(p)
Bei der **Restore-Operation** (Schritt (3)(a)):
Für jedes Segment s des aktuellen Zustands, s' des Sicherungszustands:
Für jede Seite p∈ [p(va(s)),..,p(ve(s))]:
Falls mod(p): Kopieren Seite p'->p
Zurücksetzen mod(p)

Die Lokalitätseigenschaften eines Modells werden damit, wenn schon nicht explizit, durch Sichern/Rücksetzen ausgewählter Prozesse, wenigstens implizit zur Aufwandsreduzierung ausgenutzt.

## 5.3 Zusammenfassung und Vergleich von FRED und OLGA

In diesem Kapitel wurden zwei neue Verfahren zur zeitlichen Synchronisation entwickelt, die prinzipiell sowohl für die verteilte discrete-event-Simulation als auch für integrierte Simulation mit realen Testobjekten tauglich sind und deren wichtigste Eigenschaften in Tab 5.2 einander gegenübergestellt sind.

| Kriterium | zeitl. enge Kopplung (FRED) | zeitl. lose Kopplung (OLGA) |
|---|---|---|
| Allgemeingültigkeit | ja | ja |
| Transparenz f. Simulations-programmierung | ja | ja |
| Reale Testobjekte (RTO)integrierbar | ja | ja |
| ASP-Bedarf pro Knoten | vergleichbar mit zentraler Sim(ZS) | doppelter dyn. ASP-Bedarf (Daten, Keller) wie ZS |
| Implementierung/Portierung | | |
| - Schnittstelle der NWS innerhalb des Systemkerns | Zeitmodusverwaltung | Protokolltreiber, Gerätetreiber |
| - HW-Unterstützung | dediziertes Komm. medium für Zeitmodus | nicht erforderlich |
| - SW-Implementierungs- und Portierungsaufwand | gering | hoch, schwer portierbar |
| Rechnernetz (-HW) als RTO integrierbar | möglich | nicht sinnvoll |
| Versionen des Systemkerns f. Simulationsexperimente/Echtzeit-umgebung | identisch | Verschieden |
| Implementierungsbedingte Interferenz | hoch | gering |

**Tab.** 5.2: Vergleich der Netzwerksynchronisationen FRED und OLGA

- Beide Verfahren sind anwendbar auf beliebige gekoppelte I/O-Systeme mit zeitdiskret veränderlichen Ein-/Ausgabesignalen.
- Ihre Synchronisationsaufgaben sind transparent für die Programmierung der Simulationsanwendungen (Sichern und Rücksetzen über R- und W-Prozesse werden ohne deren Zutun abgewickelt).
- Beide Verfahren sind speziell auf die Bedürfnisse einer Simulation mit realen Testobjekten (R-Prozessen) abgestimmt. Die Integration der R-Prozesse macht bei zeitlich lose gekoppelten Systemen (OLGA) mehr Probleme als bei zeitlich eng gekoppelten (FRED). Dies liegt an der eingeschränkten Reproduzierbarkeit von Echtzeitmessungen, wenn reale Testobjekte innerhalb eines Experimentes zurückgesetzt und ihr Ablauf wiederholt wird (5.2.4.4).

Im Speicherbedarf schneidet FRED besser ab, weil kein Mehraufwand zur Zustandssicherung dynamischer Daten anfällt, so daß der volle ASP-Ausbau des Zielrechners dem realen Testobjekt nicht nur im Einsatz, sondern bereits im Experiment zugute kommen kann.

Obwohl beide NWSen unterhalb der durch die BS-Kern-Dienste und die Menge der Geräte- bzw. Rechnernetztreiber gebildeten Schnittstelle liegen, ist ihre Rolle innerhalb des Systemkerns unterschiedlich. Während FRED direkt eine netzglobale virtuelle Zeit realisiert und daher in die **Zeitverwaltungen** der lokalen BS-Kerne integriert ist (vgl. Abb. 5.3), synchronisiert OLGA die Komponenten (Knoten) über ihre Ein-/Ausgabenachrichten und ist daher naheliegenderweise in die **Rechnernetz-** und **Gerätetreiber-Schnittstelle** der Experimentrechner integriert (Abb. 5.19).

In der **HW-Unterstützung** ist FRED wegen des dedizierten Bussystems zur Zeitmodus-Synchronisation anspruchsvoller als OLGA. OLGA setzt auf einem Standard-Kommunikationsmedium, z.B. token-ring-Bus, auf, das sowohl für die Synchronisation (Marker-Transport) als auch für die eigentliche Zielsystemkommunikation dient.
Andererseits ist die Implementierung und **Portierung** der **NWS-Software** auf unterschiedliche Zielrechner bei OLGA **weitaus** komplexer als bei FRED. Schuld daran ist das Zustandssicherungskonzept (5.2.5.2), dessen Implementierung stark von der Adreßraumorganisation der Experimentrechner-HW bzw. des BS-Kerns abhängt.

In dem Zielkonflikt zwischen der Integration **möglichst vieler** realer Testobjekte, und dem Ziel möglichst **unverfälschter** RTOe nehmen FRED und OLGA unterschiedliche Positionen ein.

- Unter FRED ist es vorstellbar, auch das physikalische Kommunikationsmedium als RTO im Experiment zu bewerten. Dagegen sind die Rechnernetz-Auslastung und die Nachrichtenlaufzeiten unter OLGA i. a. nicht auf den Einsatz in Echtzeitumgebung übertragbar. Hier bleibt dann nur die Simulation dieser Größen als Möglichkeit.
- FRED kann prinzipiell auch am realen technischen Prozeß eingesetzt werden, d. h. die System-SW-Konfigurationen für Experimente und Echtzeiteinsatz können identisch sein. Unter OLGA geht das nicht, weil die NWS Prozesse und Echtzeituhren zurücksetzt. Im Echtzeiteinsatz müssen die Aufrufe appl_send, appl_rec direkt auf die entsprechenden Rechnernetzdienste der Zielmaschine abgebildet werden.
- Andererseits hat OLGA einen entscheidenden Vorteil: das Verhalten des **vorhandenen** RTOe (R-Prozesse) ist weit besser auf den Echtzeiteinsatz übertragbar, also weniger verfälscht, als unter FRED, und zwar aus folgenden Gründen:

  - Unter FRED wird, wegen der netzweit chronologischen Reihenfolge **aller** Aktionen, der Ablauf eines R-Prozesses ständig von Aktionen der W-Prozesse auf beliebigen Rechnern unterbrochen, unter OLGA dagegen nur dann, wenn der R-Prozeß mit der simulierten Umgebung kommuniziert, oder bei Eintreffen des Markers (vgl. Abb. 5.22 : der Ablauf des RTO unter FRED ist viel stärker "zerhackt" als unter OLGA. Die Umlauffrequenz des Markers unter OLGA wird durch die Wahl des Lookahead-Timeout (5.2.4.1, (5)) begrenzt.
    Nun verursacht jede dieser Prozeß- und Zeitmodusumschaltungen implementierungsbedingt einen gewissen zeitlichen Fehler (Interferenz) des RTO, vgl. 5.1.1.3.

  - Unter OLGA besitzt jeder Rechner volle zeitliche Autonomie. Er könnte z.B. zeitliche Korrekturterme auf seine virtuelle Uhr aufschalten, um gerade die bei Prozeß-Zeitmodusumschaltungen lokal entstehenden Fehler zu kompensieren. Solche Korrekturterme sind aber nur Knoten-individuell zu setzen. Unter FRED sind aber alle

Uhren durch den globalen Zeitmodus zwangssynchronisiert. Eine netzweit einheitliche Zeit(-korrektur) wird i.d.R. nicht allen lokalen Erfordernissen zugleich gerecht werden.

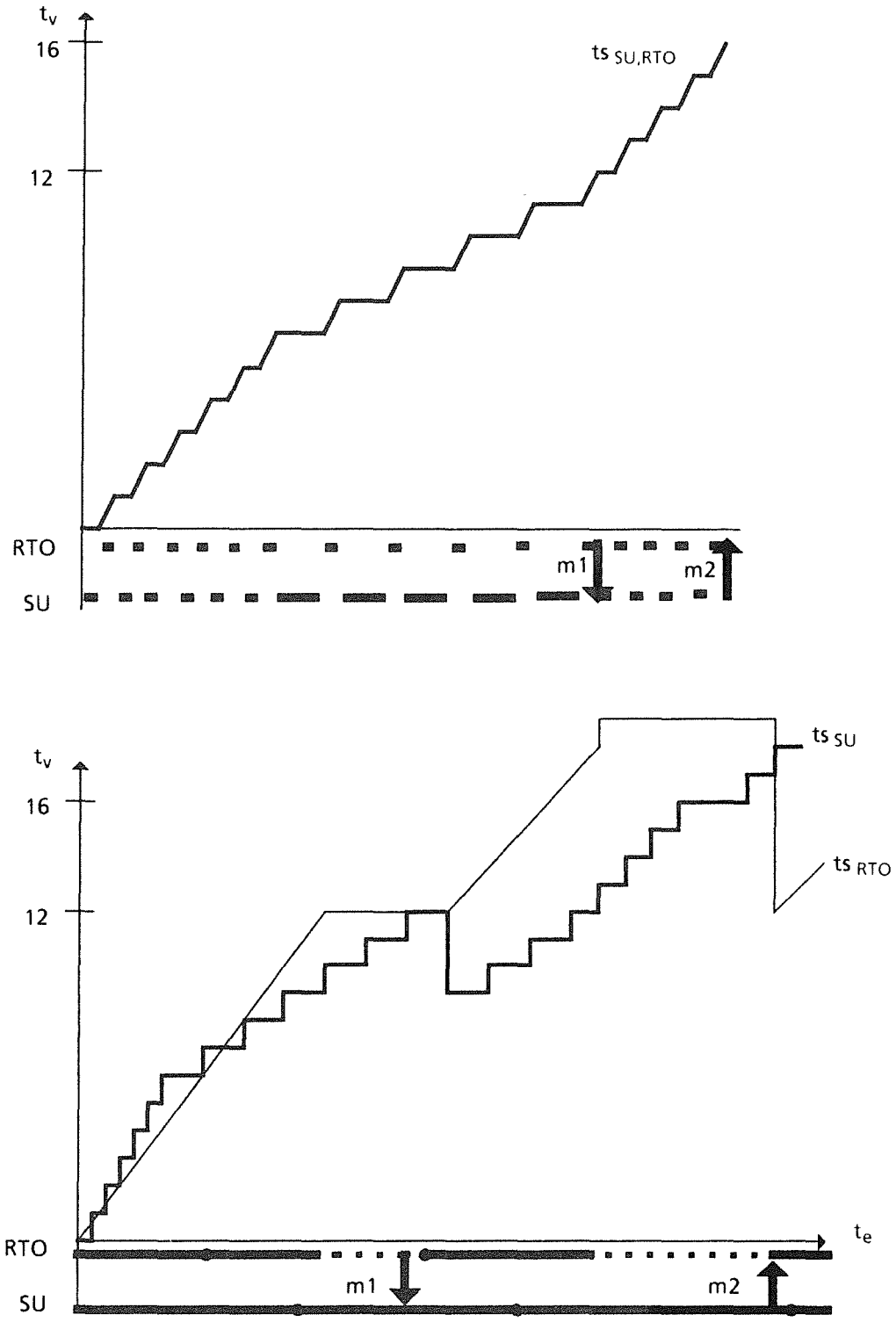



**Abb.** 5.22: Gegenüberstellung von FRED und OLGA an einem typischen Ablaufbeispiel
(SU: zeitdiskretes Simulationsmodell, 1 Ereignis pro Zeiteinheit
(m1, 12),
(m2, 16): Nachrichtenkommunikation zwischen RTO und SU)
- Zustandssicherungspunkte

## 6. Architektur verteilter Modellierungs- und Zielsysteme

Der Hauptteil der Arbeit hat sich mit der Entwicklung einer verteilten Laufzeitumgebung befaßt, welche sowohl als prozeßorientierte Simulationsschnittstelle, als auch als Echtzeit-BS-Kern verwendbar ist. Hauptanwendungen für eine solche Laufzeitmaschine sind

a) der Aufbau verteilter, i.a. heterogener Modellierungssysteme
b) die schrittweise Verfeinerung solcher Modellierungssysteme zu verteilten Zielsystemen (PFSen) und ihre interferenzarme Leistungsbewertung und Test.

In diesem Kapitel soll auf einige zur Modellierung von Prozeßführungssystemen a) benötigte typische Modellbausteine beispielhaft eingegangen und ihre Schnittstellen beschrieben werden. Es geht dabei um eine Konkretisierung von Abschnitt 3.4.5, Abb. 3.13, insbesondere des Blocks "Modellierungssysteme".
Ferner ist die Frage zu klären, ob die funktionellen Architekturen (verteilter) Simulationssysteme und (verteilter) Echtzeitsysteme tatsächlich deckungsgleich sind, anders ausgedrückt: ob der Ansatz, Simulations- und Zielsysteme bzw. Mischungen beider auf einem gemeinsamen verteilten BS-Kern aufzubauen, auch aus funktionell höherer, anwendungsorientierter Sicht sinnvoll ist. Diese Punkt b) betreffende Frage berührt eine Vielzahl von Teilgebieten, z.B. Netzwerk-Betriebssysteme, anwendungsorientierte Kommunikationsprotokolle, verteilte Echtzeit-Programmiersprachen, Simulations- und Konfigurationssprachen für verteilte Modellierungssysteme (vgl. /MUE 86//LOU 85//LIU 87/). Daher kann hier auch keine endgültige Klärung der Frage erwartet werden, sondern die Problematik soll anhand einiger Beispiele beleuchtet werden.

### 6.1 Höhere Schichten des Betriebssystems

Die für das dynamische Verhalten eines Testlings entscheidenden Konzepte und Eigenschaften (hybrides Zeitkonzept, Ablaufsemantik in virtueller Zeit, Kopplung zwischen realem Testobjekt und simulierter Umgebung) werden im zielorientierten Simulationsansatz auf der funktionell untersten Ebene, auf der Basis der Prozesse, Kommunikationsobjekte und Dienste von BS-Kernen, definiert und implementiert. Im wirtsorientierten Ansatz sind diese Simulationskonzepte mit den Objekten der Modellierungssprache, also einer i.a. funktionell höheren Ebene verknüpft. Die Modellierungssprache setzt zwar auch auf einem Betriebssystem auf, dieses besitzt aber keine für das zeitliche Verhalten der Modelle relevanten Eigenschaften.

Komplexere, durch mehrere Schichten realisierte Dienstleistungen werden auch im zielorientierten Ansatz benötigt. Es stellen sich daher folgende Fragen:

a) Übertragen sich die Konzepte der zielorientierten Simulation leicht von der BS-Kern-Ebene auf funktionell höhere Ebenen?
b) Läßt sich die Implementierung dieser höheren Funktionen nach **bewährten** Architekturkonzepten (ohne grundsätzliche Neuentwicklungen) durchführen?

c) Lassen sich, weitergehend als b), identische Realisierungen für die höheren Funktionen angeben, die sowohl von reinen Modellierungsanwendungen als auch von Echtzeitanwendungen benutzt werden (Ökonomie-Aspekt)?

Es geht im Grunde darum, ob man das Zeitkonzept (virtuelle Zeit, Echtzeit, hybride Zeit) tatsächlich als orthogonal zur funktionellen Systemarchitektur ansehen kann. Das soll anhand höherer BS-Dienste als Beispiel für funktionelle Architektur etwas näher erläuter werden.

### 6.1.1 Duale nichtkommunizierende Treiber-Hierarchien

Der einfachste, aber in der Praxis wichtigste Fall ist der, daß höhere BS-Dienste (z.B. Dateiverwaltung, Datenbankdienste, höhere Netzwerkdienste) durch eine synchrone Anforderungs-/Rückmeldungs-Schnittstelle angefordert werden (z.B. remote procedure call).

```
req          (d, par_1,..,par_n);    d : Dienst (BS-Funktion)
wait_reply   (d, res_1,..,res_m);    par_1,..,par_n : Auftragsparameter
                                     res_1,..,res_m : Ergebnisparameter
```

Diese Dienste seien im Zielsystem nach folgendem Architekturmodell realisiert:

(1) Es existiert eine Hierarchie von Treiberinstanzen TR(i,1),..,TR(i,n(i)) (Schichten i = 1,..,k); Schicht 0 bestehe dabei aus physikalischen E/A-Geräten (z.B. Hintergrundspeicher-Controller und Laufwerk, Übertragungskanal), Schicht 1 aus Gerätetreibern.

(2) Jede Instanz TR(i,j) besitze eine Auftrags-Mailbox MQ(i,j), über die Dienste durch die höheren Schichten angefordert werden, und eine Rückmeldungs-Mailbox MR(i-1,k) zur Übernahme der Ergebnisse von Diensten tieferer Schichten.

(3) Jede Instanz werde durch einen oder mehrere zyklische Prozesse realisiert, von denen jeder
- an MQ(i,j) auf einen Auftrag (Dienstanforderung) wartet
- die internen Daten der Instanz aktualisiert
- untergeordnete Instanzen TR(i-1,k) beauftragt und auf die Ergebnisse wartet (sequentiell oder nebenläufig)
- das Auftragsergebnis an MR(i,s) (s Auftraggeber-Instanz) zurückliefert.

(4) Ferner sollen folgende Annahmen gelten:
- die Treiber-Instanzen enthalten keine expliziten Zeitverzögerungen (delays)
- Sie kommunizieren nur innerhalb der Treiber-Hierarchie, d.h. ihre externen Schnittstellen sind durch (3) vollständig beschrieben.

Die Aufgabe besteht darin, ihre Dienste sowohl für reale Testobjekte ('R-Anwendungen') als auch für Simulationsmodelle ('W-Anwendungen') nutzbar zu machen, zunächst unter der vereinfachenden Annahme, daß keine problembedingten logischen und zeitlichen

**Abhängigkeiten**, insbesondere kein Informationsaustausch, zwischen R- und W-Anwendungen über diese Dienste realisiert werden soll.

**Beispiele**:

- Dienste in Kommunikationsprotokollen, mit Hilfe derer W-Anwendungen und R-Anwendungen über getrennte logische Verbindungen, aber einen gemeinsamen physikalischen Übertragungskanal kommunizieren, jedoch nicht R- mit W-Anwendungen kommunizieren.
- Dateiverwaltungs- oder Datenbankdienste, über die R- und W-Anwendungen auf (bis auf einen möglichen gemeinsamen lesenden Zugriff) getrennte Datenbestände zugreifen, die auf demselben physikalischen Datenträger abgelegt sind. Virtuelle Speicherverwaltung ist ein wichtiger Spezialfall.

Diese Aufgabe ist durch folgende Konfiguration leicht lösbar (Abb. 6.1):

- Alle höheren Treiberinstanzen TR(i,j) für $i>1$ sind doppelt instantiiert als R-Treiber $TR_R(i,j)$ und $TR_W(i,j)$, wobei $TR_{R[W]}(i,j)$ über R-[W-]Kommunikationsobjekte nur mit Nachbarinstanzen **seiner** Klasse $TR_{R[W]}(i-1,k)$,$TR_{R[W]}(i+1,l)$ kommuniziert. Jede Instanz besitzt ferner private, zur Partnerinstanz gleicher Ebene disjunkte lokale Datenobjekte (z.B. Zustandsinformation).
- Die Instanzen auf Ebene 1 (Gerätetreiber) kommunizieren paarweise über ein gemeinsames Vorschaltgerät und physikalisches E/A-Gerät wie in 4.2.5.5 beschrieben.

Diese Lösung überträgt die folgenden wesentlichen Eigenschaften gemeinsam benutzter Hardware-Betriebsmittel auf höhere logische Betriebsmittel:

- SW-Entwurf und Implementierung sind bis auf die Existenz getrennter (R- und W-)Dienstleistungsprozesse **identisch**.
- Die gesamte Treiber-Hierarchie der $TR_R(i,j)$ wird in vollem Umfang als reales Testobjekt bewertet, und ihre zeitliche Dynamik ist unabhängig von der Inanspruchnahme derselben Diensten durch W-Anwendungen über $TR_W(i,j)$.
- Die Inanspruchnahme von Diensten durch W-Anwendungen ist zeitlich transparent. Für einen beliebigen W-Prozeß p gilt stets

```
{tv(p)=T}
    req          (d, par_1,..,par_n);
    wait_reply   (d, res_1,..,res_m);
{tv(p)=T}
```

  unabhängig von der Komplexität des Dienstes d und von der tatsächlichen Belastung der bearbeitenden Treiber-Instanz $TR_W(i,j)$. Dies folgt induktiv über die Anzahl i der Schichten und die Anzahl der Dienstaufrufe in jeder Schicht i:

  $i=1$: physikalische E/A-Aufträge sind nach 4.2.5.5 für W-Prozesse zeitlich transparent

  $i>1$: Bei der Ankunft einer Dienstanforderung in MR(i,j) sind eine unbekannte Anzahl von Diensten auf gleicher Schicht bereits in Bearbeitung (nach Induktionsvoraussetzung zeitlich transparent). Zur Bearbeitung des aktuellen Dienstes d werden nur Dienste tieferer Schichten angefordert und die Ergebnisse erwartet (nach Induktionsvoraussetzung zeitlich transparent), oder lokale Rechenoperationen ohne delays ausgeführt (zeitlich transparent) und schließlich die Ergebnisse an den

Aufrufer zurückgemeldet. Da dieser zum virtuellen Zeitpunkt seines Dienstaufrufes bereits wartet, kann auch hier die Zeit nicht fortschreiten.

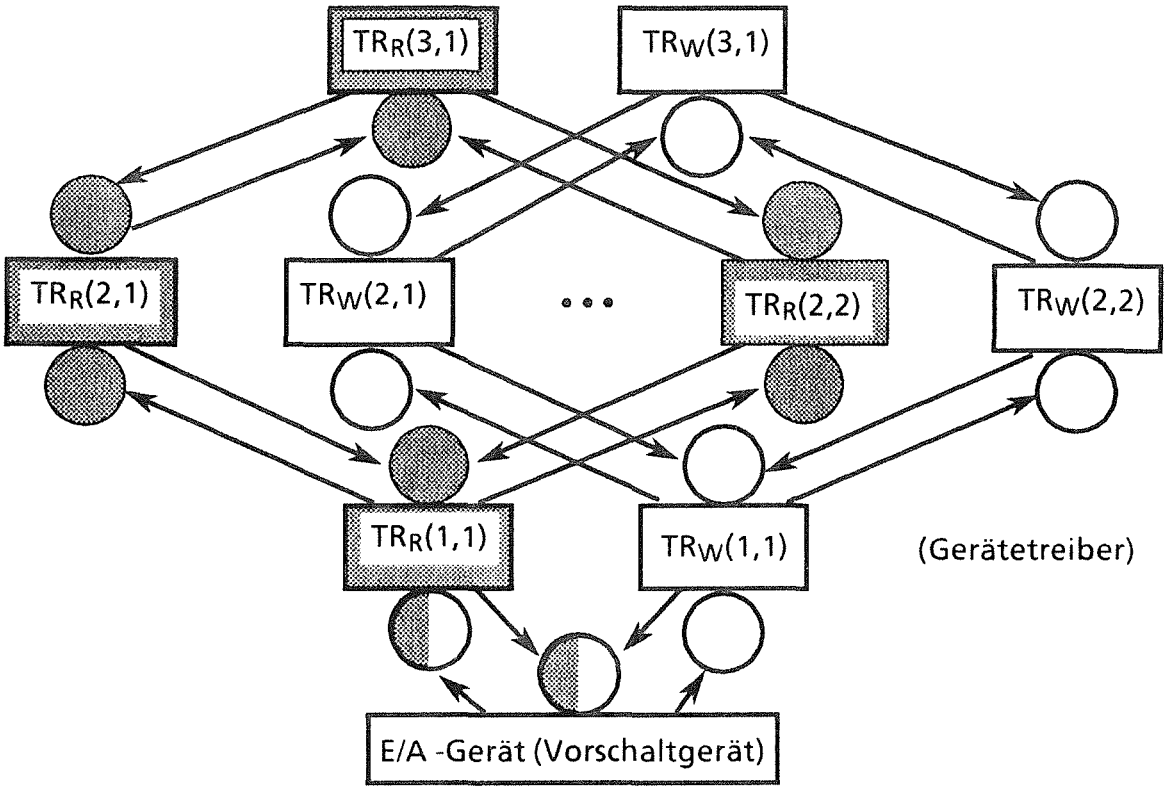



**Abb.** 6.1: Duale nichtkommunizierende Treiber-Hierarchien

### 6.1.2 Duale kommunizierende Treiber-Hierarchien

Falls im Unterschied zu 6.1.1 eine echte logische Abhängigkeit zwischen R- und W-Anwendungen über die höheren BS-Funktionen besteht, entsteht eine Treiberhierarchie wie in Abb. 6.2 dargestellt, in der die Partnerinstanzen $TR_R(i,j)$, $TR_W(i,j)$ zusätzlich über RW-Kommunikationsobjekte MS(i,j) miteinander gekoppelt werden müssen, um sich zu synchronisieren, z.B. einen gegenseitigen Ausschluß zu erreichen.

**Konsequenzen**

- Die Treiber-Instanzen für Simulationsexperimente und Einsatz unterscheiden sich (zumindest durch die zusätzlichen Kommunikationsobjekte MS(i,j)), d.h. es entsteht zusätzlicher Realisierungsaufwand.
- Durch W-Anwendungen angeforderte Dienstleistungen sind i.a. nicht zeitlich transparent.

**Beispiel**
Wenn ein W-Prozeß zur Zeit T eine Leseoperation auf einer Dateneinheit D1 startet, welche durch einen R-Prozeß gesperrt ist, der gerade eine Änderung durchführt, so **kann** die Leseoperation zeitlich gar nicht transparent sein. Denn die Änderung durch den R-Prozeß nimmt i.a. ein virtuelles Zeitintervall [T1,T2] mit T2>T1 in Anspruch, und jede nach T1 begonnene Leseoperation muß mindestens bis T2 warten, um einen konsistenten Zustand von D1 zu erhalten.

Kooperierende Treiber-Hierarchien in Abb. 6.2 entstehen dann, wenn das reale Testobjekt eines Rechners nur unvollständig durch R- oder S-Prozesse repräsentiert ist. In Abb. 6.3 (linke Seite) greifen z.B. R- und W-Prozesse direkt auf gemeinsame, in einer lokalen Datenbasis eines Experimentrechners K abgelegte Daten zu. Entweder die W-Prozesse repräsentieren in Wirklichkeit selbst R-Prozesse desselben Knotens K, oder sie stehen für rechnerexterne Benutzer, die in K zumindest eine "Ersatzinstanz" (vgl. /DRW 87/, in Abb. 6.2 als File-Server angedeutet) benötigen, um zugreifen zu können. In beiden Fällen fehlen also gewisse Prozesse des realen Testobjektes dieses Knotens. Würde man sie zumindest als S-Prozesse hinzunehmen, so wäre die Schnittstelle zwischen R-, S- und W-Prozessen identisch mit der Geräteschnittstelle (Kommunikationsmedium), d.h. auf den Fall 6.1.1 reduziert. Vertritt man dagegen - Abb. 6.2 links vor Augen - den Standpunkt, daß die **Realisierung** einer **konkreten** Verteilung in diesem Stadium noch gar nicht interessiert, so ist die Spezifikation als R-Prozesse verfrüht und stattdessen W-Prozesse angebrachter. Es scheint also, daß Konstruktionen wie in 6.1.2 vermeidbar sind, ohne bei der Verfeinerbarkeit der Simulationssysteme zu realen Testobjekten Wesentliches zu verlieren.

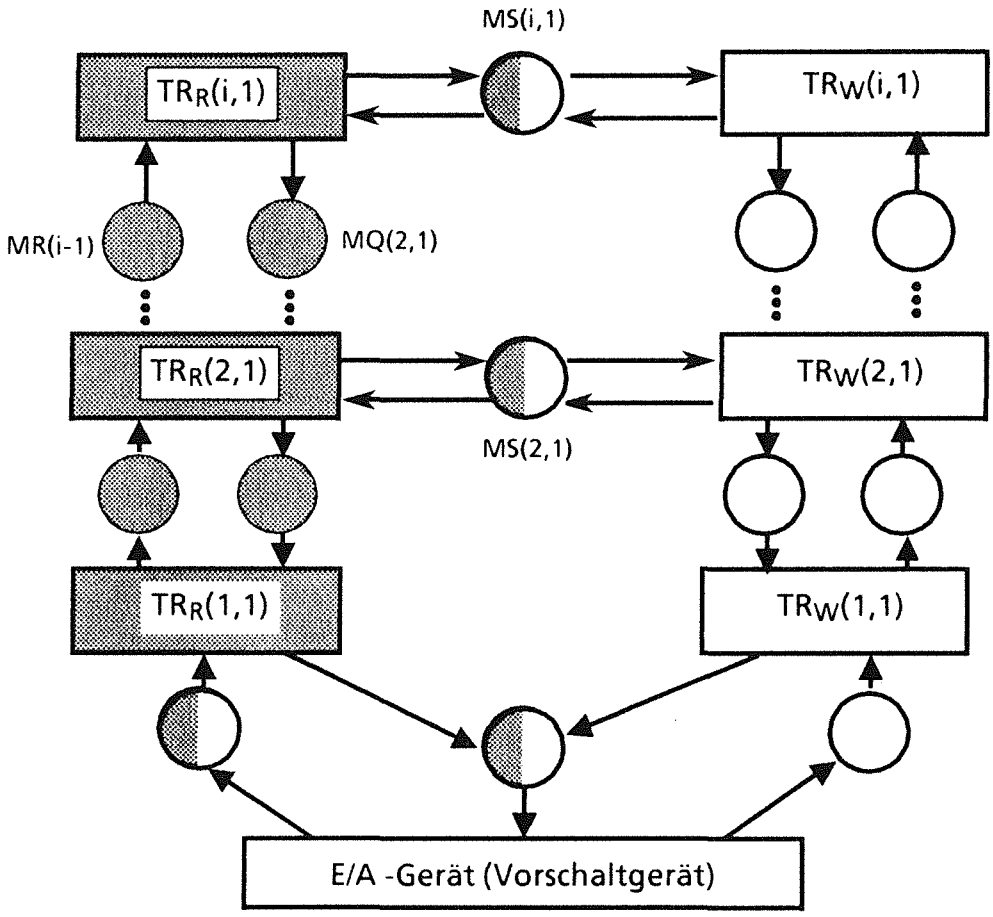



**Abb.** 6.2: Duale kommunizierende Treiber-Hierarchien

### 6.1.3 Netzwerk-BS-Dienste (Beispiel: abgesetztes Senden an Mailbox)

Ein wichtiger Fall höherer Dienste sind Netzwerk-BS-Dienste, d.h. Dienste, die auf BS-Kern-Ebene bereits existieren und nun netzweit für R- und W-Prozesse angeboten werden sollen. Diese sind, anders als in 6.1.1, für W-Prozesse nicht notwendig zeittransparent, aber ihre funktionelle und zeitliche Semantik soll für W-Prozesse in jedem Fall **dieselbe** bleiben, ob lokal oder netzweit angewandt. Als Beispiel soll z.B. der Dienst send__msg (Kap. 4.1.3) auf **netzglobale** Kommunikationsobjekte erweitert werden. Dieser Dienst remote__send ist zeitlich nicht transparent für einen W-Prozeß, wenn die Mailbox begrenzte Kapazität besitzt und buffer__option = b spezifiziert ist. Er wird für die lokationstransparente Verteilung von Modellkomponenten auf beliebige Rechner benötigt.

Remote__send an eine globale Mailbox mbx__g kann für W-Prozesse (Abb. 6.4) im Prinzip genauso wie für R-Prozesse realisiert werden :

- Durch das Prinzip des Fernaufrufs (remote-procedure-call) wird der Aufrufer (W-Prozeß) solange blockiert wie nötig.
- Es werden folgende Hilfsprozesse (alles W-Prozesse) eingesetzt:
  - Ein **Katalog-Prozeß**, der die Zuordnung
    globaler Mailbox-Namen(mbx__g) →(Knoten(k__mb),lokaleMailbox(mbx__l))
    trifft und im Fall globaler Mailboxes eine Dienstanforderung (req__send) an den Rechner k__mb sendet. Die Zuordnung zur Ziel-Mailbox kann in mbx__g explizit enthalten sein, falls die Allokation von Mailboxes auf Rechner bereits zur Übersetzungszeit feststeht, oder sie wird dynamisch über verteilte Namensdienste ('name server' /PAC 84//VAS 84/) ermittelt.
  - Ein **Vermittlerprozeß**, der nichtblockierende Systemdienste lokal aufruft bzw. Rückmeldungen ausgeführter Systemdienste verarbeitet.
  - Ein **Surrogatprozeß**, der bei abgesetzten blockierten Systemdiensten durch den Vermittlerprozeß dynamisch erzeugt wird, anstelle des Aufrufers wartet und die Ergebnisse (reply) an diesen zurücksendet.

Der Surrogat-Prozeß ist konzeptionell als "Dienstnehmer-Ersatz" einzustufen /DRW 87/, der den modellbedingten Wartezustand an der Mailbox so wie der aufrufende W-Prozess p erfährt, während die Aktionen aller übrigen Hilfsprozesse zeitlich transparent sind. Eine effizientere Implementierung kommt auch ohne Surrogat-Prozeß aus, wobei der Vermittlerprozeß bei blockierenden Systemdiensten disjunktiv auf deren Ende und auf neue Systemdienstanforderungen reagiert.

Bei Verwendung der OLGA-Netzwerksynchronisation als "Transportmedium" wird übrigens jeder der beteiligten Knoten bei der Abwicklung von send__msg einmal zurückgesetzt, der Knoten k__mb durch die Systemdienstanforderung, der Knoten k durch die Rückmeldung.

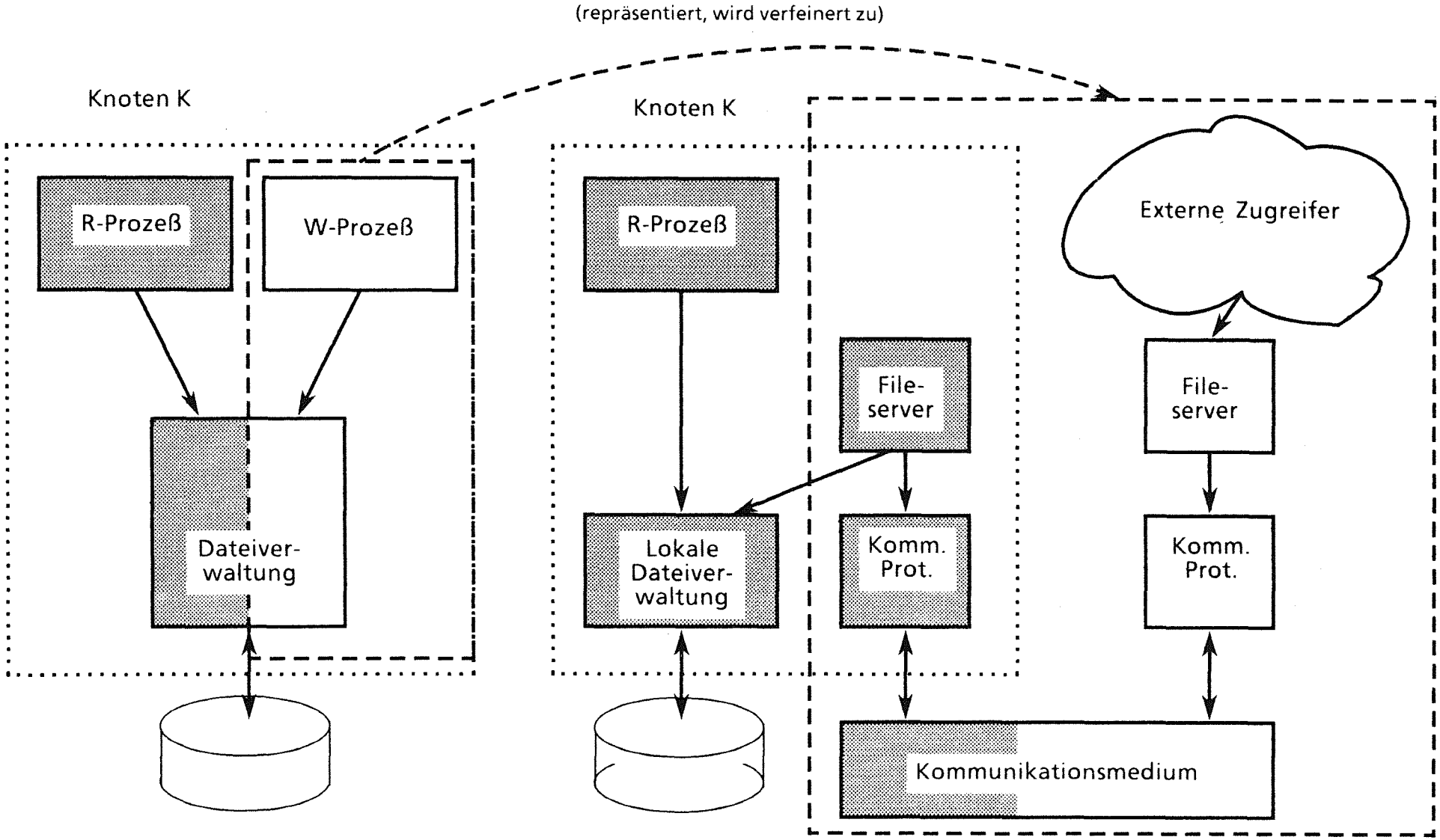



**Abb.** 6.3 Unvollständiges reales Testobjekt

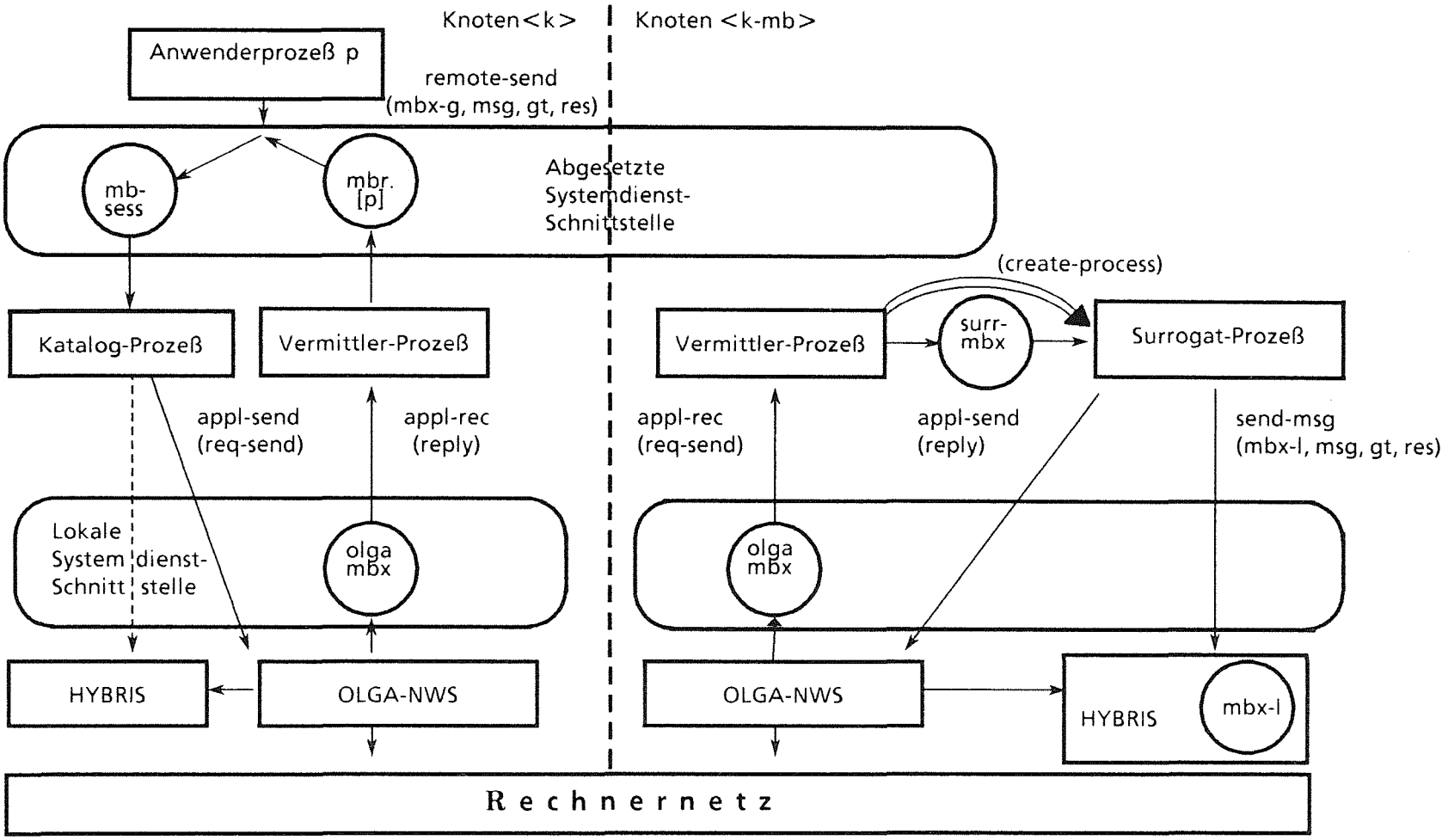



**Abb.** 6.4 Beispiel für Netzwerk-BS-Dienst (abgesetzte Mailbox-Kommunikation)

### 6.1.4 Protokoll-Hierarchien

Auch die Implementierung von Kommunikationsprotokollen in verteilten Systemen ist ein Beispiel für das Architekturmodell der Treiberhierarchien von 6.1.1. In Bezug auf die Leistungsbewertung gibt es wieder die Extremfälle

a) Kommunikations-SW, die nur zum Informationsaustausch zwischen Simulationsmodellen dient und selbst zeittransparent abläuft,
b) Kommunikations-SW als Teil des realen Testobjektes.

Gegenüber 6.1.1 ist die Situation aber insofern komplizierter, als eine vollständige reale Bewertung nicht möglich ist, sondern **zusätzlich Simulationsmodelle** benötigt werden. Dies gilt jedenfalls bei Verwendung der zeitlich lose gekoppelten NWS (OLGA). Die realen Übertragungszeiten im Netz unter OLGA sind nicht auf den Echtzeiteinsatz übertragbar (vgl. 5.3), letztere müssen durch ein Simulationsmodell des Kommunikationsmediums nachgebildet werden. Dieses Modell wird als 'virtuelles LAN' (v__lan) bezeichnet und seine Schnittstellen in 6.2.2.1 beschrieben. Kommunikations-SW als reales Testobjekt erfordert also zusätzlich zum realen Datenaustausch im verteilten Testbett die Übertragungszeit-Simulation durch v__lan. Beide benötigen selbst mindestens die Transportdienste der OLGA-NWS oder -besser- die netzweite Mailbox-Kommunikation nach 6.1.3, d.h. zeittransparente Dienste (a). Instanzen, die funktionell Aufgaben derselben Protokollschicht wahrnehmen, sind daher übereinander angeordnet (replizierte Schichten, Abb. 6.5), und nicht spiegelbildlich nebeneinander wie in Abb. 6.1.

Abb. 6.5 gibt ein anschauliches Beispiel der Datenflüsse bei realer, simulierter und zeittransparenter Kommunikation. 3 Experimentrechner seien über ein gemeinsames LAN gekoppelt, 2 von ihnen (E1,E2) enthalten ein reales Testobjekt einschließlich Kommunikations-SW bis zur Verbindungsschicht 2 (link layer bzw. media access layer). Die Zugangspunkte zum physikalischen Rechnernetz in Schicht 2, d.h. zum Senden und Empfangen von Datenpaketen an den vorgegebenen Ein-/Ausgabe-Ports, sind im Experiment ersetzt durch die Dienste transmit__vlan, receive__vlan des Simulationsmodells v__lan, so daß die Datenpakete auf dem "Umweg" über v__lan (auf E3) von E1 nach E2 übertragen werden. Dabei wird die globale Mailbox-Kommunikation und der Transportdienst der OLGA-NWS (Schicht $\geq 4$) benutzt und ihre Übertragungszeit simuliert.
In Abb. 6.5 dient v__lan neben der Kommunikation des realen Testobjektes auch der simulierten Nutzung von Kommunikationsdiensten, hier durch 2 W-Prozesse PW1, PW2, die zwei virtuellen Rechnerknoten V1 und V2 zugeordnet sind. Anders als bei PR1, PR2 wird das Übertragungsprotokoll zwischen V1 und V2 nicht real benutzt und auch nicht explizit modelliert, sondern die Übertragungszeit wird nur pauschal durch v__lan berücksichtigt.
Die W-Prozesse PW3,PW4 schließlich stehen für "abstrakte" Prozesse (z.B. Teilprozesse eines technischen Prozesses), deren Interaktion keiner Interrechner-Kommunikation entspricht und daher zeitlich transparent ist (v__lan wird bei der Kommunikation umgangen).

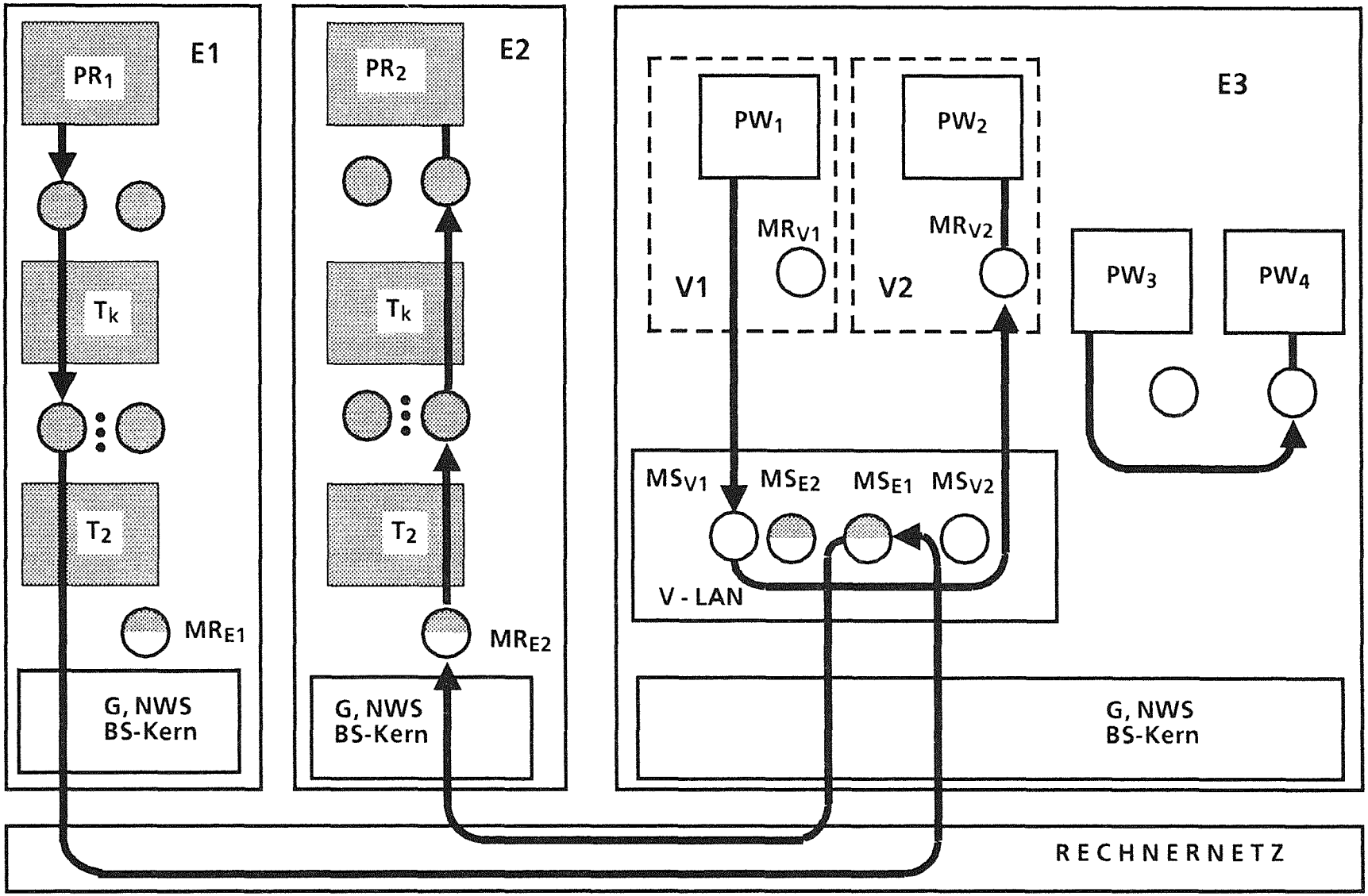



**Abb.** 6.5 Informationsflüsse bei realer, simulierter und zeittransparenter Kommunikation

## 6.2 Beispiele vorgefertigter Modellbausteine

Der im Hauptteil der Arbeit (Kap.3 bis 5) entwickelte Systemkern stellt für sich noch kein Simulationssystem dar; hierzu muß der Block "Modellierungssystem" in 3.4.5 mit Inhalt gefüllt, d.h. Modellbausteine bereitgestellt werden, die häufig wiederkehrende Simulationsaufgaben mit geringem Anpassungsaufwand zu lösen gestatten. Unterschiedliche Anwendungen des Kernsystems erfordern i.a. auch unterschiedliche Modellierungssysteme.

I.f. wird beispielhaft auf einige der zur Leistungsbewertung von PFSen wichtigsten Komponenten eingegangen:

- Modellkomponenten technischer Prozesse mit kontinuierlichem oder zeitdiskreten Zustandsverlauf
- Modellkomponenten einer virtuellen Rechnerumgebung, insbesondere
  - Modell der HW-Betriebsmittel eines Rechnernetzknotens (CPU, ASP, lokale E/A-Geräte)
  - Modell eines Rechnernetzes (Kommunikationsmedium).

Dabei wird nur auf die wesentlichen Schnittstellen eingegangen

- von den Komponenten bereitgestellte Operationen zur Laufzeit
- Hinweise auf benötigte Dienste des Kernsystems
- Schnittstellen, die die anwendungsspezifische Parametrierung der Komponenten betreffen, also für den Simulationsprogrammierer oder für übergeordnete Modellgenerierungssysteme wichtig sind.

### 6.2.1 Kontinuierlich-diskrete Prozeßelemente

I.f. werden kontinuierliche (Teil)Prozesse diskutiert, die zusätzlich diskrete Zustandsübergängen enthalten (vgl. 3.1.1); rein zeitdiskrete Prozesse können durch Spezialisierungen und Vereinfachungen des Bausteins erhalten werden.

**(1) Mathematische Beschreibung**

**Kontinuierlicher Teil**

Der Zustandsverlauf x(t) eines Prozeßelementes (PE) sei gegeben als Lösung eines Systems gewöhnlicher, nichtlinearer Differentialgleichungen 1ter Ordnung (vgl. Abb. 3.1 in 3.1.1 hinsichtlich der Bezeichnungen, Dimensionen etc).

$$x' = F(x, u, \xi, x_1,..,x_j)$$

mit

x Zustandsgrößen ($\in X$)

u Stellgrößen ($\in U$)

$\xi$ externe Störgrößen ($\in$ XS)

$x_1,..,x_j$ Kopplungsgrößen von anderen PEen 1,..,j ($\in X_1 \times .. \times X_j$)

F: $X \times U \times XS \times (X_1 \times ... \times X_j) \rightarrow X$

erfülle die lokale Lipschitz-Bedingung zur eindeutigen Lösbarkeit des Anfangswertproblems für jeden festen Parameterwert $u,\xi,x_1,...,x_j$. Der Zustandsraum X und alle Eingangsgrößen des PE seien i.f. zu

$XI := X \times U \times XS \times X_1 \times ... \times X_j$

zusammengefaßt.

$y = Fy(x, u, \xi, x_1,..,x_j)$

$z = Fz(x, u, \xi, x_1,..,x_j)$

mit

Fy: $XI \rightarrow Y$ Meßgrößenfunktion

Fz: $XI \rightarrow Z$ Regelgrößenfunktion

seien die direkt meßbaren Größen bzw. die Regelgrößen (Leistungsgrößen) des PE. Schließlich sei

$XO \subset X$

der Teil der Zustandsgrößen, der als Kopplungsgrößen wiederum andere PEe beeinflußt.

**Diskreter Teil**

Auf dem Definitionsbereich XI von F ist zusätzlich eine u.U. leere Menge von Prädikaten bzw. "Zustandsflags"

$P_i : XI \rightarrow \{true,false\} (1 \leq i \leq np)$

definiert. Jedem Wert $xi \in XI$ wird ein **diskreter Begleitzustand**

$xd \in XD := \{0,1\}^{np}$

durch

$d: XI \rightarrow XD$ mit $d(xi)_j = 1 \Leftrightarrow P_j(xi)$ $(1 \leq j \leq np)$

zugeordnet. Die Urbilder der Elemente von XD partitionieren den Zustandsraum XI in $2^{np}$ disjunkte Bereiche, die zum Teil auch leer sein können, wenn bestimmte Zustandsflag-Kombinationen unmöglich sind (Abb. 6.6).

Die wichtigsten der üblicherweise bei kontinuierlich-diskreter Prozeßsimulation (vgl GPSS /SCH 84/, DISCO /HEL 80/, SAMOA /ILU 84/, GASP VI /RIC 82/, SLAM /ORY 85/) vorkommenden diskreten Ereignisse können als Spezialfälle durch solche Prädikate einheitlich dargestellt werden, u.a.

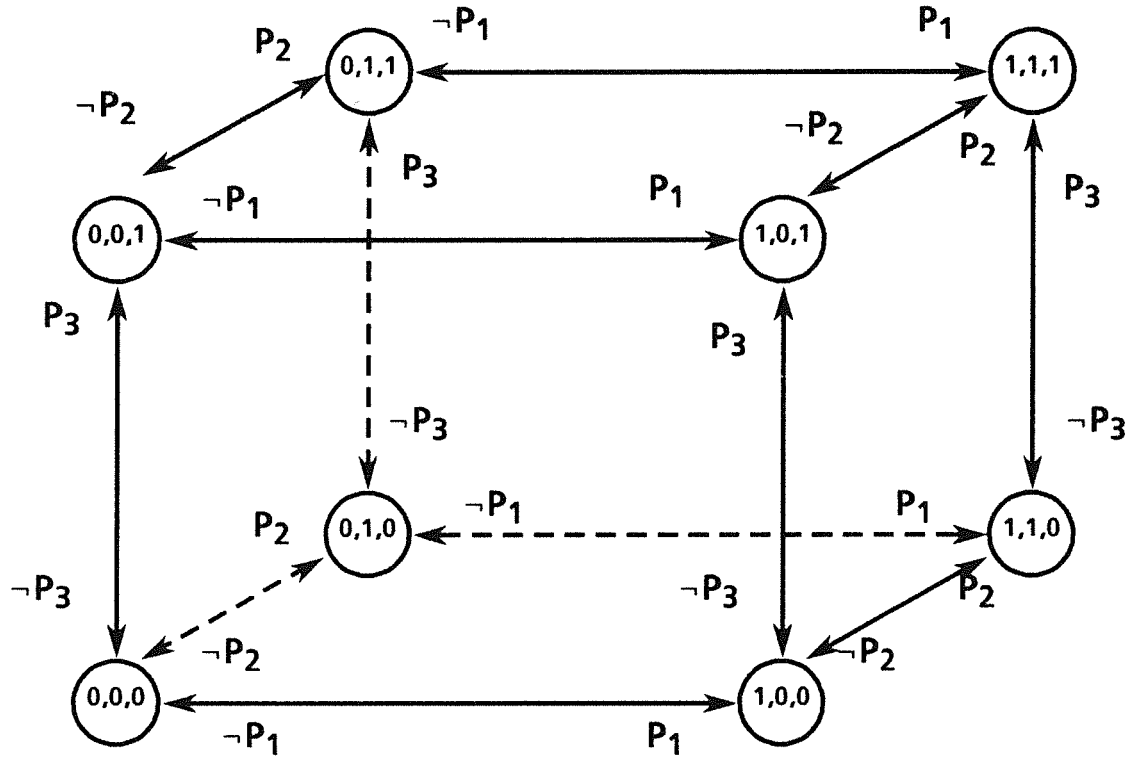



**Abb.** 6.6: Übergänge des diskreten Begleitzustandes

- Grenzwertüberschreitungen, auch 'Zustands-crossings' genannt, z. Bsp.
  - Die Ofentemperatur x1 übersteigt 1200Grad C (P1 ≡ (x1 > 1200))
  - Der Druck x1 von Behälter B1 übersteigt den Druck x2 von B2 (P2 ≡ (x1 > x2)).
- "Diskontinuitäten" der Systemgleichung, d.h. der Zustandsverlauf wird in unterschiedlichen **Arbeitsbereichen** des technischen Prozesses durch unterschiedliche DGS-Systeme bzw. Systemparameter beschrieben:

$$F(xe) := \begin{cases} F_1(xi) & \text{falls } d(xi) = (0,0,..,0) \\ \ldots & \ldots \quad \ldots \\ F_{2np}(xi) & \text{falls } d(xi) = (1,1,..,1) \end{cases}$$

- Zustands-Crossings, deren Definition selbst vom Arbeitsbereich abhängt
- Explizit zeitabhängige Bedingungen (der Zustand x kann insbesondere eine zeitliche Komponente enthalten).

Sowohl Änderungen des diskreten Begleitzustands (Abb. 6.6) als auch Änderungen externer Eingangsgrößen, z.B. Stellgrößen, bewirken i.a.

- sprungförmige Änderungen des kontinuierlichen Zustandsverktors x (wird, wie in Bsp. (4) von 5.2.1.3, eine Kugel elastisch an einer Wand reflektiert oder stößt sie mit einer anderen Kugel zusammen, so ändert sich ihr Geschwindigkeitsvektor sprungförmig).

- Änderungen der Ausgangsgrößen xo∈XO (z.B. Alarmausgabe bei Grenzwertüberschreitung, signifikante Änderung einer Kopplungsgröße für ein beeinflußtes PE).

Diese Einflüsse sind in den autonomen bzw. externen Zustandsübergangsfunktionen δ__a bzw. δ__u zusammengefaßt:

$$\delta\_\_a : XI \rightarrow X \times XO$$

$$\delta\_\_a(xi) := \begin{cases} \delta\_\_a_1(xi) & \text{falls } d(xi) = (0,0,..,0) \\ \ldots & \ldots \\ \delta\_\_a_{2^{np}}(xi) & \text{falls } d(xi) = (1,1,..,1) \end{cases}$$

$$\delta\_\_u : XI \rightarrow X \times XO$$

**(2) Lösungsvorschrift**

Zur Lösung des kontinuierlichen Zustandsverlaufs werden Standardverfahren der numerischen Integration (explizite Einschrittverfahren, z.B. Runge-Kutta 4.Ordnung, mit dynamischer, fehlerabhängiger Schrittweitensteuerung) eingesetzt. Nach jedem Integrationsschritt ist zu prüfen, ob im nächsten Schritt bei unveränderten Eingangsgrößen eine Änderung des diskreten Begleitzustands xd stattfindet und ggf. diese zeitlich zu lokalisieren (Zeitpunkt t*). Nach dem verkürzten Integrationsschritt bis t* wird die autonome Übergangsfunktion δ__a angewandt. Dies resultiert in einem neuen Zustand x, ggf. einer Ausgabe xo an andere PE'e, und u.U. auch in einer Änderung der Differentialgleichung (d.h. mit t* beginnt ein neues AWP, vgl. Abb. 6.7a).
Sinngemäß ist bei einer Änderung externer Eingangsgrößen zum Zeitpunkt t* zu verfahren: es wird nur bis t* integriert und δ__u angewandt (Abb. 6.7b).

x(t) Lösung des AWP: $x'(t) = F(x(t),u)$, $x(t_0) := x_0$;

$t^* = \sup\{t | d(x(t),u) = d\}$

$x^* = x(t^*_{+0})$ ; $d^* = d(x^*)$

$(x_1,xo) = \delta\_\_a(x^*, u)$;

$d_1 = d(x_1)$

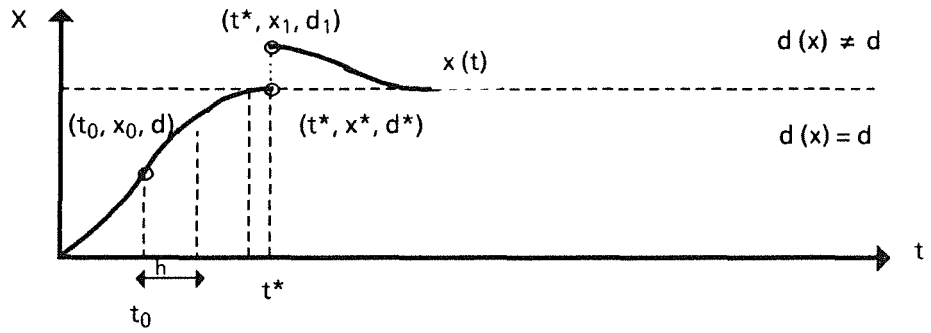



**Abb.** 6.7a: Autonomer Zustandsübergang δ__a (crossing)

Alle Interaktionen zwischen unterschiedlichen PEen erfolgen **explizit**, unter Anwendung von δ__a oder δ__u. Weil ein PE definitionsgemäß keine weiteren diskreten Ereignisse zu beachten

x(t) Lösung des AWP: $x'(t) = F(x(t),u), x(t_0) := x_0$;

$x^* = x(t_1)$

$(x_1,xo) = \delta\_\_u(x^*, u_1)$;

$d_1 = d(x_1)$

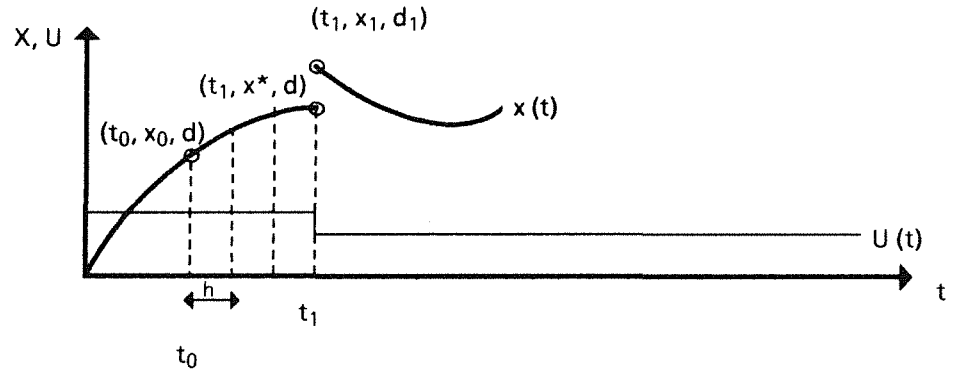



**Abb.** 6.7 b : Externer Zustandsübergang δ__u, Stellgrößenänderung $(u_1,t_1)$

hat, sind unterschiedliche PEe völlig autonom in der Wahl ihrer Integrationsschrittweiten.
Die Ein-/Ausgangsgrößen eines PE werden dabei als Treppenfunktionen behandelt. Dies ist zwar für Stell- und Störgrößenverläufe berechtigt, bedeutet aber für Kopplungsgrößen zwischen PEen, die durch ein gemeinsames DGS beschrieben sind, eine Interpolation 0ter Ordnung des zeitkontinuierlichen Lösungsverlaufes. Zwar können die Fehler durch einen a priori vereinbarten Austausch der Kopplungsgrößen zwischen PEen in einem genügend feinen Zeitraster begrenzt werden, doch sollten stark verkoppelte DGSe der Form

$$x'_1 = F1(x_1, x_2, u_1)$$
$$x'_2 = F1(x_2, x_1, u_2)$$

besser als **Ganzes** integriert (durch **ein** PE) realisiert werden, statt sie organisatorisch in zwei PEe mit Zustandsgrößen $x_1, x_2$ zu zerlegen.

**(3) Schnittstellen**

Software-technisch ist ein PE eine Instanz eines generischen Datentyps, welcher

a) einige standardisierte Schnittstellenoperationen zur Interaktion mit anderen PEen oder dem PFS (Prozeßperipherie-E/A) zur Verfügung stellt
b) zu unterschiedlichen Typen realer technischer Prozesse spezialisiert werden kann, z.B. ein Behälter oder Armgelenk eines Roboters, wobei jeder Typ durch einen Satz aktueller generischer Parameter gekennzeichnet ist, die vom Modellierer spezifiziert werden (Ein-/Ausgabedatentypen, Zustands-DGS, Prädikate)
c) unterschiedliche Exemplare desselben Typs (z.B. Behälter, Armgelenk) mit verschiedenen Anfangszuständen als Instanzen erzeugen kann.

**a) Operationen zur Laufzeit**

**create-pe** (pe__type: pe__description): pe__id;

Erzeugen einer neuen Instanz eines Prozeßelementes, die durch pe__id referenzierbar ist und deren konkreter Typ (generische Parameter, vgl. b) durch pe__type definiert ist.
Dabei werden alle für die Simulation zur Laufzeit benötigten Hilfsobjekte (vgl. c)) erzeugt und initialisiert.

**init-pe** (pe: pe__id; x:X; neighbours: sequence of pe__id);

Initialisieren des Zustands x für die Instanz pe, sowie Bekanntmachen der Instanzen von PEen, welche pe durch seine Ausgabe bei Zustandsübergängen $\delta$__a, $\delta$__u beeinflußt.

**delete-pe** (pe: pe__id);

Vernichten der Instanz pe eines PE einschließlich ihrer internen Hilfsobjekte.

**control-pe** (pe: pe__id; t:absolute__time; xi__comp: inp__selector; xi: XI);

Diskrete Änderung einer der Eingangsgrößen zum Zeitpunkt t; durch den Selektor xi__comp wird spezifiziert, welche Komponente der Eingangsgrößen von pe durch den neuen Wert xi zu ersetzen ist.

**sample-y-pe** (pe: pe__id; t:absolute__time) :Y;
**sample-z-pe** (pe: pe__id; t:absolute__time) :Z;

Abtastfunktion, die den aktuellen Wert der direkt meßbaren Größen y oder Regelgrößen z zum Zeitpunkt t liefert. Dabei wird stets exakt bis zum Zeitpunkt t integriert, unabhängig von der internen Integrations-Schrittweite. sample__z__pe liefert die Rohwerte der Leistungsgrößen eines PE (3.1.3.1) und wird zur Experimentauswertung benötigt.

Für die Operationen create__pe, init__pe und delete__pe wurde nur eine Meta-Notation angegeben; die genaue Syntax hängt von den verwendeten Konzepten der Implementierungssprache ab, z.B. generische packages und generische Instantiierung in ADA, oder Klassenhierarchien in SIMULA oder einer anderen objektorientierten Sprache.

**b) Modellierungs-Schnittstelle (generische Parameter)**

Um eine Instanz eines PE erzeugen zu können, müssen folgende Datentypen und Funktionen spezifiziert werden:

- Datentypen und Dimensionen von X, Y, Z, XS, U, Anzahl np von Prädikaten
- Datentypen der Kopplungsgrößen $X_1,...,X_j$ zu anderen PEen, Typen der Ausgabe-PEe
- Funktion F des kontinuierlichen DGS
- Prädikate $P_j$ $(1 \leq j \leq np)$ zur Beschreibung von crossings und Arbeitsbereichen
- Meß- und Regelgrößenfunktionen Fy, Fz
- Diskrete Zustandsübergangsfunktionen $\delta$__a, $\delta$__u

- Verfahrensparameter (z.B. numerisches Integrationsverfahren, Bereichsangaben zu der Integrationsschrittweite, Fehlerschranken für den lokalen Diskretisierungsfehler, zeitliche Auflösung bei der Lokalisierung von crossings etc.).

Diese Modellierungsschnittstelle ist disjunkt zu der Schnittstelle zur unterliegenden Laufzeitmaschine, also unabhängig vom Kernsystem zur Simulation. Sie besteht nur aus Datentypen und Funktionsprozeduren in einer höheren Sprache, in der die PEe realisiert werden. Dennoch bleibt die direkte Programmierung dieser Funktionen für den Endbenutzer i.a. zu mühsam und zeitraubend. Zwei Wege zur Abhilfe bieten sich an:

(i) Verwendung einer existierenden kontinuierlich/diskreten Simulationssprache als Eingabe-Schnittstelle, welche dem Benutzer Arbeit in der Formulierung, Umsortierung, Auflösung von Gleichungen abnimmt und als Ausgabe die obigen Datentypen und Funktionsprozeduren erzeugt. Diese können über externe Bibliotheken eingebunden werden. Diese Lösung kann jedoch nicht voll befriedigen:
   - es ist nicht garantiert, daß der erzeugte Code, insbesondere die diskreten Teile, der oben angegebenen Form entspricht, daß also reine Funktionsprozeduren ohne globale Variable und ohne Schnittstellen zur Ablaufsteuerung erzeugt werden. In manchen Simulationssprachen, z.B. GPSS, wird nicht strikt zwischen problembezogenen Aspekten und laufzeit-bezogenen wie z.B. Einplanungen in der Zeitliste, getrennt.
   - der Modellierungskomfort solcher Sprachen ist begrenzt, weil sie meist als Allzweck-Sprachen konzipiert und zum Teil nicht interaktiv sind.

(ii) Entwicklung von Dialogkomponenten anstelle neuer Simulationssprachen, die auf spezielle Anwendungen der Prozeßsimulation zugeschnitten sind, interaktiv mit dem Benutzer die Systemgleichungen erstellen und, gestützt auf symbolische Formelmanipulationssysteme, Analyse-Werkzeuge für DGSe etc., Code für die Datentypen und Funktionen erzeugen und darüber hinaus auch Vorschläge für die Lösungsverfahren und Schrittweitensteuerung machen können. Solche Komponenten sind Teil des übergeordneten Entwicklungssystems. Ansätze hierzu finden sich z.B. in /MAR 84/.

**c) Realisierung**

Für die Realisierung von PEen werden folgende interne (verborgene) Funktionen benötigt: die Auswertung des diskreten Begleitzustands d, die zeitdiskrete Integrationsfunktion $F_\Delta$, welche das numerische Integrationsverfahren aufruft, und eine Funktion C zur Crossing-Detektion. Ein Server-Prozeß (W-Prozeß), der über ein W-Kommunikationsobjekt die über die Laufzeit-Schnittstelle (a) gestellten Anforderungen empfängt, steuert die Simulation des PE. Benötigt wird die zeitüberwachte Form der Empfangsoperation nach 4.1.3. Der Server-Prozeß ist in seiner Struktur sehr ähnlich dem zur Simulation von discrete-event-Komponenten in Beispiel 4.2.1.3 angegebenen (Abb. 4.4).

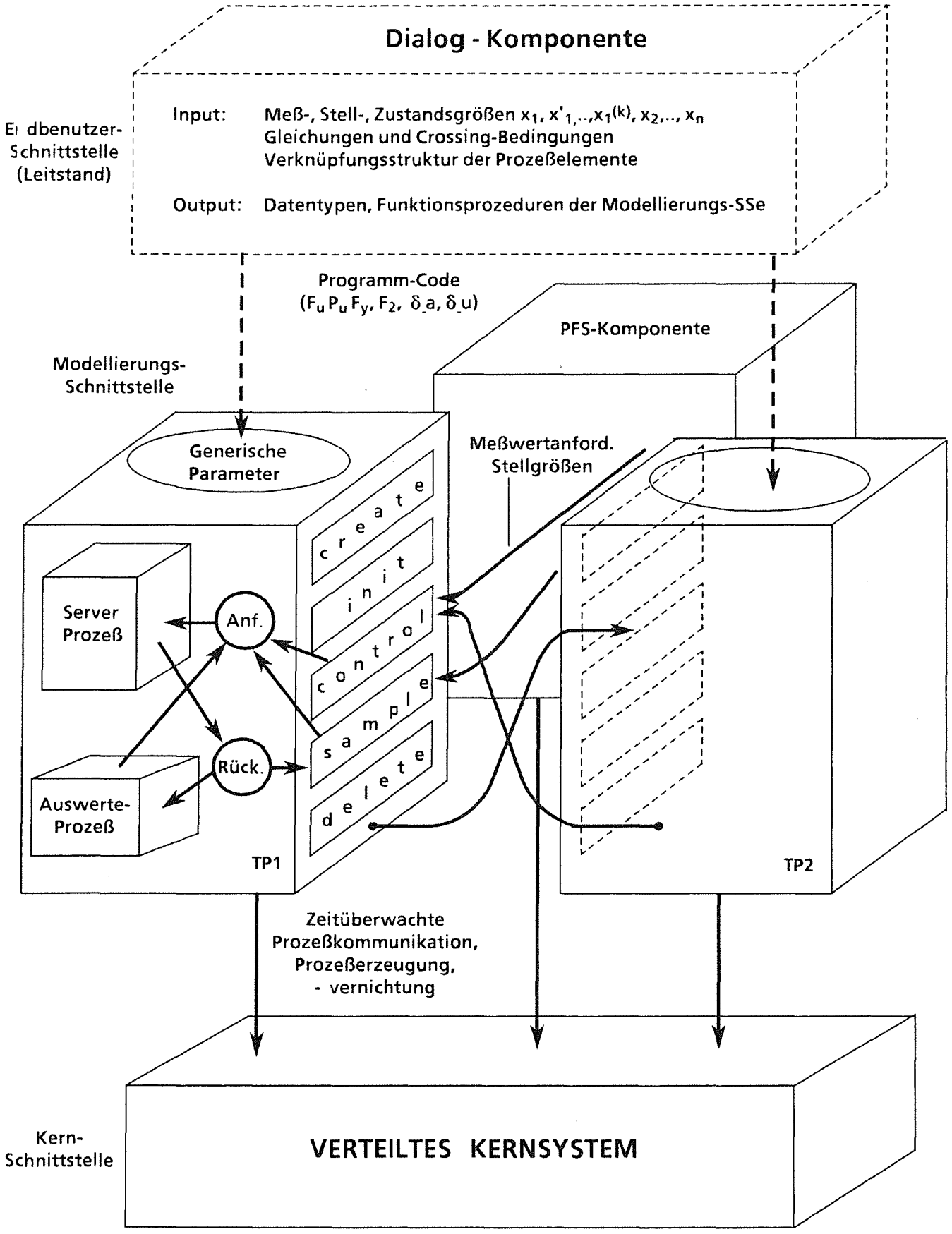



**Abb**. 6.8: Schnittstellen kontinuierlich-diskreter PEe

Hinzu kommt ein zweiter Auswertungsprozeß, der die Regelgrößen in einem frei wählbaren Zeitraster abtastet und an das Auswertungssystem weiterleitet.

Abb. 6.8 zeigt die wesentlichen Schnittstellen von PEen im Überblick.

### 6.2.2 Virtuelle Betriebsmittel in Mehrrechnersystemen

Aufgabe dieser Komponenten ist es, die Konkurrenz von PFS-Komponenten um Hardware-Betriebsmittel im Entwurfsstadium nachzubilden, um ihre Einflüsse auf Antwortzeiten und Regelgrößen unter verschiedenen statischen oder dynamischen BM-Zuordnungen und Zuteilungsstrategien vorherzusagen. Virtuelle BM gehören zu den Standard-Modellbausteinen praktisch aller Simulationssysteme, die zur Leistungsvorhersage von DV-Systemen dienen. Folgende primäre Anforderungen werden gestellt (vgl. 3.4.5):

- Dienste für den Verbrauch von HW-BMen in purer Form anzubieten:
  - CPU-Zeitverbrauch
  - ASP-Bedarf
  - Bedarf an E/A-Transfers, Kommunikationsaufkommen
- Standardmodelle für die wichtigsten Zuteilungs- und Bedienstrategien zur Verfügung zu stellen
- Meßpunkte zur Erfassung BM-orientierter Leistungsgrößen (Auslastungen, Warteschlangen-Längen) bereitzustellen
- administrative Operationen zu definieren, um Zuordnungen zwischen BM-Benutzern und BMen dynamisch herzustellen und lösen zu können.

**Nicht** angestrebt wird

a) Standardmodelle für logische, SW-gestützte Betriebsmittel mit funktionell vielfältigen Diensten anzubieten, wie z.B. bei der Simulation von Kommunikationsprotokollen oder Betriebssystemen üblich. Ein virtuelles E/A-Gerät, z.B. Hintergrundspeicher, bietet demnach nur E/A-Transfers auf Geräteebene an, gegeben durch ihren Datenumfang und ggf. weitere Auftragsparameter, aber keine höheren Dienste auf Dateien oder Datenbanken. Wenn nötig, können solche Modelle aber aus den im Testbett real existierenden logischen Betriebsmitteln (vgl. 6.1.1-6.1.4) und den dabei benutzten (virtuellen) HW-BMen zusammengesetzt werden.

b) Die **Implementierung** eines **im Testbett real** existierenden Betriebsmittels, insbesondere seiner Zuteilungsregeln, detailliert nachzubilden. Wenn eine derart feine Modellierung notwendig ist, sieht die integrierte Simulation ohnehin seinen Einsatz als reales Testobjekt vor. Die einzige Ausnahme bildet die Nachbildung der Kommunikationsmedien, z.B. Ethernet, die zwar Teil des Zielsystems sind, aber nicht sinnvoll als reales Testobjekt im Experiment

bewertet werden können. Für sie käme ersatzweise eine detaillierte interne Modellierung in Frage.

Als Beispiele greifen wir i.f. die Schnittstellen eines Rechnernetz- und eines Rechnerknotenmodells heraus.

#### 6.2.2.1 Rechnernetzmodell

Der Modellbaustein v__lan, der in 6.1.4 bereits benutzt wurde (Abb. 6.5), ist durch folgende Topologie- und Übertragungseigenschaften gekennzeichnet:

- Es wird eine Direktübertragung (vollduplex) von $n \geq 2$ Knoten über ein gemeinsames v__lan modelliert. Jeder Knoten k ist über zwei Kommunikationsobjekte mit v__lan verbunden, die den Sende- und Empfangspufferspeicher repräsentieren:
  MR__r (v__lan,k), MS__s (v__lan,k)
  Jeder angeschlossene Knoten k ist entweder ein **physikalischer Experimentrechner**, der als Zielrechner für ein reales, über v__lan kommunizierendes Testobjekt fungiert, oder ein **virtueller Rechner**, der auf einem beliebigem Experimentrechner modelliert wird (6.2.2.2). In beiden Fällen ist der Knoten eindeutig durch seinen Namen identifizierbar.
  Rechnernetze, in denen keine Direktverbindung zwischen allen Knoten besteht, z.B. sternförmige Topologien, können durch Verknüpfung mehrerer v__lan über Vermittlungsknoten aufgebaut werden.

- Von Knoten i nach j zu übertragende Nachrichtenpakete werden als Aufträge zunächst in MS__s (v__lan, i) abgelegt und sind spezifiziert durch
  - **Nutzdaten**, die physikalisch von MS__s (v__lan, i) nach MR__r (v__lan, j) übertragen werden, i.a. über ein unterliegendes reales Rechnernetz.
  - **virtuelle Länge**, die mit der realen Länge der Nutzdaten nur bei realen Testobjekten als Benutzer übereinstimmt. Sie allein - und nicht die reale Länge - bestimmt die simulierte Übertragungszeit.

- Unterschiedliche interne Realisierungen von v__lan mit identischen Benutzerschnittstellen sind möglich, je nach Detaillierungsgrad der **Übertragungszeit-Simulation**:
  - grob.
    eine nebenläufige Übertragung zwischen beliebigen Paaren von Sendern und Empfängern wird simuliert; die Übertragungszeiten werden nach einer lastabhängigen, für das zu modellierende Rechnernetz typischen **Verteilungsfunktion** ermittelt. Die Last ergibt sich aus dem Umfang aller momentan zu übertragenden bzw. in den Sendepuffern befindlichen Nachrichten. Modelle dieser Art werden z.B. in /MUE 86/ und in /LES 86/ verwendet.

- fein;
  die interne Auftragsabwicklung durch die physikalische Rechnernetzschicht (z.B. Ethernet, CSMA-CD Protokoll IEEE 802.3 /WOL 80/) wird im Detail modelliert.

- Fehlerhafte Übertragungen sollen nach einer vorgebbaren Wahrscheinlichkeit simuliert und dem Empfänger gemeldet werden.

- Die Verzögerung aus Sicht der Anwendung ist nicht allein durch die Eigenschaften des Kommunikationsmediums und die Last gegeben, sondern auch durch die Kapazität und Synchronisationseigenschaften der Sende-/Empfangs-Kommunikationsobjekte (blockierend oder überschreibend, vgl. 4.1.3).

- Die Verbindung zwischen einem einzelnen Knoten und v__lan kann dynamisch aufgehoben, oder v__lan kann vernichtet werden. Da hiermit auch die Sende-/Empfangs-Kommunikationsobjekte einschließlich der gespeicherten Nachrichten vernichtet werden, bedeutet dies die Nichterreichbarkeit eines Knoten über v__lan, bzw. den Ausfall von v__lan.

**Operationen**

**create-vlan** (v__lan: lan__id; <performance__attributes>);

Erzeugt lokal auf dem ausführenden Experimentrechner ein Objekt v__lan und die zu seiner Simulation benötigten Hilfsobjekte (z.B. W-Prozeß als Server-Prozeß). v__lan ist ein experimentweit bekannter, eindeutiger Name des virtuellen Rechnernetzes. <performance__attributes> bezeichnet die charakteristischen Größen zur Leistungsspezifikation, in erster Linie

- Übertragungszeit-Verteilungsfunktion (lastabhängig)
- Wahrscheinlichkeit von Übertragungsfehlern.

Es wird ein interner und benutzer-transparenter Katalogdienst benötigt, der allen Experimentrechnern die interne Auftrags-Schnittstelle, z.B. Mailbox-Identifikation, mitteilt, über die weitere Anforderungen unter dem Namen v__lan bearbeitet werden (z.B. connect__vlan).

**connect-vlan** (v__lan: lan__id; node: node__id; <access__attributes>);

Erzeugt die Sende-/Empfangs-Kommunikationsobjekte

MS__s (v__lan, node), MR__r (v__lan, node)

und stellt dadurch die Verbindung zwischen einem Knoten node und v__lan her. Kapazität und Zugriffseigenschaften der Kommunikationsobjekte werden durch <access__attributes> festgelegt.

node bezeichnet entweder einen Zielrechner für ein RTO, oder einen virtuellen Rechner, der auf einem Experimentrechner e__node erzeugt wurde (durch create__vnode, vgl. 6.2.2.2).

Die MS__s, MR__r werden wie folgt auf die Experimentrechner verteilt (vgl. das Beispiel zu Abb. 6.5):

MS__s (v__lan, node) ----> Experimentrechner von v__lan

MR__r (v__lan, node) ----> node , falls node Experimentrechner

e__node, falls node virtueller Rechner auf e__node

**disconnect-vlan** (v__lan: lan__id; node: node__id);

Löst die Verbindung zwischen einem virtuellen Rechner oder Zielrechner node und v__lan auf, d.h. vernichtet insbesondere MS__s (v__lan, node), MR__r (v__lan, node). Dadurch werden auch alle Nachrichten im Sendepuffer MS__s (v__lan, node) gelöscht, und bereits in Übertragung befindliche, an MR__r (v__lan, node) gerichtete Nachrichten kommen nicht mehr an.

**delete-vlan** (v__lan: lan__id);

Vernichtet alle internen Objekte des Simulationsmodells v__lan, und alle noch existierenden Verbindungen von v__lan zu Zielrechnern oder virtuellen Rechnern. Es können also keine weiteren Aufträge für v__lan mehr bearbeitet werden. Das virtuelle Rechnernetz kann aber unter demselben Namen später wieder neu aufgebaut werden.

**transmit-vlan** (v__lan : lan__id;
dest : node__id;
data : message;
vlength : integer);

Sendet die Nachricht data von MS__s (v__lan, source) nach MR__r (v__lan, dest), falls

- v__lan existiert
- der Aufrufer ein Zielprozeß des Experimentrechners source, oder ein dem virtuellen Rechner source zugeordneter Wirtsprozeß ist und der Rechner source mit v__lan verbunden wurde.

Simuliert wird die Übertragungsdauer der Nachricht als Funktion der Länge v__length, der momentanen Belastung und der Leistungsspezifikation von v__lan. Fehlerhafte Übertragungen werden mit vorgegebener Wahrscheinlichkeit simuliert.

**receive-vlan** (v__lan : lan__id;
**out** source : node__id;
**out** data : message;
**out** vlength : integer;
**out** res : error__code);

Es wird eine Nachricht von MR__r (v__lan, dest) empfangen, wobei dest der Zielrechner oder der zugeordnete virtuelle Rechner des aufrufenden Prozesses. Die Ergebnisse der Übertragung sind der Absender(Knoten), die virtuelle Nachrichtenlänge sowie der Übertragungsstatus (fehlerhaft/fehlerfrei).

#### 6.2.2.2 Rechnerknotenmodell

Die einleitenden Bemerkungen von 6.2.2 gelten auch für das Rechnerknotenmodell (virtueller Rechner, VR abgekürzt), dessen Eigenschaften i.f. zusammengefaßt werden.

- Ein W-Prozeß und dessen Betriebsmittelverbrauch soll einem VR (dynamisch) zugeordnet werden können. Ebenso lassen sich im Prinzip auch beliebige weitere, von diesem Prozeß erzeugt Objekte auf einen virtuellen Rechner beziehen. Um die Schnittstellen eines VR nicht unnötig aufzublähen und so zu einer vollständigen Betriebssystem-Simulation zu gelangen, wird die Zuordnung von Objekten zu VRn auf die wichtigsten Objekttypen, (W-)Prozesse und Kommunikationsobjekte, beschränkt.

- Lokalität der Objekterzeugung bezüglich der virtuellen Rechner: Ein W-Prozeß erzeugt immer nur seinem eigenen VR zugeordnete Prozesse und Kommunikationsobjekte. Ein erster Prozeß (Initialisierungsprozeß) wird zusammen mit VR generiert und VR zugeordnet.
  Einem VR zugeordnete W-Prozesse und Kommunikationsobjekte haben ansonsten dieselben Eigenschaften wie gewöhnliche W-Prozesse und W-Kommunikationsobjekte des hybriden BS-Kerns.
  Insbesondere kann ein solcher W-Prozeß neben den betriebsmittel-bezogenen, VR-lokalen Diensten auch netzglobale Dienste beanspruchen wie z.B. abgesetzte Mailbox-Kommunikation oder Kommunikation über ein virtuelles LAN. Ein VR ist also ein Objekt, das zusätzliche, betriebsmittelorientierte Dienste für W-Prozesse anbietet, aber keine Schicht, welche Dienstleistungen unterer Schichten verbirgt.

- Lokalität der Objekterzeugung bezüglich der Experimentrechner: Alle demselben VR zugeordneten Objekte werden auch auf demselben physikalischen Experimentrechner erzeugt und allokiert; es können sich aber mehrere VR auf einem Experimentrechner befinden. Dies geschieht aus Effizienzgründen: Prozesse, die dieselben Betriebsmittel (VR) beanspruchen, sind als Simulationsmodelle i.a. stark miteinander verkoppelt. Die intensive Kooperation dieser Modelle soll möglichst experimentrechnerlokal, statt über die Netzwerksynchronisation erfolgen. (vgl. Beispiel in 5.2.3).

- Ein W-Prozeß beansprucht nur die Betriebsmittel seines eigenen VR, spezifiziert durch Verbrauchsangaben, deren Form vom Detaillierungsgrad der benutzten HW-BM und ihrer Zuteilungsstrategien abhängt. Diese Parameter werden bei der Erzeugung des VR festgelegt (s.u.).

- Wird ein VR vernichtet, so werden mit dem Initialisierungsprozeß automatisch auch alle baumartig erzeugten und VR zugeordneten Unterprozesse und Kommunikationsobjekte vernichtet, unabhängig von ihrem Zustand. Dadurch wird die Überführung von VR in den Grundzustand (reset), bzw. der Ausfall von VR modelliert. Der VR kann aber unter demselben Namen später neu erzeugt werden (z.B. durch einen W-Prozeß zur Experimentsteuerung).
  Die Spezifikation der übergeordneten statistischen Ausfall-/Wiederanlauf-Modelle ist aber nicht Gegenstand dieser Arbeit (siehe /LES 86/).

**Operationen**

```
create-vnode   (node      : node_id;
                <resource_attributes>;
                iproc     : load_module;
                <iproc_attributes>;
           out  iproc     : process_token);
```

Erzeugt und initialisiert einen virtuellen Rechner mit Namen node, einschließlich der Verwaltungsobjekte (z.B. Betriebsmittel-Warteschlangen, Bedienprozesse), lokal auf dem Experimentrechner des aufrufenden W-Prozesses. Dieser W-Prozeß ist selbst **keinem** VR als SW-Prozeß zugeordnet, sondern typischerweise ein Prozeß der Experiment-Initialisierung und -steuerung. node muß ein netzweit eindeutiger Name sein. <resource_attributes> spezifiziert die rechnerlokalen Betriebsmittel (vgl. Tab. 6.1).
Ein erster Initialisierungsprozeß iproc (ein W-Prozeß) wird ebenfalls erzeugt, node zugeordnet und gestartet.

```
create-vprocess   (proc : load_module;
                   <proc_attributes>;
              out  proc : process_token);
```

Der aufrufende W-Prozeß, der einem virtuellen Rechner VR zugeordnet sein muß, erzeugt und startet einen experimentrechnerlokalen W-Prozeß über dem Ladeobjekt proc, der ebenfalls VR zugeordnet wird. <proc_attributes> spezifiziert, wie <iproc_attributes> in create_vnode, die virtuellen Prozeß-Attribute, die für die zeitliche Bewertung auf VR wichtig sind, z.B.

- initialer Speicherbedarf für Code- und Kellersegment; der Prozeß wird zum frühesten Zeitpunkt gestartet, zu dem dieser Bedarf durch den virtuellen ASP-Pool des VR befriedigt werden kann!
- virtuelle Prozeßpriorität auf VR
- Zeitliche Angaben über den Prozeßstart (Zeitpunkt, ggf. Periode, zeitliche Frist).

Hierbei wird der untergeordnete Dienst create_process des hybriden Kerns für W-Prozesse benutzt.

**create-vmailbox** (<mailbox__attributes>,
**out** mb : mailbox__token);

Analog zu create__vprocess, erzeugt create__vmailbox ein experimentrechnerlokales W-Kommunikationsobjekt, das dem VR des aufrufenden W-Prozesses zugeordnet wird und dessen Attribute (Kapazität, Pufferstrategie, vgl. 4.1.3) in <mailbox__attributes> zusammengefaßt sind.

**delete-vprocess** (proc : process__token);

**delete-vmailbox** (mb : mailbox__token);

Vernichtet einen W-Prozeß proc bzw. W-Kommunikationsobjekt mb und löst die Zuordnungen in dem zugeordneten VR.

**delete-vnode** (node : node__id);

Vernichtet den VR node und seine internen BM-Verwaltungsobjekte, sowie alle noch existierenden W-Prozesse und -Kommunikationsobjekte, die durch create__vprocess bzw. create__vmailbox dynamisch erzeugt und node zugeordnet wurden.

Mit Hilfe der folgenden Dienste nehmen W-Prozesse virtuelle BM ihrer zugeordneten VR in Anspruch. Dies wird je nach Art des BM näher spezifiziert durch verschiedene Verbrauchsangaben (vgl. Tab. 6.1). Die von den W-Prozessen erfahrene dynamische Verzögerung hängt ab von den spezifizierten Zuteilungsstrategien und der momentanen Last der VR zugeordneten W-Prozesse.

**vnode-work** (<usage>);
**vnode-req-asp** (<usage>);
**vnode-req-ea** (unit : device__unit;
<request__specification>);

| | BM-Beschreibung | BM-Zuteilung | BM-Anforderung (<usage>, <request__specification>) |
|---|---|---|---|
| CPU | grob: Taktzeit<br>fein: Instruktions-typen aufge-schlüsselt | - Zeitscheiben<br>- FIFO<br>- statische Priorität (präemptiv/ nichtpräemptiv)<br>- Antwortzeit- Priorität | grob: Anzahl Takte<br>fein: Instruktionsprofil |
| ASP | (Sorten, Ausbau)<br>z. Bsp.<br>- Code, Konstanten, stat. Daten<br>- Keller<br>- dynamische Daten | (vgl. /WET 84/, Kap. 3.2):<br>Kandidatenauswahl<br>ASP-Organisation (Tabellenverfahren, Halbierungsverfahren)<br>Freispeicher-sammlung | (ASP-Art, Umfang) |

| | BM-Beschreibung | BM-Zuteilung | BM-Anforderung (<usage>, <request_specification>) |
|---|---|---|---|
| E/A | grob: | | |
| | - Gerätetypen | FIFO | Gerät, |
| | - Geräte, Transferraten | | Transfer-Umfang |
| | fein: | | |
| | - Gerätetypen | - FIFO | Gerät, |
| | - Controller,E/A-Bus | - Auftrags-Priorität | Transfer-Umfang, |
| | - Geräte-Einheiten | - geräteorientierte | Auftraggeber, |
| | - Controller-Transferrate | Strategie | E/A-Operation, |
| | - Positionierzeit (Hintergrund-Speicher) | | Ort der Daten |

**Tab.** 6.1: Beispiele für Betriebsmittel- und BM-Verbrauchsspezifikation

## 7. Zusammenfassung und Ausblick

In dieser Arbeit wurde eine neue Systemarchitektur für die integrierte Leistungsvorhersage und -analyse verteilter Echtzeitsysteme, speziell Prozeßführungssysteme, entwickelt. Den verteilten Zielmaschinen der PFSe, bestehend aus Echtzeit-BS-Kernen, E/A-Geräte- und Rechnernetz-Schnittstellen sowie Zielsprachen, wird eine virtuelle Zeitführung als funktionelle Erweiterung unterlegt, die als Spezialfälle die konventionelle Echtzeitführung, die zeitdiskrete Simulationsuhr oder beliebige Mischungen ermöglicht. Damit können kooperierende Prozeßsysteme aus realen Testobjekten (R-Prozessen) und prozeßorientierten Simulationsmodellen (W-Prozessen) koordiniert ablaufen.

Die Kommunikations-Schnittstelle zwischen realen Testobjekten und simulierter Umgebung ist so gestaltet, daß das Leistungsverhalten des RTO in simulierter Umgebung übertragbar auf eine verhaltensgleiche Echtzeitumgebung ist. Der durch die Erweiterungen bereitgestellte Simulationsansatz ist allgemeingültig, d.h. den Zeigler'schen discrete-event-Netzen oder dem SIMULA Koroutinenmodell äquivalent. Die erweiterten Zielmaschinen können also auch als konventionelle Simulationsmaschinen benutzt werden. Das Simulationskonzept ist darüber hinaus unabhängig von einem bestimmten (Hersteller-)BS-Kern und der Art seiner Kommunikations- und Synchronisationsdienste, Zuteilungsregeln, sequentiellen Implementierungssprachen oder E/A-Geräten; es wird als eine Vorschrift zur Übertragung **gegebener** Dienste einer Zielmaschine in virtuelle Zeit eingeführt.

Dieses Konzept besitzt folgende Hauptanwendungen:

- **Leistungsbewertung im hierarchischen Entwurf**
  Prozeßführungssysteme werden von ausführbaren, abstrakten Entwürfen schrittweise zu einsatzfähigen Zielversionen verfeinert, wobei Modellierungs- und Zielsprache stets identisch sind, und anhand einheitlicher, anwendungsorientierter Leistungsgrößen des geschlossenen Wirkungskreises bewertet. Durch die reale Ersetzung von PFS-Entwürfen innerhalb simulierter Experimentumgebung wird das Problem der Modellvalidierung entscheidend entschärft.
- **Verteilte, sprach-heterogene Modellierungssysteme**
  Der erweiterte Systemkern kann als universelle, verteilte Laufzeitumgebung für verteilte Simulationsanwendungen mit heterogenen Modellierungssprachen dienen.
- **Interferenzarmes Testen verteilter Echtzeitsysteme**
  Die virtuelle Zeitführung erlaubt, an Haltepunkten den Ablauf eines verteilten PFS auf einzelnen Knoten anzuhalten und interaktiv zu inspizieren, ohne den globalen zeitlichen Ablauf nennenswert zu verfälschen. Bisher waren für dieses Problem keine umfassenden Lösungsansätze bekannt.

Die wichtigsten Lösungsschritte werden i.f. noch einmal zusammengefaßt. Nach der Festlegung der Zielsysteme (technische Prozesse, PFSe), ihrer Entwicklungsstadien und Leistungsgrößen

wurde die Erweiterung der Ablaufsemantik beliebiger Prozeßsysteme auf virtuelle Zeit formal spezifiziert. Die Prozeßaktionen wurden klassifiziert in grundsätzlich - auch bei Prozeßblockierung - zeittransparente, in explizite Zeitverzögerungen und in zeitüberwachte Kommunikations- und Synchronisationsoperationen. Die Kooperation eines solchen Prozeßsystems mit einem durch ein I/O-System beschriebenen realen Testobjekt mit zeitdiskreten Ein-/Ausgabesignalverläufen wurde dann als "Echtteilesimulation in Nicht-Echtzeit-Umgebung" spezifiziert. Es wurden der Zustandssteuerbarkeit und -beobachtbarkeit äquivalente Bedingungen angegeben, unter denen das zeitliche Verhalten dem realen, in Echtzeit operierenden Wirkungskreis entspricht.

Anschließend wurde das systemtheoretische Konzept umgesetzt in eine Architektur, in dem als reales Testobjekt nur noch Rechensysteme (HW, BS+Anwender-SW) oder deren Subsysteme auftreten. Zunächst wurde in Kap. 4 gezeigt, wie ein Echtzeit-BS-Kern für Monoprozessorsysteme zu einem "hybriden BS-Kern" für die integrierte Simulation erweitert wird. Hervorzuheben sind dabei insbesondere

- die Aufteilung der Prozeßzustände für W-Prozesse in betriebsmittelbedingte/modellbedingte
- die begrenzt asynchrone Kommunikations-Schnittstelle zwischen R- und W-Prozessen
- der abstrakte Datentyp der Virtuellen Uhr mit Simulationsuhr und Echtzeituhr als Sonderfällen
- die Koordinierung von Zeitführung und Prozessorzuteilung für R- und W-Prozesse
- die Organisation der von R- und W-Prozessen gemeinsam benutzten passiven, räumlich aufgeteilten BM (ASP) und die Ansteuerung der E/A-Geräte.

Neben der realen Nutzung der Betriebsmittel der Zielmaschine durch R-Prozesse und der zeittransparenten Nutzung durch W-Prozesse wurde auch die simulierte Nutzung durch S-Prozesse, als dritter Prozeßklasse, zugelassen. Anhand der objektorientierten Architektur des BS-Kerns (mit/ohne Erweiterungen) konnte auch die Ökonomie der Erweiterungen gezeigt werden.

In Kap. 5 wurden neue Verfahren für die zeitliche Synchronisation mehrerer über Rechnernetz(e) lose gekoppelten Monoprozessorsysteme entwickelt:

- ein zeitlich eng gekoppeltes (FRED), das eine netzglobale virtuelle Uhr implementiert. Zur Effizienzverbesserung des zeitlich eng gekoppelten Verfahrens wurde ein neues, intervallgesteuertes Ablaufmodell für W-Prozesse eingeführt.
- ein zeitlich lose gekoppeltes (OLGA), das auf einer individuellen Vorausschau der Knoten mit Zurücksetzen (rollback) basiert. OLGA verwendet eine statische Speicherverwaltung mit nur einer Sicherungskopie des Zustands; Zustandssicherung und Zurücksetzen operieren auf Knotenebene. Dadurch können die Monoprozessor-BS-Kerne aus Kap. 4 unverändert in verteilten Systemen wiederverwendet werden.

Beide Verfahren sind schwach korrekt und terminieren für beliebige zeitdiskrete Simulationsmodelle, sind transparent für die Modellprogrammierung und geeignet zur Integration realer Testobjekte (R-Prozesse). In OLGA mußte hierzu die eingeschränkte

Reproduzierbarkeit des Zeitverhaltens von R-Prozessen nach dem Zurücksetzen in Form von Unsicherheitsintervallen berücksichtigt werden.

Der Vergleich von FRED und OLGA ergab, daß FRED einen direkten Übergang vom Simulationsexperiment zum Echtzeiteinsatz erlaubt, daß aber aufgrund der globalen virtuellen Zeit erhebliche Interferenzprobleme der realen Testobjekte auftreten können.

OLGA ist dagegen selbst nicht für den Echtzeiteinsatz geeignet; andererseits sind wegen der zeitlichen Autonomie der Knoten die vorhandenen realen Testobjekte in den Simulationsexperimenten viel weniger von Interferenz betroffen.

In Kap. 6 wurde die Anwendung des in Kap. 4 und 5 entwickelten Systemkerns als verteilte reale Ablaufmaschine bzw. Simulationsmaschine an einfachen Beispielen erläutert und die Schnittstellen einiger Modellkomponenten (kontinuierlich-diskreter Modelle technischer Prozesse, Rechnerknoten- und Rechnernetzmodelle) vorgestellt.

**Ausblick**

Die Haupterkenntnis der Arbeit besteht darin, daß die integrierte Simulation für Echtzeitsysteme die Vorteile simulativer Methoden (entwurfsorientierte Modellierung, anwendungsorientierte Leistungsgrößen, Interferenzfreiheit) und echtzeitorientierter Methoden (Ausnutzung der Information des Zielsystems, absolute Leistungsdaten, Wegfall überflüssigen Modellierungsaufwands) vereint und als BS-Architektur auch realisierbar ist. Weiterentwicklungen sind jedoch sowohl in technischer als auch in methodischer Hinsicht angezeigt:

**(1) Effizienz der verteilten Simulation**

Die Effizienz der Verfahren zur NWS muß verbessert werden, insbesondere für größere Knotenanzahlen ('up-scaling'). Für die lose gekoppelten Verfahren bieten sich zwei Ansätze an:

- Ausnutzung von Eigenschaften des Zeitverhaltens spezieller Modellkomponenten, um den Aufwand des Zurücksetzens zu reduzieren. Prüfstein bleibt dabei die völlige Anwendertransparenz und Allgemeingültigkeit der Verfahren. D.h. falls die Eigenschaften nicht erfüllt sind, aus denen das Verfahren Nutzen zieht, darf dies nur zu einem Verlust an Effizienz, aber nicht zu inkorrekten Ergebnissen oder Nichtterminierung führen, wie bei den Vorhersehbarkeitseigenschaften (Kap. 5.2). Alle notwendige Information muß in den Modellen selbst enthalten sein, anders als z.B. in /WLU 87/, wo der Modellierer die Semantik von abstrakten Datentypen (Warteschlangen) zusätzlich zum eigentlichen Modell spezifiziert, um den Rücksetzaufwand für die unterliegende NWS (Time-Warp) zu minimieren.
- Erweiterung des zweistufigen Verfahrens OLGA (vgl. 5.2.3) auf ein k-stufiges, hierarchisch-rekursives, in dem je m Komponenten, durch einen virtuellen Ring verbunden, zu einer Komponente höherer Ordnung zusammengefaßt und durch eine eigene Sicherungskopie vertreten werden. Bei n Komponenten ergeben sich O ($\log_m$ (n)) Kommunikationsebenen und

ebensoviele Sicherungskopien pro Komponente. Hierarchische Verfahren wurden bereits für zeitlich eng gekoppelte /ZEI 85/ und für zeitlich lose gekoppelte Simulationsmodelle nach defensiven Verfahren /PRA 88/ publiziert.

**(2) Anwendungen**

Die Konzepte und Dienste des Systemkerns wurden zwar an einfachen Testbeispielen (Kugelfallversuch /SCR 84/, Transportstation) überprüft, doch kann erst eine größere verteilte Anwendung zeigen, welche sprachlichen und sonstigen Erweiterungen auf Anwendungsebene nötig sind, um die integrierte Simulation optimal anwenden zu können. Hier würde sich z.B. der Entwurf und die Leistungsbewertung eines Netzwerk-BS auf der Grundlage von Monoprozessor-BS-Kernen anbieten. In diesem Rahmen wäre auch die Realisierung des Entwicklungs- und Auswertungssystems (Kap. 3.5) in Angriff zu nehmen.

**(3) Innovative Rechnerarchitekturen**

Von hohem Interesse ist die Frage, wie sich die integrierte Simulation auf grundsätzlich andere Zielrechnerarchitekturen überträgt, z.B. OCCAM-Transputernetze oder Datenflußrechner. Es ist zu erwarten, daß die Interferenzprobleme infolge gemeinsamer Betriebsmittelbenutzung in solchen hochparallelen Architekturen entschärft werden, andererseits die Anforderungen an die NWS weiter steigen.

**(4) Konstruktionslehre**

Als wesentliche Entwicklungs- und Bewertungsstadien eines PFS wurden der ausführbare abstrakte Entwurf, der Entwurf in virtueller Rechnerumgebung, die Implementierung mit simuliertem und mit realem Betriebsmittelbedarf auf der Zielmaschine genannt. Wünschenswert wäre nun eine systematische Verfeinerungsmethodik oder -strategie eines PFS entlang der Dimensionen

- **funktionelle** Detaillierung, z.B. durch virtuelle Maschinen und ihre Dienste,
- Detaillierung des **Betriebsmittelbedarfs**, von der abstrakten Zeitverzögerung bis zur Zeitmessung, insbesondere die Ersetzungsreihenfolge simulierter durch reale Komponenten.

**(5) Leistungs-Diagnose**

Bei der integrierten Simulation sind Änderungen im Leistungsverhalten immer das Ergebnis expliziter, vom Entwickler beabsichtigter und kontrollierter Änderungen im Entwicklungsobjekt (und nicht von impliziten Änderungen in der Testumgebung oder Modellierungsfehlern im Entwicklungsobjekt). Insofern ist die "Symptomerhebung", als die Erfassung von Leistungsgrößen,gegenüber bekannten experimentellen Verfahren verbessert. Damit ist aber noch nicht beantwortet, **weshalb** bestimmte Entwurfs- oder Implementierungsentscheidungen in einem komplexen verteilten SW-System z.B. Leistungsverschlechterungen zur Folge haben, welche Hilfsgrößen, z.B. Betriebsmittelauslastungen, zur genaueren Diagnose inspiziert werden müssen und welche Abhilfemöglichkeiten bestehen. Im Bereich der Engpaßanalyse und Optimierung von Fertigungsstraßen, wo ein unmittelbarer wirtschaftlicher Anreiz besteht,

existieren solche Werkzeuge zur Leistungsdiagnose (Expertensysteme, vgl./SCH 87/), aber noch kaum Vergleichbares für verteilte Echtzeit-Rechensysteme mit ihren andersartigen Leistungsgrößen (man vgl. z.B. /LKK 86/). In /SMI 88/ wurden anwendungsübergreifende Entwurfsprinzipien, wie das Lokalitäts-, Fixierungs- oder Dominanzprinzip formuliert, in deren Licht konkrete programmiertechnische Entscheidungen, etwa die Art der Kommunikationsoperationen oder die Verteilung der Rechenaufgaben, auf ihre letzlichen Auswirkungen (globale Antwortzeiten und Regelgrößen) bewertet werden könnten. Könnte man diese Prinzipien quantifizieren, also als zusätzliche Leistungsgrößen mitbewerten, käme man in der Leistungsdiagnose und -synthese vielleicht ein Stück weiter.

**Literaturverzeichnis**


/ADA 83/ The Programming Language ADA Reference Manual
Ed.: G.Goos, J. Hartmanis
LNCS 155, Springer Verlag, Berlin, 1983

/ADE 83/ Adelsberger H.
Modelling and Simulation in ADA
1st European Simulation Congress, Aachen, 1983, pp.273-280

/AFH 85/ Andres C., Fleischmann A., Holleczek P., Trautner M.
Testen von verteilten Programmen zur Prozeßautomatisierung
Angewandte Informatik No.2, 1985, pp.69-76

/ALF 85/ Alford M.W.
SREM at the age of eigh; The distributed computing design system
Computer 18 (4), 1985, pp.36-46

/BBK 86/ Bayan R., Bonnet C., Kung A., Kichgässner W., Landwehr R., Schwarz B.
Requirements for a Real-Time ADA Runtime Kernel and Proposed Kernel Interface
GMD-Arbeitsbericht 204, März 1986

/BCM 75/ Baskett F., Chandy K.M., Muntz R.R., Palacios F.G.
Open, Closed and Mixed Networks of Queues with Different Classes of Customers
Journal ACM, Vol. 22, 1975, pp. 248

/BCM 87/ Bagrodia L., Chandy K.M., Misra J.
A Message-Based Approach to Discrete-Event Simulation
IEEE Transactions on Software Engineering 13 (6), 1987, pp. 654-665

/BEL 87/ Berry O., Lomow G.
The potential speedup in the Optimistic Time Warp Mechanism for Distributed Simulation
2nd international conference on computers and applications
Beijing, China, 23.-27.6.1987, pp. 694-698

/BES 82/ Berg H.K., Smith M.G.
A Distributed System Experimentation Facility
Proc 3rd Int. Conference on Distributed Computing Systems
Ft. Lauderdale, Fl., USA, Okt. 1982, pp.324-329

/BEZ 87/ Bezevin J.
Timelock: A Concurrent Simulation Technique and its Description in Smalltalk - 80
1987 Winter Simulation Conference
Atlanta, Ga., 14.-16. 12. 1987, pp. 503-506

/BIL 84/ Birtwistle G., Luker P.
Dialogs for simulation
Conf. on Simulation in Strongly Typed Languages
La Jolla, Ca., USA, 2nd-4th February, 1984, pp.90-95

/BIR 79/ Birtwistle G.
DEMOS - A System for Discrete-Event Modelling based on SIMULA
Mac Millan, 197

/BMS 82/ Beilken C., Mattern F., Spenke M.
Entwurf und Implementierung von CSSA - Beschreibung der Sprache, des Compilers und des Mehrrechnersimulationssystems
Teil A: Konzepte
MEMO-SEKI-82-03-A, Universität Kaiserslautern, 1982

/BOO 86/ Booch G.
Object-Oriented Development
IEEE Transactions on Software Engineering 12 (2), 1986, pp.211-221

/BRL 82/ Brayer K., Lafleur V.
A Testbed Approach to the Design of a Computer Communication Network
Computer 15(10), 1982, pp.15-23

/BRY 82/ Bryant R.M.
Discrete system simulation in ADA
Simulation 39 (4), 1982, pp.111-121

/BUM 88/ Burkhart H., Millen R.
Techniken und Werkzeuge der Programmbeobachtung am Beispiel eines Modula-2 Monitorsystems
Informatik Forschung und Entwicklung 1988 (3), pp. 6-21

/CAV 83/ Cavouras J.C.
Computer System Evaluation Through Supervisor Replication
Computer Journal 26(2), 1983, pp.134-141

/CHM 79/ Chandy K.M., Misra J.
Specification, Synthesis, Verification and Performance Analysis of Distributed Programs - A Case Study: Distributed Simulation
Seminar on Distributed Data Sharing Systems
Aix en Provence, Frankreich, 16.-18.5.1979

/CHM 81/ Chandy K.M., Misra J.
Asynchronous Distributed Simulation via a Sequence of Parallel Computations
CACM 24(11) Special Issue on Simulation Modelling and Statistical Computation, 1981, pp.198-206

/CLE 84/ Chu W.W., Leung K.K.
Task Response Time Model and its Applications for Real-Time Distributed Processing Systems
Real-Time Systems Symposium 1984, pp.225-236

/CLU 85/ Cleary J.G., Lomow G.A., Unger B.W., Xiao Z.
JADE's IPC Kernel for distributed Simulation
Proc. 13th SIMULA Users' Conference,
University of Calgary, Canada, 28th-30th August, 1985
pp.1-8

/CMM 79/ Cheriton D.R., Malcolm M.A., Melen L.S., Sager G.R.
Thoth, a Portable Real-Time Operating System
CACM 22 (2), 1979, pp.105-115

/COG 88/ Comfort J.C., Gopal R.R.
Environment paritioned distributed Simulation with Transputers
Distributed Simulation 1988, San Diego, Ca.,
2.-5. 2. 1988, pp. 103-108

/CON 85/ Concepcion A.I.
Mapping distributed simulators onto the hierarchical
multi-bus microprocessor architecture
Conf. on Distributed Simulation, San Diego, Ca., Jan. 1985,
pp.8-13

/COR 85/ Coolahan J.E., Roussopoulos N.
A Timed Petri Net Methodology for Specifying Real-Time
System Timing Requirements
Int. Workshop on Timed Petri Nets, Turin, Juli 1985, pp.24-31

/DEC 80/ PDP11 Peripherals Handbook
KWV11-A Programmable Real-Time Clock
Digital Equipment Corporation, Maynard, Mass., 1980

/DEM 84/ Decker H., Maierhofer J.
Very high level model description and simulation
Conf. on Simulation in Strongly Typed Languages
La Jolla, Ca., USA, 2nd-4th February, 1984, pp.44-48

/DIN 78/ Informationsverarbeitung 1
DIN Taschenbuch 25
Beuth Verlag GmbH, Berlin, Köln, 1978

/DKO 86/ Didic M., Kohlhepp P., Oberle R.
Design considerations for a distributed real-time nuclear
reactor safety system
Computer Systems Science and Engineering 1(2), 1986
pp.82-92

/DOB 84/ Downes V.A., Bosch R.T.
Discrete Event Simulation with ADA
UKSC Conference on Computer Simulation,
Univ. of Bath, England, 12.-14.9.1984, pp.68-78

/DOM 73/ Dörfler W., Mühlbacher J.
Graphentheorie für Informatiker
de Gruyter, Berlin, New York, 1973

/DRW 87/ Drobnik O., Wettstein H.
Eine Architektur heterogener verteilter Systeme
Informationstechnik 29 (6), 1987, pp. 411-419

/EFR 86/ Estrin G., Fenchel R.S., Razouk R.R., Vernon M.K.
SARA: Modelling, Analysis, and Simulation Support for Design
of Concurrent Systems
IEEE Trans. SW-Engineering 12 (2), 1986, pp.293-311

/ELL 83/ Ellis J.T.
ADL/ADS - A Testbed Tool for Experimentation with Real-Time DDP Architectures
Real-Time Systems Symposium '83, Arlington, Va., USA, 6.-8.12.1983, pp.89-99

/FAE 79/ Färber G.
Prozeßrechentechnik
Springer Verlag, Berlin, 1979

/FAE 84/ Färber G.
Architektur zukünftiger Prozeßrechnersysteme
Prozeßrechner 1984, 4.GI/GMR/KfK-Fachtagung Karlsruhe, 26.-28.9.1984,
Informatik Fachberichte 86, Springer Verlag, pp.22-41

/FAP 83/ Faulk S.R., Parnas D.L.
On the Uses of Synchronization in Hard Real-Time Systems
Real-Time Systems Symposium '83, Arlington, Va., USA, 6.-8.12.1983, pp.101-109

/FBW 82/ Franta W.R., Berg H.K., Wood W.T.
Issues and Approaches to Distributed Testbed Instrumentation
Computer, 15(10), 1982, pp.71-81

/FEN 87/ Fenner P.R.
The Flex/32 for real-time multicomputer simulation
3rd Conf. on Multiprocessors and Array Processors
San Diego, Ca., 1987, pp. 127-133

/FER 78/ Ferrari D.
Computer Systems Performance Evaluation
Prentice Hall, Englewood Cliffs, 1978

/FOE 78/ Föllinger O.
Regelungstechnik
Elitera-Verlag, Berlin, 1978

/FOJ 85/ Fortier P.J., Juttelstad P.
Real-Time Hardware/Software Simlation, Design, and Use as a Performance Evaluation and Prediction Tool
Proc. 18th Annual Hawaii International Conference on System Sciences, 1985, pp.58-66

/FOR 83/ Forsstrom K.S.
Array Computers in Real-Time Flight Simulation
Computer 16 (6), 1983, pp.62-70

/FRA 77/ Franta W.R.
The Process View of Simulation
Elsevier-North Holland, New York, 1977

/FRN 82/ Franzkowiak G.H., Naro R.W.
An Analytical Model for Evaluation of Distributed Multiprocessor Systems with Shared Common Resources
Proc 3rd Int. Conference on Distributed Computing Systems
Ft. Lauderdale, Fl., USA, Okt. 1982, pp.786-791

/FUJ 88/ Fujimoto R.M.
Performance measurements of ditributed simulation strategies
Distributed Simulation 1988, San Diego, Ca.
2.-5. 2. 1988, pp. 14-20

/GAF 88/ Gafni A.
Rollback mechanisms for optimistic distributed simulation systems
Distributed Simulation 1988, San Diego, Ca.
2.-5. 2. 1988, pp. 61-67

/GAI 86/ Gait J.
A Probe Effect in Concurrent Programs
Software Practice and Experience 16(3), 1986
pp.225-233

/GEA 77/ Gear C.W.
Simulation: conflicts between real time and software.
Mathematical Software III, Academic Press Inc., 1977,
pp.121-138,

/GHU 85/ Gremminger K., Hütter R.
GROOPI Systemübersicht
Interner Bericht, Kernforschungszentrum Karlsruhe, Feb. 1985

/GOB 84/ Godwin A.N., Bulmer G.
Solving a real-time simulation problem in ADA
Conf. on Simulation in Strongly Typed Languages
La Jolla, Ca., USA, 2nd-4th February, 1984, pp.151-155

/GOF 88/ Gordon A.J., Finkel R.A.
Handling Timing Errors in Distributed Programs
IEEE Trans. on Software Engineering 14 (10)
Oct. 1988, pp. 1525-1535

/GRT 86/ Groselj B., Tropper C.
Pseudosimulation: An Algorithm for Distributed Simulation
with Limited Memory
Int. Journal of Parallel Programming 15 (5), 1986, pp 413-456

/HAH 87/ Halsall F., Hui S.C.
Performance monitoring and evaluation of large embedded
systems
Software Engineering Journal, Sept. 1987, pp.184-192

/HAR 87/ Harter P.K.
Response Times in Level-Structured Systems
ACM Trans. on Computer Systems 5 (3), 1987, pp. 232-248

/HAU 88/ Hauser J.P.
Prototyping Communication System Software with the Distribution
Simulation and Prototyping Testbed
Military Communications Conference, San Diego, Ca.
23.-26. 10. 1988, Vol. 3, pp. 1027-1033

/HEL 80/ Helsgaun K.
DISCO - a SIMULA-based language for continuous combined and
discrete simulation
Simulation 35(1), 1980, pp.1-12

/HHK 85/ Harter P.K., Heimbigner D.M., King R.
IDD: an Interactive Distributed Debugger
Int. Conf. on Distributed Computing Systems, Paris,
13.-17.5.1985, pp.498-506

/HLA 84/ Heidelberger P., Lavenberg S.S.
Computer Performance Evaluation Methodology
IEEE Transactions on Computers 33(12), 1984,
pp.1195-1220

/HOO 84/ Hooper J.W.
A Prototyping Language for Distributed Systems
Summer Computer Simulation Conference, 23.-25.7.1984,
Boston, Mass., USA, pp. 172-179

/HUG 84/ Hughes D.J.F., Gunadi H.
S/Pascal: A portable simulation language based on Pascal
Annual SCS Multiconference, Mission Bay, USA, 2.-4.2.1984,
pp.116-120

/IEE 86/ Futurebus - a Standard Specification for an Advanced
Backplane Bus
IEEE, P896.1/D7.3, Mai 1986

/ILU 84/ Inkster J., Lomow G., Unger B.
Combined discrete and continuous simulation in ADA
Conf. on Simulation in Strongly Typed Languages
La Jolla, Ca., USA, 2nd-4th February, 1984, pp.16-21

/INT 81/ Intel iAPX 432 Object Primer
Doc. 171858-001, Rev.B,
Intel Corporation, Santa Clara, CA, USA, 1981

/INT 85a/ Intel iRMX 86 Operating System, Volume 1
(Nucleus User's Guide), Doc. 148001-001
Intel Corporation, Santa Clara, CA, USA, 1985

/INT 85b/ Intel iRMX 86 Operating System, Volume 3
(iSDM 286 system debug monitor, iRMX 86 dynamic debugger),
Doc. 148003-001
Intel Corporation, Santa Clara, CA, USA, 1985

/INT 87/ Intel Microprocessor and Peripheral Handbook, Vol II
(8253 Programmable Interval Timer)
Intel Corporation, Santa Clara, CA, USA, 1987

/ISE 77/ Isermann R.
Digitale Regelsysteme
Springer Verlag, Berlin - Heidelberg - New York, 1977

/ISO 85/ Information Retrieval, Transfer, and Management for OSI
International Organization for Standardization (ISO),
Doc. ISO/TC97/SC21 No.711-713, August 1985

/JAL 82/ Jacobson P.A., Lazowska E.D.
Analyzing Queueing Networks with Simultaneous Resource
Possession
CACM, 25 (2), 1982, pp.142-151

/JBH 85/ Jefferson D. et. al.
Implementation of time warp on the Caltech hypercube
Conf. on Distributed Simulation, San Diego, Ca., Jan. 1985,
pp.70-75

/JBW 87/ Jones S.H., Barkan R.H., Wittie L.D.
Bugnet: A real time distributed debugging system
6th Symp. on Reliability in Distributed Software and Database Systems
Williamsburg, VA., 17.-19. 3. 1987, pp. 56-65

/JEF 83/ Jefferson D.
Virtual Time
Report TR-83-213, University of Southern California,
Los Angeles, Ca 90089, USA, 1983

/JEF 87/ Jefferson D. et. al.
Distributed Simulation and the Time Warp Operating System
ACM Operating Systems Review 1987, pp. 77-93

/JES 83/ Jefferson D., Sowizral H.
Fast Concurrent Simulation Using the Time Warp mechanism,
Part 1: Local Control, Rand Note N-1906AF, The Rand
Corporation, Santa Monica, Ca., 1983

/JEW 84/ Jefferson D., Witkowski A.
An Approach to Performance Analysis of Timestamp-driven
Synchronization Mechanisms
ACM SIGACT/SIGOPS Symp. on Principles of Distibuted
Computing, Juni 1984, pp.243-253

/JLS 87/ Joyce J., Lomow G., Slind K., Unger B.
Monitoring Distributed Systems
ACM Trans. on Computer Systems 5 (2), pp. 121-151

/KAN 84/ Kasahara H., Narita S.
Integrated Simulation System for Design and Evaluation
of Distributed Computer Control Systems
9th IFAC Triennial World Congress, Budapest, Ungarn, 1984,
pp.2669-2674

/KDK 88/ Kopetz H. et. al.
Entwurf und Bewertung fehlertoleranter, verteilter Echtzeitsysteme
am Beispiel MARS
Concurrency '88, pp. 379-398

/KEL 85/ Keller H.B.
Unterstützung der Prozeßführung im nuklear-chemischen
Bereich durch den Einsatz der Simulationstechnik
2. Symposium Simulationstechnik, Bad Münster am Stein-
Ebernburg, 24.9.-26.9.1985, pp.453-458

/KES 77/ Kessels J.L.W.
An Alternative to Event Queues for Synchronization
in Monitors
CACM 20 (7), 1977, pp.500-503

/KFJ 79/ Kain R.Y., Franta W.R., Jelatis G.D.
Chimpnet: A Network Testbed
Computer Networks, Vol. 6, 1979, pp.447-457

/KIM 89/ Kim K.H.
An Approach to Experimental Evaluation of Real-Time Fault-Tolerant Distributed Computing Schemes
IEEE Trans. on Software Engineering 15 (6)
June 1989, pp. 715-725

/KLA 83/ Klaeren H.A.
Algebraische Spezifikation
Springer Verlag 1983

/KLE 75/ Kleinrock L.
Queueing Systems, Vol. II: Applications.
John Wiley, New York, 1975

/KLM 82/ Kopetz H., Lohnert F., Merker W., Pauthner G.
The Architecture of MARS
Report MA 82/2, Technische Universität Berlin, 1982

/KLU 85/ Kluge W.E.
An Approach to Computer System Modeling Based on Petri Nets
In: Methodologies for Computer System Design
W.K.Giloi, B.D.Shriver (ed.), North Holland, 1985

/KNG 84/ Knight J.C., Gregory S.T.
A Testbed for Evaluating Fault-Tolerant Distributed Systems
14th Conf. on Fault Tolerant Computing Systems
Kissimmee, Fl., USA, 20.-22.6.1984, pp.300-305

/KOM 85/ Kopetz H., Merker W.
MARS: A Maintainable Real Time System
IEEE Computer Architecture Technical Committee Newsletter,
June 1985, pp.70

/KOH 81/ Kohlhepp P.
Ein System für Entwurf, Implementierung und Echtzeitsimulation von Prozeßführungssoftware.
Fachtagung Prozeßrechner 1981, München, 10.-11. März 1981.
Informatik-Fachberichte (39), Springer Verlag, pp.65-75

/KOH 82/ Kohlhepp P.
Leistungsbewertung zeitkritischer Prozeßführungssysteme
Unveröffentlichter Bericht, 1982

/KRS 83/ Krishna C.M., Shin K.G.
Performance Measures for Multiprocessor Controllers
Performance '83, College Park, Maryland, USA, Mai 1983
pp.229-250

/KUM 86/ Kumar D.
Simulating Feedforward Systems Using a Network of Processors
Annual Simulation Symposium, Tampa, Fl., USA, 12.-14.3.1986
pp.127-144

/KVM 75/ Kiviat P.J., Villanueva R., Markowitz H.M.
The Simscript II Programming Language
CACI, Los Angeles, 1975

/LAL 84/ Lauber R., Lempp P.
Integrierte Rechnerunterstützung bei der Durchführung von Automatisierungsprojekten
Prozeßrechner 1984, 4.GI/GMR/KfK-Fachtagung
Karlsruhe, Sept. 1984, pp.336-345

/LAM 78/ Lamport L.
Time, Clocks, and the Ordering of Events in a Distributed System
CACM 21(7), 1978, pp.558-565

/LAS 85/ Lamport L., Schneider F.B.
Formal Foundation for Specification and Verification
In: Distributed Systems. Methods and Tools for Specification.
M. Paul, H.J. Siegert (Eds.), LNCS 190, Springer Verlag, 1985
pp.203

/LBR 85/ LeBlanc R.J., Robbins A.D.
Event-Driven Monitoring of Distributed Programs
Int. Conf. on Distributed Computing Systems, Paris,
13.-17.5.1985, pp.515-522

/LES 86/ Leszak M.
Modellierungs- und Simulationsinstrumentarium für fehlertolerante verteilte Systeme angewendet auf verteilte Datenbankverwaltungssysteme
Diss., Technische Universität Berlin, 1986

/LEV 81/ Levi P.
Betriebssysteme für Realzeitanwendungen
Datakontext-Verlag, Köln, 1981

/LFL 87/ Ludewig J., Färberböck H., Lichter H. et. al.
Software-Entwicklung durch schrittweise Komplettierung
Requirement Engineering Conference, St. Augustin, Mai 1987
(GMD-Studien 121), pp. 113-124

/LIE 86/ Liefländer G.
Introduction to EOS, Basic Concepts
Interner Bericht, Universität Karlsruhe, 1986

/LIR 86/ Little J.H., Riter R.R.
Prototype Real-Time Simulation Software for the Concurrent Multiprocessing Environment
IEEE/AIAA 7th digital avionics systems conference
Fort Worth, Texas, 13.-16.10.1986, pp. 209-211

/LIU 87/ Li X., Unger B.
Languages for Distributed Simulation
Conf. on Simulation and AI, San Diego, Ca., 14.-16.1.1987,
pp. 35-40

/LKK 86/ Lehmann A., Knödler B., Kwee K., Szczerbicka H.
Dialogue-oriented and knowledge-based modelling in a typical PC environment
In: Intelligent Simulation Environment. Luker, Adelsberger (Ed.). Simulations Series 17(1), 1986

/LOU 85/ Lomow G., Unger B.
Distributed Software Prototyping and Simulation in JADE
INFOR 23(1), 1985, pp.69-89

/MAA 84/ Makam S.V., Avizienis A.
An Event-Synchronized System Architecture for Integrated Hardware and Software Fault-Tolerance
4th Int. Conf. on Distributed Computing Systems,
San Francisco, Ca., USA, 1984, pp.357-365

/MAH 84/ Mahjoub A.
On the Static Evaluation of Distributed Systems Performance
The Computer Journal 27(3), 1984, pp. 201-208

/MAO 83/ Marzullo K., Owicki S.
Maintaining the Time in a Distributed System
ACM SIGACT/SIGOPS Symp. on Principles of Distibuted Computing, Monreal, Canada, 17.-19.8.1983, pp.295-305

/MAR 84/ Marr G.R.
SIM_BY_INT - The next generation simulation language
Annual SCS Multiconference, Mission Bay, USA, 2.-4.2.1984, pp.121-123

/MAS 85/ May D., Shepard R.
OCCAM and the Transputer
In: Reijns G.L., Dagless E.L. (Eds.):
Concurrent Languages in Distributed Systems, North Holland, 1985, pp.19-33

/MAT 84/ Magnenat-Thalmann N., Thalmann D.
Animated types and actor types in computer simulation and animation
Conf. on Simulation in Strongly Typed Languages
La Jolla, Ca., USA, 2nd-4th February, 1984, pp.51-56

/MCS 81/ McCormack W.M., Sargent R.G.
Analysis of Future Event Set Algorithms for Discrete Event Simulation
CACM 24(12), 1981, pp.801-812

/MIS 86/ Misra J.
Distributed Discrete Event Simulation
ACM Computing Surveys 18(1), 1986, pp.39-64

/MLE 84/ MacLeod I.M.
Data Consistency in Sensor-Based Distributed Computer Control Systems
4th Int. Conf. on Distributed Computing Systems,
San Francisco, Ca., USA, 1984, pp.440-446

/MUE 84/ Mühlhäuser M.
A Distributed Environment for Development and Performance
Evaluation of Network Applications
Interner Bericht Nr.14/84, Universität Karlsruhe

/MUE 86/ Mühlhäuser M.
Entwicklungsunterstützung für anwendungsorientierte
Verteilte Programme
Diss., Universität Karlsruhe, 1986

/MUE 88/ Mühlhäuser M.
Using Distrubuted Simulation for Distributed Application Development
Annual Simulation Symposium Tampa, Fl.
16.-18. 3. 1988, pp. 189-206

/MWM 88/ Madisetti V., Walrand J., Messerschmitt D.
WOLF: a rollback algorithm for optimistic distributed simulation systems
1988 Winter Simulation Conference, San Diego, Ca.
12.-14. 12. 1988, pp. 296-305

/NGD 86/ Neiders G.K., Goldstein A.S., Davidson J.E.
Evaluation of Prototype Digital Flight Control Algorithms
in Hardware-in-the-loop Environment
IEEE/AIAA 7th digital avionics systems conference
Fort Worth, Texas, 13.-16.10.1986, pp. 547-554

/NUT 83/ Nutt G.J.
An Experimental Distributed Modeling System
ACM Trans. on Office Information Systems 1(2), 1983,
pp.117-142

/ORY 85/ O'Reilly J.J.
SLAM II: A Tutorial
In: Simulation Technical Committee Newsletter, Mai 1985,
pp.14-17

/OVN 85/ Overstreet C.M., Nance R.E.
A Specification Language to Assist in Analysis of
Discrete Event Simulation Models
CACM 28 (2), 1985, pp. 190-201

/PAC 84/ Parng T.-M., Chang S.-C.
Logical Design of High-Level Protocols for Local Area
Network
4th Int. Conf. on Distributed Computing Systems,
San Francisco, Ca., USA, 1984, pp.166-172

/PAJ 84/ Pajak H.G.
Synchronous Debugging of real-time microprocessor-based
control systems
Simulation No.1, 1984, pp.5-11

/PEA 80/ Peacock J.K.
Distributed Simulation Using a Network of Processors
CCNG T-Report T-87, University of Waterloo, Canada, 1980

/PEF 84/ Pence J.A., Finkel D.
Modeling a computer system with SIMAN on the IBM PC
Annual SCS Multiconference, Mission Bay, USA, 2.-4.2.1984,
pp.107-111

/PES 83/ Peterson J.L., Silberschatz A.
Operating System Concepts
Addison-Wesley, Reading, Massachusetts, USA, 1983

/PIM 83/ Pimentel J.R.
Real-time simulation using multiple microcomputers
Simulation 40 (3), 1983, pp.93-104

/POH 87/ Pohlmann W.
Simulated Time and the ADA Rendezvous
4. Symposium Simulationstechnik
Zürich, 9.-11.9.1987, pp.92-102

/PRA 88/ Prakash A., Ramamoorthy C.V.
Hierachical Distributed Simulations
8th International Conference on Distributed Computing Systems
San Jose, Ca., 13.-17. 6. 1988, pp. 341-348

/PRP 79/ Pritsker A.A.B., Pegden C.D.
Introduction to Simulation and SLAM
John Wiley, New York, 1979

/PSS 85/ Posten C., Scheithauer R., Schulze W.
SIGID: ein praxisorientiertes Simulationspaket mit
Echtzeitelementen
Automatisierungstechnik 33 (12), 1985, pp.373-378

/PWM 79/ Peacock J.K., Wong J.W., Manning E.G.
Distributed Simulation Using a Network of Processors
Computer Networks No.3, 1979, pp.44-56

/RAL 87/ Ratheal S., Lombardi F.
A Software Testbed for the Design and Evaluation of
Computer Systems
Microprocessing nad Microprogramming 19 (1), 1987, pp 49-58

/RED 84/ Reijns G.L., Dagless E.L. (Ed.)
Concurrent Languages in Distributed Systems - Hardware
Supported Implementation
Proc. of the IFIP WG 10.3 Workshop, Bristol, 26.-28.3.1984

/REM 88/ Reed D.A., Malony A.D.
Parallel discrete event simulation : The Chandy-Misra Approach
Distributed Simulation 1988, San Diego, Ca.
2.-5. 2. 1988, pp. 8-13

/REY 82/ Reynolds P.F.
A Shared Resource Algorithm for Distributed Simulation
Proc. Annual Symposium on Computer Architectures,
Austin, Tx., 26.-29.4.1982, pp.259-266

/REY 83/ Reynolds P.F.
Active Logical Processes and Distributed Simulation
Proc. of the Winter Simulation Conference, 1983,
pp.262-264

/REY 88/ Reynolds P.F.
Heterogeneous distributed simulation
1988 Winter Simulation Conference, San Diego, Ca.
12.-14. 12. 1988, pp. 206-209

/RIC 82/ Rimvall M., Cellier F.E.
The GASP VI Simulation Package for Process-Oriented
Combined Continuous and Discrete System Simulation
10th IMACS World Congress on System Simulation and
Scientific Computation, Montreal, Canada, 8.-13.8.1982,
pp.413-416

/ROB 84/ Roberts S.D.
Simulation with Insight
Proc. of the Winter Simulation Conference, Dallas, Tx., USA,
28.-30.11.1984, pp.23-32

/ROH 73/ Rohlfing H.
SIMULA
BI Hochschultaschenbücher, Mannheim, 1973

/SAJ 87/ Sajkowski M.
Protocol Verification Using Discrete-Event Models
IIASA Workshop on Discrete-Event Systems: Models and
Applications, Sopron, Ungarn, 3.-7.8.1987

/SAN 77/ Sanguinetti J.W.
Performance prediction in an Operating System Design
Methodology
Diss., University of Michigan, USA, 1977

/SAN 79/ Sanguinetti J.
A technique for integrating simulation and system design.
Conference on Simulation, Measurement and Modelling of
Computer Systems. Boulder, USA, August 1979, pp.163-172

/SCH 84/ Schmidt B.
Der Simulator GPSS-FORTRAN Version 3
Fachberichte Simulation
Springer Verlag, Berlin, 1984

/SCH 87/ Schmidt R.
Einsatzmöglichkeiten der Simulation in der Werkstattsteuerung
4. Symposium Simulationstechnik
Zürich, 9.-11.9.1987, pp.520-538

/SCK 84/ Schnieder E., Kraft K.H.
Dynamische Modelle für die Signalverarbeitung mit
Prozeßrechnern
Regelungstechnik 32(1), 1984, pp.12-17

/SCR 84/ Schrott G.
Fallstudie: ein zeitkrtischer Prozeß
Informatik Spektrum 7 (2), 1984, pp.102-106

/SCS 84/ Conference on Simulation in Strongly Typed Languages
La Jolla, Ca., USA, 2nd-4th February, 1984

/SCT 82/ Schrott G., Tempelmeier T.
Messungen in einem Prozeßrechnersystem mit einem eigenen Meßrechner
TUM-I8202, Institut für Informatik, Technische Universität München, 1982

/SKR 85/ Schwan K., Kaelbling M., Ramnath R.
A Testbed for High-Performance Parallel Software
OSU-CISRC-TR-85-5, Ohio State University, Columbus, Ohio, USA, 1985

/SLR 81/ Schrott G., Lehnhoff-Roßkopf S.
Manual zum Prozeßrechner-BS MOBS
Technische Universität München, TUM-INFO/10-81-00-200/1, Oktober 1981

/SMD 83/ Smith R.W., McDonald W.C.
A Flexible Distributed Testbed for Real-Time Applications
Proc. 16th Annual Hawaii International Conference on System Sciences, 1983, pp.339-348

/SMI 84/ Smith E.T.
Debugging Tools for Message-Based, Communicating Processes
4th Int. Conf. on Distributed Computing Systems,
San Francisco, Ca., USA, 1984, pp.302-310

/SMI 87/ Smith E.
Kausalität und Temporalität in der Modellbildung
4. Symposium Simulationstechnik
Zürich, 9.-11.9.1987, pp.127-134

/SMI 88/ Smith C.U.
Applying Synthesis Principles to Create Responsive Software Systems
IEEE Trans. on Software Engineering 14 (10)
Oct. 1988, pp. 1394-1408

/SMM 86/ Shanmugan K.S., Manning T.C., Minden G.J.
Block-Oriented Systems Simulator (Boss)
1986 IEEE Military communications conference (Vol. 3)
Monterey, Ca., 5.-9.10.1986, Beitrag 36.1

/SPO 84/ Standridge C.R., Pritsker A.A.B., O'Reilly J.
Integrated Simulation Support System Concepts and Examples
Annual Simulation Symposium, Tampa, Fl., USA, 14.-16.3.1984, pp.141-160

/STU 88/ Stünkel B.
Aufbau eines objektorientierten BS-Kerns mit virtuelle Zeit zur integrierten Leistungsbewertung von Echtzeitsystemen
Diplomarbeit, Universität Karlsruhe, Jan. 1988

/SWR 87/ Seliger G., Wieneke-Toutaoui B., Rabe M.
Simulationsunterstützung bei der Planung und im Betrieb
von flexiblen Fertigungssystemen4. Symposium Simulationstechnik
Zürich, 9.-11.9.1987, pp.512-519

/TEM 87/ Tempelmeier T.
Performance Analysis of a Microprogrammed Real-Time
Operating System with an Interrupt-and-Abort Discipline
Microprocessing and Microprogramming 19 (3), 1987, pp. 233-251

/THB 83/ Thomasian A., Bay P.
Queueing Network Models for Parallel Processing of Task Systems
Int. Conf. on Parallel Processing, Bellaire, MI, USA, Aug. 1983
pp.421-428

/TOK 88/ Tokuda H., Kotera M.
Real-Time Tool Set for the ARTS Kernel
Real Time Systems Symposium, Huntsville, Al.,
6.-8. 12. 1988, pp. 289-299

/TRB 84/ Trauboth H.
Einsatz der Informationstechnik im Betrieb von Prozeßanlagen
Eingeladener Hauptvortrag, 14. GI Jahrestagung
Braunschweig, 2.-5. 10. 1984

/UBC 84/ Unger B., Birtwistle G., Cleary J., Hill D. et. al.
Jade: A simulation and software prototyping environment
Conf. on Simulation in Strongly Typed Languages
La Jolla, Ca., USA, 2nd-4th February, 1984, pp.77-83

/UNB 82/ Unger B.W., Bidulock D.S.
The Design and Simulation of a Multicomputer Network
Message Processor
Computer Networks No.6, 1982, pp.263-277

/UNG 88/ Unger B.W.
Distributed simulation
Winter Simulation Conference, San Diego, Ca.
12-14. 12. 1988, pp. 198-205

/VAS 84/ Vasudevan R.
Designing Name Servers for Distributed System
Real-Time Systems Symposium 1984,
pp.28-35

/WAL 78/ Walke B.
Realzeitrechner-Modelle
Oldenbourg Verlag, München, Wien, 1978

/WAR 79/ Ward S.L.
Simulation and Real-Time System Design: an Integrated
Approach
Diss., Northwestern University, Illinois, USA, 1979

/WBC 83/ Witten I.H., Birtwistle G.M., Cleary J., Hill D.R. et al.
JADE: A Distributed Software Prototyping Environment
Operating Systems Review 17(3), 1983, pp.10-23

/WEI 85/ Weinmann A.
Die zu erwartende Stabilitätsgrenze von Regelungen mit zufälliger Abtastperiode
Automatisierungstechnik 33 (8), 1985, pp.257-258

/WEL 83/ Wells B.H.
Distortion of simulation results due to sampling
Summer Computer Simulation Conference, 11.-13.7.1983, Vancouver, BC, Canada, pp.102-107

/WEM 88/ West J., Mullarney A.
ModSim : A language for distributed simulation
Distributed Simulation 1988, San Diego, Sa., 2.-5. 2. 1988, pp. 155-159

/WET 84/ Wettstein H.
Architektur von Betriebssystemen
Carl Hanser Verlag, München, 1984

/WHP 85/ White K.H., Paulk C.
A Distributed Real-Time Testbed Using a Time-Sharing Computing System
International computer software and applications Conference, Chicago, Ill., USA, 1985, pp.182-188

/WLU 87/ West D., Lomow G., Unger B.
Optimising time Warp using the semantics of abstract data types
Conf. on Simulation and AI, San Diego, Ca., 14.-16.1.1987, pp. 3-8

/WOL 80/ Wolfberg N.E. (ed)
The Ethernet Handbook, 2nd Edition
North Holland, Amsterdam, 1980

/WON 84/ Wong G.Y.
Ada-based distributed computer system modeling language
Conf. on Simulation in Strongly Typed Languages
La Jolla, Ca., USA, 2nd-4th February, 1984, pp.156-160

/WSY 83/ Wyatt D.L., Sheppard S., Young R.E.
An Experiment in Microprocessor-Based Distributed Digital Simulation
Proc. of the Winter Simulation Conference, 1983, pp.271-277

/ZAS 86/ Zave P., Schell W.
Salient Features of an Executable Specification Language and Its Environment
IEEE Transactions on Software Engineering 12(2), 1986, pp.312-325

/ZAV 82/ Zave P.
Testing Incomplete Specifications of Distributed Systems
Symp. on Principles of distributed computing systems, Ottawa, Canada, August 1982, pp.42-48

/ZAV 84/ Zave P.
The Operational Versus the Conventional Approach to Software Development
CACM 27(2), 1984, pp.104-118

/ZEI 76/ Zeigler B.P.
Theory of Modelling and Simulation
John Wiley, New York, 1976

/ZEI 84/ Zeigler B.P.
Multifacetted Modelling and Discrete Event Simulation
Academic Press, London, New York, 1984

/ZEI 85/ Zeigler B.P.
Discrete event formalism for model based distributed simulation
Conf. on Distributed Simulation, San Diego, Ca., Jan. 1985, pp.3-7

# Anhang

## A.1 Abkürzungen und Symbole

### (1) Abkürzungen (alphabetisch)

| | |
|---|---|
| **ASP** | Arbeitsspeicher |
| **BM** | Betriebsmittel |
| **BS** | Betriebssystem |
| **DEVN** | discrete-event-Netz (Menge gekoppelter discrete-event-Systeme) |
| **DEVS** | discrete-event-System |
| **DGS** | (gewöhnliches) Differentialgleichungssystem |
| **GVT** | globale virtuelle Zeit (bei zurücksetzenden Verfahren zur verteilten Simulation) |
| **LAN** | Lokales Rechnernetz |
| **MLT** | minimum link time, kleinste Eingabeverbindungszeit einer Modellkomponente in einer verteilten Simulation |
| **NWS** | Netzwerksynchronisation (Instanz zur Realisierung eines Synchronisationsverfahrens zur verteilten Simulation) |
| **PE** | Prozeßelement (abstraktes Modell eines technischen Prozesses oder Teilprozesses) |
| **PFS** | Prozeßführungssystem |
| **RTO** | reales (in Echtzeit bewertetes) Testobjekt in einer Simulation |
| **SU** | simulierte Umgebung (in einer Echtteilesimulation) |
| **TP** | technischer Prozeß |
| **TW** | Time Warp (rücksetzendes Verfahren zur verteilten Simulation) |
| **UISR** | Uhr-Unterbrechungsroutine |
| **VG** | Vorschaltgerät (einem physikalischen E/A-Gerät vorgeschaltetes Simulationsmodell der Bearbeitungszeit) |
| **VLAN** | Modell eines Kommunikaitonsmdiums (LAN) |
| **VR** | Modell eines Rechnerknotens |
| **VSE** | verteilte Simulation mit zeitlich enger Kopplung |
| **VSL** | verteilte Simulation mit zeitlich loser Kopplung |
| **VVS** | Verfahren zur verteilten Simulation |
| **WUP** | Uhrprozeß (W-Prozeß) |
| **ZK** | (starke) Zusammenhangskomponente der Kommunikationstopologie in einem verteilten Simulationsmodell |
| **ZS** | zentrale Simulation |

### (2) Allgemeine Notation

a) Mathematische Symbole

| | |
|---|---|
| **R** | Menge der reellen Zahlen |
| **N** | Menge der natürlichen Zahlen |
| **B** | Menge der logischen Werte (true, false) |
| $\mathbf{R}^k$ | k-dimensionaler Vektorraum über R |
| a mod b | modulo-Operation, Rest bei ganzzahliger Division |
| a div b | ganzzahlige Division |
| $\wedge, \vee, \neg,$ | logisches und,oder,nicht, |
| $\Rightarrow, \Leftrightarrow, \exists, \forall$ | impliziert,äquivalent; es existiert (Existenzquantor), für alle (Allquantor) |
| $\in, \subseteq, \cup, \cap, \setminus$ | Element,Inklusion,Vereinigung,Durchschnitt Komplement von Mengen |
| $\neq$ | ungleich |
| $\times$ oder $\Pi$ | Cartesisches Produkt von Mengen |

| | |
|---|---|
| $2^X$ | Potenzmenge (Menge aller Teilmengen) von X |
| $\|X\|$ | Kardinalität (Anzahl Elemente) der Menge X |
| f:X->Y od. f:X→Y | Abbildung (Funktion) von der Menge A in die Menge B |
| dom(f) | Definitionsbereich der Funktion f (= X) |
| im(f) | Bildbereich der Funktion f (CY) |
| f∘ g | Hintereinanderausführung von Funktionen f:X->Y, g:Y->Z |
| ≡ | Identität von Prädikaten P:X-> **B** |

b) Folgen

| | |
|---|---|
| **F**(X) | Menge der endlichen Folgen mit Elementen in X |
| \|f\| | Anzahl der Elemente einer Folge f∈ **F**(X) |
| insert: **F**(X)×X-> **F**(X) | Einfügen eines Elementes ∈X in Folge |
| remove: **F**(X)×X-> **F**(X) | Ausfügen eines Elementes ∈X aus Folge |
| leer: **F**(X)-> **B** | Folge leer? |
| erst: **F**(X)-> X | erstes Element einer Folge |

c) Zeitbereiche, Zeitfunktionen, Ein-/Ausgabefunktionen

**T** Zeitbereich (kontinuierlich (**R**) oder diskret (**N**))

[t0,t1],(t0,t1),[t0,t1),(t0,t1] t0,t1∈ **T**
Zeitintervalle (offen, abgeschlossen, links abgeschlossen, rechts abgeschlossen)

μ(M) f. M⊂T Maß der Menge M (M Vereinigung von Zeitintervallen)

ω: [t0,t1] -> X
Zeitfunktion (zeitlicher Verlauf) mit Werten in X über Zeitintervall [t0,t1]

| | |
|---|---|
| (X,T) | Menge aller Zeitfunktionen über Zeitbereich T und Wertebereich X |
| (X,T) | Menge der Zeitfunktionen über Zeitintervallen fester Länge T |
| $(X,T)^d$ | differenzierbare Zeitfunktionen (kontinuierlicher Zeitbereich T) |
| $(X,T)^c$ | sonstige zeitkontinuierliche Funktionen |
| $(X,T)^{sc}$ | stückweise konstante Zeitfunktionen (Treppenfunktionen) |
| $(X,T)^e$ | Ereignisfunktionen x: [t0,t1]-> XT∪{∅}, ∅ Leersymbol x(t)≠∅ nur an endlich vielen Stellen |

| | |
|---|---|
| ω1 • ω2 | Konkatenation zweier Zeitfunktionen ω1: [t0,t1)->X, ω2: [t1,t2)->X ω1 • ω2: [t0,t2)->X |
| $\omega\|_{[t,t']}$ | Restriktion von ω auf Zeitintervall [t, t'] |
| $\emptyset\|_{[t,t']}$ | Leere Zeitfunktion über [t, t'] |
| $\omega\|_{T>}$ | Restriktion von ω auf [t0,T], wobei dom(ω) = [t0,t1] (t1 ≥ T) |
| $\omega\|_{<T}$ | Restriktion von ω auf [t1-T,t1], wobei dom(ω) = [t0,t1) |

$x_{[t0,T]} \in (X,T)^e$ "Sprungfunktion" mit

$$x_{[t0,T]}(t) = \begin{cases} x \in X & \text{für } t = t0; \\ \emptyset & \text{sonst} \end{cases}$$

| | |
|---|---|
| F: (X,T) -> (Y,T) | Ein-/Ausgabefunktion (dom(F(ω)) = dom(ω) ∀ ω∈ (X,T)) einer I/O-Funktionszuordnung, I/O-System oder DEVS |
| $\delta\_\emptyset$ | autonome Zustandsübergangsfunktion eines DEVS |
| $\delta\_ex$ | externe Zustandsübergangsfunktion eines DEVS |
| λ | Ausgabefunktion eines I/O-Systems oder DEVS |
| ta | Zeitpunkt des nächsten autonomen Zustandsübergangs eines DEVS |
| SELECT | Rangfolgefunktion einer Menge gekoppelter DEVS-Komponenten |

| | |
|---|---|
| | mit zeitgleichen Zustandsübergängen |
| infl | Menge der von einem DEVS beeinflußten DEVS-Komponenten in einem gekoppelten System ('influencees') |

d) Syntaxspezifikation, z.B. Parameter von Schnittstellendiensten

| | |
|---|---|
| <a \| b> | Element bzw. Parameter a oder b alternativ |
| {a} | a optional |

**(3) Bezeichnungen mit spezieller Bedeutung**

a) Hierarchischer Entwurf (Kap. 2.3)

| | |
|---|---|
| $E_i$ | Detaillierungsstadium eines Systementwurfes |
| $M_i$ | abstrakte Maschine in einem Systementwurf |
| $S_i, W_i$ | Sprache, bzw. Modellierungssprache einer abstrakten Maschine |
| $Z_i$ | konkrete Objekte u. Operationen einer Maschine |
| $M_S$ | Maschine, die durch eine Sprache S definierte Dienstleistungen zur Verfügung stellt |
| $M_Z$ | Zielmaschine |
| $M_W$ | Simulationsmaschine |

b) Leistungsgrößen (Kap. 3.1.3)

| | |
|---|---|
| $G, G_N$ | quadratisches Gütekriterium der Regelgrößen eines TP |
| d.tb | Basiszeit einer Dateneinheit d |
| $a(v), a_{max}(v), a_{min}(v), ar_{max}(v)$ | Alter einer Variablen bzw. Reaktionszeit eines Datenausgangs |
| BIPZ (r,i) | Intervallbezogene Auslastung physikalischer BM (zeitlich) |
| BIPR (r,i) | Intervallbezogene Auslastung physikalischer BM (räumlich) |
| BIL (r,d,i) | Intervallbezogene Auslastung logischer BM |
| BALZ (r,d,req,i) | Aufgabenbezogene Belastung logischer BM |

c) Prozeß-Annotationen (Kap. 3.3.1, 4.2.1)

| | |
|---|---|
| c(p) | Kontrollzustand eines Prozesses p |
| z(p) | Prozeßzustand eines Prozesses p (in Sinne des BS-Kerns) |
| d(p) | Zustand der dynamischen Daten eines Prozesses p |
| at (p,<a>) | p befindet sich unmittelbar vor oder in Aktion <a> |
| after (p,<a>) | p befindet sich unmittelbar nach Aktion <a> aber noch nicht in der nächsten Aktion |
| {P} <a> {Q} ≡ | (at(p,<a>)⇒P) ⇒ (after(p,<a>)⇒Q) (P,Q Prädikate) |

d) Zeitliche Zuordnungen (Kap. 3.3, 3.4.3, 4.2)

| | |
|---|---|
| tv(p) | Virtuelle Zeit der Aktionen eines einzelnen Prozesses p (Teil des Prozeßzustands) |
| tv__su | untere zeitliche Schranke für die nächste Ausgabe eines prozeßorientierten Simulationsmodells bei leerer Eingabe |
| $t_v$:A ->T | Virtuelle Zeit aller Aktionen (eines Rechnerknotens) in einem Experiment |
| $t_e$:A ->T | Echtzeit bei der Ausführung der Aktionen in einem Experiment |
| ts:im($t_e$) ->T | Zuordnung Echtzeit -> virtuelle Zeit (Bewertungszeitfunktion) eines Experimentrechnerknotens |

e) Prozeßzustände, Prozeßmengen im BS-Kern (Kap. 4.2)

| | |
|---|---|
| r | rechnend |
| si | simulierend (S-Prozeß) |
| b | bereit |
| w__co | wartend auf Kommunikation oder Synchronisation |
| w__zeit | wartend auf Ablauf zeitlicher Wartebedingung |
| w__bm | wartend auf passive BM (ASP) |
| w__ea | wartend auf E/A-Transfers |
| it | inaktiv oder terminiert |
| | |
| FB[R] | bereite oder rechnende R- oder S-Prozesse |
| FB[W] | " " " W- Prozesse |
| FR | rechnende Prozesse |
| FS | simulierende (S-)Prozesse |
| | |
| FW[R] | blockierte R- oder S-Prozesse (beliebige Wartezustaände) |
| FWB | auf passive BM (ASP) wartende W-Prozesse |
| FWEA | auf E/A-Transfers wartende W-Prozesse |
| FBB | W-Prozesse in bm-bedingten Wartezuständen (FB[W] ∪ FWB ∪ FWEA) |
| FMB | W-Prozesse in modellbedingtem Wartezuständen |
| FI | W-Prozesse mit offenem Zeitintervall (Kap. 5.1.2) |
| | |
| ce__min (zl) | kleinste zeitliche Wartebedingung in Zeitliste zl (∞, falls leer(zl)) |
| | |
| tb(t,p,z) | zeit-, auftrags- und zustandsabhängige Gerätebearbeitungszeit (Kap. 4.2.5) |

f) Verteilte Simulation (VSE) (Kap. 5.1)

| | |
|---|---|
| $N_F,N_R,N_D$ | Knotenmengen für lokale Zeitmodi F, R, D |
| F,R,D, | selbstbestimmte und |
| $F_D,R_D,F_R$ | erzwungene Zeitmodi bei zeitlich enger Kopplung |

g) Verteilte Simulation (VSL) (Kap. 5.2)

| | |
|---|---|
| $\mathbf{K}$ | Indexmenge der Modellkomponenten |
| $E \subset \mathbf{K} \times \mathbf{K}$ | Kommunikationstopologie der Modellkomponenten (gerichtete logische Verbindungen) |
| $T_i$ | lokale Simulationszeit einer Modellkomponente $K_i$ |
| $T_{ij}$ | Kopplungszeit der log. Verbindung $(i,j) \varepsilon E$ |
| TI | maximale Kopplungszeit der Eingabeverbindungen einer Komponente $K_i$ |
| $MLT_i$ | minimale Kopplungszeit der Eingabeverbindungen einer Komponente $K_i$ |
| $TS_i$ | Sicherungszeitpunkt einer Komponente |
| $TM_i$ | Eintrag der lokalen Simulationszeit einer Komponente $K_i$ im Marker |
| $TA_i$ | Eintrag des Zeitpunktes der nächsten Ausgabe einer Komponente $K_i$ im Marker, bei leerer Eingabe |
| TA_min | kleinster Eintrag $TA_i$ im Marker |
| Tmin | kleinster Eintrag insgesamt " " |
| sb | Sicherungszustand einer Komponente |
| $F_T$: (X,T) -> (Y,T) | Ein-/Ausgabefunktion für Zeitintervalle der Länge T |
| $F_{T,\delta}$: (X,T) -> (Y,T+$\delta$) | Vorausschauende Ein-/Ausgabefunktion für Zeitintervalle der Länge T um Betrag $\delta$ |

h) Technische Prozesse und Prozeßelemente (Kap. 3.1, 6.2.1)

| | |
|---|---|
| F | Zustands-DGS |
| Fy | Meßgrößenfunktion |
| Fz | Regelgrößenfunktion |
| x | Zustandsgrößenvektor |
| y | Meßgrößenvektor |
| z | Regelgrößenvektor |
| u | Stellgrößenvektor |
| $\xi$ | Störgrößenvektor |
| $x_i, x_{ij}$ | Kopplungsgrößen zu anderen Teilprozessen |
| $P_i$ | Crossing-Prädikate eines PE |
| d(x) | diskreter Begleitzustand eines PE |
| $\delta$_a | autonome Zustandsübergangsfunktion eines PE |
| $\delta$_u | externe Zustandsübergangsfunktion eines PE |

## A. 2 Zustandsübergänge der OLGA-Netzwerksynchronisation zu Abb. 5.20 (vgl. Kap. 5.2.4, 5.2.5)

$[..]_D$,$[..]_E$ bedeuten: die geklammerten Bedingungen oder Aktionen sind spezifisch für Simulation von discrete-event-Netzen (D), bzw. für Echtteilesimulation (E).

| Zustands-übergang | Name in Abb. 5. 20 | Bedingung | Aktion |
|---|---|---|---|
| L→L | rec__mar__forward | rec__net(mar) ∧ (lnt∨tlt(mar)≤$T_K$) | send__net(m__update(mar,false)) |
| L→L | appl__rec__present | [rec__net(m)∧m∈$I_K$] | [oiq__put(m); oiq__get(m); clock__init(m.t);e(k):=0; save; ]$_E$ |
| L→L__M | rec__mar__lookahead | rec__net(mar)∧ ¬lnt∧tlt(mar)>$T_K$ | clockentry__set(l__ce, tlt(mar)) |
| L→SW | appl__send__first | appl__send(m) | ooq__put(m) |
| LW→R, SW→R | rec__msg__past | [rec__net(m)∧ m.t∈ [$TS_K$,$T_K$]]$_D$ [rec__net(m)∧ m.t<$I_K$]$_E$ | oiq__put(m); rollback; clockentry__set(r__ce, oiq.TI) |
| SW→SW | appl__rec__present | [rec__net(m)∧ m.t∈$I_K$]$_E$ | [oiq__put(m); oiq__get(m); clock__init(m.t);e(k):=0; save; ]$_E$ |
| SW→SW | appl__send__more | appl__send(m)∧m.t=ooq.T0 | ooq__put(m) |
| SW→SW | rec__mar__no__grant | rec__net(mar)∧ ¬a__send(mar, [erst(ooq) rang]$_D$) | send__net(m__update(mar, true, [erst(ooq).rang ]$_D$)) |
| SW→S | rec__mar__grant | rec__net(mar)∧ a__send(mar,[erst(ooq).rang]$_D$) | - |
| S→S | appl__send__ooq | ¬leer(ooq) | [erst(ooq).t:= Tmin(mar); ]$_E$; send__net(erst(ooq)); ooq__get(erst(ooq)) |
| S→L | return__grant | leer(ooq) | send__net(ms__update(mar, rec__set)); save; [clock__init(Tmin(mar)); e(k):=0; ]$_E$ |
| L__M→L | lookahead__timeout | lookahead__timeout ∨ lnt | send__net(m__update(mar, false) |
| L__M→S | appl__send__granted | appl__send(m) ∧a__send(mar,[m.rang]$_D$) | ooq__put(m); clockentry__cancel(l__ce) |
| L__M→SW | appl__send__not granted | appl__send(m) ∧¬a__send(mar,[m.rang]$_D$) | ooq__put(m); clockentry__cancel(l__ce); send__net(m__update(mar, true, [m.rang ]$_D$)); |

| | | | |
|---|---|---|---|
| R→R | rec__msg__more | rec__net(m)∧<br>m.t=oiq.TI | oiq__put(m) |
| R→R | rec__mar | rec__net(mar) | send__net(mar;<br>[mar__rec:=true]$_D$ |
| R→R__T | rollback__timeout | rollback__timeout<br>[∧mar__rec]$_D$ | [send__net(m__update(mar,false))]$_D$ |
| R__T→R__T | appl__rec__oiq | ¬leer(oiq) | oiq__get(erst(oiq)) |
| R__T→L | rollback__save | leer(oiq) | save;[e(k):=0;]$_E$ |