6. Simulative Leistungsmodelle

Rückblick

Wo stehen wir in unseren Überlegungen? Wir hatten erkannt (Kap. 1), daß Leistungsbewertung einerseits auf Objektexperimente abgestützt, andererseits unter Einsatz von Leistungsmodellen betrieben werden kann. Wir hatten uns (Kap. 2 und 3) die Grundlagen für die Durchführung von Objekt-Experimenten erarbeitet - und auch deren Problematik im Hinblick auf den zu leistenden Aufwand erkannt. Wir haben uns anschließend auf den Modellierungspfad begeben und uns dabei (Kap. 4 und 5) völlig auf mathematische Modelle konzentriert. Auf der Basis der Verkehrsnetze als strukturierender Vorstellung haben wir insbesondere die meßtechnisch motivierten Techniken der Betriebsanalyse studiert und bis zu dem beachtlichen Niveau der (Pseudo-)separablen Netze vorangetrieben. Wir hatten dabei darauf hingewiesen, daß die "eigentlichen" separablen Netze in stochastischem Kontext entwickelt wurden und daß wir die diesbezüglichen "analytischen" Modelle in dieser Vorlesung nur ausschnittsweise behandeln (s. aber andere Spezialveranstaltungen).

Kritik

Trotz der beträchtlichen Größe der Modellklasse der separablen Netze verbleiben eine Fülle realer Phänomene, die durch diese Klasse nicht abgedeckt werden, die also mit separablen Netzen nicht modellierbar sind. In unserer Begeisterung über die erzielten Erfolge bei der Modellklasse mit Produktformlösung haben wir solche Einschränkungen ein wenig aus den Augen verloren. Erinnern wir uns kurz an einige der "schmerzvollsten": Unsere "Stationen" sind i.w. auf die bekannte "einfache Struktur" beschränkt (was uns jede "Hierarchisierung" der Maschine zunächst verbietet); wir können keine begrenzten Puffer / Speicher berücksichtigen (die Stationen besaßen alle unbegrenzte räumliche Kapazität, welche Tatsache in der Annahme Abgangszeitpunkt=Ankunftszeitpunkt bei Stationswechsel verborgen war); wir sind auf ganz bestimmte Familien von Scheduling-Disziplinen festgelegt (vgl. die diversen "Homogenitäts"-Annahmen der Betriebsanalyse bzw. die eingeschränkte Menge analysierbarer Disziplinen der separablen Netze); die Kundenbewegung im Netz war statisch festgelegt (kein dynamisches, u.U. zustandsabhängiges "routing"); eine gleichzeitige Belegung mehrerer Stationen durch einen Kunden ist ausgeschlossen (und damit die gleichzeitige Belegung mehrerer Betriebsmittel, inklusive aller Parallelitäten der Abarbeitung); … Blättern Sie ruhig durch die Vorgängerkapitel noch einmal durch, diesmal mit dem kritischen Blick auf: Was geht denn alles **nicht**?

Auswege?

Als Antwort auf den jetzt zweifellos laut werdenden Ruf nach Vergrößerung der behandelbaren Modellklasse (d.h. nach Möglichkeiten zur modelltechnischen Berücksichtigung zusätzlicher Phänomene der realen Welt) können wir auf verschiedenen Wegen voranschreiten:

- Bleiben wir im (nicht ausgeführten) analytischen Bereich, dann ist zu berichten, daß explizite Lösungen, die wesentlich über die separablen Netze hinausgehen, trotz eindringlicher Bemühungen bisher nicht gefunden wurden. Wir könnten, im Bereich stochastischer Modelle, bescheidener werden und uns, statt um explizite Lösungen, um numerische Lösungen bemühen. In der Tat, die Ableitung analytischer Lösungen für "einheitliche, gleichmäßige" (in stochastischer Sprechweise: "stationäre") Betriebsperioden reduziert sich mathematisch auf die Lösung (sehr großer) linearer Gleichungssysteme. Methoden und Techniken zur numerischen Lösung auch großer linearer Gleichungssysteme stehen andererseits in großer Vielfalt zur Verfügung und können hier nutzbar gemacht werden. Wir kommen in Kap. 7 (leider nicht sehr ausführlich) auf diese Möglichkeit zurück. Wir können noch bescheidener werden und, statt

weitere explizite Lösungen?

numerische Lösungen

approximative Lösungen

exakter Lösungen, approximative Lösungen anstreben. Hier ist in den vergangenen Jahren in Ausweitung exakter analytischer Lösungen tatsächlich "einiges" vorgeschlagen und ausprobiert worden. Näheres dazu (wieder nur recht kurz) ebenfalls in Kap. 7.

Simulation

- Wir können allerdings auch einen drastischeren Schnitt machen und den Bereich der mathematischen Modellierung ganz verlassen, natürlich ohne auf Modellierung als Prinzip zu verzichten. Knüpfen wir dazu an Kap. 1 (in der Gegend der Abb. 1.19) wieder an: Ein Kalkül war gefragt, der aus der Beschreibung von Maschine und Last den zugehörigen Leistungswert auf formalem Wege bestimmen sollte. Und eine Idee war schon dort skizziert: Wie wäre es, wenn wir dabei ein Programm im Auge hätten, das bei seinem Ablauf den dynamischen Vorgang des Betriebs eines Rechensystems (Bearbeitung einer Last durch eine Maschine) "imitierte" - die Idee der **Simulation** war damit geboren.

Lernziele

Wir werden in diesem Kapitel die initialen Grundlagen simulativer Leistungsmodelle, leider nur sehr kurz, kennenlernen (s. aber andere Spezialveranstaltungen). Wir werden dazu (und dies bezeichnet direkt unsere Lernziele),

- verschiedene Konzepte der Simulation diskutieren und den von uns benötigten Typus, die rechnergestützte ereignisorientierte stochastische Simulation, von anderen Simulationstypen abgrenzen;

- Denkweisen und zugehörige Modellbildungen der ereignisorientierten Simulation kennenlernen und mit ihnen umzugehen verstehen, wobei insbesondere die Erfassung des Phänomens "Zeit" in einem Simulator(-Programm) in zwei unterschiedlichen Ansätzen (event-scheduling und process-interaction) zu begreifen und einzuüben ist;

- über notwendige Unterstützungen nachdenken, die eine zur Implementierung von Simulatoren geeignete Programmiersprache uns bieten sollte und (außer bei dem genannten "Zeit"-Aspekt) insbesondere bei Datenstrukturen, Zufallszahlen-Generatoren und statistischer Auswertung fündig werden;

Literatur

Klärung "Simulation"

Es muß uns von vornherein klar sein, daß wir zwar auf Simulatoren als eine Unterklasse von Leistungsmodellen gestoßen sind, daß aber andererseits die Simulation von Rechensystemen lediglich eine Unterklasse des großen Bereichs Simulation darstellt. Demzufolge finden wir zwar in den in Kap. 1 aufgeführten Lehrbüchern zur Leistungsbewertung auch Simulations-Kapitel, werden in den Literaturzitaten zu diesem Kapitel aber mehr auf allgemeine Simulations-Literatur hinweisen.

6.1 Ereignisorientierte Simulation: Konzepte und Modellbildung

Simulation ist ein schillerndes Wort, wird es doch für eine ganze Reihe von Untersuchungs- (und andere) Techniken verwendet, die (auf den "zweiten" Blick) nur wenige Gemeinsamkeiten aufweisen. Es soll uns hier nicht darum gehen, all diese Techniken in ihren Ähnlichkeiten und Verschiedenheiten kennen und unterscheiden zu lernen. Wohl sollten wir aber von vornherein präzisieren, was **wir** unter dem Begriff Simulation verstehen. Dazu einige Überlegungen:

Modellierung dynamischer Systeme

(i) In Kap. 1, und in den einleitenden Zeilen des vorliegenden Kapitels, haben wir schon auf intuitiver Grundlage skizziert, wofür wir Simulation einsetzen wollen: Es ging uns darum, den Betrieb eines Rechensystems (Bearbeitung einer Last durch eine Maschine) quantitativ zu bewerten. Wir haben das Wort Simulation erstmals verwendet, als wir uns überlegten, wie wir wohl diese Bewertung mittels eines Ersatz-Systems (anstelle des zu untersuchenden Objekt-Systems), also mittels eines **Modells** bewerkstelligen könnten. Der Betrieb eines Rechensystems ist ein über der Zeit ablaufender Vorgang, wir bewerten ein (zeitabhängig-sich-verhaltendes) **dynamisches System**. Simulation ist demnach (für uns) eine Modellierungstechnik zur Untersuchung/Analyse dynamischer Systeme. (Sollte Ihnen diese Feststellung allzu "natürlich" erscheinen: Unter dem vielgebrauchten Begriff Monte-Carlo-Simulation subsumiert man im allgemeinen numerische Techniken zur Auswertung bestimmter Integrale, also keine direkte Modellierungstechnik; wird diese Technik zur Analyse eines Systems eingesetzt, dann i.allg. zur Untersuchung zeitunabhängiger oder zu festen Zeitpunkten bestehender Eigenschaften, also zur Analyse eines statischen Systems.)

rechnergestützte Simulation

(ii) Als Vorgang der Modellanalyse hatten wir auf intuitiver Basis im Kopf, den Vorgang Rechensystembetrieb im Modell zu imitieren, also über einer (evtl. "künstlichen") Zeit nachzuspielen. Wir hatten als Informatiker wohl auch unmittelbar vor Augen, diese Imitation mittels ablaufender Programme in einem Rechner zu vollziehen. Simulation ist für uns eine **rechnergestützte** Technik. (Dies ist ebenfalls nicht so ganz natürlich: Ein strategisches Planspiel, in betriebswirtschaftlichen, volkswirtschaftlichen, militärischen, … Bereichen, läßt sich durchaus auch durch Gruppen von Menschen vollziehen; diese Ansätze werden ebenfalls als Simulationen apostrophiert.)

Modellierung zustandsdiskreter Systeme

(iii) Wir haben uns in Abschn. 2.1 ausführlich darüber unterhalten, daß wir unser Objekt-System Rechenbetrieb als **zustandsdiskretes** System auffassen sollten. Wenn sich diese Auffassung als irgendwie hilfreich für unsere Simulations-Bemühungen erwiese, dann sollten wir sie auch einsetzen, sollten (unseren Typ von) Simulation also als Analysetechnik für zustandsdiskrete Systeme verstehen. (Auch hier existieren offensichtlich andere Zielsetzungen zuhauf: Simulationen physikalischer, chemischer, thermodynamischer Prozesse, von Vorgängen der Nationalökonomie, der ökologischen Veränderungen, des Wetterverlaufs, … werden sicher die Zustandskontinuierlichkeit der zu analysierenden Systeme zu akzeptieren haben!)

Modellierung ereignisorientierter Systeme

(iv) Wir haben in Abschn. 2.1 ebenfalls diskutiert, daß in zustandsdiskreten Systemen der Begriff "Ereignis" eine besondere und hilfreiche Rolle spielt: Der Zustandsverlauf über der Zeit, eine sog. Trajektorie, ist hier eine reine Treppenkurve (vgl. Abb. 2.1.1); die Zeitpunkte der (sprunghaften) Zustandsveränderung bezeichneten wir auch als Ereigniszeitpunkte; wir hatten mit dem Ereignis-Begriff zusätzlich etwas herumgespielt, indem wir uns (in Abschn. 2.1) klarmachten, daß wir je nach Lust und Laune die Tatsache der Zustandsveränderung selbst

oder alternativ den Umstand/Auslöser der Zustandsveränderung als Ereignis auffassen konnten. Konzentrieren wir uns ausschließlich auf die zweite Interpretation (also: jede Zustandsveränderung wird von einem bestimmten Ereignis ausgelöst/verursacht), dann ist der zeitabhängige Zustandsverlauf des zu analysierenden Systems (sein "Verhalten") völlig von der Folge eintretender Ereignisse beherrscht: Ein Ereignis (Sie erinnern sich aus Abschn. 2.1: Es beinhaltete zumindest einen Zeitpunkt und einen Typ) tritt zu "seinem" Zeitpunkt auf, findet das System in bestimmtem Zustand und führt aufgrund "seines" Typs zu einer bestimmten Zustandsveränderung. Sonst passiert (in dieser Vorstellung) absolut nichts. Wir haben damit die Vorstellung des **ereignisorientierten** Systems entwickelt, (nochmals:) eines Systems, dessen Verhalten einzig und allein von der Folge eintretender Ereignisse "getrieben" wird.

Einsatz der ereignisorientierten Sicht für zustandskontinuierliche Systeme

Als Nebenbemerkung: Wir haben die Vorstellung der ereignisorientierten Systeme aus der Festlegung auf zustandsdiskrete Systeme entwickelt, indem wir erkannten, daß **jede** Zustandsänderung eines solchen Systems als von einem Ereignis ausgelöst betrachtet werden konnte. Beim Einsatz dieser Vorstellung sind wir andererseits durchaus nicht auf zustandsdiskrete Systeme beschränkt: Warum sollte ein Ereignis nicht Anlaß sein, eine kontinuierliche Zustandsgröße in ihrem Wert zu verändern? Die Eindeutigkeit der Abbildung Zustandsveränderung $\leftrightarrow$ Ereignis allerdings ist bei einem zustandskontinuierlichen System dann nicht gegeben, wenn sich eine kontinuierliche Zustandsgröße zeitkontinuierlich zu verändern vermag. Dennoch, die Zustandsdiskretheit war eine "hilfreiche" Auffassung (vgl. iii), indem sie die Vorstellung des ereignisorientierten Systems half aus der Taufe zu heben. Wir halten an beiden Vorstellungen fest.

stochastisch-statistische Auffassung

(v) Wir waren verschiedentlich darauf gestoßen worden (vgl. Abschn. 3.1), daß eine detailliert-deterministische Beschreibung der Vorgänge in einem RS, aber auch eine deterministische (Leistungs-)Bewertung des RS-Betriebs große Probleme mit sich brachte. Wir haben jeweils erkannt, daß ein **stochastisch-statistischer** Ansatz uns große Erleichterungen verschafft - und sogar zu "aussagekräftigeren" Ergebnissen verhilft. Der kritischste Punkt bestand jeweils aus der Beschreibung von Rechen-Lasten, wo sich stochastische Vorstellungen besonders verdient machten (z.B.: Ein "job" benötigt Prozessorzeit in einem Umfang, der durch eine ZV beschrieben wird), allerdings mit der Folge, daß der gesamte Betrieb als stochastischer Prozeß zu sehen war - mit den daraus resultierenden Implikationen für den Bewertungsvorgang. Wollen wir uns die erarbeiteten Erleichterungen nicht wieder verbauen, sollten wir also in unseren simulativen Imitations-Vorhaben eine Dynamik auf stochastischer Grundlage vorsehen.

Zusammenfassung

Wir sind jetzt soweit, die gesuchte Präzisierung des Begriffs Simulation mit Hilfe der diversen Stichworte vornehmen zu können: Wir beabsichtigen, dynamische, ereignisorientierte, stochastische Systeme zu simulieren; wir betreiben rechnergestützte, ereignisorientierte, stochastische Simulation.

Grundprinzipien Simulator

Wie machen wir das, praktisch gesehen? Nun, der dynamische Vorgang der Zustandsveränderungen des modellierten Systems ist mittels des dynamischen Vorgangs der Exekution des Programms "Simulator" nachzuvollziehen. Bei einem dynamischen System interessiert uns zu (potentiell allen, oft aber nur:) gewissen Zeitpunkten, wie das System zu diesen Zeitpunkten "ist", also sein Zustand. Der Zustand eines Programms (wir beziehen uns ausschließlich auf imperative Programmiersprachen) zu irgendeinem Zeitpunkt seiner Exekution ist in den Werten all seiner Programmvariablen festgehalten. Soll also aus diesen Variablenwerten "imitierenderweise" der Zustand des simulierten Systems abgelesen

werden können, dann gilt es als ersten Schritt, den **Zustandsraum** des Objektsystems (evtl. geeignet abstrahiert) **auf die Datenstruktur** des Simulators abzubilden. In Simulationsterminologie wird der diesbezügliche Teil der Simulator-Datenstruktur (i.w. eine Menge von "Zustandsvariablen") auch als **statische Struktur** des Modells bezeichnet. Bei der "Erfindung" der statischen Struktur hat man es bei simulativen Modellen leichter als bei anderen Modelltypen (vgl. die Vorgängerkapitel!): Wir können nämlich sehr "problemnah" arbeiten. Unsere Vorstellung vom Objektsystem ist in der Regel bereits hochstrukturiert: Das System besteht aus gewissen **Objekten** (die zueinander in bestimmten Beziehungen stehen); ein Objekt besitzt gewisse **Attribute**, die sein "So-Sein" charakterisieren (einige Attribute weisen zeitlich konstante Werte auf und beschreiben die Art des Objekts; andere Attribute haben zeitlich veränderliche Werte und beschreiben den Zustand des Objekts). Der wesentliche Schritt der Modellbildung bezüglich der statischen Modellstruktur (oben mit der Bemerkung "evtl. geeignet abstrahiert" mehr verschleiert als erklärt) ist also kraft unserer Vorstellung vom Objektsystem bereits vollzogen; die Zusammenfassung der Werte-Räume aller zeitveränderlichen Attribute entspricht dem (minimalen) Zustandsraum des Modells, eine zugeordnete Menge von Attributs-(Zustands-)Variablen ist geeignet als (minimale) Simulator-Datenstruktur. Falls Ihnen diese simple Modellbildung mysteriös erscheint, machen Sie sich doch bitte klar: Wenn wir über "irgend etwas" sprechen, sprechen wir in Wahrheit über unsere "Vorstellung davon", über ein (unser!) "mentales Modell" (welches nicht notwendig korrekt, geeignet, ... sein muß). Die Simplizität der Modellbildung bei der Simulation beruht darauf, daß sich solch ein mentales Modell direkt, ohne Abstriche umsetzen läßt, und daß uns der Schritt der Bildung des mentalen Modells selbst (die "eigentliche" Abstraktion Realität → Modell) meist gar nicht bewußt ist.

statische Struktur

mentales Modell

Die Datenstruktur eines Simulators repräsentiert also den Zustandsraum des Modells (bleiben wir ab jetzt immer beim "Modell"), Werte von Attribut-Variablen repräsentieren zu beliebigem Zeitpunkt seinen Zustand. Ein dynamisches Modell verändert seinen Zustand über der Zeit - die Attributwerte müssen demnach ("Nachspielen") zu geeigneten Zeitpunkten verändert werden. Verändern von Attributwerten im Simulator(-Programm) bedeutet Wertzuweisungen an die Attribut-Variablen; das Imitieren von Zustandsveränderungen erfolgt damit zwangsläufig per Ausführung gewisser Programmcode-Teile des Simulators. Wann erfolgen welche Zustandsänderungen? In einem ereignisorientierten Modell zu Ereigniszeitpunkten, entsprechend der Art des eingetretenen Ereignisses. Um also die Dynamik des Modells in den Griff zu bekommen, müssen wir uns nach der (abgehandelten) statischen Struktur zwangsläufig um seine (Simulationsterminologie:) **temporale Struktur** bemühen: Es gilt, alle Arten/Typen von Ereignissen zu identifizieren (d.h. alle Auslöser von Zustandsänderungen) und festzuhalten, welche Zustandsänderungen anläßlich des Eintretens jedes der Ereignistypen stattfinden. In Programmierungs-Jargon umgesetzt: Für jeden Ereignistyp ist ein "Stück" Programm-Code erforderlich, eine sog. **Ereignisroutine**; wir gehen davon aus, daß bei Eintritt eines Ereignisses die gemäß Ereignistyp zugeordnete Ereignisroutine durchlaufen wird; diese hat also alle Attributsveränderungen bezüglich dieses Ereignistyps, ggf. abhängig von den vorgefundenen Attributswerten ("zustands"-abhängig), durchzuführen, d.h. die entsprechenden Wertzuweisungen vorzunehmen. Insgesamt zu diesem Punkt: **Zustandsveränderungen** des Modells werden **auf Ereignisroutinen** des Simulators abgebildet.

temporale Struktur

Beispiel: Bankschalter

Beispiel 6.1.1: Sei der "Betrieb" an einem Bankschalter in einem simulativen Modell zu erfassen. Unser mentales Modell ist ohne viel Nachdenken greifbar. Hinter dem Bankschalter befindet sich ein Angestellter, der diverse Tätigkeiten zu verrichten vermag. Wir verspüren ein unmittelbares Bedürfnis nach Präzisierung: Es ist genau ein Angestellter; er verrichtet eine Tätigkeit nach der anderen jeweils vollständig; er geht nicht weg, wenn er nichts zu tun hat; er legt auch keine sonstigen Pausen ein. Vor dem Bankschalter spielt sich der Publikumsverkehr ab. Präzisierung: Kunden treffen einzeln am Schalter ein; sie stellen sich an, wenn sie warten müssen; sie haben einen ganz bestimmten Auftrag an den Angestellten im Sinn; sie gehen erst weg, wenn ihr Auftrag erfüllt ist. (Sie bemerken: Es handelt sich um nichts anderes als um eine "freundliche" Verkleidung der "FCFS-Station", vgl. Selbsttest 4.2.9)

Objekte:	**Attribute:**
(mehrere Exemplare von:)	(je Kunden-Exemplar:)
• Kunde	- Identifikation: ID, Wertemenge NAMEN, beliebig codiert - Auftrag: TASK, Wertemenge AUFTRAGSARTEN, beliebig codiert - Warteposition: P, Wertemenge "dran" POSITIONEN, Codierungsvorschlag: P {0} {1,2,…}
(genau ein Schalter, strukturiert in:)	
• Bedienplatz	- Tätigkeit: B, Wertemenge "untätig" AUFTRAGSARTEN (s. oben) - Bedienter: K, Wertemenge "keiner" NAMEN (s. oben)
• Warteschlange	- Füllungsgrad: N {0,1,2,…} - Warteliste: WS, Liste der Länge N aus (z.B.) Kunden ID's (s. oben), in Ankunftsreihenfolge sortiert

Sie bemerken: Hier ist vieles "doppelt gemoppelt", der Zustandsraum ist sehr redundant und von einem minimalen Zustandsraum weit entfernt. Wir lassen es im Moment dabei, kommen aber später darauf zurück.

Temporale Struktur? Zustandswechsel erfolgen offensichtlich genau anläßlich des Eintreffens eines Kunden (Ankunft) und des Endes der Bedienung eines Kunden (Bedienende). Wir haben es also mit genau zwei Ereignistypen und den zugehörigen Ereignisroutinen zu tun. Auf der Basis der gewählten statischen Struktur müssen wir notieren:

Ereignistypen: **Ereignisroutinen:**

- Ankunft:
  (eines bestimmten Kunden)

```
{ Notiere Attribute,
  d.h. ID- und TASK-Werte,
  dieses Kunden };
IF B="untätig"
THEN    BEGIN
        P{-Attribut dieses Kunden}:=0
                                {für "dran"};
        B:=TASK {dieses Kunden};
        K:=ID {dieses Kunden}
        END
ELSE    BEGIN
        N:=N+1;
        P{-Attribut dieses Kunden}:=N;
        { Verlängere Liste WS
          hinten um einen Eintrag,
          ID dieses Kunden }
        END
```

- Bedienende:
  (eines bestimmten Kunden)

```
{ Lösche alles von jenem Kunden,
  dessen ID=K };
IF N=0 {keine Wartenden}
THEN    BEGIN
        B:= "untätig";
        K:= "keiner"
        END
ELSE    BEGIN
        B:=TASK {des vordersten in WS};
        K:=ID {des vordersten in WS};
        { Lösche vordersten WS-Eintrag };
        N:=N-1
        {noch was?}
        END
```

Sie bemerken: Das war jetzt doch komplexer als erwartet. Und wir sind auch noch (durch die Wahl der statischen Struktur) selbst daran schuld!

Ende Beispiel 6.1.1

Redundanz des Zustandraums

Lerneffekt aus dem Beispiel? Es ist schon richtig, daß unser Verständnis von einem zu analysierenden / zu modellierenden System, also unser mentales Modell, uns unmittelbar befähigt, statische und temporale Struktur eines zugehörigen ereignisorientierten Simulators zu notieren. Allerdings hat ein zu unbedachtes Vorgehen dabei seinen Preis: Wir erhalten u.U. einen recht redundanten Zustandsraum, der recht komplexe (und irrtumsanfällige!) Ereignisroutinen nach sich zieht. Letzteres übrigens ganz prinzipiell: Redundanz heißt wechselseitige Abhängigkeit von Zustandsgrößen; heißt, daß bestimmte Relationen über Zustandsgrößen aufrechtzuerhalten sind; heißt, daß Veränderung einer Zustandsgröße zur Veränderung von anderen zwingt. Einer der diesbezüglichen Fälle aus dem Beispiel: Die ID des Kunden, dessen P "dran" signalisiert, muß in K des Bedienplatzes notiert sein.

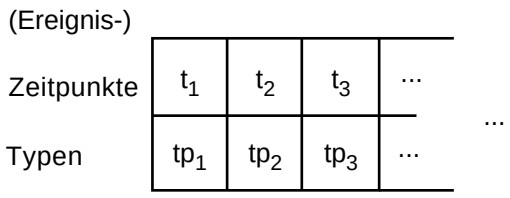
Mit Festlegung der statischen Struktur (Zustandsraum) und der temporalen

Ereignis-Zeitpunkte

Struktur (Zustandsübergangsregeln) eines ereignisorientierten Simulators sind wir noch nicht am Ziel. Wir müssen vielmehr jetzt dafür sorgen, daß (im ablaufenden Programm "Simulator") gewisse Ereignisse zu gewissen Zeitpunkten stattfinden. Bei der Erfindung eines dafür geeigneten programmtechnischen Mechanismus' ist die "Ereignisorientiertheit" des Modells eine zentrale Stütze: Zustandsänderungen sollen ja nur zu Zeitpunkten von Ereignissen stattfinden; zwischen zwei aufeinanderfolgenden, zeitlich "benachbarten" Ereignissen bleibt der Zustand unverändert (programmtechnisch ist absolut nichts zu tun); zwischen zwei aufeinanderfolgenden Ereignissen verstreicht lediglich "Zeit" (dies muß programmtechnisch nachvollzogen werden); da alle Zwischenzeitpunkte zwischen zwei aufeinanderfolgenden Ereigniszeitpunkten völlig geschehnislos verlaufen, können wir "Zeitablauf" aber als diskontinuierlichen Vorgang modellieren, der von einem Ereigniszeitpunkt unmittelbar auf den folgenden "springt" (programmtechnisch können wir eine globale Variable t = "momentane Zeit" vorsehen, welcher der Reihe nach die Zeitpunkte der Ereignisse zugewiesen werden).

Ereignisliste

Die Implementierung des skizzierten Prinzips kann in verschiedenen Formen geschehen. Deren einfachste ist die Realisierung eines "Kalenders", einer Liste aus Einträgen darüber, **wann welches Ereignis** eintritt, einer sog. **Ereignisliste** (Abb. 6.1.2).



Ereigniszeitpunkte t_i, i=1,2,..., monoton nichtfallend

Abbildung 6.1.2: Ereignisliste

Zusätzlich ist eine "zentrale Simulatorschleife" vorzusehen, welche eine globale Variable Zeitvariable t

- der Reihe nach auf die Werte t_i setzt
- und anschließend in die gemäß t_{pi} "zugehörige" Ereignisroutine verzweigt.

Zentrale Simulatorschleife

Als prinzipielle Möglichkeit dafür vgl. Abb. 6.1.3.

```
.
.
WHILE {Simulator läuft}
DO BEGIN
    {lies "vordersten" (Abb. 6.1.2: "ganz linken")
     Eintrag der Ereignisliste, Werte seien: t_x, tp_x};
    t:=t_x; {"momentane Zeit" t "springt auf" t_x}
    {verzweige zu Ereignisroutine, die durch tp_x gekennzeichnet ist,
    Annahme: Ereignisroutine gibt nach Ablauf Kontrolle hierher zurück};
    {lösche "vordersten" Eintrag der Ereignisliste}
    END;
.
.
```

Abbildung 6.1.3: Zentrale Simulatorschleife

Insgesamt: Wir simulieren "Zeit" durch diskontinuierliches "Vorrücken" der Variablen t, simulieren Zustandsveränderungen ("Trajektorie"), indem wir zu allen Ereigniszeitpunkten die jeweils zugehörige Ereignisroutine ausführen.

Zeitbegriffe

Lassen Sie uns eine häufige Quelle von Mißverständnissen hier von vornherein ausschließen: Wir haben es mit einer Reihe verschiedener Zeitbegriffe zu tun. Da ist, erstens, die **Objektzeit**, jene Zeit, in der (zumindest hypothetisch) der Betrieb eines realen Systems abläuft. Da ist, zweitens, die **Modell-** oder **Simulationszeit**, jene Zeit, die wir im Programm "Simulator" manipulieren (indem wir die Zeitvariable t auf verschiedene Werte setzen); die Modellzeit imitiert die Objektzeit, ist also "identisch" zu ihr (bis auf Translationen, wie etwa: Modellzeit "startet bei 0"). Da ist, drittens, die **Realzeit** des exekutierenden Simulatorprogramms, (das ja in irgendeinem realen Zeitintervall abläuft) nur der Vollständigkeit halber erwähnt und hier ziemlich belanglos. Und da ist, viertens, die sog. **virtuelle Zeit** des exekutierenden Simulatorprogramms (vgl. auch Abschn. 2.2: "Uhren"), jene Zeit, welche die (dieses Simulatorprogramm exekutierende) CPU benötigt, bzw. bis zu einem bestimmten Stand der Modellzeituhr (Simulation noch nicht beendet) benötigt hat; diese letzte Zeit, auch **CPU-Zeit** benannt, ist (obwohl sie mit dem simulierten System absolut nichts zu tun hat) für uns u.a. deshalb interessant, weil wir ja gelegentlich auch den Ressourcenbedarf des Simulators zu bedenken haben - Simulatoren sind notorische "Langläufer", ihre "Effizienz" (wieviel Modellzeit in wieviel CPU-Zeit) ist regelmäßig ein Problem. Zwischen den beiden allein als interessant verbleibenden Größen Modellzeit und CPU-Zeit besteht ein auf den ersten Blick verwirrender Zusammenhang. Modellzeit verstreicht "zwischen" Ereigniszeitpunkten; da wir im Simulator aber von Ereigniszeitpunkt zu Ereigniszeitpunkt "springen", insbesondere keinerlei Code ausführen, wird zwischen Ereigniszeitpunkten keine CPU-Zeit verbraucht. Andererseits geschehen die Zustandsübergänge anläßlich des Eintretens von Ereignissen in unserer Vorstellung spontan, in verschwindender Modellzeit; da wir im Simulator zum Vollzug der Ereignisübergänge aber Programmcode zu durchlaufen haben (die Ereignisroutinen), wird zu Ereigniszeitpunkten ganz deutlich CPU-Zeit verbraucht. Kurz: Endliche Modellzeitintervalle (zwischen Ereignissen) werden in verschwindender CPU-Zeit nachvollzogen, verschwindende Modellzeitintervalle (zu Ereigniszeitpunkten) in endlicher CPU-Zeit. Aus der Klärung der verschiedenen Zeitbegriffe fällt unmittelbar praktisch Verwertbares ab: Die Exekutionszeit eines Simulators (CPU-Zeit) ist von der Anzahl zu simulierender Ereignisse bestimmt - nicht (zumindest nicht direkt) von der zu überstreichenden Modellzeit (= nachzubildenden Objektzeit). Eine Erhöhung der Simulationseffizienz (z.B.:= Modellzeit/CPU-Zeit) muß, wo erforderlich, an den Ereigniszahlen ansetzen und damit am Abstraktionsniveau unseres mentalen Modells; hier gilt i.allg. (Nachdenken!): höheres Abstraktionsniveau → weniger Ereignisse.

Zusammenhang Modellzeit/ CPU-Zeit

Füllen der Ereignisliste

Nach diesem Ausflug zurück zur Hauptlinie: Wir wissen inzwischen, wie der technische Ablauf einer ereignisorientierten Simulation zu organisieren ist - auf Basis einer "gefüllten" Ereignisliste. Wie aber kommen die einzelnen Ereigniseinträge dorthin? Sie alle vor Ablauf der eigentlichen Simulation dort eintragen zu wollen (die vielleicht spontan aufkommende Idee), ist in sich widersprüchlich: Kennen wir nämlich alle Ereignisse (Zeitpunkt + Typ), dann bräuchten wir nicht mehr zu simulieren - Ereignislisten und Trajektorien sind im Prinzip aufeinander abbildbare Darstellungen eines System-"Verhaltens"; es kann notwendig sein, aus einer Verhaltensbeschreibung Bewertungsgrößen auszurechnen (vgl. Kap. 2) - aber "simuliert" muß dazu nicht mehr werden.

Bleibt als Möglichkeit, die Ereignisliste **während** der Simulation zu füllen. Wie

Ermitteln der Zukunft

das? Nun, wenn wir den Betrieb des realen Systems Schritt für Schritt simulieren, ist es uns auf Basis unseres mentalen Modells möglich, immer wieder "ein Stückchen" in die Zukunft zu schauen. Nehmen wir zur Erklärung unseren Bankschalter aus Bsp. 6.1.1 (oder den äquivalenten, abstrakteren "single-server", vgl. Kap. 4/5): Wir werden sicher zu spezifizieren haben, wann Kundenankünfte erfolgen. Wird etwa angenommen, daß (genau) alle 5 min ein Kunde eintrifft, dann wissen wir anläßlich einer Ankunft (Ereignis), daß 5 min später (ein Stückchen in der Zukunft) wieder eine Ankunft (Ereignis) stattfinden wird; oder wird, im stochastischen Gewande, angenommen, daß Zwischenankunftsabstände u.i.v. sind gemäß einer Abstands-ZV A, dann können wir uns anläßlich einer Ankunft eine Realisierung a von A "beschaffen" (dies ist die Domäne von "Zufallszahlengeneratoren") und wissen dann, daß a ZE später wieder eine Ankunft erfolgen wird. Ähnlich: Zum Zeitpunkt des Beginns einer Bedienung wissen wir, daß eine ganz bestimmte Zeit später (Bankschalter: Zeit, die für "Auftrag" benötigt wird; single-server: Realisierung s einer Bedienzeit-ZV S) ein Bedienende erfolgen wird.

Zukunft planen

Wann immer "ein Stückchen Zukunft" sichtbar wird, müssen wir es nur noch "vormerken" - technisch, indem wir einen entsprechenden Eintrag (Ereigniszeitpunkt, Ereignistyp) in die Ereignisliste eintragen, genauer gesagt: einsortieren, denn die monoton nichtfallende Folge der Ereigniszeitpunkte muß aufrechterhalten werden. Wenn wir dann im Verlauf der Simulation "an die Stelle" dieses Eintrags kommen (Ereigniszeitpunkt nach Ereigniszeitpunkt nachsimulierend), finden wir das Ereignis automatisch, zum richtigen Modellzeitpunkt, und können entsprechend handeln (Ereignisroutine).

Wer hat zu planen?

Gelegentlich ist, ein Stück in die Zukunft blickend, ein Ereignis einzuplanen, technisch also eine Operation auf der Dateistruktur "Ereignisliste" auszuführen. Wo im Simulatorcode steht der Aufruf der entsprechenden Operation? Als einzige Möglichkeit bieten sich die Ereignisroutinen an: Sie werden ja "anläßlich" eines Ereignisses durchlaufen, bei ihrer Codierung weiß man über das neu aufscheinende Stückchen Zukunft Bescheid. Die Ereignisroutinen erhalten damit eine zweite wesentliche Aufgabe zugewiesen: Neben der Veränderung des Modellzustands (wie besprochen) müssen sie die Planung der Modellzukunft vornehmen (jeweils ein Stückchen davon, soweit vorhersehbar).

Beispiel: M/M/1

Beispiel 6.1.4: Unser Bankschalter aus Bsp. 6.1.1, in der abstrahierten Form (zusätzliche Annahmen) eines "M/M/1"-Systems (exponentiell verteilte Ankunftsabstände, exponentiell verteilte Bedienzeiten)

Statische Struktur:

Wir wählen (diesmal!) einen minimalen Zustandsraum unter Ausnutzung unserer betriebsanalytischen Kenntnisse.

Objekte:	**Attribute:**
• Wartesystem	- Füllungsgrad: n $\mathbf{N}_0$ (zählt Anwesende incl. Bedientem)

Temporale Struktur:

Ereignistypen: **Ereignisroutinen:**

• Ankunft:

```
n:=n+1;
{ sortiere Ereignis (Ankunft, t+a) in Ereignisliste ein;
  t ist gegenwärtiger Zeitpunkt,
  a ist Realisierung der exponentiell, mit Parameter λ,
                    verteilten Zwischenankunfts-ZV A  };
IF n=1 {"vorher" war Wartesystem leer}
THEN
BEGIN
  { sortiere Ereignis (Bedienende, t+s) in Ereignisliste ein;
    t ist "jetzt",
    s ist Realisierung der exponentiell, mit Parameter μ,
                    verteilten Bedienzeit-ZV S  }
END
```

• Bedienende:

```
n:=n-1;
IF n>0 {System ist nicht leer}
THEN { sortiere Ereignis (Bedienende, t+s)
                    in Ereignisliste ein  }
```

Bemerke: Dank unseres wohlüberlegten Zustandsraums (keinerlei Redundanz) sind alle Manipulationen des Zustands einfach und durchsichtig; die Zukunftsplanung nimmt im Code der Ereignisroutinen vergleichsweise großen Raum ein.

Ende Beispiel 6.1.4

Notationsverabredungen

Um die Arbeit des Aufschreibens zu erleichtern, lassen Sie uns im folgenden davon ausgehen, das Einsortieren eines Ereignisses des Typs "type" zum Zeitpunkt "time" in die Ereignisliste sei mittels einer Anweisung

```
PLAN(type,time)
```

zu erledigen; ferner, das "Ziehen" einer ZV-Realisierung "wert" einer ZV mit Verteilung "verteilungskennung" und etwa nötigen Parametern "parameterliste" sei mittels

```
wert:= DRAW(verteilungskennung(parameterliste))
```

zu vollziehen. Aus der Ereignisroutine für "Bedienende" des Bsp. 6.1.4 wird damit übersichtlicher:

```
n:=n-1;
IF n>0
THEN PLAN(Bedienende, t+DRAW(negexp(μ)))
```

Simulations-Initialisierung

Wir sind mit den Prinzipien der ereignisorientierten Simulation fast fertig. Eine wesentliche "Kleinigkeit" noch: Irgendwie muß das ganze Simulieren ja anfangen (was nur möglich ist, wenn ganz zu Anfang etwas in der Ereignisliste steht), und irgendwie muß es auch wieder aufhören. Erfinden wir also schnell ein Stück Code "Simulationshauptroutine", die bei "Start Simulation" begonnen wird zu exekutieren; auf unser Beispiel abgestellt:

Simulations-hauptroutine

```
BEGIN
    t:=0 {Modellzeit startet bei 0);
    PLAN(Ankunft,DRAW(negexp(λ)))     {eine erste Ankunft wird festgelegt!};
    {sorge für Simulationsabbruch, vgl. unten};
    {zentrale Simulatorschleife der Abb. 6.1.3};
    {nochwas?}
END
```

Das "sorge für Simulationsabbruch" ist in Zusammenhang mit WHILE {Simulator läuft} der zentralen Simulatorschleife zu codieren - dem Sinn nach z.B. "bis Modellzeit > Grenze"; es gibt andere Möglichkeiten dafür.

Beobachtungen

Und noch eine (wirklich allerletzte) Bemerkung: Bis jetzt läuft unser Simulator noch spurlos ab - wir haben ja keinerlei Beobachtungen aufgezeichnet. Aufzeichnen von Beobachtungen im Ereignis-Kontext kennen wir aber schon - vgl. eventing aus Abschn. 2.1; hier ist im Prinzip alles genauso wie dort besprochen: Die Meßstellen (vgl. Abb. 2.2.2) für Aufzeichnungen sind völlig natürlich in den Ereignisroutinen vorhanden (**dritte** Aufgabe für Ereignisroutinen). Die Bewertung der Aufzeichnungen, falls unmittelbar benötigt, haben ihren Platz im "nochwas" der Simulationshauptroutine.

6.2 Spezifikation von Simulatoren: Wünschenswerte Hilfsmittel

Einschätzung Simulation

Es wird Ihnen im Verlauf des vorangegangenen Abschnitts aufgefallen sein, daß
- einerseits das Prinzip der ereignisorientierten Simulation elegant und gut verständlich ist und zur Strukturierung von Simulator-Programmen erheblich beiträgt;
- andererseits das Programmieren von Simulatoren sicher zu den "komplexeren" Programmieraufgaben zählt, zumindest auf dem anspruchslosen Sprachniveau, das wir bisher eingesetzt haben.

Hilfen:

Erleichterungen wären wünschenswert. Wenn wir dabei einzig ans "Programmieren" von Simulatoren denken, läuft dieser Wunsch auf Forderungen an die zu verwendenden Programmiersprachen hinaus, sei es in Richtung zu fordernder sprachlicher Konstrukte (hinsichtlich Syntax und Semantik) allgemeiner Programmiersprachen, sei es in Richtung der Definition spezifischer, "simulationsgeeigneter" Hochsprachen. Rein programmiersprachlich zu argumentieren, heißt hier allerdings ein wenig zu kurz greifen. Erleichterung ist ja gefragt nicht wegen irgendwelcher Notationsvereinfachungen sprachlicher Art an sich, sondern um den Vorgang der Umsetzung eines mentalen Modells in eine zugehörige, präzise, per Rechner behandelbare Beschreibung zu unterstützen. (Aus diesem Grunde heißt der vorliegende Abschnitt auch nicht "Programmierung" sondern "Spezifikation", d.h. genaue Beschreibung, von Simulatoren.) Die Unterstützung eines solchen Umsetzungsvorgangs bedient sich (zwangsweise?) immer zunächst bestimmter Denkwelten / Modellwelten, die (auf problemnahem Niveau) in erster Linie das "Erdenken" von Spezifikationen zu erleichtern haben und erst in zweiter Linie eine geeignete, einfache Notation (von Modellspezifikationen, im Rahmen einer Modellwelt). Mysteriös? Blättern Sie zurück: Ohne daß uns dies bewußt wurde, haben wir im Vorgängerabschnitt eine solche Denkwelt / Modellwelt bereits entwickelt. Sie beinhaltete eine "Ereignisliste" sowie deren Manipulation durch eine "zentrale Simulatorschleife" und durch "Ereignisroutinen"; etwa entstandene Wünsche nach Notationsvereinfachungen basieren auf dieser Denkwelt(!) - PLAN(type,time) möge als schlagendes Beispiel dienen. Diese Denkwelt ist, wie wir schon bald sehen werden, beileibe nicht die einzig mögliche. Aus einem weiteren Grund noch greifen wir zu kurz, wenn wir bei "Erleichterung" allein an Sprach-Charakteristika denken: Zur Spezifikation von Modellen könnten interaktive Menue-Techniken, graphische Darstellungen, etc. ebenso infrage kommen.

Sprache?

Spezifikation

Modellwelt

Bleiben wir aber zunächst bei den Wünschen im sprachlichen Bereich, beim "Programmieren" von Simulatoren. Wünschenswert wären Hilfsmittel in folgenden Bereichen:

Datenstrukturen

- Höhere (rekursive) Datenstrukturen, wie Listen, Schlangen, Stacks etc. leicht aufbauen zu können (oder direkt als Sprachelemente zur Verfügung zu haben), stellte eine erhebliche Erleichterung der Programmierung "unserer Art" von Modellen dar (denken Sie an die "Stationen", die "Netze von Stationen", etc.). Auch die Möglichkeit der dynamischen Instantiierung von Datenobjekten sollte hochwillkommen sein (ein "neuer" Kunde betritt das System, ein neues Exemplar der Datenstruktur zur Aufnahme seiner Attribute wird nötig). Insgesamt weisen Sprachen der ADT- (Abstrakter Datentyp-) Denkweise Vorteile, weisen objekt-orientierte Sprachen durch die zusätzliche Kapselung von Objekt-(Typ-) Operatoren zusätzliche Vorteile auf. Wir dürfen uns hier nicht verlieren; prüfen Sie aber bei Interesse SIMULA, SMALLTALK, MODULA, ADA, C++, JAVA (s. Literaturliste) auf ihre diesbezüglichen Fähigkeiten.

"Zeit"

- Die Manipulation des Phänomens "Zeit" stellt eine Besonderheit von Simulatoren dar. Wir haben mit der "Ereignisliste" eine Möglichkeit dafür kennengelernt. Diese Auffassung läßt sich auch (insbesondere wenn wir die Datenstruktur "Liste" zur Verfügung haben, s.oben) leicht "zu Programm" bringen. Einmal auf diese Besonderheit aufmerksam geworden, sollten wir aber überlegen, ob sich vielleicht alternative Denkwelten zur Erfassung von "Zeit" anbieten. Wir widmen dieser Idee noch einigen Platz in Abschn. 6.3. Die Verfolgung dieser Linie führt auf spezielle Simulationssprachen.

Stochastik/ Statistik

- Mit der Stochastik/Statistik haben wir es immer wieder zu tun. Aufgefallen dürfte Ihnen sein, daß wir das "Ziehen" einer Realisierung einer ZV bestimmter Verteilung in Abschn. 6.1 etwas stiefmütterlich behandelt haben. Ich verweise bei diesem (notwendig auszufüllenden) Punkt auf die Literatur. Ferner haben wir bisher nur angedeutet, daß wir Läufe von (stochastischen) Simulatoren auch werden "auswerten" müssen - statistische Verfahren rücken erneut ins Bild. Auch hier ist ein Ausbau unserer Kenntnisse erforderlich - s. Literatur. Vom Standpunkt der Anforderungen an eine für Simulation geeignete Sprache wünschen wir uns auf beiden Gebieten Angebote.

6.3 Modellvorstellungen zur "Zeit" in ereignisorientierten Modellen

Wir kennen bereits eine Modellwelt zur Behandlung des Phänomens "Zeit". Diese Modellwelt basiert auf einer endlichen Menge von Ereignissen und ihnen zugeordneten Zustandswechseln (implementiert: Ereignisroutinen), auf einem Kalender aus zukünftigen, für diskrete Zeitpunkte vorgemerkten Ereignissen (implementiert: Ereignisliste) und einer iterativen Dynamik der Art "berücksichtige nächstes Ereignis und vergiß es" (implementiert: Simulationshauptschleife). Dieser Ansatz trägt den Namen **event scheduling** (Ereignisplanung). Er läßt sich in jede Programmiersprache einbetten (einfach, wenn Listenstrukturen zugreifbar sind; etwas schwieriger, wenn Listen explizit, z.B. auf ARRAYs, implementiert werden müssen). Er kann aber durchaus zum Sprachbestandteil spezieller Simulationssprachen erhoben werden; solche Sprachen bieten Anweisungen der Art

Ansatz "event scheduling"

```
PLAN <event_type>,<event_time>
```

an, dazu notwendigerweise Möglichkeiten der Vereinbarung von Ereignisroutinen und ihrer Identifikation mit bestimmten <event_types>. Die Simulationshauptschleife muß bei diesen Sprachen nicht explizit programmiert werden, sie ist als Bestandteil des Laufzeitsystems standardmäßig vorhanden. Eine konkrete Sprache gemäß event-scheduling Ansatz ist SIMSCRIPT, KiVM68. Unter sehr starker Verkürzung dieser umfangreichen Simulationssprache: Die charakteristische Anweisung ist hier z.B. in der Form

Beispielsprachen

```
SCHEDULE AN <event> AT <time expression>
```

verfügbar. Um auch "jüngere" Beispiele zu geben: GASP (s. Prit74) ist eine Einbettung in FORTRAN, die ebenfalls den event-scheduling-Ansatz verfügbar macht (als Spracheinbettung nicht ganz so deutlich wie SIMSCRIPT), neben ereignisorientierten Zustandsänderungen aber auch zeitkontinuierliche erfaßt. SLAM (s. Prit84) ist eine Sprache, die event-scheduling- und (die weiter unten besprochenen) process-interaction-Denkwelten anbietet (und ebenfalls, neben den ereignisorientierten, zeitkontinuierliche Zustandsänderungen zu simulieren erlaubt).

Andere Ansätze

Neben dem event-scheduling-Ansatz existieren als Modellvorstellung der "Zeit"-Bewältigung eine ganze Reihe alternativer, z.T. deutlich stärker abstrahierender und (daher!) leichter zu handhabender Ansätze. Die einschlägige Literatur listet typischerweise zwei Alternativen, die sog. **activity-scanning** und **process-interaction** Ansätze (s. z.B. Fish73). Angesichts der Vielfalt existierender Varianten (s. Kreu86) täuscht diese strenge Eins-aus-Drei-Klassifikation aber mehr Systematik vor, als tatsächlich gegeben ist. Wir machen es uns hier noch etwas einfacher, indem wir auch den (praktisch selten eingesetzten) activity-scanning Ansatz undiskutiert lassen. Den verbleibenden (praktisch häufig eingesetzten) process-interaction Ansatz wollen wir dagegen in einigem Detail diskutieren.

Ansatz "process interaction"

Sehen wir uns dazu ganz konkret unseren Bankschalter, Bsp. 6.1.1, nochmals an. Um zu spezifizieren, wie die Dynamik des Modells gesehen wird, müssen wir nicht notwendig den einzelnen Ereignissen "nachrennen". Jeweils eine ganze Menge von Ereignissen gehören ja offensichtlich zusammen, sind Ereignisse "im Leben" wohlidentifizierbarer Einheiten des Modells. So "kommt ein Kunde und stellt sich an" (Ereignis!), "kommt dran" (Ereignis!), "ist fertig und geht weg" (Ereignis!); drei Ereignisse also, die das Leben des Objekts "Kunde" ausmachen. (Bitte lesen Sie diese Beispiele nicht als notwendige, sondern als mögliche Struk-

turierung; wir werden ja abstrakter, rücken unserem mentalen Modell näher; was in diesem mentalen Modell wohlidentifizierbare Objekte sind, in welchen Abschnitten ihr Leben verläuft, und welche Ereignisse sie daher absolvieren, spiegelt unsere in Grenzen frei gewählte Vorstellung.) Außer den Kunden gibt es den Bankangestellten, der "eine Bedienung beginnt" (Ereignis) und später "beendet" (Ereignis), eine nächste Bedienung "beginnt", ...; eine unbegrenzte Menge von Ereignissen also, die das Leben des Objekts "Bankangestellter" ausmachen (bitte nehmen Sie an dieser recht unmenschlichen Reduzierung des Angestellten keinen Anstoß!). Des weiteren existiert eine Umwelt, die einen Kunden "in die Bank einspeist" (Ereignis), später einen weiteren Kunden "einspeist" (Ereignis), später ...; eine unbegrenzte Menge von Ereignissen, die das Leben des Objekts "Umwelt" (begrenzt auf den hier relevanten Ausschnitt) ausmachen. Diese Zusammengehörigkeit von Ereignissen gibt der oben besprochene event-scheduling-Ansatz nicht wieder, obwohl unsere Modellvorstellung ganz wesentlich auf ihr beruht. Wir könnten also den Versuch wagen, die verschiedenen Objekt-"Leben" als zentrales Strukturierungsmittel zu verwenden (Einzelereignisse stellen dann eine Unterstrukturierung dar), statt wie bisher alles auf Einzelereignisse aufzubauen (und die Gesamt-Leben aus diesen zusammenzusetzen); zu erhoffen hätten wir uns in diesem Fall eine abstraktere, näher am mentalen Modell liegende Modellwelt.

Verhaltensmuster/ Prozeßmuster

Der Zirkel schließt sich! Wir haben erneut die "Vorschriften" erdacht, gemäß derer ein dynamisches Objekt sich entlang der Zeit "verhält", die Verhaltensmuster/Prozeßmuster also, die uns bereits in Kap. 4,5 hilfreich waren. Um es zu wiederholen: Auch hier sind wir bei der Vorstellung gelandet, das zeitabhängige Verhalten eines dynamischen Systems sei getragen (generiert) durch eine Menge wohlidentifizierbarer dynamischer Objekte (Prozesse), deren jedes (jeder) sich im Verlauf der Zeit entsprechend festgelegter Vorschriften (Prozeßmuster) verhält. Eine Menge/ein System paralleler (zeitlich nebeneinander ablaufender) Prozesse bestimmt unsere Modellwelt - einerseits. Andererseits können diese parallelen Prozesse nicht (in umgangssprachlichem Sinne:) wechselseitig unabhängig ablaufen. Sie erfassen ja ein "System", haben zumindest gelegentlich "etwas miteinander zu tun", müssen "interagieren" (process-interaction! Ansatz). Klar doch, im Beispiel ist das "Bedientwerden" (als Bestandteil eines Kunden-Prozesses) und das "Bedienen" (als Bestandteil des Angestellten-Prozesses) derselbe Vorgang; wenn wir auch in unterschiedlichen Prozessen zu denken beginnen, müssen doch (im Beispiel:) zumindest Bedienanfang und Bedienende in beiden Prozessen je simultan geschehen, müssen die beiden Prozesse zu (mindestens) diesen beiden Zeitpunkten synchronisiert werden.

parallele Prozesse

Interaktion

Forderungen für Realisierung

Damit haben wir einen Rahmen für die Realisierung eines process-interaction Ansatzes abgedeckt. Er besteht aus den Notwendigkeiten,

(i) Prozeßmuster notieren zu können,
(ii) Prozesse (gemäß bestimmten Musters) starten zu können;
(iii) Prozesse "interagieren" lassen zu können.

Prozeßmuster

An eine programmiersprachliche Realisierung denkend, ist (i) eher einfach: Alle höheren Programmiersprachen kennen "Prozeduren", die ja auch "benannte Ablaufmuster" sind - so ähnlich ließen sich auch Prozeßmuster notieren.

Prozeßstart

Zu (ii) ließe sich ebenfalls recht einfach eine Anweisungsart

```
START_PROCESS_OF <process pattern>
```

oder (mit Namensgebung für den neuen Prozeß)

START <process name> OF <process pattern>

erfinden, ihre Semantik mit herkömmlichen Sprachmitteln aber nicht realisieren: Es kann sich nicht um ein Analogon zum Prozeduraufruf handeln; Prozeduren geben ja die Ablaufkontrolle (erst) nach erfolgtem "Durchlauf" an die Aufrufstelle zurück (CALL-Semantik); ein Prozeßmuster dürfte aber nur "ein Stück weit" exekutiert werden (nämlich jenes Stück Code, das eine in Modellzeit 0 erfolgende Zustandsänderung nachvollzieht) und dann "anhalten" (genau dort, wo eine in endlicher Modellzeit stattfindende Aktivität zu erfassen ist), daraufhin einem anderen Prozeßmuster Gelegenheit geben, "ein Stück" Code zu exekutieren (eine Zustandsänderung, die zum nächsten berücksichtigten Modellzeitpunkt stattfindet), u.s.f. Eine Programmiersprache, in der dieser Ablauf erfaßbar wäre, muß schon ein Koroutinen-Konzept (vgl. SIMULA, Pool87), ein Task-Konzept (vgl. ADA, Ledg80) oder ein thread-Konzept (vgl. JAVA, GoJS97) anbieten! Dies, wenn wir an einen Einzelprozessor als Simulations-ausführende Instanz denken; bei einem Mehrprozessorsystem wird die Sache noch wesentlich komplexer (wir bleiben für alles Folgende bei einem Einzelprozessor).

Interaktion

Zu (iii) schließlich haben wir noch gar nicht hinreichend gründlich nachgedacht, um überhaupt konkrete Wünsche an eine Programmiersprache äußern zu können. Vertiefen wir uns also in das Interaktions-Problem. Wieder kommt alles darauf an, wie wir auf Spezifikationsebene gerne denken möchten. Ganz nahe an den zu modellierenden Problemen ist eine Denkweise, welche die Eigenständigkeit der Prozesse im System paralleler Prozesse betont: Die Prozesse haben wechselseitig im Prinzip keine Kenntnis voneinander; jeder von ihnen "marschiert" jeweils so weit er kann (exekutiert sein Prozeßmuster bis an eine Stelle, die eine Fortsetzung explizit verhindert) und notiert die Bedingungen für eine (spätere) Fortsetzung. Kompliziert? Absolut nicht! Dem einzelnen Kunden unseres Bankschalter-Beispiels schreiben wir das einfache, nur auf ihn bezogene Prozeßmuster vor:

{Stell' Dich an};
{Warte bis Du dran bist};
{Laß' Dich bedienen};
{Geh' weg}

Dies ist leicht verständlich und enthält gleich zwei "Umstände", die den Ablauf des einmal gestarteten Kundenprozesses hemmen (besser gesagt "beide" Arten solcher Umstände - es werden keine weiteren auftreten):

WAIT_UNTIL

(i) "Warte bis x" formuliert eine Bedingung für eine Fortsetzung; ist sie erfüllt, erfolgt Fortsetzung (im Beispiel: der Schalter könnte ja gerade frei sein), wenn nicht, muß gewartet werden, bis sie erfüllt ist. Wir erfinden gleich eine Notation für diese Art Warten:

WAIT_UNTIL <boolean expression>

führe zum Anhalten, wenn der Boole'sche Ausdruck (kurz: b) FALSE ist, setze genau dann (und genau zu dem Modellzeitpunkt) fort, wenn b den Wert TRUE erreicht (ohne oder mit Warten). Zwangsläufig folgt daraus: b ist im Simulatorprogramm über bestimmten Programmvariablen notiert, die gewisse Modell-Zustandsvariable nachbilden ("dran" ist das oben verwendete typische Beispiel); ist b=FALSE (muß der Kunde also warten), kann er nur über eine Wertveränderung von in b referenzierten Variablen TRUE werden;

die dazu notwendige Ausführung eines Stücks Code kann aber nur im Rahmen der Ausführung eines anderen Prozeßmusters erfolgen - der angehaltene Prozeß wartet ja, sein Code wird nicht weiter ausgeführt (im Beispiel wäre es Aufgabe des Bankschalters, das "dran" eines Kunden auf TRUE zu setzen). So ganz nebenbei haben wir damit die oben gefragte Interaktion von Prozessen erfunden (zumindest eine mögliche Form): Das Setzen und Abfragen von (im Modell:) Zustandsvariablen bzw. (im Programm:) Datenobjekten, die für eine Menge von Prozessen gemeinsam sichtbar / zugreifbar sind.

PAUSE_FOR

(ii) "Laß' Dich bedienen" formuliert im Grunde ein (Modell-)Zeitintervall, das vor Fortsetzung des Prozesses zu verstreichen hat. "Im Grunde", weil wir wieder unterschiedlich denken können: Das Bedientwerden (auf seiten des Kunden) und das Bedienen (auf seiten des Angestellten) sind ja, wie wir erkannten, dieselbe Aktivität. Wir können die Kontrolle über die Dauer dieser Aktivität (schon im mentalen Modell!) beim Kunden sehen, der dann genauer formulieren würde "bediene mich für (z.B.) 3 min"; wir können sie aber auch beim Bankangestellten sehen, womit der Kunde dann zu formulieren hätte "warte bis der Bankangestellte mit Deiner Bedienung fertig ist" - ein Typ (i) WAIT_UNTIL also. Wer auch immer die Dauer eines Zeitintervalls spezifiziert (lassen wir diese Aufgabe für das Weitere beim Kunden), muß ausdrücken können, daß vor einer Prozeßfortsetzung (Modell-)Zeit zu verstreichen hat. Erfinden wir gleich wieder eine Notation für diese (zweite) Art von Warten:

PAUSE_FOR <arithmetic expression>

führe zu einem Anhalten (der Exekution) des Prozesses für ein (Modell-) Zeitintervall, dessen Länge durch den Wert des arithmetischen Ausdrucks gegeben ist. (Wieder diese "Gehirnakrobatik": Während im realen System hochaktiv bedient wird, passiert im Modell absolut nichts außer dem Ablaufen der Zeit zwischen Bedienanfang und Bedienende!).

process interaction und Sprache

Die wesentliche Gedankenarbeit ist damit geleistet. Was folgt, sind zwangsläufige Abrundungen. Offensichtlich ist aber schon jetzt, daß sich der process-interaction Ansatz (im Gegensatz zum event-scheduling Ansatz) nicht in beliebigen Programmiersprachen notieren läßt; zu verwendende Sprachen müssen diverse speziell geeignete Fähigkeiten haben, wenn es nicht sogar spezifische Simulationssprachen sind, die process-interaction Sprachelemente beinhalten.

Beispiel: Bankschalter Verabredungen

Üben wir unser neu gewonnenes process-interaction Verständnis am Beispiel des Bankschalters. Wir treffen dazu eine Reihe von Verabredungen, um die Notation nicht völlig in Pseudocode vornehmen zu müssen:

Prozeßmuster

- Ein Prozeßmuster beinhaltet
  - einen Vereinbarungsteil, der prozeßeigene Zustandsobjekte / Datenobjekte auflistet; jeder nach diesem Muster geSTARTete Prozeß wird je ein Exemplar jedes dieser Objekte "besitzen"; auf ein Objekt eines Prozesses kann mittels
    <process name>.<object name>
    zugegriffen werden;
  - einen Codeteil, der das Verhalten jedes nach diesem Muster geSTARTeten Prozesses festlegt; er enthält insbesondere die für uns interessanten Anweisungen der WAIT_UNTIL, PAUSE_FOR, START -Arten; in diesem Codeteil kann ein laufender Prozeß auf "sich selbst" und "seine eigenen" Datenob-

jekte mittels des Prozeß"namens" CURRENT zugreifen.

- Ein Simulationsmodell enthält einen speziellen, einzelnen "Simulationshauptprozeß", mit dessen Ausführung die Simulation beginnt, und der die Aufgabe hat, Initialisierungen vorzunehmen und die Dauer der Simulation festzulegen. *Hauptprozeß*

- Es gebe einen speziellen Datentyp *Queues*
  FIFOQUEUE OF <entry type>
  Ein FIFOQUEUE-Objekt enthält FCFS-geordnete Objekte des Typs <entry type>. Als <entry type> sind auch Prozesse (eines bestimmten Prozeßmusters) zugelassen. In ADT-Denkweise exportiere ein FIFOQUEUE-Objekt die Funktionen / Prozeduren / Operatoren "enqueue", "delete", "first", "empty", auf die in der üblichen dot-Notation zugegriffen wird. Um nicht zu formal zu werden, zur Erläuterung eine Syntax- und Semantik-Klärung am Beispiel-Programm:

```
queue: FIFOQUEUE OF INTEGER;     {deklariert queue als (leere)
                                    FCFS-Schlange für INTEGER-Einträge}
b: BOOLEAN; i: INTEGER;          {nur zur Hilfe}
b:=queue.empty;                  {weist Wert TRUE zu}
queue.enqueue(1); queue.enqueue(2);
                                 {tragen Werte 1 und 2 ein, 1 ist "vorne", 2 "danach"}
i:=queue.first;                  {weist Wert 1 zu}
b:=queue.empty;                  {weist Wert FALSE zu}
queue.delete;                    {löscht "vorderes" 1, läßt 2 zurück}
i:=queue.first;                  {weist Wert 2 zu}
.
.
```

Beispiel 6.3.1: Vergleiche Bsp. 6.1.1,6.1.4; wir verwenden die M / M / 1-Spezifikationen, d.h. exponentiell verteilte Ankunftsabstände (Par. λ), exponentiell verteilte Bedienzeiten (Par. μ). *Beispiel*

Prozeßmuster:

- kunde
  Zustandsobjekte/Datenobjekte:
  dran: BOOLEAN

  Code:

```
CURRENT.dran:= FALSE             {Initialisierung};
schalter.ws.enqueue(CURRENT)     {stell' Dich an};
WAIT_UNTIL CURRENT.dran          {warte bis Du dran bist};
PAUSE_FOR DRAW(negexp(μ))        {Bedienung "M"};
schalter.belegt:= FALSE          {vgl. *)};
schalter.ws.delete               {Abgang}
```

- umwelt
  Code:

```
LOOP
   START_PROCESS_OF kunde        {Ankunft};
   PAUSE_FOR DRAW(negexp(λ))     {Zwischenankunfts-Intervall "M"};
ENDLOOP
```

- bankschalter

```
Zustandsobjekte/Datenobjekte:
    belegt: BOOLEAN;
    ws: FIFOQUEUE OF kunde;

Code:
    LOOP
    WAIT_UNTIL (NOT CURRENT.ws.empty)    {bis ein Kunde wartet};
    REPEAT
        CURRENT.ws.first.dran:= TRUE;
        CURRENT.belegt:= TRUE                {vgl. *)};
        WAIT_UNTIL (NOT CURRENT.belegt)    {bis Bedienung beendet, vgl. *)};
    UNTIL CURRENT.ws.empty
ENDLOOP
```

Simulationshauptprozeß:

```
t:=0                                    {Setze Modellzeit, vgl. **)};
START_PROCESS_OF umwelt;
START schalter OF bankschalter          {Name in kunde benötigt};
PAUSE_FOR {Länge Simulation, vgl. **)};
{Auswertungen}
```

Diskussion Beispiel

Bemerkungen

*) Das ist nicht so sehr schön, denn es durchbricht unsere Vorstellung der wechselseitigen "Unkenntnis" der Prozesse voneinander. Geht es anders? Wenn wir dabei bleiben, daß (wie hier angenommen) WAIT_UNTIL einzig über Zuständen / Objektwerten definiert ist, und wenn wir dem "bankschalter" als autonomem Prozeß ersparen wollen, auf das "beendet"-Setzen der "kunden" zu vertrauen, könnte er das für ihn notwendige Erkennen eines Bedienendes aus der Tatsache des "Weggegangenseins" des Bedienten schöpfen. Also: bankschalter merke sich, wem er "dran" gesagt hat, und WAIT_UNTIL {Warteschlangen-Erster ungleich Gemerkter}. Um das zu notieren, hätten wir aber auch noch "unsere" Notationen für pointer / Verweis / Referenz -Variable und -Operationen einführen müssen. Zu viel für hier!

**) Üblicherweise ist die Modellzeitvariable t schon implizit zu Anfang auf "0" gesetzt. Und: Für die Festsetzung der per Simulator zu überstreichenden Modellzeit gibt es eine ganze Reihe anderer möglicher "Abbruchkriterien" als nur diese Modellzeit selbst.

Ende Beispiel 6.3.1

Folgerungen

Ein erstes Fazit über den process-interaction Ansatz läßt sich unschwer ziehen: Das Denken in parallelen Prozessen ist als Modellwelt problemnah, angenehm und weniger komplex als das Denken in Einzelereignissen (des event-scheduling Ansatzes). Es ist dies jedenfalls so lange, wie die Interaktion zwischen Prozessen keine wesentliche Rolle spielt. Bei der Organisation der Interaktion tauchen unerwartete Schwierigkeiten auf (ich hoffe, Sie haben diese im Bsp. 6.3.1 bemerkt), die wir auch noch nicht völlig geklärt haben. Sehen wir ein Interaktionsdetail aus Bsp. 6.3.1 ganz präzise an: "schalter", ein Prozeß des Prozeßmusters "bankschalter", wartet gelegentlich bis "NOT CURRENT.belegt"; auf Denkebene ist dies das Warten auf Beendigung eines laufenden Bedienungsintervalls, das vom Bedienten, einem Prozeß des Musters "kunde", mittels "schalter.belegt:=FALSE" signa-

weitere Präzisierung nötig?

lisiert wird. Wann, ganz präzise, läuft "schalter" weiter? Nähmen wir die parallelen Prozesse beim Wort, so genau dann, wenn "belegt" auf "FALSE" springt; und dann, nähmen wir die parallelen Prozesse beim Wort, wären (zumindest) zwei Prozesse gleichzeitig aktiv: Der "kunde" Prozeß wendete sich seinem "delete" zu, der "schalter" seiner "ws.empty"-Prüfung. Und dann, nähmen wir die parallelen Prozesse beim Wort, ginge die Sache auch (potentiell) schief: Ist nämlich der soeben zu Ende Bediente der letzte anwesende Kunde (nach seinem Weggehen die Warteschlange also leer), und ist "schalter" um ein Quentchen schneller als der parallel aktive "kunde"-Prozeß, (führt "schalter" also seine "ws.empty"-Prüfung aus, ehe "kunde" sein "delete" ausführen konnte), dann wendet sich "schalter" fälschlich der Bedienung eines weiteren Kunden zu (jenes, der "eigentlich" schon weg ist). Hier (und in vielen ähnlichen Fällen) ist dringend weitere (Semantik-) Klärung erforderlich.

ausführende Instanz

Diese Semantik-Klärung kommt nicht umhin, die Simulations-ausführende Instanz genauer zu betrachten. Hier hatten wir uns schon bei der ersten Diskussion der START-Anweisungen auf einen Einzelprozessor festgelegt. (Bei einem Mehrprozessorsystem als ausführende Instanz müßten wir im folgenden völlig anders argumentieren.) Ein Einzelprozessor kann aber nicht mehrere Programme realiter parallel ausführen (nicht mehrere Prozesse physikalisch zeitparallel aktiv bearbeiten). Er wird notwendigerweise Parallelität derart "vorgaukeln" müssen, daß er in Koroutinen-Manier die aktiven Programme / Prozesse jeweils exklusiv, wechselseitig zeitversetzt "ein Stückchen" exekutiert / fortschreiten läßt. Eine präzise Definition dieser "Stückchen" wird die gesuchte Klärung bringen. Damit noch nicht genug, wird zur Klärung erneut die alte Zeitachse des event-scheduling Ansatzes "auf niedrigerer Ebene" ins Spiel kommen.

technischer Ablauf process interaction

Nehmen wir nun endlich die Klärung vor, indem wir den technischen Ablauf einer Simulation der process-interaction Denkwelt entwerfen (s. Abb. 6.3.2). In einer geeigneten Programmiersprache selbst zu implementieren, oder von einer speziellen Simulationssprache implizit (Laufzeit-System) anzubieten, ist wieder eine zentrale Simulatorschleife (analog zu, aber etwas anders als, Abb. 6.1.3), die auf einer Ereignisliste (analog zu, aber etwas anders als, Abb. 6.1.2) arbeitet.

(a) Ereignisliste mit Einträgen (t_i,tp_i),
gemäß t_i monoton nicht-fallend geordnet, wo
t_i: Modellzeitpunkte
tp_i: Verweise auf Prozesse samt (impliziten oder expliziten)
Verweisen "auf die Stelle, wo sie (in ihrem Prozeßmuster) stehen"

(b) Zentrale Simulatorschleife

```
t:=0            {setzt Anfang Modellzeit};
{setze Erstereignis, nämlich "Start Simulationshauptprozeß", vgl. *)};
LOOP
  {lies "vordersten" Eintrag der Ereignisliste, Werte seien: tx,tpx};
  t:=tx         {"momentane" Zeit "springt"};
  {aktiviere Prozeß gemäß tpx, vgl. *)};
  {entferne Ereignisliste-Eintrag};
  WHILE {UNTIL-Bedingungen TRUE, vgl. *)}
  DO {aktiviere einen diesbezüglichen Prozeß, vgl. *)}
ENDLOOP
```

Abbildung 6.3.2: Organisationsteil für process-interaction Simulation

Diskussion

*) Diskussion Abb. 6.3.2:
Immer wieder ist ein bestimmter Prozeß zu "aktivieren", d.h. die Exekution seines Prozeßmusters dort fortzusetzen, "wo er gerade steht". Er kann stehen:
- ganz am Anfang; in diesem Falle wurde er aufgrund einer START-Anweisung vorgemerkt, die implizit (à la PLAN) zu einem Eintrag in die Ereignisliste umgesetzt wurde (auch das spezielle "Erstereignis"-Setzen funktioniert genau so; alle anderen STARTs werden zur selben Modellzeit, aber "nach" dem gerade bearbeiteten Eintrag vorgemerkt);
- an einer PAUSE_FOR Anweisung; in diesem Falle wurde er für bekannten Zeitpunkt (à la PLAN) in der Ereignisliste vorgemerkt;
- an einer WAIT_UNTIL Anweisung; in diesem Fall ist er nicht in der Ereignisliste vorgemerkt; vielmehr wird seine Bedingung durch die Aktivitäten eines in der Ereignisliste vorgemerkten, aktivierten Prozesses erfüllt (bzw. durch die Aktivitäten eines Ereignislisten-Vorgemerkten aktivierten Vorgängers, …),.

weitere Fragen

Wir wollen uns nicht in weitere Details vertiefen (z.B. was ist, wenn mehr als ein WAIT_UNTIL-Wartender fortsetzen könnte, was ist mit PAUSE_FOR 0, mit WAIT_UNTIL TRUE, …). Wohl sollten wir aber explizit feststellen, daß durch die Festlegung, wo ein Prozeß "stehen" kann (bzw. wo er nicht stehen kann) auch die "Stückchen" exklusiver Abarbeitung eines Prozeßmusters geklärt sind: Ein Prozeß läuft bis zu einem WAIT_UNTIL, bis zu einem PAUSE_FOR oder eben bis zu seinem Ende (gemäß Prozeßmuster).

Implementierungsfragen

Wenn Sie sich in die Lage des Implementierers des Codes versetzen, der WAIT_UNTIL Anweisungen in maschinennäheren Code umsetzt (bei einer Simulationssprache also des Compiler- bzw. Laufzeitsystem-Implementierers), dann bekommen Sie sicherlich Bedenken bzgl. der Laufzeiteffizienz dieses Sprachkonstruktes: Der Boole'sche Ausdruck b einer Anweisung WAIT_UNTIL b war in keiner Weise eingeschränkt, er kann sich auf beliebige Variablen des Simulatorprogramms beziehen; dies aber bedeutet, daß **jede** Wertzuweisung an eine Variable potentiell das b **jedes** WAIT_UNTIL-wartenden Prozesses betreffen kann; was wiederum impliziert, daß immer wenn ein Prozeß anhält (d.h. ein Stück Prozeßmuster durchlaufen wurde, in dem potentiell Variablenzuweisungen enthalten waren) auch sämtliche b's aller WAIT_UNTIL-Wartenden bezüglich (jetzt) möglicher Fortsetzung zu überprüfen sind. Dies kann sehr aufwendig werden! (Man kann sich "raffiniertere" Lösungen ausdenken - komplex sind sie alle.)

Ineffizienzen

Auswege

Auswege aus dieser Effizienz-Bedrohung? Hier zwei verschiedene, die in einander entgegengesetzte Richtungen weisen:

unbeschränktes Warten, PASSIVATE

- Näher hin zur Implementierung (→ weiter weg vom mentalen Modell):
  Wir streichen das ineffiziente WAIT_UNTIL und ersetzen es durch eine PASSIVATE-Anweisung, die einen ausführenden Prozeß (unbedingt!) anhält. Damit kann ein Prozeß zwar noch feststellen, ob er fortgesetzt werden kann (Überprüfung der Bedingung b des gestrichenen WAIT_UNTIL), kann sich ggf. auch korrekterweise selbst anhalten (IF NOT b THEN PASSIVATE), kann aber nicht mehr ausdrücken, wann er wieder "aufzuwecken" ist. Wenn er dies nicht selbst kann, muß es notgedrungen ein anderer Prozeß übernehmen: Eine

ACTIVATE

ACTIVATE <process name> Anweisung ist zwingend erforderliches Pendant zum PASSIVATE. Damit müssen sich die Prozesse des Systems gegenseitig "kennen" und müssen darauf achten, daß sie (wenn ihre Aktivitäten zum Auf-

wecken eines anderen Prozesses führen können) entsprechende ACTIVATEs ausführen. Insgesamt wird der Simulator-Code damit wieder länger und komplexer, aber eben i.allg. effizienter (da aus der Kenntnis des Problems eine Reihe unnötiger "b"-Überprüfungen entfallen). Wenigstens eine kleine Bestätigung des Komplexer-Werdens am Bsp. 6.3.1: Ein "kunde" muß bei Ankunft daran denken, daß "ws" gerade "empty" sein könnte, daß "schalter" sich also PASSIVATEd haben könnte - und muß ggf. "schalter" ACTIVATEn. Eine verbreitete process-interaction Simulationssprache, die diesen PASSIVATE / ACTIVATE Ausweg wählt, ist SIMULA (z.B. Pool87, Fran77).

Warten auf spezielle Bedingungen

- Näher hin zum mentalen Modell (→ weiter weg von der Implementierung): Noch näher am mentalen Modell und dennoch effizienter? Ja, aber unter Aufgabe der Allgemeinheit des "b" der WAIT_UNTIL-Anweisungen. Nur noch auf ganz bestimmte b's kann gewartet werden, wobei solche Einschränkungen (sollen sie nicht wirklich hinderlich sein) sorgfältig auf die zu simulierenden Systeme abgestimmt sein müssen. Wir beschreiten damit einen Weg hin zu sog. **Szenario-Sprachen**, die nicht mehr zur (hier: ereignisorientierten) Simulation generell, sondern zur Simulation bestimmter Problemklassen entworfen sind. Ein verbreitetes Beispiel einer Szenario-Sprache ist GPSS (z.B. Gord75) - entwickelt zur Spezifikation von "Warteschlangensystem"-Problemen (also gerade für Probleme, wie wir sie unter der Überschrift Leistungsmodellierung vor uns haben; nicht sofort begeistert sein: GPSS ist ansonsten keine sehr moderne Sprache). In GPSS notiert sich das "kunde"-Prozeßmuster (als Beispiel) sehr einfach mittels einer SEIZE-Anweisung, einem WAIT_UNTIL <ganz besonders b>, das Warten auf Freisein nämlich einer Bedien-"facility", wie sie für Probleme einer bestimmten Problemklasse ("Szenario") charakteristisch ist.

SEIZE schalter	Die "facility" schalter wird belegt, falls diese frei ist - sonst wird gewartet; der "kunde" hat nicht einmal mehr Kenntnis von irgendeiner Warteschlange.
ADVANCE s	Jetzt hat der Kunde den Schalter und behält ihn für ein Zeitintervall der Länge s (die Zeit wird ADVANCEd, vgl. PAUSE_FOR).
RELEASE schalter	Und jetzt wird Schalter wieder freigegeben (und implizit u.U. ein in SEIZE Wartender erlöst).

Lassen Sie uns unseren Ausflug hier abbrechen, nochmals mit dem Hinweis auf Kreu86, wo eine Vielzahl problemnäherer und implementationsnäherer Simulations-Modellwelten vorgestellt und diskutiert werden.

Leerseite

Literatur **Ereignisorientierte Simulation**
(Discrete Event Simulation)

Allgemeine Lehrtexte (Auswahl)**:**

BaCN96 Banks J., Carson J., Nelson B.; Discrete-event system simulation; Prentice-Hall 1996

Cass93 Cassandras C.; Discrete event systems: Modeling and performance analysis; Aksen Ass. 1993

DeVa89 Delaney,W. / Vaccari,E.: Dynamic models and discrete event simulation; Marcel Dekker 1989

Fish78 Fishman,G.S.: Principles of discrete event simulation; Wiley 1978

Fish95 Fishwick,P.A.; Simulation model design & execution: Building digital worlds; Prentice-Hall 1995

Gord78 Gordon,G.: System simulation (2nd ed); Prentice-Hall 1978

Jain91 Jain,R.: The art of computer systems performance analysis; Wiley 1991

Krüg75 Krüger,S.: Simulation; Grundlagen, Techniken, Anwendungen; de Gruyter 1975

LeSm79 Lewis,T.G. / Smith,B.J.: Computer principles of modeling and simulation; Houghton Mifflin 1979

MaGn72 Maisel,H. / Gnugnoli,G.: Simulation of discrete stochastic systems; Science Research Associates 1972

McDo87 MacDougall,M.H.: Simulating computer systems; Techniques and Tools; MIT Press 1987

Mitr82 Mitrani,I.: Simulation techniques for discrete event systems; Cambridge University Press 1982

Neel87 Neelamkavil,F.: Computer simulation and modelling; Wiley 1987

Sieg91 Siegert,H-J.: Simulation zeitdiskreter Systeme; Oldenbourg 1991

Formale Modellierungs-Aspekte vorherrschend:

Zeig76 Zeigler,B.P.: Theory of modelling and simulation; Wiley 1976

Zeig84 Zeigler,B.P. (ed): Multifacetted modelling and discrete event simulation; Academic Press 1984

Statistik-Aspekte vorherrschend:

BrFS87 Bratley,P. / Fox,B.L. / Schrage,L.E.: A guide to simulation; Springer 1987 (2nd ed)

Klei74 Kleijnen,J.P.C.: Statistical techniques in simulation; Parts I, II; Dekker 1974

KlvG92 Kleijnen,J.P.C. / van Groenendaal,W.; Simulation: A statistical perspective; Wiley 1992

LaKe82 Law,A.M. / Kelton,W.D.: Simulation modeling and analysis; McGraw-Hill 1982
LaKe91 McGraw-Hill 1991 (2nd ed)

Mihr72 Mihram,A.G.: Simulation; Statistical foundation and methodology; Academic Press 1972

MiCo68 Mize,J.H. / Cox,J.G.: Essentials of simulation; Prentice-Hall 1968

Sprach-Aspekte vorherrschend:

FrPW90 Frauenstein,T. / Pape,U. / Wagner,O.:
Objektorientierte Sprachkonzepte und Diskrete Simulation; Springer 1990

Kreu86 Kreutzer,W.: System simulation - Programming styles and languages;
Addison-Wesley 1986

Page88 Page,B. et al; Simulation und moderne Programmiersprachen; Modula-2, C, Ada;
Springer 1988

sowie für jede **Simulationssprache** mindestens ein Buch, z.B.:

BoKP76 Bobillier,P.A. / Kahan,B.C. / Probst,A.R.: Simulation with GPSS and GPSS V;
Prentice-Hall 1976

Fran77 Franta,W.R.: The process view of simulation; North-Holland 1977

KiVM68 Kiviat,P.J. / Villanueva,R. / Markowitz,H.M.; The SIMSCRIPT II programming language;
Prentice-Hall 1968

KiPr69 Kiviat,P.J. / Pritsker,A.A.B.;
Simulation with GASP II: A FORTRAN based simulation language; Prentice-Hall 1969

Page91 Page,B.: Diskrete Simulation - Eine Einführung mit Modula-2; Springer 1991

Pool87 Pooley,R.J.: An introduction to programming in SIMULA; Blackwell Scientific 1987

Prit74 Pritsker,A.A.B.; The GASP IV simulation language; Wiley 1974

Prit84 Pritsker,A.A.B.: Introduction to simulation and SLAM II; Wiley 1984

Schr91 Schriber,T.J.; An introduction to simulation using GPSS / H; Wiley 1991

weitere Referenzen des Textes:

Hoff87 Hoffmann, H.-J.: SMALLTALK verstehen und anwenden; Hanser 1987

Ledg80 Ledgard,H.: ADA - An introduction; Springer 1980

Wirt82 Wirth,N.: Programming in MODULA-2; Springer 1982

GoJS97 Gosling J., Joy B., Steele G.L.: Java TM - die Sprachspezifikation; Addison-Wesley 1997