3. Spezifikation von Simulatoren

3.1 Konzeptuelle Bedürfnisse

Auffällig:
- Prinzip der ereignisorientierten Simulation
  zwar elegant, verständlich, "strukturiert"
- Programmieren von Simulatoren aber eher
  "komplexere" Programmieraufgabe,
  zumindest auf benutztem (anspruchslosem) Sprachniveau

Erleichterungen wünschenswert!

"Programmieren" von Simulatoren:

→ Forderungen an zu verwendende Programmiersprachen
- sprachliche Konstrukte (Syntax + Semantik)
  allgemeiner Programmiersprachen
- Definition spezifischer,
  "simulationsgeeigneter" Hochsprachen

Aber: mehr als "Notationsvereinfachungen" gefragt:
- Vorgang der Umsetzung mentales Modell in
  zugehörige, präzise Beschreibung zu unterstützen
  (Nicht "Programmierung" sondern "Spezifikation")
- (dazu nötig:) Einführung Denkwelten/Modellwelten,
  die (auf problemnahem Niveau)
  "Erdenken" von Spezifikationen erleichtern

Mysteriös?
(Unbewußt) solche Denkwelt/Modellwelt bereits entwickelt:
- "Ereignisliste" samt Manipulation durch
  - "zentrale Simulatorschleife" und
  - "Ereignisroutinen"
- Notationswünsche, basierend auf dieser Denkwelt(!)
  zB PLAN(type,time)

Diese Denkwelt ist nicht die einzig mögliche (s.später)

Auch sonst: "Erleichterung = Sprach-Charakteristika"
nicht zeitgemäß
Spezifikation mittels interaktiver Menue-Techniken,
graphischer Darstellungen, … , hochaktuell

Zunächst dennoch sprachliche Wünsche
Willkommen wären Hilfsmittel in folgenden Bereichen:

- Höhere (rekursive) **Datenstrukturen**,
  Listen, Schlangen, Stapel etc. hilfreich
  zB "Warteschlange aus Kunden",
  auch: "Ereignisliste"
  Dynamische Instantiierung von Datenobjekten
  zB "neuer Kunde trifft ein" → neues Exemplar
  Datenstruktur nötig
  ADT-basierte Sprachen,
  objekt-orientierte Sprachen
  durch Kapselung von Objekt-(Typ-) Operatoren Vorteile
  Prüfe zB SIMULA, SMALLTALK, MODULA,
  ADA, C++, JAVA, … (s. auch Page88, FrPW90)

- **Manipulation** Phänomen "**Zeit**"
  ist Besonderheit von Simulatoren
  "Ereignisliste" sicher eine Möglichkeit dafür
  (und leicht "zu programmieren")
  Alternative Denkwelten zur Erfassung von "Zeit" s. 3.2
  (führt auf spezielle Simulationssprachen)

- Immer wieder: **Stochastik/Statistik**
  - "Ziehen" von Realisierung einer ZV
    bestimmter Verteilung nur "gestreift", s. 3.3
  - "Auswertung" von Läufen (stochastischer) Simulatoren
    erfordert statistische Verfahren, s. 4. u. 6.

  Angebote "in Sprache" erwünscht

3.2 Modellvorstellungen zur "Zeit" in ereignisorientierten Modellen

Bereits bekannte Modellwelt:

- endliche Menge von Ereignistypen
  und ihnen zugeordnete Zustandswechsel
  (implementiert in: Ereignisroutinen),
- Kalender aus zukünftigen,
  für diskrete Zeitpunkte vorgemerkten Ereignissen
  (implementiert in: Ereignisliste)
- iterative Dynamik der Form
  "berücksichtige nächstes Ereignis und vergiß es dann"
  (implementiert in: Simulationshauptschleife)

Ansatz trägt Namen **event scheduling** (Ereignisplanung)

event scheduling in jede Programmiersprache einbettbar
- einfach, wenn Listenstrukturen zugreifbar
- schwieriger, wenn Listen explizit zu implementieren
  (zB in ARRAYs)

event scheduling durchaus als Sprachbestandteil
spezieller Simulationssprachen einführbar

dann
- existieren Anweisungen der Art
  PLAN <event_type>,<event_time>

- gibt es Möglichkeiten der Vereinbarung von
  Ereignisroutinen und zu ihrer Identifikation
  mit bestimmten <event_types>

- wird Simulationshauptschleife Standard-Bestandteil
  des Laufzeitsystems (nicht explizit zu programmieren)

Beispiele:

- SIMSCRIPT (MaHK63, KiVM68):
  Charakteristische Anweisung verfügbar in der Form
  SCHEDULE AN <event> AT <time expression>

- GASP (KiPr69,Prit74,PrYo75):
  event-scheduling Einbettung in FORTRAN (+ PL/I)
  (als Spracheinbettung nicht so deutlich wie SIMSCRIPT);
  umschließt (neben ereignisorientierten)
  auch zeitkontinuierliche Zustandsänderungen

- SLAM (Prit84) bietet (neben event-scheduling)
  auch process-interaction-Denkwelt (s. unten) an,
  umschließt ebenfalls (neben ereignisorientierten)
  auch zeitkontinuierliche Zustandsänderungen

Neben **event-scheduling** existieren
(als Modellvorstellung für "Zeit"-Organisation)
eine Reihe alternativer Ansätze:
z.T. deutlich stärker abstrahierend
und (daher!) leichter zu handhaben

Typischerweise gelistet (zB Fish73):
- **activity-scanning**
- **process-interaction** Ansätze

Tatsächlich:
größere Vielfalt, weniger Systematik (Kreu86)
zB "ABC approach" (Tysz99):
Verbindung activity-scanning / event scheduling

event scheduling / activity scanning / process interaction :

drei verschiedene "Sichten" der Dynamik
eines ereignisorientierten Systems

damit auch drei verschiedene Angebote,
mentales Modell zu strukturieren / zu spezifizieren

Zur Begriffsbildung:

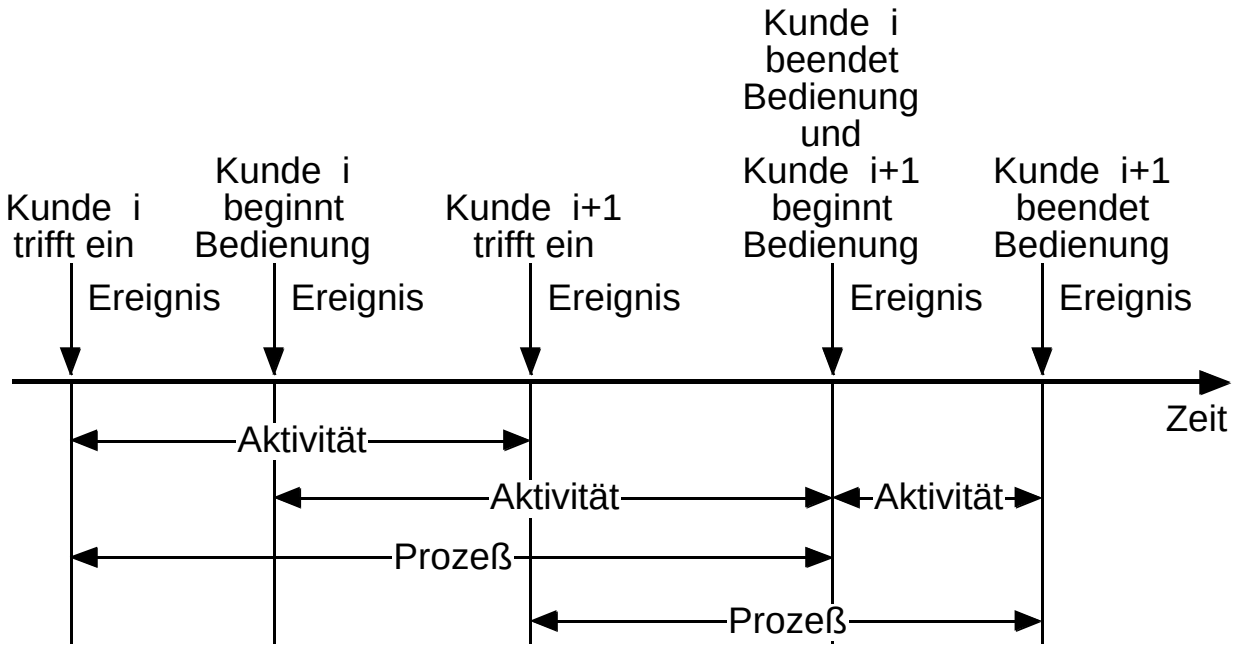


Abbildung 3.2.1: Geschehnisablauf beim Bankschalter

Bemerkung: Es gibt andere Möglichkeiten,
Ereignisse, Aktivitäten, Prozesse zu definieren!

Entsprechend Abb. 3.2.1:
- drei genannte "Sichten"
- und (zugehörige) konzeptuelle Rahmen der Zeitsteuerung

gelistet in Folge steigender Abstraktion

event scheduling: bereits bekannt, verbreitet,
explizite Manipulation eines Kalenders

activity scanning: nur wenig implementiert
(zB CSL, ECSL - Buxt62/67,Clem66,
SIMON(75) - Hill75)

Denkwelt: Es gibt Menge von Aktivitäten, die "wissen",
- wann / ob sie beginnen oder enden können
- was anläßlich Beginn / Ende zu tun ist
(iw Zustandsänderungen)

Implementierungsidee:
Aktivität besitzt (und setzt) eigene "Uhr"
Aktivität zugeordnet ist "Aktivitätsroutine"
Aktivitäten werden (wiederholt) "angestoßen"

Schema 3.2.2: activity scanning am Beispiel 2.1

- Aktivitäten sind Ankunftserzeugung,
Bedienung

- Initiierung: {implizit: alle Uhren auf "anfang" gesetzt}
{Setze Uhr Ankunftserzeugung (>"anfang")}
{bereite Auswertung vor}

- Ablauf:
Zeit := "anfang";
WHILE {Simulation nicht beendet}
DO BEGIN
{finde Uhr mit kleinster "zukünftiger" Zeit}
{setze "momentane Zeit" auf diese Zeit}
REPEAT UNTIL {keine Aktivität spricht an}
{aktiviere der Reihe nach alle Akt.Routinen}
END

- Beendigung:
{Auswertungsaktivitäten}

- Aktivitätsroutinen

  - Ankunftserzeugung

```
IF {Zeit ≥ Uhr Ankunftserzeugung}
THEN BEGIN
        {erzeuge Kunden, → Warteschlange};
        {setze Uhr Ankunftserzeugung}
        END
```

  - Bedienung

```
IF {Zeit ≥ Uhr Bedienung}
THEN BEGIN
        IF {Schalter belegt}
        THEN BEGIN
                {entlasse Kunden};
                {befreie Schalter}
                END
        IF NOT {Warteschlange leer}
        THEN BEGIN
                {wähle Kunden aus};
                {Kunden → Schalter};
                {belege Schalter};
                {setze Uhr Bedienung}
                END
        END
```

Bemerke:
Im Vergleich zu event scheduling faßt activity scanning gewisse (im Problemdenken "zusammengehörige") Ereignisse in Aktivitäten zusammen

höheres Abstraktionsniveau

→ **problemnähere Spezifikation**

process interaction: verbreiteter Ansatz Fran77
typisch: SIMULA, zB Rohl73, Pool87

"Auf dem eingeschlagenen Weg" weiter,
größere Menge von Ereignissen "zusammengehörig":

- ein (jeder!) Kunde
  - "trifft ein und stellt sich an" (Ereignis!)
  - "kommt dran" (Ereignis!)
  - "ist fertig und geht weg" (Ereignis!)

  drei Ereignisse also,
  die das "Leben" jedes Objekts "Kunde" ausmachen
  (Bitte beachten:
  Dies ist nur eine mögliche Strukturierung;
  was in dem - abstrakter werdenden - Modell
  Objekte sind, wie ihr "Leben" verläuft,
  welche Ereignisse sie absolvieren,
  entspricht - in Grenzen - frei wählbarer Vorstellung)

- der Bankangestellte
  - "beginnt eine Bedienung " (Ereignis)
  - "beendet sie" später (Ereignis)
  - eine nächste Bedienung "beginnt"
  - usw.

  eine unbegrenzte Menge von Ereignissen also,
  die das "Leben" d. Objekts "Bankangestellter" ausmachen

- eine Umwelt
  - "schickt" einen Kunden in die Bank (Ereignis)
  - später einen weiteren (Ereignis)
  - usw.

  eine unbegrenzte Menge von Ereignissen also,
  die das "Leben" des Objekts "Umwelt" ausmachen

Versuch lohnenswert,
verschiedene Objekt-"Leben"
als zentrales Modell-Strukturierungsmittel zu nutzen
(Einzelereignisse sind "Unterstrukturierung")

Zu erhoffen:
abstraktere, näher an Vorstellung liegende Modellwelt

Vorstellung:
zeitabhängiges Verhalten eines dynamischen Systems
ist getragen (generiert) durch
Menge dynamischer Objekte (Prozesse),
deren jedes (jeder) sich entsprechend
festgelegter Vorschriften (Prozeßmuster) verhält

→ **Menge/System "paralleler" Prozesse**
Diese können nicht wechselseitig unabhängig ablaufen,
erfassen "System",
haben gelegentlich "etwas miteinander zu tun",
müssen "interagieren" ("process-interaction")
Im Beispiel: "Bedientwerden"
(als Bestandteil Kunden-Prozeß)
und "Bedienen"
(als Bestandteil Angestellten-Prozeß)
ein Vorgang, in beiden Prozessen "simultan"

→ **Prozesse zu solchen Zeitpunkten zu synchronisieren**

Damit Rahmen für Realisierung process-interaction Ansatz:
Erforderlich sind Möglichkeiten,

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

An programmiersprachliche Realisierung denkend:

(i) **Prozeßmuster notieren**
eher einfach:
Höhere Programmiersprachen kennen "Prozeduren", "benannte Ablaufmuster"; Prozeßmuster "so" notieren.

(ii) **Prozesse** (gemäß bestimmten Musters) **starten**
zwar (ebenfalls recht einfach) Anweisungsart erfindbar
START_PROCESS_OF <process pattern>
oder (mit Namensgebung für den neuen Prozeß)
START <process name> OF <process pattern>
gewünschte Semantik aber
(mit herkömmlichen Sprachmitteln) nicht realisierbar:

Kann kein Analogon zu Prozeduraufruf sein, da
- Prozeduren Ablaufkontrolle (erst) nach "Durchlauf" an Aufrufstelle zurückgeben (CALL-Semantik)
- ein Prozeßmuster aber nur "ein Stück weit" exekutiert werden darf
(jenes Stück Code, das eine in Modellzeit 0 erfolgende Zustandsänderung nachvollzieht),
dann aber "anhalten" muß
(dort, wo eine in endlicher Modellzeit stattfindende Aktivität zu erfassen ist),
um einem anderen Prozeßmuster zu ermöglichen, "ein Stück" Code zu exekutieren
(eine Zustandsänderung, die zum "nächsten" berücksichtigten Modellzeitpunkt stattfindet), usf

Programmiersprache mit "solchem" Ablauf benötigt zB
- Koroutinen-Konzept (vgl SIMULA)
- Task-/Thread-Konzept (vgl ADA, Ledg80; JAVA), oä

(Abhilfen auch mit "multiple entries", "switch", C++ vgl Tysz99; aber "Gedächtnis"!)

Dies bei Einzelprozessor als Simulations-Instanz,
bei Mehrprozessorsystem wesentlich komplexer!
Wir bleiben (bis auf Weiteres) bei **Einzelprozessor**

(iii) **Prozesse "interagieren" lassen**
noch gar nicht hinreichend durchdacht,
um konkrete Wünsche an PS äußern zu können

Vertiefung in Interaktions-Problem:

- Wie mögen wir (auf Spezifikationsebene) denken?

  "Nahe am zu modellierenden Problem"
  betont Eigenständigkeit der Prozesse

  Prozesse haben im Prinzip keine Kenntnis voneinander,
  jeder "marschiert" soweit er kann
  (exekutiert Prozeßmuster bis an Stelle,
  die Fortsetzung explizit verhindert)
  "notiert" Bedingungen für (spätere) Fortsetzung

- Kompliziert? Absolut nicht!

  Dem einzelnen Kunden (des Bankschalters)
  vorzuschreibendes, auf ihn bezogenes Prozeßmuster:

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

- Leicht verständlich, enthält gleich zwei "Umstände",
  die Ablauf eines gestarteten Kundenprozesses "hemmen"
  (genauer: "beide" Arten solcher Umstände,
  es werden keine "sonstigen" auftreten):

identifizierte "Umstände" des Wartens
(a) "Warte bis Du dran bist"
(b) "Laß' Dich (für eine Weile) bedienen"

zu "Umstand" (a):
"Warte bis x" formuliert **Bedingung für Fortsetzung**
- wenn erfüllt, Fortsetzung
  (im Beispiel: Schalter könnte frei sein),
- wenn nicht, Warten bis erfüllt

"Erfindung" Notation für diese Art Warten:

WAIT_UNTIL <boolean expression>

- setze genau dann (und zu dem Zeitpunkt) fort,
  wenn Boolescher Ausdruck (b) Wert TRUE erreicht
- dh entweder unmittelbar (ohne Warten), falls b TRUE
  oder später (mit Warten), dann wenn b TRUE erreicht

Daraus zwangsläufig:

- b im Simulatorprogramm über Programmvar'n notiert,
  die gewisse Modell-Zustandsvariable nachbilden
  ("dran" ist verwendetes typisches Beispiel)

- falls b=FALSE (muß Kunde also warten),
  kann es nur über Wertveränderung von
  (in b referenzierten) Variablen TRUE werden

- dazu notwendig Ausführung von Code;
  kann aber nur per Ausführung anderen Prozeßmusters
  erfolgen - angehaltener Prozeß wartet,
  sein Code nicht weiter ausgeführt
  (im Beispiel: Bankschalter könnte "dran"
  eines Kunden auf TRUE setzen)

Eine Form der gefragten Interaktion von Prozessen erfaßt:
Setzen und Abfragen von
(im Modell:) Attributen bzw
(im Programm:) Zustandsvariablen,
die für Menge von Prozessen (gemeinsam) zugreifbar

zu "Umstand" (b):
"Laß' Dich bedienen" formuliert (im Grunde) **Zeitintervall**, das vor Fortsetzung des Prozesses zu verstreichen hat

"Im Grunde", weil wir unterschiedlich denken können
- Bedientwerden (auf seiten des Kunden) und
- Bedienen (auf seiten des Angestellten)

sind ein und dieselbe Aktivität

Kontrolle über Dauer dieser Aktivität kann (schon mental!)
- beim Kunden liegen
  (dann genauer formuliert: "bediene mich für (zB) 3 min")
- beim Bankangestellten liegen
  (dann Formulierung:
  "warte bis Bankangestellter mit Bedienung fertig",
  dh ein Typ (a) WAIT_UNTIL)

Wer immer Dauer Zeitintervall spezifiziert, muß ausdrücken, daß vor Prozeßfortsetzung (Modell-)Zeit verstreichen soll

"Erfindung" 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 Wert arithm. Ausdruck gegeben ist

Wieder "Gehirnakrobatik":
Während im realen System hochaktiv bedient wird,
passiert im Modell absolut nichts
außer dem Ablaufen der Zeit zwischen
Bedienanfang und Bedienende!

Wesentliche Gedankenarbeit geleistet, folgen Abrundungen
Offensichtlich aber:
process-interaction Ansatz
(im Gegensatz zu event-scheduling Ansatz)
nicht in beliebigen PS notierbar,
spezielle Fähigkeiten gefordert,
wenn nicht sogar spezifische Simulationssprachen
mit process-interaction Sprachelementen

Beispiel: **Bankschalter in process-interaction Sicht**

Dafür zunächst Notationsvereinbarungen:

- Prozeßmuster beinhaltet
  - Vereinbarungsteil, der
    prozeßeigene Zustandsobjekte/Datenobjekte auflistet;
    jeder nach diesem Muster geSTARTete Prozeß
    besitze je ein Exemplar jedes dieser Objekte;
    Zugriff auf ein Objekt eines Prozesses mittels
    <process name>.<object name>
  - einen Codeteil, der Verhalten jedes nach
    diesem Muster geSTARTeten Prozesses festlegt;
    enthält insbesondere Anweisungen der Arten
    WAIT_UNTIL, PAUSE_FOR, START;
    im Code kann laufender Prozeß auf "sich selbst"
    und "seine eigenen" Datenobjekte zugreifen über
    Prozeß"namen" CURRENT ("THIS", "MY")

- Simulationsmodell enthält speziellen
  "Simulationshauptprozeß",
  mit dessen Ausführung Simulation beginnt, und der
  - Initialisierungen vornimmt
  - Dauer Simulation festlegt

- Es gibt speziellen Datentyp

  FIFOQUEUE OF <entry type>

  FIFOQUEUE-Objekt enthält
  FCFS-geordnete Objekte des Typs <entry type>
  als <entry type> auch Prozesse
  (eines bestimmten Prozeßmusters) zugelassen

  FIFOQUEUE-Objekt exportiert
  Funktionen / Prozeduren / Operatoren / Methoden
  "enqueue", "delete", "first", "empty"
  Zugriff in üblicher dot-Notation

  Syntax- und Semantik-Klärung an Programmskizze:

```
queue: FIFOQUEUE OF INTEGER;
            {deklariert + initialisiert queue als (leere)
             FCFS-Schlange für INTEGER-Einträge}
b: BOOLEAN; i: INTEGER;       {Hilfsgrößen}
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 3.2.3: Vergleiche Bsp. 2.1,2.4;
Bankschalter mit M/M/1-FCFS-Spezifikationen,

- exponentiell verteilte Ankunftsabstände (Parameter λ)
- exponentiell verteilte Bedienzeiten (Parameter μ)

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, vgl *)};
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;    {vgl *)}
      CURRENT.belegt:= TRUE;
      WAIT_UNTIL (NOT CURRENT.belegt)
                        {bis Bedienung beendet, vgl *)};
    UNTIL CURRENT.ws.empty
  ENDLOOP
  ```

Simulationshauptprozeß:

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

Bemerkungen

*) Unschön, durchbricht Vorstellung
"wechselseitige Unkenntnis der Prozesse"

Anders? Wenn (wie angenommen)
WAIT_UNTIL einzig über Objektwerten definiert
und wenn "bankschalter" nicht auf
"beendet"-Setzen der "kunden" vertrauen soll,
könnte er Bedienende aus
"Weggegangensein Bedienter" erkennen.
Also bankschalter merke, wem "dran" gesagt,
und WAIT_UNTIL {WS-Erster ungleich Gemerkter}

**) Üblicherweise Modellzeitvariable t implizit
zu Anfang auf "0" gesetzt.
Und: Für Festsetzung der zu überstreichenden
Modellzeit gibt es Reihe anderer
"Abbruchkriterien" als nur Modellzeit selbst

Ende Beispiel 3.2.3

Erstes Fazit process-interaction Ansatz:

- Denken in parallelen Prozessen als Modellwelt
problemnah, angenehm und kompakter als
Denken in Einzelereignissen (event-scheduling)

- Dies jedenfalls so lange, wie
Interaktion der Prozesse keine wesentliche Rolle spielt

- Bei Organisation Interaktion tauchen Schwierigkeiten auf
(vgl Bsp. 3.2.3);
diese noch nicht völlig geklärt

Dazu Interaktionsdetail aus Bsp. 3.2.3 ganz präzise:

- "schalter", ein Prozeß des Prozeßmusters "bankschalter", wartet gelegentlich bis "NOT CURRENT.belegt"; auf Denkebene ist dies Warten auf Beendigung laufenden Bedienungsintervalls, das vom Bedienten, einem Prozeß des Musters "kunde", mittels "schalter.belegt:=FALSE" signalisiert

- Wann genau läuft "schalter" weiter?
  Nähmen wir die parallelen Prozesse beim Wort,

  - genau dann, wenn "belegt" auf "FALSE" springt

  - und dann wären zwei Prozesse gleichzeitig aktiv: "kunde"-Prozeß wendete sich seinem "delete" zu, "schalter" seiner "ws.empty"-Prüfung

  - und dann ginge die Sache (potentiell) schief:

    ist "schalter" um ein Quentchen schneller als parallel aktiver "kunde"-Prozeß, ("schalter" führt "ws.empty"-Prüfung aus, bevor "kunde" sein "delete" ausführt),

    dann wendet sich "schalter" fälschlich erneut diesem Kunden zu (jenem, der "eigentlich" schon weg ist)

- Hier (und in vielen ähnlichen Fällen) dringend weitere (Semantik-)Klärung erforderlich

- Semantik-Klärung muß (notwendigerweise) die Simulations-ausführende Instanz genauer betrachten

Festlegung war: **Einzelprozessor**
(bei Multiprozessor völlig anders zu argumentieren)

Einzelprozessor

- kann nicht mehrere Programme realiter parallel ausführen

- muß Parallelität derart "vorgaukeln", daß
(in Koroutinen-Manier) aktive Programme/Prozesse
jeweils exklusiv, wechselseitig zeitversetzt
"ein Stückchen" exekutiert werden

- präzise Definition dieser "Stückchen" bringt
gesuchte Klärung;
bei dieser Klärung kommt Zeitachse
(des event-scheduling)
"auf niedrigerer Ebene" wieder ins Spiel

Abbildung 3.2.4 - Entwurf des technischen Ablaufs
einer Simulation der process-interaction Denkwelt:

In geeigneter Programmiersprache selbst zu implementieren,
oder von einer speziellen Simulationssprache implizit
(Laufzeit-System) anzubieten,

- ist wieder zentrale Simulatorschleife
(analog zu, aber etwas anders als, Abb. 2.3),

- die auf einer Ereignisliste arbeitet
(analog zu, aber etwas anders als, Abb. 2.3)

(a) **Ereignisliste** mit Einträgen (t_i,tp_i),
gemäß t_i monoton nicht-fallend geordnet, wo
t_i: Zeitpunkte (der "Modellzeit")
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: t_X,tp_X};
t:=t_X {"momentane" Zeit "springt"};
{aktiviere Prozeß gemäß tp_X, vgl *)};
{entferne Ereignisliste-Eintrag};
WHILE {UNTIL-Bedingungen TRUE, vgl *)}
DO {aktiviere einen diesbezüglichen Prozeß, vgl *)}
ENDLOOP

Abbildung 3.2.4: Organisationsteil für process-interaction Simulation

*) **Diskussion**:
Immer wieder bestimmter Prozeß zu "aktivieren",
dh Exekution Prozeßmuster dort fortzusetzen,
"wo Prozeß gerade steht".

Kann stehen:

- ganz am Anfang; dann wurde er aufgrund einer START-Anweisung vorgemerkt, die implizit (à la PLAN) zu Eintrag in Ereignisliste umgesetzt wurde (auch "Erstereignis"-Setzen funktioniert so; alle anderen STARTs werden zur selben Modellzeit, aber "nach" gerade aktuellem Eintrag, vorgemerkt);
- an einer PAUSE_FOR Anweisung; dann wurde er für bekannten Zeitpunkt (à la PLAN) in Ereignisliste vorgemerkt;
- an einer WAIT_UNTIL Anweisung; dann ist er nicht in Ereignisliste vorgemerkt; vielmehr wird seine Bedingung durch Aktivitäten eines in der Ereignisliste vorgemerkten, aktivierten Prozesses erfüllt werden (hoffentlich!) (bzw durch Aktivitäten eines durch Aktivitäten eines Vorgemerkten aktivierten Vorgängers,...)

Weitere Details hier nicht diskutiert
(zB was, wenn mehrere WAIT_UNTIL-Wartende fortsetzen könnten, was ist mit PAUSE_FOR 0, WAIT_UNTIL TRUE,...).
Aber festzuhalten:
Durch Festlegung, wo Prozeß "stehen" kann,
auch "Stückchen" exklusiver Abarbeitung
von Prozeßmustern geklärt:

Prozeß läuft bis zu einem **WAIT_UNTIL**,
bis zu einem **PAUSE_FOR** oder eben
bis zu seinem **Ende** (gemäß Prozeßmuster)

Implementierung von WAIT_UNTIL Anweisungen
(bei Simulationssprache Compiler bzw Laufzeitsystem)
unmittelbar Bedenken bzgl **Laufzeiteffizienz**!

- Boole'sches b eines WAIT_UNTIL b
  in keiner Weise eingeschränkt,
  kann sich auf beliebige Variablen beziehen
- bedeutet, daß **jede** Wertzuweisung an Variable
  potentiell b **jedes** WAIT_UNTIL-Wartenden betrifft
- impliziert, daß immer wenn ein Prozeß anhält
  sämtliche b's aller WAIT_UNTIL-Wartenden bezüglich
  Fortsetzung zu überprüfen sind (+ "bei mehreren" ??)

Dies kann aufwendig werden!

Auswege?

Hier **zwei** "in entgegengesetzte Richtungen":

- **Näher hin zur Implementierung**
  (→ weiter weg vom mentalen Modell):
  - streiche ineffizientes WAIT_UNTIL
  - ersetze es durch PASSIVATE-Anweisung,
    die ausführenden Prozeß (unbedingt!) anhält.

  Damit kann Prozeß
  - noch feststellen, ob fortzusetzen
    (Überprüfung b des gestrichenen WAIT_UNTIL),
  - kann sich auch selbst anhalten
    (IF NOT b THEN PASSIVATE),
  - kann aber nicht mehr ausdrücken,
    wann er wieder "aufzuwecken" ist.

  Notgedrungen:
  - (Wieder-)Aktivierung durch anderen Prozeß
  - ACTIVATE <process name> Anweisung erforderlich
  - Prozesse müssen einander "kennen" und
    müssen entsprechende ACTIVATEs ausführen.

  Simulator-Code damit länger / komplexer, aber effizienter
  So arbeitet SIMULA (zB Pool87, Fran77)

- **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
  (muß auf zu simulierende Systeme "abgestimmt" sein)

  Schritt zu sog. **Szenario-Sprachen**,
  nicht mehr zur (hier: ereignisorientierten)
  Simulation generell,
  sondern zur Simulation bestimmter Problemklassen

So arbeitet GPSS (Gree72,Schr74,Gord75)
- entwickelt zur Spezifikation von "Warteschlangensystem"-Problemen
- keine sehr moderne Sprache

In GPSS notiert sich "kunde"-Prozeßmuster (als Beispiel) sehr einfach mittels
- einer SEIZE-Anweisung,
  einem WAIT_UNTIL <ganz besonderes b>,
  auf Freisein nämlich einer Bedien-"facility",
  wie für Probleme bestimmter Problemklasse ("Szenario") charakteristisch

SEIZE schalter	"facility" schalter wird belegt, falls diese frei ist - sonst wird gewartet
ADVANCE s	Jetzt hat Kunde Schalter, behält ihn für Dauer s
RELEASE schalter	Jetzt wird Schalter freigegeben (implizit ein in SEIZE Wartender erlöst)

War nur "Ausflug", s. etwa Kreu86

3.3 Generierung von Realisierungen von ZVn

Bei Modellspezifikation gelegentlich

Beschreibung von Größen ("stochastische" Simulatoren)
durch **ZV mit bestimmter Verteilung**
diese etwa charakterisiert durch Verteilungsfunktion
FY(y) := P[Y ≤ y]
(so Bsp. 2.4: Bedienzeit an Schalter negativ exponentiell)

Folglich, im Verlauf der Simulation, immer wieder benötigt
Realisierungen dieser ZV
(so Bsp. 2.4: DRAW(negexp(μ)))

Wunsch:
"Ziehen" von Realisierungen y aus der Verteilung von Y

Gesucht Software-Realisierung, dh
Stück Programm-Code, das

- bei i-ter Exekution Wert y_i liefert,
  der als Realisierung von Y aufgefaßt werden kann

- bei wiederholter Exekution Folge y_1, y_2, ... liefert,
  welche als unabhängige, aus der Verteilung von Y
  gezogene Stichprobe aufgefaßt werden kann, dh

  Folge enthält Werte, die
  - so "häufig" in Intervalle des Wertebereichs fallen,
    wie FY dies vorschreibt
  - "voneinander unabhängig" sind

"Zufall" aus "deterministischem Code" ??

Generierung von "**Pseudo-Zufallszahlen**" (**ZZ**)
mittels "Pseudo-Zufallszahlen-Generatoren"

- liefern Folgen, die "so aussehen als ob … "
- ermitteln ZZ "effizient"

Realisierung (üblicherweise) in zwei Schritten:

(i.1) Codierung Mechanismus', der
über $\{ j \in N_0 \mid 0 \leq j \leq m-1 \}$ gleichverteilte, diskrete ZV Z
realisiert

(i.2) Transformation von Z derart, daß
über [0,1) gleichverteilte, kontinuierliche ZV U
realisiert

(ii) Transformation von U derart, daß
(im Anwendungsfall spezifizierte)
ZV Y (mit gegebener Verteilungsfunktion FY)
realisiert

Zu (i.1): Diskret gleichverteilte ZZ

Zwei Wege beschritten:

- Basis (große) Tafel von (vorbereiteten) ZZ (zB RAND55)

  Tafel aus Beobachtung "wirklich" zufälliger Phänomene
  (zB Beobachtung radioaktiven Zerfalls,
  Verwendung # Spaltungen in festen Intervallen)

  mittels bestimmten Algorithmus ZZ nach ZZ
  aus Tafel entnommen

- Generierung durch
  Algorithmus, der ZZ nach ZZ errechnet

In beiden Fällen:
- **Reproduzierbare** (eben nicht zufällige) Folgen von ZZ,
- aber **hinreichend überraschend**, um zufällig zu wirken:
  "Pseudo"-Zufallszahlen!

Übrigens: Reproduzierbarkeit geradezu erwünscht!
ua um Programmtesten zu ermöglichen
Und: "hinreichend überraschend", dh "zufällig",
muß gezeigt bzw (statistisch) überprüft werden!

Kongruenz - Generatoren

- Lineare Kongruenzgeneratoren
  (weitverbreitet, Urquelle: Lehm51)

  ZZ-Erzeugung gemäß:

  z_0 **N** $(0 \le z_0 < m)$ gesetzt: als "seed" bezeichnet

  $z_{i+1} \equiv az_i + c \pmod m$

  wo: m **N**, m>1, a **N**, c $\mathbf{N_0}$

  ($0 \le z_i < m; i \ge 0$)

  Beispiel: a = 3, c = 0, m = 10

i	0	1	2	3	4
z_i	1	3	9	7	1

  ↑ seed

  zeigt auch typische (potentielle) Schwachpunkte:

  - "Periode" unvermeidbar

  - Nicht alle (prinzipiell möglichen) Zahlen
    0…m-1 erreicht

"maximale Periode" (= m) erzielbar durch geeignete Wahl von a und c (abhängig von m) ?

Bedingungen für maximale Periodenlänge m:

Sei kanonische Primzahlzerlegung von m:

$p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_r^{\alpha_r}$ (= m)

wo $p_i \in \mathbf{P}$, $p_i \neq p_j$ für $i \neq j$, $\alpha_i \in \mathbf{N}$, $1 \leq i \leq r$, $r \in \mathbf{N}$

Linearer Kongruenzgenerator besitzt Periodenlänge m genau dann wenn

- c relativ prim zu m dh ggT(c,m) = 1
- $a \equiv 1 \pmod{p_i}$, $1 \leq i \leq r$ dh $a \in \{ p_i k+1 \mid k \in \mathbf{N_0} \}$
- 4 teilt m $a \equiv 1 \pmod 4$

zB UNIX (Jain91): $m = 2^{32}$,
$a = 1103515245$,
$c = 12345$

$c \geq 1$ Gemischter Kongruenzgenerator
$c = 0$ Multiplikativer Kongruenzgenerator
(wäre erwünscht, da "effizienter")

Weitere praktische (Effizienz-)Erwägungen:
Modul- (m-) Wahl an (maschinenabhängige) Arithmetik + Zahlendarstellung anpassen:

$m = 2^b$ ermöglicht bei Binärmaschine einfache mod-Operation per Abschneiden bzw Schieben/Abschneiden

$m = 2^{p-1}$ nutzt bei Binärmaschine mit Wortbreite p gesamte Darstellungsmöglichkeit ≥ 0

Für multiplikative Kongruenzgeneratoren gilt

m **P**: Maximal erreichbare Periodenlänge: m-1
wird erreicht, wenn
- a Primitivwurzel mod m, dh
  ggT(a,m) = 1 und m-1 = min{ k **N** | $a^k \equiv 1 (\text{mod } m)$ }
- und seed **N**

zB (Jain91): m = 2^{31} -1 = 2147483647,
a = 7^5 = 16807 ("viel verwendet"),
a = 48271 oder 69621 ("beste")

$m=2^b$: Aus Effizienzgründen erwünscht, aber
maximal erreichbare Periodenlänge nur: 2^{b-2}
wird erreicht, wenn
- $a \equiv 3 (\text{mod } 8)$ oder $a \equiv 5 (\text{mod } 8)$
- und seed { 2k+1| k $\mathbf{N_0}$ }

zB (Jain91): m = 2^{31}, $a = 65539 \equiv 3 (\text{mod } 8)$
("RANDU", "nicht gut")
m = 2^{35}, $a = 5^{13} \equiv 5 (\text{mod } 8)$
("SIMULA/UNIVAC", "nicht gut")

- Mehrfach-rekursive Kongruenzgeneratoren
  (Fibonacci, Extended Fibonacci, …)

- Matrixgeneratoren …

Weitere Generator-Typen
Tausworthe-, gemischte, …
(vgl zB Jain91, Sieg91, BaCN96, Tysz99)

Erfinden Sie keine neuen !

Zu (i.2): Kontinuierlich [0,1)-gleichverteilte ZZ

(i.1) lieferte Generator für
n-periodische Folge ganzer Zahlen z_i
über Zahlenmenge [0:m-1]

Bei voller Periode n = m (bzw n = m-1)
alle i [0:n-1] (bzw i [1:n-1])
je Periode genau einmal "getroffen",

falls Treffen "regellos":

Generator für "Pseudo"
[0:m-1]- (bzw [1:m-1]-) gleichverteilte ZV Z

Transformation

U = Z / m

liefert entsprechend "Pseudo"-
[0,1)-gleichverteilte kontinuierliche ZV U
(bzw (0,1]-, (0,1)-, [0,1]- gleichverteiltes U)

Streng genommen: diskrete u_i-Werte z_i / m, z_i **N**

Aber: Werte liegen idealerweise sehr "dicht",

bei Beachtung der
Maschinen-Darstellung von Zahlen
sogar optimal dicht: "alle darstellbaren" Zahlen
(entspricht üblicher Approximation:
FK bzw GK bzw "real" für "reell")

Forderungen an die $u_i = z_i / n$ (bzw an Z-Generator):

(i) "gleichverteilt":

global (unendliche Folgen) durch "volle Periode"
bei hinreichend ("Zahlendarstellung") großem m
offensichtlich erfüllt;

lokal ("Regellosigkeit") durch statistische Tests zu überprüfen
(zB Chi-Quadrat-, Kolmogoroff-Smirnoff-Tests, vgl 5.3)

(ii) "unabhängig":

Pseudo-Unabhängigkeit zu klären
durch Untersuchung (und möglichst Ausschluß)
sequentieller "Regelmäßigkeiten" der Zahlenfolge

Untersuchbar:

- Regelmäßigkeiten des Z-Generators
  (Bei Kongruenzgeneratoren liegen
  aufeinanderfolgende Zahlen-Paare, -Tripel, … k-Tupel
  auf beschränkter Anzahl 2-, 3-, k-dimensionaler
  Hyperebenen:
  Lage, Zahl, Abstand der Ebenen beurteilbar)

- mit statistischen Verfahren aufdeckbare
  sequentielle Abhängigkeiten
  (Autokorrelationsfunktion, vgl Fish73/78, LaKe82/86,
  Unabhängigkeitstests,
  zB Runs-Test,vgl 6.2, … ,
  vgl Mitr82, Jain91, BaCN96, Tysz99)

Erfinden Sie keine neuen Generatoren !!

Zu (ii): ZZ gemäß geforderter Verteilung von Y, spezifiziert durch Verteilungsfunktion FY(y)

Ausgangspunkt (aus (i)):
[0,1)-gleichverteilte kontinuierliche ZV U, dh

$$FU(u) = \begin{cases} 0 & ; \quad u < 0 \\ u & ; \quad 0 \le u < 1 \\ 1 & ; \quad u \ge 1 \end{cases}$$

Unterschiedliche Transformationsmethoden
(y-Werte aus u-Werten) existieren (hier: nur Auswahl)

INVERSION (Umkehrmethode)

Fallunterscheidung:

- Y kontinuierlich verteilt
- Y diskret verteilt

- Sei Y **kontinuierliche** ZV mit Bild(Y) = I
  (Wertebereich kontinuierliches Intervall I)
  und FY streng monoton steigend auf I

- Gesucht:
  Transformation Y = t(U), so daß FY(y) "wie gefordert"

- Weg:

  $FY(y) = P[Y \le y]$, y I
  $= P[t(U) \le y]$, y I

  Annahme (später zu prüfen):
  t streng monoton steigend auf Def(t) := Bild(U)
  t^{-1} streng monoton steigend auf I

  $FY(y) = P[U \le t^{-1}(y)]$, y I
  $= FU(t^{-1}(y))$, y I
  $= t^{-1}(y)$, y I
  $t(FY(y)) = y$, y I

  $t = FY^{-1}$ auf Bild(U) (Annahme bestätigt)

  $Y = FY^{-1}(U)$

  Die gesuchte Transformation ist gegeben durch die Umkehrfunktion der Verteilungsfunktion FY auf I

Beispiel: (Negativ-) Exponential-Verteilung mit Parameter λ

$(0 < \lambda < \infty)$

$FY(y) = 1 - e^{-\lambda y}$, $y \geq 0$

(FY auf $\mathbf{R}^+$ streng monoton steigend)

$\rightarrow$ $e^{-\lambda y} = 1 - FY(y)$, $y \geq 0$

$-\lambda y = \ln (1 - FY(y))$, $y \geq 0$

$y = - (1 / \lambda) \ln (1 - FY(y))$, $y \geq 0$

($y \geq 0$: $FY(y) = u \Leftrightarrow y = FY^{-1}(u)$, $FY(y)$ [0,1))

$Y = - (1 / \lambda) \ln (1 - U)$

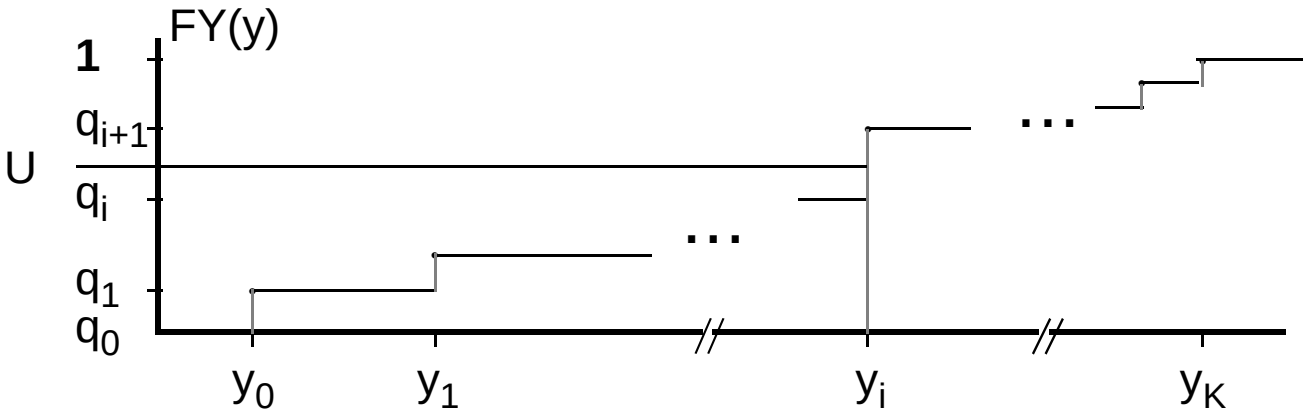
bzw: $Y = - (1 / \lambda) \ln (U)$ nachdenken!

Sei Y **diskrete** ZV mit Bild(Y) = { y_i **N₀** | i I },

wo I Indexmenge {0,1,...,k} **N₀** und $y_{i-1} < y_i$ (i > 0),

und geforderter Verteilung $P[Y=y_i]$, i I

hier: explizite Umkehrfkt. existiert nicht (FY ist Treppenfkt.)!
aber: Algorithmus angebbar, der "Umkehr"-Idee nutzt



Partitionierung von [0,1)-Intervall in | I | = k+1
halboffene Intervalle $[q_i,q_{i+1})$, so daß $q_{i+1} - q_i = P[Y=y_i]$:

$$
\begin{aligned}
P[Y=y_i] &= FY(y_i) - FY y_{i-1}) && , i \quad I \\
&= q_{i+1} - q_i && , i \quad I \\
&= FU(q_{i+1}) - FU(q_i) && , i \quad I \\
&= P[q_i \le U < q_{i+1}] && , i \quad I
\end{aligned}
$$

Somit

für eine "gezogene" ZZ u∈[0,1)

bestimme jenes i ∈ I, für welches gilt

$FY(y_{i-1}) \le u < FY(y_i)$

(bzw setze i=0 für $u<FY(y_0)$)

y_i ist die (aus u transformierte) Realisierung von Y

Beispiel:

Geometrische Verteilung mit Parameter α $(0 < \alpha < 1)$

$FY(i) = 1 - \alpha^i$, $i \in \mathbf{N}$ $(y_i := i + 1)$

Suche i, so daß für $u \in [0,1)$:

$FY(i) \le u < FY(i+1)$

hier sogar: geschlossene Formel!

$$Y = 1 + \left\lfloor \frac{\ln(1-U)}{\ln \alpha} \right\rfloor \quad \text{bzw.} \quad Y = 1 + \left\lfloor \frac{\ln U}{\ln \alpha} \right\rfloor$$

Bemerkung:

Umkehrmethode basiert
(im kontinuierlichen und im diskreten Fall)
auf Kenntnis der Verteilungsfunktion
(aber zB Normalverteilung:
kein Ausdruck in geschlossener Form bekannt)

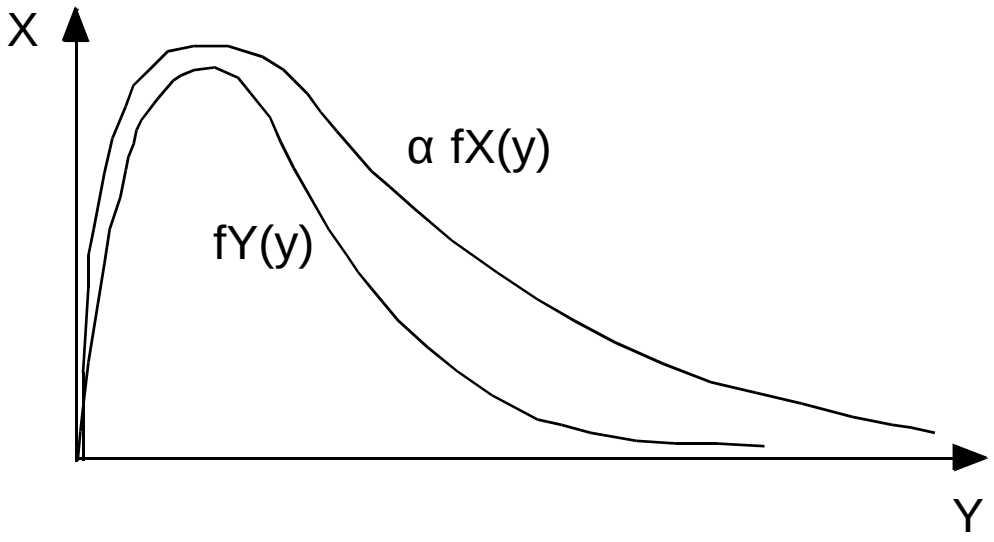
VERWERFUNG

anwendbar für diskrete und kontinuierliche ZV

hier: Y kontinuierlich
Kenntnis der Dichtefunktion fY

Existiere andere Dichtefunktion fX(y) und $\alpha > 0$,

so daß für alle y: $fY(y) \le \alpha\, fX(y)$
(fX ist "majorisierende Dichte")



Stichprobe y aus fX
(Existenz Generator vorausgesetzt)
kann mit Wahrscheinlichkeit $fY(y)\ (\alpha\ fX(y))^{-1}$
als Stichprobe aus fY akzeptiert werden

Prozedur:

REPEAT
"ziehe" y entsprechend Dichte fX;
"ziehe" x aus [0,α fX(y)]-Gleichverteilung
UNTIL $x \leq fY(y)$;
{akzeptiere y}

Effizienz des Verfahrens:

- abhängig von α :
  α möglichst nahe 1, da (E[Anz. Durchläufe] = α)

- abhängig von Effizienz der Generierung für fX

- Steigerung möglich, falls ... (vgl BrFS87)

KOMPOSITION

Realisierungen von Y werden zusammengesetzt
aus Realisierungen (mehrerer) anderer Zufallsvariablen.

Beispiele:
Hyperexponential-Verteilung, Erlang-Verteilung

Effizienz der Verfahren:
Anzahl zusammensetzender Komponenten
bestimmt Anzahl der Ziehungen aus
Basis-Gleichverteilung (ZV Z bzw U)

Spezielle Verfahren für Spezielle Verteilungen
vgl Literatur

Bemerkungen

- Bereitstellung ZZ-Generatoren:

  Basis- (Z-) Generator ermittelt z_i aus
  (meist genau einem) Vorgänger - z_{i-1},
  neues z_i ist Vorgänger des nächsten, z_{i+1}

  Bei Implementierung Generator als
  Funktion / Prozedur / Unterprogramm (zB namens "gen")
  (ohne "Gedächtnis"!)
  ist Vorgänger-z_i generatorextern "weiterzutragen"

  Bereitstellung von ZZ-Generatoren daher
  - als mindestens einparametrige Funktion
    (weitere Parameter zur Charakterisierung
    der Verteilung und Verteilungs-Parameter)
  - und Forderung an Benutzer, diesen "speziellen"
    Parameter aktuell mit eigens zu vereinbarender
    globaler Variabler (zB "s") zu besetzen

Aufruf (mit Zuweisung) also zB

a := gen(s)

wobei s Ein- und Ausgabeparameter von gen ("Tragen"), ohne direkte Interpretation auf Benutzerebene

Initialwert von s wird vom Benutzer (einmal) als "seed" gesetzt; das ermöglicht
- exakte Reproduktion eines "Laufs" eines stochastischen Simulators (erwünscht für Testzwecke)
- gewollte Verschiedenheit von "Replikationen"eines Laufs (erwünscht zur Imitation "zufälliges" System)

- Entwicklung ZZ-Generatoren:

  Im statistischen Sinne "brauchbare", "gute" Generatoren müssen vielfältige Tests erfolgreich überstehen:

  Unterstützung seitens Simulations-Sprachen und -Tools in Form der Bereitstellung ausgetesteter (möglichst effizient arbeitender) ZZ-Generatoren dringend erforderlich

Erfinden Sie keine neuen Generatoren !!!

Und: Verwenden Sie keine (zwar durchaus verbreiteten, aber) anerkannt "schlechten" wie zB die "mid-square technique", Urquelle: vNeu48, "mid-product technique", Urquelle: Fors48

LEERSEITE

3.4 Einige Aspekte der Einbettung des event-scheduling Ansatzes in allgemeine HLL

- Verfügbar sei:
  Allgemeine HLL ohne spezielle Simulationskonzepte

- Gesucht ist:
  Implementierungsschema für discrete-event Simulatoren;
  nur event-scheduling Ansatz "einfach" einbringbar

- Zu bedenken (vgl "Konzeptuelle Bedürfnisse", 3.1):

  (i) höhere Datenstrukturen, insbesondere Listen
  (ii) damit auch Zeitmanipulation
  über spezielle "Ereignisliste"
  (iii) Realisierung von Zufallsvariablen (vgl 3.3)
  (iv) Statistische Auswertung (später)

- Verschiedene Fälle zu betrachten (HLL-Fähigkeiten)

  (α) ohne pointer- (bzw Referenz-) Konzept,
  ohne dynamische Speicherzuweisung

  (α1) ohne record-Konzept
  (α2) mit record-Konzept

  (β) mit pointer- und record- Konzept

  (β1) ohne dynamische Speicherfreigabe
  (β2) mit dynamischer Speicherfreigabe

(Im folgenden alles à la PASCAL geschrieben,
obwohl wir über diverse HLLs reden)

Zu (i): Listen

Datentyp: Folge von Einträgen
+ Operatoren auf Folge

Deklaration / Vereinbarung im Prinzip:
<list_id>: LIST OF <entry_type>

Eintragstyp?

- Von den Problemen her "sitzen" in Listen Objekte, die durch ihre Attribute genauer charakterisiert sind
- Attributstypen sicher heterogen,
  - ideal also RECORD als <entry_type>
  - bei (α1) Ausweg:
    Abbildung aller Attributstypen auf homogenen einfachen Typ (ia INTEGER, REAL),
    Implementierung <entry_type> in ARRAY

zB

TYPE a_eintrag (es wird ja verschiedene geben)

(α1) = ARRAY[1..2] OF INTEGER

ab (α2) = RECORD
attr1: <type_1>
attr2: <type_2>
END

und Folge von Einträgen

- erwünschte Deklaration:
  a_liste: LIST OF a_eintrag
  aber: "LIST OF" nicht vorhanden, zu implementieren
  damit Listen-Vereinbarung implementierungsabhängig

- Implementierungen

  (α) in ARRAYs
  impliziert Beschränkung maximaler Listenlänge "maxlength"
  erfordert eigene "Speicherverwaltung",
  da Menge enthaltener Einträge dynamisch
  ("Objekte kommen und gehen")

  (β) in verzeigerten Strukturen,
  wobei (β1) erneut eigene Speicherverwaltung fordert
  ("Objekte gehen auch wieder")

Operatoren ?

- naheliegend: Prozeduren, Funktionen
  bei Sprachen ohne ADT- OO- Konzept (leider) ohne
  "Zusammengehörigkeit von Instanzen+Operationen"

- welche?
  initial sicherlich:
  - PROCEDURE init(<list_id>,<Zusatzinformationen>)
    {erzeugt leere Liste}
  - PROCEDURE add(<list_id>,<entry>,<noch 'was?>)
    {fügt Eintrag zu}
  - PROCEDURE delete(<list_id>,<was?>)
    {löscht Eintrag}

Implementierungsentscheidungen bzgl <entry>:

- für (β) (mit pointer-Konzept)
  kann <entry> Verweis auf einzutragendes Objekt sein,
  daher unabhängig von Listen-Implementierung
  (namentlich) bekannt,
  Listen können auf diesen Verweisen arbeiten.
  Hat nicht nur Vorteile:
  Listen müssen darauf vertrauen, daß
  Einträge nicht "von außerhalb" manipuliert werden

- für (α) (ohne pointer-Konzept)
  kann <entry> nur Eintrag selbst oder dessen Name sein;
  Eintrag muß in die Liste kopiert werden.

  Entscheidung:
  <entry> ist Eintragsname, Eintrag in Liste kopiert.
  add (uä) hat Resultatparameter <position>
  (implementierungsabhängigen Typs),
  der (späteres) delete (uä) erlaubt.

  Hat nicht nur Nachteile:
  "Abschotten" (information hiding) ermöglicht,
  Einschränkung externen Eingriffs in Liste erreicht

  somit

  PROCEDURE add(<list_id>,<buffer>,<position>)

  PROCEDURE delete(<list_id>,<position>)

Weitere "Operatoren" wünschenswert / notwendig?

- Um "Hineingreifen" in Liste zu vermeiden:
  - PROCEDURE examine(<list_id>,<position>,<buffer>)
    {kopiert aus Liste}
  - PROCEDURE change(<list_id>,<position>,<buffer>)
    {überschreibt Eintrag in Liste}

- Weiterhin sicher hilfreich:
  - FUNCTION empty(<list_id>): BOOLEAN
    {prüft, ob Liste leer}
  - FUNCTION first(<list_id>): <position_type>
    {liefert Verweis auf ersten Eintrag}
  - FUNCTION last(<list_id>): <position_type>
    {liefert Verweis auf letzten Eintrag}
  - FUNCTION next(<list_id>,<position>): <position_type>
    {liefert Verweis auf "folgenden" Eintrag}

Zu (ii):Ereignisliste

Spezielle, "sortierte" Listen benötigt,
- sowohl hier (Ereignisliste)
- als auch allgemein (Prioritäten, …)

zB aufsteigend geordnet bzgl "Schlüssels"
Typ Schlüssel? Um allgemein zu sein: REAL, INTEGER

Eintragen mit
- PROCEDURE enqueue
  (<list_id>,<key>,<buffer>,<position>)
  {fügt gemäß aufsteigenden "keys" ein}
- zusätzliche Möglichkeiten u.a.
  add → append (hinten), prepend (vorne)

Weitere ("interne") Implementierungsüberlegungen

"Verkettungs"-Information zu halten:

Minimum:
einfache (Vorwärts- oder Rückwärts-) Verkettung
\+ Verweis auf (erstes oder letztes) "Einstiegselement"

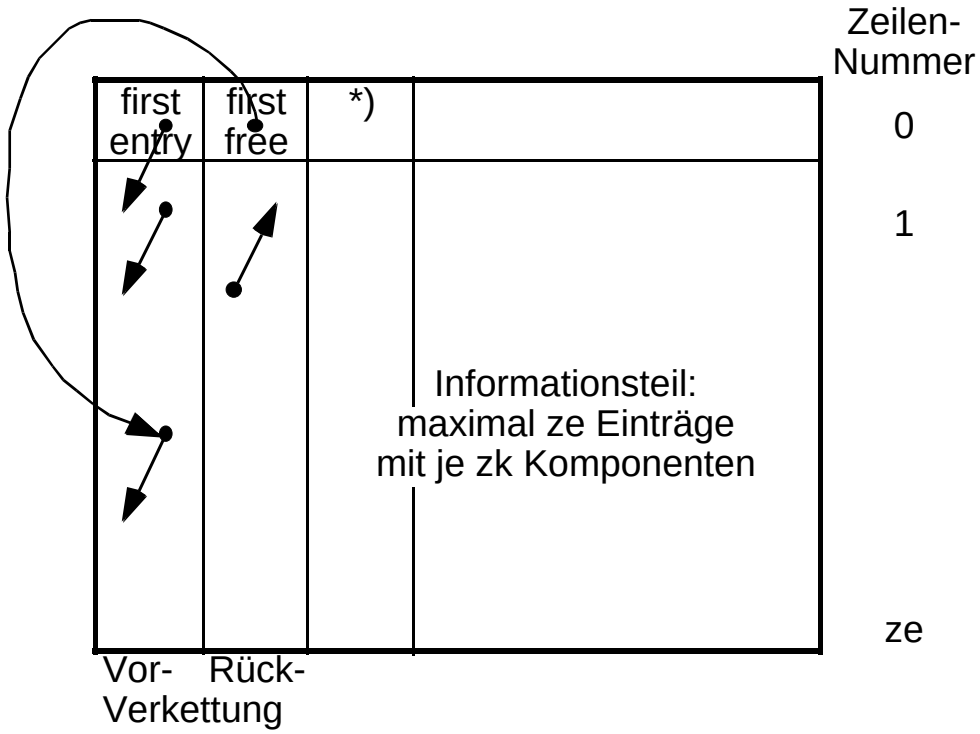
Falls kombinierte Verwendung
first / prepend und last / append (add)
sollte erster und letzter Eintrag leicht zugreifbar sein

Falls "delete" häufige Operation
(bei einfacher Verkettung sequentieller Durchlauf)
kann doppelte Verkettung lohnen
(dann doppelt+zyklisch naheliegend,
so daß erster und letzter Eintrag leicht zugreifbar)

Implementierungsvorschlag für Fall (α1):

zweidimensionaler ARRAY
- Folge für Einträge: Typ homogen
- INTEGER? ungünstig für Informationsteil (zB Ereignisliste: ganzzahlige Zeit)
- REAL? ungünstig für Verweise

Entscheidung: INTEGER



gewählte Variante:
- doppelte Verkettung für "besetzte" Zeilen
- einfache (Vorwärts-)Verkettung für "freie" Zeilen

*) könnte Information tragen: "sorted" (1), "unsorted" (0) list key?
- zusätzliche Spalte oder
- Teil Eintrag (zB Verabredung: erste Komponente)

Je Liste damit Deklaration:

VAR <list_id>: ARRAY[0..ze,-1..zk] OF INTEGER

(PASCAL-Schwäche:
ze, zk-Werte müssen bekannt sein)

<position_type> ist INTEGER

Prozedur-Beispiel:

PROCEDURE init
(VAR <list_id>: ARRAY; ze,zk: INTEGER)

(reicht nicht für PASCAL:
ARRAY[,] OF INTEGER

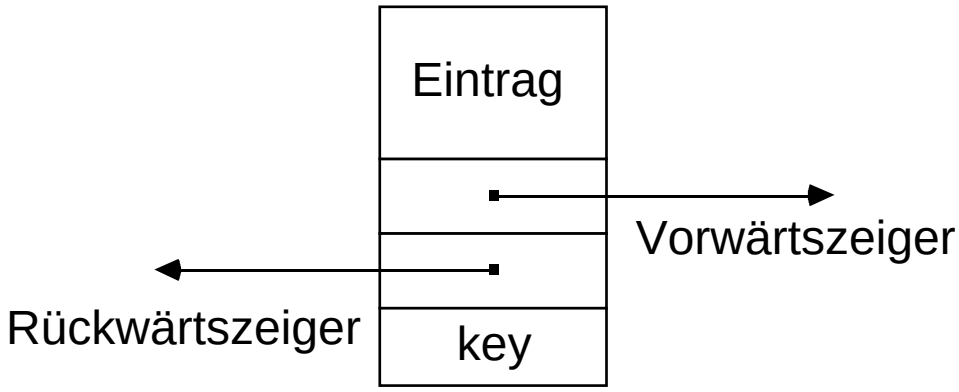
dann aber für jede Dimensionsvariante
neues "init" (!)
oder nur **ein** (ze,zk)-Paar
für alle verwendeten Listen (!))

(weitere - im Mittel platzsparende - Alternative:

ein array für **alle** Listen)

Implementierungsvorschlag für Fall (β2):
(anderes Extrem)

- (Listen-)"element" (intern, doppelte Verkettung):



Zeiger müssen Typ ↑<element_type> haben

- <element_type> ?

```
RECORD
    eintrag: <entry_type>;
    next, prev: ↑<element_type>;
    key: ?
END
```

- Wenn (wie anzunehmen)
  <entry_type> unterschiedlich für unterschiedliche Listen,
  dann (bei strenger Typisierung wie zB PASCAL)
  <element_type> auch unterschiedlich

  Ausweg für (einigermaßen) homogene Lösung?

  varianter record

- Lösung mit varianten records (zB):

```
TYPE
    entry_variant = (art1,art2,…);          {alle aufzuführen}
    pointer = ↑element;
    element = RECORD
                    key: REAL;
                    next, prev: pointer;
                    CASE list_variant: entry_variant OF
                      art1: (<component11>: <type11>;
                                                …
                             <component1n>: <type1n> );
                      art2: (                …                );
                                                …
                END;
    list = RECORD
            firstelement, lastelement: pointer;
            sorted: BOOLEAN
                    {Entscheidung nötig:
                      uU element-Varianten nochmals
                      schachteln nach sorted/unsorted}
          END
```

- Deklaration somit je Liste

```
VAR <list_id>: list
```

- Prozedur-Beispiel
  (<position_type> ist "pointer")

```
PROCEDURE init(VAR <list_id>: list)
```

- Weitere Alternativen

  init, add, … , delete müssen in Fällen (β)
  mit NEW und DISPOSE (bei PASCAL) oä arbeiten

  Im Falle (β1),
  dh kein DISPOSE verfügbar (ist nicht PASCAL-Standard)
  müssen eigene "Äquivalente" entworfen werden
  (etwa: "nnew", "ddispose"),
  die eigene Speicherverwaltung betreiben:
  - "ddispose"-te Plätze in "free_list" halten
  - in "nnew" wieder verwenden

  Fall (α2) schließlich: ARRAY OF RECORD

- Speziell zu (ii), Ereignisliste:

  Muß sortiert sein;
  Informationen je Eintrag:
  Zeitpunkt, Ereignisart, uU Verweise/Informationen
  | |
  uU key |
  (α1): als INTEGER verschlüsselt
  ab (α2): TYPE event = (e_typ1,e_typ2,…)
  + Eintrags-Komponente
  ereignisart: event

- Schließlich: Skizze zur "Simulationshauptschleife"
  Fall (α)

```
{Initialisierungen, insbesondere "maxtime"};
REPEAT
    examine(eventlist, first(eventlist), buffer);
    t:= {ereigniszeit aus buffer}
    CASE {ereignisart aus buffer}
        {e_typ1}:  BEGIN
                       {Ereignisroutine anstoßen,
                           i.allg. Prozeduraufruf}
                   END;
        {e_typ2}:  BEGIN
                       …
                   END;
            …              …
    END;
    delete(eventlist, first(eventlist));
UNTIL t > maxtime {ungenau};
{Auswertungen}
```

CASE ist PASCAL-typisch,
andere Möglichkeiten IF THEN ELSE, SWITCH, …

- Ausprobieren !!

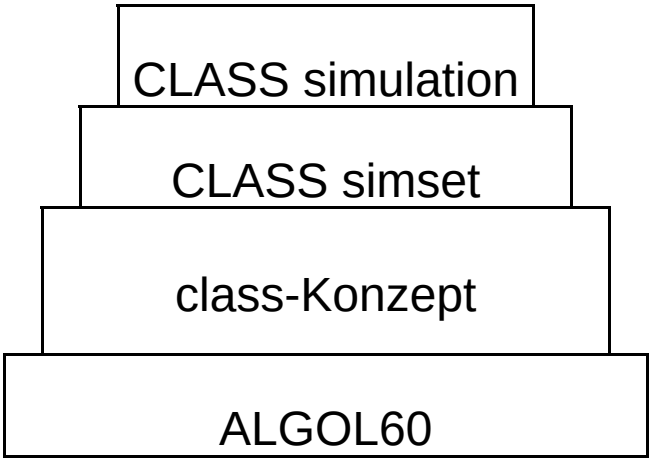
  vgl auch: Kreu86 (Einbettung in PASCAL)
  Page88, Page91 (Modula-2, C, Ada)
  Evan88, Fish95 (Datenstrukturen für event list,
  + bewertende Vergleiche)

3.5 (Wesentliche Konzepte der) Programmiersprache SIMULA

SIMULA ist (wie PASCAL) Sprache der ALGOL-Familie
Haupt-Entwicklungspunkte: 1965, 1967,
1987 (Standardisierung)

Lit: BDMN73, Rohl73, (Fran77), Lamp82, (Mitr82), Pool87, SIS87

"über das Übliche hinausgehende" Konzepte:
Konzeptpyramide (Mitr82)



Konzepte:

(iv) parallele Prozesse, Ereignisliste
(iii) Listen-Manipulation
(ii) Objektorientierte Konzepte, Koroutinen
(i) das "Übliche" in HLLs

(i) zu den "üblichen" Konzepten:

und ihrer syntaktischen Ausprägung in SIMULA

- voll blockorientiert

<program> ::= <block>
<block> ::= BEGIN
<declaration part>;
<statement part>
END
<statement> ::= <block> | …

- Deklarationsbeispiele

  INTEGER i,j;
  REAL alpha,z;
  REAL ARRAY a[1:n,0:k+1]

  alle Variablen mit Deklaration initialisiert,
  keine Konstanten, keine records

- zu Ausdrücken

arithmetische Operatoren	+ , - , * , / , // , **
	sqrt , sin , abs , …
Vergleichsoperatoren	< , > , <= , >= , ¬ =
logische Operatoren	AND , OR , IMP , ¬

  bedingte Ausdrücke erlaubt

- zu Anweisungen

  Zuweisung <variable> := <expression>

  bedingte und iterative Anweisungen
  - IF … THEN
  - IF … THEN … ELSE …
  - WHILE … DO … (kein REPEAT)
  - FOR … STEP … DO … (sehr allgemein)

  alle Klammerungen mit
  - BEGIN … END

- zu Prozeduren und Funktionen

  [<type>] PROCEDURE <identifier>
  (<formal parameters>);
  <specifications>;
  <procedure body>;

  optionales <type> kennzeichnet Funktion und Resultattyp
  Spezifikationsteil kennzeichnet Parametertypen + Übergabeart
  3 Übergabearten angeboten:
  by value, by reference, by name
  defaults je nach Parametertyp

- zu input / output

  Standards kümmerlich!

  Beispiele:

  ```
  n := inint;
  x := inreal;
  inimage;        {"Zeilen"wechsel}

  outint(n,…);    { … Formatierungsangaben}
  outfix(x,…);
  outtext("<text>");
  outimage        {"Zeilen"wechsel}
  ```

  Bessere Möglichkeiten unter Zuhilfenahme
  des Klassenkonzepts und (vordefinierter) Systemklassen

(ii) Das Klassenkonzept

- zunächst: "wie (allgemeines !) RECORD-Konzept"

```
CLASS <identifier>      (<formal parameters>);
                        <specifications>;
                        <block>;
zB
    CLASS man(age,size,weight);
        INTEGER age; REAL size,weight;
        BEGIN
            BOOLEAN overweight;
            overweight:=weight>size-100
        END man;
```

Neues Objekt ("instance")
vom Typ man
mit Attributen age, size, weight, overweight
erzeugt mit zB

```
    NEW man(42,180,75.2);
```

"body" (<block>) bei Instantiierung ausgeführt

Zuweisung Verweis auf solches Objekt mittels
:- ("denotes", "bezeichnet")
an Variable passenden Typs

Vereinbarung solcher Variabler:

```
    REF(<class identifier>) <list of identifiers>
```

im Beispiel etwa

```
    REF(man) tom, harry;        {Initialwert: NONE}
```

und

```
    tom :- NEW man(42,180,75.2)
    harry :- tom                {harry "alias" tom}
```

Objekte, auf die kein Verweis mehr existiert,
werden "automatisch" vernichtet ("garbage collection")

weitere Deklarationsbeispiele

```
REF (man) ARRAY[1:5];
REF (man) PROCEDURE a(…)
```

Zugriff auf Klassen-(Objekt-)Attribute mit dot-Notation

```
BOOLEAN critical;
    …
critical:=tom.overweight
```

aber: totale Abschottung eingeschachtelter Blöcke
(von außen kein Zugriff auf "innere"Variable)

Hantieren mit dot-Notation erleichtert durch INSPECT

```
INSPECT tom
    DO critical:=overweight
    [OTHERWISE {print warning}]
```

Test auf "Gleichheit" bzw "Ungleichheit" von Referenzen
mit == bzw =/=

Zugriff "auf sich selbst" innerhalb Klassenvereinbarung

```
THIS man
```

- Zusätzlich zu diesem reinen RECORD-Konzept
  können Klassen eigene Operatoren / "Methoden" besitzen
  (ADT- / "OO"- Denkweise !)
  zB

```
CLASS man(age,size,weight); {Spezifikationen}
    BEGIN
        BOOLEAN overweight;

        PROCEDURE aging(y); INTEGER y;
            BEGIN age:=age+y END aging;

        overweight:= …
    END man;
```

  und aktiviert werden mit zB

```
tom.aging(2)
```

- Typisch objektorientierter Aspekt "Vererbung"
  ("inheritance")
  eingeführt über "Klassenhierarchie", wobei
  "untere" / "innere" Klassen per "Klassen-Präfix"
  Attribute "oberer" / "äußerer" Klassen erben / übernehmen
  zB mit

```
CLASS person(name,address) … ;
```

  und

```
person CLASS man( … ) … ;
```

  haben alle Objekte der Klasse man
  automatisch die Attribute
  - sowohl von person (ererbt), nämlich name, address,…
  - als auch von man (zusätzlich) zB age, size,…

- Weitere Feinheiten

  Da ja sowohl mit NEW person
  als auch mit NEW man
  Objekte erzeugbar, benötigt man uU beim Zugriff von
  außen Unterstützung bei der genauen "Qualifikation";
  ermöglicht durch QUA<class_id>
  und INSPECT … WHEN <class_id>

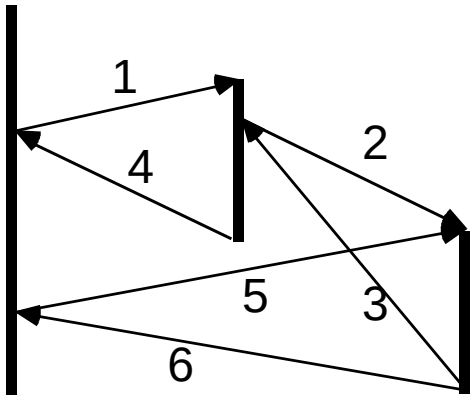
  Bei Inkarnierung eines Objektes
  wird class-body durchlaufen;
  mittels INNER läßt sich
  Durchlaufen des nächst-unteren class-body's
  an (durch INNER) definierten Ablaufpunkten erzwingen

  Ausprägungen von Attributen einer Oberklasse
  können je nach Unterklasse verschieden sein.
  VIRTUAL verlagert genaue Erklärung nach unten / innen
  zB

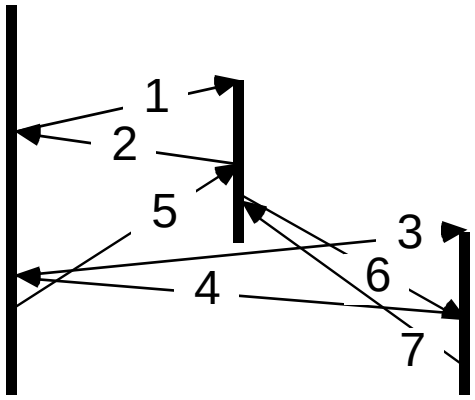
```
CLASS a;
    VIRTUAL PROCEDURE p;
    BEGIN
        … ; p; …
    END a;

a CLASS b;
    BEGIN
        PROCEDURE p … ;
        …
    END b;
```

- Klassen realisieren des weiteren **Koroutinen-Konzept**

  Ablauf-Prinzip bei "üblichen" Prozeduren / Funktionen



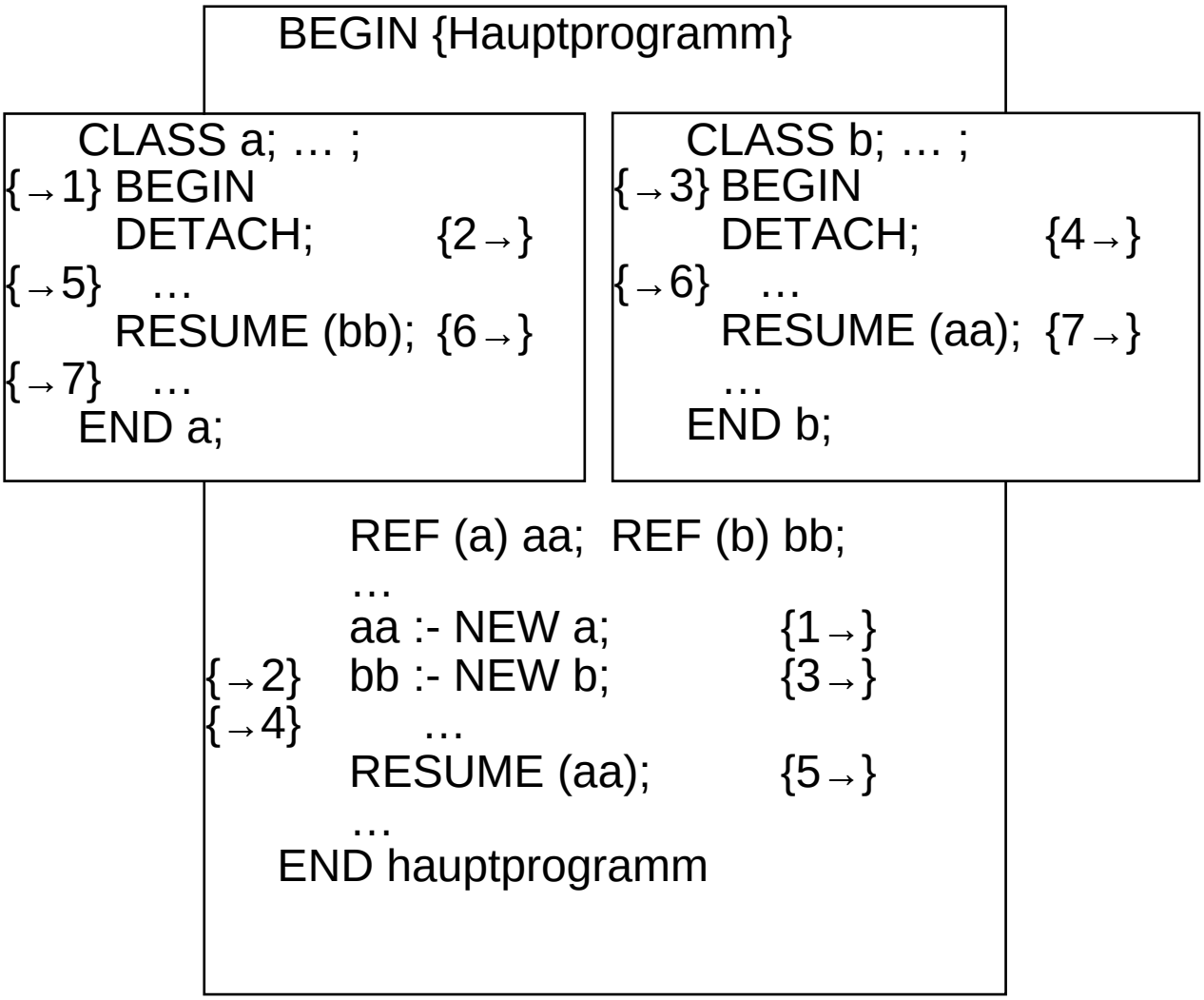
  davon prinzipiell verschiedener, verzahnter Ablauf
  (war ja dringend gefragt, vgl 3.2)



  In SIMULA wird class-body anläßlich NEW
  (bei Inkarnation Objekt) durchlaufen.
  Durchlauf kann mittels
  DETACH (in diesem body)
  gestoppt werden (zunächst!),
  Kontrolle geht zurück an Aufrufenden (der NEW sagte)

  DETACHtes Klassenobjekt a kann mit
  RESUME (a)
  wieder aufgenommen werden (an Unterbrechungsstelle)

Typisches Ablaufbeispiel:



Mittels dieses spracheigenen Koroutinenkonzepts wird (später) die anwendungsbezogene

Systemklasse "simulation"

implementiert werden

(iii)Die system class "simset"

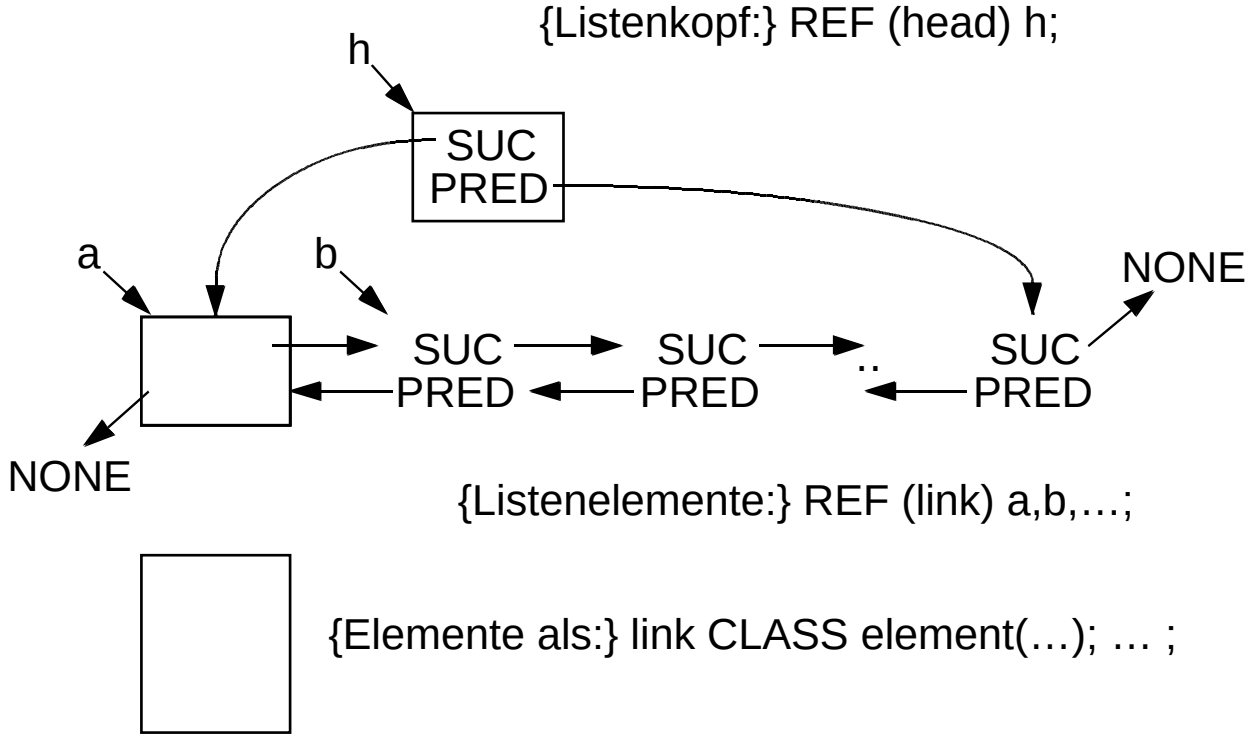
- Vordefinierte (System-)Klasse, enthält
  Konzepte für Deklaration und Manipulation
  doppelt verketteter Listen (über beliebigen Objekttypen)

- Zugriff auf Konzepte über Präfix vor eigenem Programm
  simset BEGIN {eigener SIMULA-Code} END

- top level Struktur der Klasse:

```
CLASS simset;
BEGIN
    CLASS linkage; … {realisiert Vor- und Rück-
                                        verzeigerung};
    linkage CLASS head; … {Listenkopf: Liste selbst};
    linkage CLASS link; … {Listenelementköpfe};
END simset;
```

- Architektur:



- Operatoren (Minimalausbau) genauer:

```
CLASS linkage;
   BEGIN  REF (link) PROCEDURE suc; … ;
                  {liefert Verweis auf Folge-Element};
          REF (link) PROCEDURE pred; … ;
                  {liefert Verweis auf Vor -Element};
          {damit Zugriff von außen ausgeschlossen}
   END linkage;


linkage CLASS head;
   BEGIN  PROCEDURE clear; … {leert Liste};
          REF (link) PROCEDURE first; …
                  {liefert Verweis auf erstes Element};
          REF (link) PROCEDURE last; …
                  {liefert Verweis auf letztes Element};
          BOOLEAN PROCEDURE empty; …
                  {prüft ob Liste leer};
          INTEGER PROCEDURE cardinal; …
                  {liefert Anzahl Listenelemente};
   END head;


linkage CLASS link;
   BEGIN  PROCEDURE out; … ;
          PROCEDURE into(h); REF (head) h; … ;
          PROCEDURE precede(x);
                                 REF (linkage) x; … ;
          PROCEDURE follow(x);
                                 REF (linkage) x; … ;
                  {vgl Semantik-Erklärungen}
   END link;
```

Ein Objekt kann
zu **einem** Zeitpunkt
nur **einer** Liste angehören

Eine Liste kann
Objekte **verschiedener** Typen
(dh verschiedener Unterklassen von link)
enthalten

Zur Erklärung der link-Operationen:

- Seien lk, lk1, lk2 Objekte einer link-Unterklasse, (also Listenelemente),
  hd ein Objekt der Klasse head (also eine Liste)

- lk.into(hd) lk wird letztes Listenelement in Liste hd
  lk1.precede(lk) lk1 wird vor lk eingefügt, in Liste, der lk angehört
  lk2.follow(lk) lk2 wird nach lk eingefügt, in Liste, der lk angehört
  lk.out lk wird entfernt, aus Liste, der es angehört

- Einfaches Anwendungsbeispiel:

```
simset BEGIN
        INTEGER i;
        REF (head) liste;
        link CLASS ganz(i); INTEGER i; ;
        liste :- NEW head;
        FOR i:=1 STEP 1 UNTIL 10 DO
            NEW ganz(i).into(liste);
        END beispiel
```

(iv)Die Systemklasse "simulation"

- Vordefinierte (System-)Klasse, enthält
  Konzepte der **ereignisorientierten** Simulation,
  in **process-interaction** Sicht
  insbesondere
  Konzept **interagierender Prozesse**
  Konzept einer **Zeitachse**

- Zugriff auf Konzepte über Präfix
  simulation BEGIN {eigener SIMULA-Code} END
  simulation ist Unterklasse von simset,
  enthält damit alle **Listenkonzepte** (wie besprochen)

- Beliebige Klassen(objekte) können mithilfe
  Präfix "process"
  zu Prozessen erklärt werden,
  die an dynamischem Ablauf teilnehmen

- simulation-Zeitachse enthält (wie üblich) Elemente,
  die jeweils ein Ereignis charakterisieren,
  konkret Objekte mit 2 Attributen
  - EVTIME Zeitpunkt des Ereigniseintritts
  - PROC Verweis auf (zu diesem Zeitpunkt)
    zu aktivierenden Prozeß;

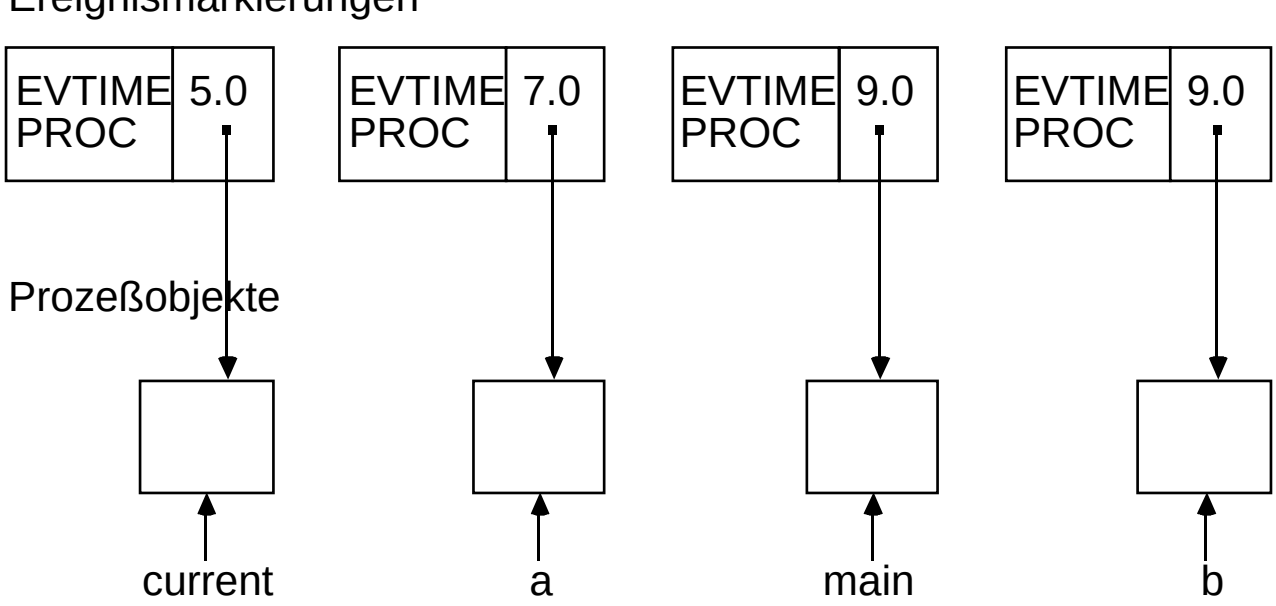
  ist gemäß EVTIME aufsteigend sortiert,
  Elemente mit gleicher EVTIME können zusätzlich
  in setzbarer Folge angeordnet werden

- "current" liefert immer Verweis auf den
  im ersten Element der Zeitachse notierten Prozeß
  (dies ist der "aktuelle", gerade aktive Prozeß)

- "time" liefert immer die zugehörige Aktivierungszeit
  (dh die "momentane" Zeit, die Modellzeit)

- Zeitachse kann Ereignis bzgl des "uneigentlichen" Prozesses "main" enthalten (des Simulationsblocks; jenes Blocks, der Präfix "simulation" trägt)

- Beispiel Zeitachse (Liste) mit 4 Ereignissen



current verweist auf jeweils aktiven Prozeß

kann während aktiver Phase (seine) nächste bestimmen mittels REACTIVATE {zu Zeitpunkt}
HOLD {für Zeitintervall}

(Vormerkung Zeitachse, Zustand: aktiv → vorgemerkt)

kann sich während aktiver Phase deaktivieren mittels
PASSIVATE
WAIT

(keine Vormerkung Zeitachse, Zustand: aktiv → passiv)

Simulationsmechanismus benutzt Koroutinenkonzepte. Zur Vermeidung (sehr versteckter) Fehler ist von expliziter Verwendung von DETACH, RESUME etc abzuraten

Übersicht über **Systemklasse "simulation"**

```
simset
    CLASS simulation;
        BEGIN
            link CLASS process; … ;
            REF (process) PROCEDURE current; … ;
            REAL PROCEDURE time; … ;
        COMMENT scheduling-Prozeduren (o. A. Parameter);
            PROCEDURE hold; … ;
            PROCEDURE passivate; … ;
            PROCEDURE wait; … ;
            PROCEDURE cancel; … ;
            PROCEDURE activate; … ;
            …
            REF ("simulationsblock") main;
        COMMENT   folgen Aktionen zur Initialisierung
                  der Zeitachse, Anfangszeit "0";
        END simulation;
```

zentrale Rolle spielt **Klasse "process"**:

```
link CLASS process;
        BEGIN   BOOLEAN PROCEDURE idle; … ;
                BOOLEAN PROCEDURE terminated; … ;
                REAL PROCEDURE evtime; … ;
                REF (process) PROCEDURE nextev; … ;
            DETACH;          {nach Instantiierung: Stop}
            INNER:           {nach Aktivierung: Unterklasse}
            "TERMINATE"      {nach Unterklasse: Schluß}
        END process;
```

Objekte der Klasse "process"
(auch Objekte einer ihrer Unterklassen)
sind **Prozeßobjekte**

Da process Unterklasse von link,
können Prozeßobjekte in (simset-) Listen geführt werden

Prozeßobjekt befindet sich nach "Generierung"
im Zustand "detached",
LSC (local sequencing counter) weist auf
erste (Benutzer-) Anweisung des Prozeßobjektes.

Dieses nach "Aktivierung" des Prozesses durchlaufen
(nicht notwendig, typischerweise nicht "am Stück");
wenn aber ganz durchlaufen, Rückkehr
und nachfolgend "Terminierung".

Insgesamt vier unterschiedliche "Zustände" für Prozeßobjekt;

- ist in Zeitachse repräsentiert
  - als erstes Element, Zustand: aktiv **active**
  - als "späteres" Element, Zustand: vorgemerkt **suspended**

- ist nicht in Zeitachse repräsentiert
  - Aktionen erschöpft, Zustand: beendet **terminated**
  - Aktionen noch nicht erschöpft, Zustand: passiv **passive**

Zustand mittels process-Prozeduren
- idle
- terminated
- evtime

überprüfbar / erkennbar

Die process-Prozeduren idle, terminated, evtime liefern:

Prozeß-Zustand	Ergebnisse von idle	terminated	evtime
aktiv	false	false	momentaner Zeitpunkt
vorgemerkt	false	false	Zeitpunkt der Vormerkung
passiv	true	false	Fehler
beendet	true	true	Fehler

Zu den **simulation-Prozeduren / -Funktionen**:

- REF (process) PROCEDURE current; … ;
  Verweis auf soeben aktiven Prozeß (bzw aktives "main")

- REAL PROCEDURE time; … ;
  aktuelle Simulationszeit (≡ current.evtime)

- PROCEDURE hold(interval); REAL interval; … ;
  beendet (einstweilen) Aktionen von current;
  current wechselt: aktiv → vorgemerkt zu time+interval,
  (neuer Eintrag in Zeitachse, interval<0 → =0)
  Reaktivierung des (bisherigen) current
  (später) automatisch,
  zum Vormerkzeitpunkt, an "Unterbrechungs"stelle (LSC !)

- PROCEDURE passivate; … ;
  beendet (einstweilen) Aktionen von current;
  current wechselt: aktiv → passiv,
  (Eintrag in Zeitachse gelöscht)

- PROCEDURE wait(s); REF (head) s; … ;
  current wird in (FCFS-)Liste s eingetragen
  und wechselt: aktiv → passiv

- PROCEDURE cancel(x); REF (process) x; … ;
  Ein (etwaiger) Eintrag von x in Zeitachse wird gelöscht
  x wechselt: (aktiv, vorgemerkt) → passiv,
  (passiv, terminiert) → keine Wirkung
  cancel(current) ≡ passivate

Die **Aktivierungsprozeduren**

(da kein "Warten auf Zustand" vorgesehen,
müssen alle Aktivierungen explizit vorgenommen werden)

{ ACTIVATE / REACTIVATE } <process_expression_1>

[{ AT / DELAY } <time> [PRIOR]]
[{ BEFORE / AFTER } <process_expression_2>]

Verweise <process_expression_1> auf x, dann

- ACTIVATE wirkungslos, falls x nicht passiv
- REACTIVATE wirkungslos, falls x terminiert
- REACTIVATE wirkungsgleich mit cancel + ACTIVATE,
  falls x aktiv oder vorgemerkt
- REACTIVATE wirkungsgleich mit ACTIVATE,
  falls x passiv

{ ACTIVATE | REACTIVATE } <process_expression_1>

[{ AT | DELAY } <time> [PRIOR] | { BEFORE | AFTER } <process_expression_2>]

ACTIVATE / REACTIVATE x ohne einen der (optionalen) Zusätze
bewirkt unmittelbare Aktivierung von x,
aufrufender Prozeß: aktiv → vorgemerkt (mit gleicher Zeit),
aufgerufener Prozeß: → aktiv (neues current).

Zeitzusätze

AT t	DELAY t
bewirken Vormerkung zur (Simulations-)Zeit	
t	time+t
(falls t<time: AT time)	(falls t<0: DELAY 0)

Zusatz
PRIOR
bewirkt Vormerkung vor allen anderen Vormerkungen derselben Vormerkungszeit
(ohne PRIOR: danach)

Verweise <process_expression_2> auf y, dann erfolgt Vormerkung von x (falls y vorgemerkt oder aktiv)

BEFORE y	unmittelbar vor y, zur gleichen Zeit
AFTER y	unmittelbar nach y, zur gleichen Zeit

Ist dabei y weder vorgemerkt noch aktiv,
oder x==y,
dann resultiert Wirkung cancel(x)

Beispielskizzen:

Der einfache (Bank-)Schalter in SIMULA-Denkweise

(nicht in SIMULA-Notation),

in drei alternativen Versionen (Wechsel der "Akteure)

Version 1

Schalterabfertigung ist Prozeß,
Ankünfte werden dem "Hauptprogramm" (main) überlassen,
Kunden sind "passiv" ("werden herumgeschaufelt")

Prozeß Bedienung:

```
FOREVER DO
BEGIN
    WHILE {kundenliste nicht leer} DO
    BEGIN     {pausiere für Bedienzeit}
              {entferne Kunden}
    END;
    {warte auf Ankunft → warte (unbeschränkt)}
END
```

Hauptprogramm:

```
{setze  simulationsdauer auf Wert,
        abbruch auf FALSE,
  initialisiere kundenliste,
  initialisiere schalter (ist "Bedienung") }
WHILE ¬ abbruch DO
BEGIN
    {pausiere für Zwischenankunftszeit};
    IF {simulationsdauer überschritten}
    THEN  abbruch:=TRUE
    ELSE  BEGIN
              {erzeuge neuen Kunden und füge ihn
                                    in kundenliste ein};
              {aktiviere Schalter, falls dieser passiv}
          END
END;
{uU: pausiere für gewisse Zeit ("Leerlaufenlassen")}
```

Version 2

Schalterabfertigung, Kundenankunftserzeugung
sind Prozesse,

Prozeß Bedienung: wie Version 1

Prozeß Kundenstrom:

```
FOREVER DO
BEGIN    {pausiere für Zwischenankunftszeit};
         {erzeuge neuen Kunden, → kundenliste};
         {aktiviere schalter, falls dieser passiv}
END;
```

Hauptprogramm:

```
{ setze simulationsdauer auf Wert,
  initialisiere kundenliste,
  initialisiere schalter (ist "Bedienung"),
  initialisiere und aktiviere Kundenstrom };

{ pausiere für simulationsdauer };

{uU:   halte Kundenstrom an,
       pausiere für gewisse Zeit }
```

Version 3

Schalterabfertigung,
Kundenankunftserzeugung,
jeder Kunde sind Prozesse,

Es gibt eine Warteschlange
für wartendende Kunden(-Prozesse)

Prozeß Kunde:

```
{stell' Dich an;
    aktiviere Schalter und warte bis Du dran bist:
    ACTIVATE schalter AFTER current;
    WAIT (kundenliste)                              };
{pausiere für Bedienzeit};
{geh' weg: entferne Dich aus Kundenliste};
```

Prozeß Bedienung:

```
FOREVER DO
BEGINWHILE {kundenliste nicht leer} DO
        BEGIN{aktiviere ersten ("dran") Kunden};
                {warte auf Bedienende:
                    REACTIVATE current AFTER "dran"}
        END;
        {warte auf Kundenankunft: warte (unbegrenzt)}
END;
```

Prozeß Kundenstrom:

```
FOREVER DO
BEGIN   { pausiere für Zwischenankunftszeit };
        { initialisiere, aktiviere neuen Kunden }
END;
```

Hauptprogramm:

```
{ setze simulationsdauer auf Wert,
  initialisiere kundenliste,
  initialisiere schalter (ist "Bedienung"),
  initialisiere und aktiviere Kundenstrom };

{ pausiere für simulationsdauer };
{ halte Kundenstrom an };
{ pausiere für gewisse Zeit }
```

Viel Spaß beim "Üben"