RECHENZENTRUM TH MÜNCHEN
ARBEITSGRUPPE BETRIEBSSYSTEME

INTERNSCHRIFT Nr. 15

THEMA:

Simulation

VERFASSER:

v.CONTA

DATUM:

WS 68/69

FORM DER ABFASSUNG

- ENTWURF
- X AUSARBEITUNG
- ENDFORM

SACHLICHE VERBINDLICHKEIT

- X ALLGEMEINE INFORMATION DISKUSSIONSGRUNDLAGE
- ERARBEITETER VORSCHLAG
- VERBINDLICHE MITTEILUNG
- VERALTET

ÄNDERUNGSZUSTAND

BEZUG AUF BISHERIGE INTERNSCHRIFTEN

Vorkenntnisse aus:

Erweiterung von:

Ersatz für:

BEZUG AUF KÜNFTIGE INTERNSCHRIFTEN

Vorkenntnisse zu:

Erweiterung in:

Ersetzt durch:

ANDERWEITIGE LITERATUR

ARBEITSSEMINAR UBER BETRIEBSSYSTEME

WS 1968/69

F. Simulation

(von Conta)

1. Was versteht man unter Simulation

Von Simulation kann man immer dann sprechen, wenn ein tatsächlicher Ablauf anhand eines zugeordneten vereinfachten Modells untersucht wird. Man sagt dann, der tatsächliche (simulierte) Ablauf werde in dem Simulationsmodell simuliert. So verstanden ist fast jede Denktätigkeit Simulation.

Als Aufgabe für die Rechenanlage sind z.B. folgende Simulationsvorgänge von praktischer Bedeutung:

a) Simulation eines Analogrechners:
Die einzelnen Programmteile des betreffenden Simulationsprogramms simulieren das Verhalten von Kondensatoren, Widerständen etc., soweit es für das Resultat von Bedeutung ist. Eigene Simulationssprachen wie z.B. KALDAS oder SLANG für die Simulation von Analogrechnern geben Formulierungshilfen, um Verknüpfungen und Parameterwahl dieser Elemente (wie sie Steckverbindungen, Potentiometerstellungen etc. am Analogrechner entsprechen) im Simulationsmodell zu realisieren.

b) Simulation eines Kernreaktors:
Hierbei wird z.B. die Neutronenverteilung im Inneren des Kernreaktors ermittelt, indem der Weg einzelner Neutronen mit statistischen Anfangsdaten durch den Kernreaktor rechnerisch verfolgt wird. Eine statistische Überlagerung vieler solcher Neutronenbahnen gibt dann ein Bild von der zu erwartenden Neutronenverteilung im Inneren des Reaktors z.B. für eine bestimmte Stellung der Graphitstäbe.

c) Simulation des Betriebssystems einer Rechenanlage:
Will man den Durchsatz an gerechneten Aufträgen für ein gegebenes Betriebssystem erhöhen, so hat man zunächst nach auftretenden Engpässen zu suchen und muß diese dann nach Möglichkeit beheben. Wenn beispielsweise der Rechner häufig warten muß, bis benötigte Information von einem Plattenspeicher beschafft ist, so könnte man versuchen, die Platte durch ein schnelleres Speichermedium zu ersetzen oder auch die Organisation des Plattenverkehrs durch geeignete Abänderungen leistungsfähiger zu machen.

Die Simulation muß hier also Informationen liefern, die den Ablauf beim Betrieb des simulierten Betriebssystems zu analysieren gestatten und darüber hinaus die Untersuchung ermöglichen, wie sich z.B. noch nicht gekaufte Zusatzgeräte oder noch nicht programmierte andersartige Verwaltungsroutinen auswirken würden.

Als Formulierungshilfen für die Simulation eines Betriebssystems eignen sich allgemeinere Simulationssprachen wie SIMON oder CSL, die sich z.B. von einer speziell für die Simulation von Analogrechnern entwickelten Sprache erheblich unterscheiden. In solchen allgemeineren Simulationssprachen lassen sich dann neben der Betriebssystemsimulation auch sehr viele ganz andersartigen Simulationsprobleme formulieren.

Zusammenfassend können wir sagen: Der Begriff der SIMULATION als solcher ist viel zu weit gefaßt, als daß man damit fruchtbar operieren könnte. Dagegen ist die Bedeutung spezieller Simulationsaufgaben wie die der Simulation eines Analogrechners, eines Kernreaktors, eines Betriebssystems ohne viele Erläuterungen von vorn herein klar.

Simulation wird angewendet, wenn

α) der simulierte Vorgang nicht real durchgeführt werden kann (weil z.B. kein entsprechender Analogrechner zur Verfügung steht (a), oder der Kernreaktor beim Experiment explodieren könnte (b), oder weil wesentliche Anlagenteile im Fall (c) noch nicht gekauft wurden).

β) Detailinformation über den Ablauf benötigt wird, die durch unmittelbare Beobachtung desselben nur schwer zu erhalten ist. (Wo liegen die Engpässe im Fall α.)

Für uns ist hauptsächlich interessant, die mit der Simulation eines Betriebssystems zusammenhängenden Fragen zu untersuchen. Um mit dieser Fragestellung vertraut zu werden, betrachten wir im folgenden zunächst die praktische Arbeit mit dem Simulationsmodell eines Betriebssystems, wie sie N.R. NIELSEN beschrieben hat, ohne vorerst näher auf die Struktur des Simulationssystems selbst einzugehen. Dabei wollen wir versuchen, Grundaufgaben zu erkennen, die für Simulationsprobleme typisch sind, und für deren Formulierung in eine Simulationssprache entsprechende Sprachelemente bereitgestellt werden sollten.

2. Die praktische Arbeit mit Hilfe des Simulationsmodells für ein Betriebssystem

(N.R NIELSEN: The simulation of timesharing systems, CACM 10 (1967) p 397 ff)

a) Die Aufgabe

Geplant war der Ankauf eines IBM 360/67 - Timesharing Systems. Gesucht war eine optimale Konfiguration: Wieviel Kernspeicher, Anzahl und Typ der Plattenspeicher und Trommelspeicher, Ausführung der Datenübertragungswege (simplex oder halb-duplex).

Die erste große Schwierigkeit für die Lösung dieser Aufgaben hängt nicht mit der Struktur des Simulationsmodells zusammen, resultiert aber aus den Umständen, unter denen im allgemeinen solche Simulationsprobleme gelöst werden müssen: Es waren nämlich die benötigten Informationen über Hardware- und Softwareparameter des zu liefernden Systems nur sehr schwer vom Hersteller zu beschaffen und die schließlich angegebenen Werte erwiesen sich als sehr ungenau. Die mit der Simulation beauftragte Arbeitsgruppe erstellte in vielen Fällen eigene, viel ungünstigere Schätzwerte, die sich am Ende immer noch als zu optimistisch herausstellten.

Die Ähnlichkeit des Simulationssystems mit dem simulierten System (d.h. die Güte der Simulation) litt unter diesen ungenauen Annahmen zwar beträchtlich, es zeigte sich jedoch, daß die Schlußfolgerungen bezüglich optimaler Konfiguration trotzdem richtig waren und das Simulationsmodell trotz allem qualitativ brauchbare Resultate lieferte.

In Aussicht genommen war zunächst eine Konfiguration aus

31 Konsolen
1 CPU
3 Kernspeicher mit je 256 Kbyte
1 Trommelspeicher
1 großer Plattenspeicher } für paging-Zwecke
1 großer Plattenspeicher
2 kleinere Plattenspeicher } für file-Verkehr

Bandgeräte, Zeitendrucker und Kartenleser wurden vorerst nicht berücksichtigt.

b) Die Durchführung

Die Simulation der Rechenaufträge (job-Simulation) erfolgte in der Weise, daß zunächst eine Reihe von Musterprogrammen (Jobs) simuliert wurden, die in 10 verschiedene Typen zerfielen. Diese Typen unterschieden sich etwa darin voneinander, daß bei den zu einem Typ gehörigen Jobs im wesentlichen nur Rechnung anfiel, während die zu einem anderen Typ gehörigen Jobs das System im wesentlichen nur mit file-Verkehr belasteten. Wieder andere Typen wiesen starke Inanspruchnahme des paging-Mechanismus auf etc.

Für jede Konsole wurde nun eine eigene Häufigkeitsverteilung für diese Muster-Jobs vorgegeben und die Reihenfolge der Jobs an dieser Konsole als Zufallsfolge (unter Beachtung dieser Häufigkeitsverteilung) angenommen.

Zu den im folgenden beschriebenen verschiedenen Simulationsläufen vergleiche man die folgende Tabelle.

	CPU-Exek (%)	CPU-Verw (%)	CPU-Leer (%)	Länge der Warteschlange für paging	Länge der Warteschlange für platten-file-Verkehr
b1	13	40	47	>19	
realist. Job-mix und Job-mix mit wenig Konsolverkehr					
b2	43	35	22	0	13->19
geringer Konsolverkehr wenig paging					
b3	20	46	34	12	>19
geringer Konsolverkehr realist. paging-Job-mix vierfache paging-Plattenzahl					
b4	31	69	0	12	0
geringer Konsolverkehr realist. paging-Job-mix vierfache paging-Plattenzahl Plattenzugriff im Fileverkehr 0					
b5	31	62	1	8	13
geringer Konsolverkehr realist. paging-Job-mix vierfache paging-Plattenzahl doppelte file-Plattenzahl					
b6	5	76	19	>19	
Job-mix mit viel I/O vierfache paging-Plattenzahl geändertes file-Plattenmanagement (1 große file-Platte)					
b7	8	92	0	9	
Job-mix mit viel I/O 2 paging-Trommeln statt 4 Platten geändertes file-Plattenmanagement					
b8	16	81	3	11	
realist. Job-mix, normales I/O 2 paging Trommeln statt 4 Platten geändertes file-Plattenmanagement geändertes Druck-Management					

b1) Lauf mit realistischem Job-mix und bei Job-mix mit wenig Konsolverkehr

Zunächst dachte man, daß die hohe CPU-Leerzeit von 47 % darauf zurückzuführen sei, daß über die langsamen Konsolen nicht genug Aufträge angeliefert würden, um den Rechner auszulasten. Daher wiederholte man den Lauf mit einem Job-mix mit stark reduziertem Konsolverkehr. Das Resultat blieb jedoch im wesentlichen unverändert, so daß der Grund für die hohe CPU-Leerzeit anderswo zu suchen ist.

Einen Hinweis auf die mögliche Ursache gibt die lange paging-Warteschlange (Warteschlange für Ein/Ausgabe im Verkehr mit den für paging-Zwecke verwendeten Platten). Eine Überschlagsrechnung ergab Mindestwartezeiten für jede Seite von über 0.8 sec. Um die Annahme zu prüfen, ob das paging zum Engpaß wurde, wurde der folgende Lauf gemacht:

b2) Lauf bei Job-mix mit wenig Konsolverkehr und wenig paging

Der Job-mix wurde so gewählt, daß er das System kaum für paging in Anspruch nahm. Dieser Job-mix ist zwar sehr unrealistisch, doch kann man so prüfen, inwieweit das paging die Schuld trägt an der hohen CPU-Leerzeit im Lauf b1. Tatsächlich ergab sich eine um über die Hälfte geringere CPU-Leerzeit.

Damit war der paging-Plattenverkehr als Engpaß nachgewiesen. Nun galt es nach Lösungen zu suchen, denselben zu beschleunigen, so daß er auch bei realistischem Job-mix eine bessere CPU-Ausnützung gestattet.

Der erste Gedanke war hierzu, die Plattenzahl für paging-Zwecke zu erhöhen.

b3) Lauf bei Job-mix mit wenig Konsolverkehr (sonst realistisch) und vierfacher paging-Plattenzahl

Die Zahl der paging-Platten wurde vervierfacht und alle mit separaten Datenkanälen ausgestattet. Abgesehen vom reduzierten Konsolverkehr (wie im 2. Lauf von b1) wurde nun wieder mit realistischerem Job-mix gerechnet. Die Verbesserung der CPU-Leerzeit auf 34 % gegenüber b1 war deutlich, jedoch nicht so gut, wie man nach b2 erwartet hatte.

Der Verdacht lag (wegen der betreffenden Warteschlangenlänge) nahe, daß ebenso wie vorher der paging-Plattenverkehr auch der Plattenverkehr für Zwecke der file-Speicherung zu langsam arbeitet. Zur Prüfung dieser Annahme wurde probeweise mit verschwindender Zugriffszeit für die file-Verkehr-Platten gerechnet.

b4) Lauf bei Job-mix mit wenig Konsolverkehr (sonst realistisch), vierfacher paging-Plattenzahl und verschwindender Zugriffszeit im File-Plattenverkehr

Das Verschwinden der CPU-Leerzeit bestätigte den Verdacht, daß auch noch der file-Plattenverkehr beschleunigt werden muß, wenn die CPU besser ausgenutzt werden soll, als in den früheren Läufen.

Gleichzeitig kündigt sich jedoch mit dem hohen Anteil von 69 %, den Verwaltungsroutinen an der CPU-Arbeit beanspruchen, ein neuer Engpaß an.

Um einen technisch gangbaren Weg zur file-Verkehrbeschleunigung zu erproben, wurde nun die im file-Verkehr eingesetzte Plattenzahl verdoppelt (ähnlich wie früher die paging-Plattenzahl vervierfacht worden war):

b5) Lauf bei Job-mix mit wenig Konsolverkehr (sonst realistisch), vierfacher paging-Plattenzahl und doppelter Plattenzahl für file-Verkehr

Es ergibt sich, daß dies eine technisch mögliche Lösung für den Engpaß wäre, der durch den Plattenverkehr zustande kam. Jedoch scheidet sie aus finanziellen Gründen aus.

Statt dessen versuchte man eine Beschleunigung des software-Plattenmanagements zu erreichen. Die Organisation war bisher so, daß die einzelnen Plattenmoduln nacheinander jeweils vollständig aufgefüllt wurden. Das hatte zur Folge, daß z.B. das paging fast ausschließlich über einen Plattenarm abgewickelt wurde, was wegen der langsamen Beweglichkeit desselben im Mittel eine maximale Seitentransferrate von 8 Seiten je sec. erlaubte. Analoges gilt für den file-Verkehr.

Statt dessen verteilte man nun das paging über alle Moduln gleichmäßig. Die entstehenden Lücken auf den Platten wurden entweder für die file-Speicherung genutzt oder einfach frei gelassen. Da dies zwangsläufig zu schnellerem Plattenverkehr führen mußte, wurde kein Kontrollauf mit vergleichbarem Job-mix durchgeführt.

Bei den nun folgenden Läufen wurden nun auch

2 Bandgeräte
1 Drucker
1 Kartenleser

berücksichtigt. Ferner wurde nun angenommen:

1 Trommelspeicher
4 kleine Plattenspeicher } für paging-Zwecke
1 großer Plattenspeicher (mit geändertem Management) für file-Verkehr.

b6) Lauf mit viel I/O, paging und file-Verkehr, 1 Trommel + 4 Platten für paging, neues Plattenmanagement

Der Job-mix wurde nun so gewählt, daß hauptsächlich die kritischen Belastungstypen der Anlage auftraten (paging, file-Verkehr, Konsolverkehr, ferner Kartenlesen und -drucken). Dazu kamen einige Hintergrundprogramme, auf die das System während der Wartezeiten zurückgreifen konnte.

Die Resultate sind wegen des geänderten Job-mix nicht vergleichbar mit denen früherer Läufe. Jedoch war das Resultat enttäuschend mit 19 % CPU-Leerzeit, wobei die restlichen 81 % fast völlig mit Verwaltungsarbeit (76 %) ausgelastet waren, so daß nur 5 % der CPU-Zeit der eigentlichen Rechenarbeit zugute kamen.

Da der paging-Plattenverkehr sich als Engpaß erwiesen hatte, der nun besonders stark in Anspruch genommen war, und da ferner nach der Änderung des Plattenmanagements die paging-Platten schlecht ausgenützt waren wegen der vielen Leerstellen, lag es nahe, anstelle der 4 paging-Platten eine weitere paging-Trommel einzusetzen die schneller ist und deren zwar geringere Speicherkapazität dafür besser ausgenützt werden kann.

b7) Lauf mit viel I/O, paging und file-Verkehr; 2 Trommeln für paging. Neues Plattenmanagement

Die CPU-Leerzeit von 0 % stellt eine wesentliche Verbesserung dar; die der Rechnung zugute kommende CPU-Exekutionszeit hat sich aber nicht wesentlich verbessert, da der Verwaltungsanteil auf 92 % angewachsen ist.

Die Schuld für den hohen Verwaltungsaufwand wurde auch zum Teil der Druckroutine gegeben. Denn diese versuchte eine optimale Druckerausnützung zu erreichen, indem sie nur jeweils eine Zeile druckte. Der dann notwendig erfolgende Interrupt kostete jedesmal 2 ms Verwaltung für die CPU. Nun baute man die Druckroutine um, auf Ausgabe von jeweils 30 - 40 Zeile auf einmal/um die Häufigkeit der notwendigen Interrupts zu reduzieren, obwohl dies angeblich die Kapazität des Zeilendruckers nicht voll auszunutzen gestattete.

In der Tat zeigte ein Kontrollauf ein günstigeres Bild. Wir wollen jedoch abschließend nur noch einen Lauf mit realistischem Job-mix betrachten, der in etwa mit b1) vergleichbar ist.

b8) Lauf mit realistischem Job-mix, 2 Trommeln für paging, neues Platten- und Druckermanagement

Die CPU-Leerzeit von 3 %, die CPU-Verwaltungszeit von 81 % und CPU-Jobexekutionszeit 16 % können als ein gewisser Erfolg der Bemühungen angesehen werden.

Nun wurde versucht, den Verwaltungsaufwand kleiner zu machen. Die entsprechenden Verwaltungsroutinen wurden daraufhin durchgegangen, in wiefern sie schneller gemacht werden könnten, und es wurden dementsprechend kleinere Softwareparameter im Simulationsprogramm angenommen.

Dadurch mußte der Job-Durchsatz an der Anlage größer werden und es stellte sich die Frage, ob dann immer noch 2 paging-Trommeln günstiger waren als 4 Platten und 1 Trommel. Das Resultat entsprechender Simulationsläufe zeigte, daß dann die 2 Trommeln noch viel günstiger als 4 Platten und eine Trommel waren, als es schon früher der Fall war.

Weiter wurde die Frage gestellt, ob man von den 3 Kernspeichereinheiten eine für Zwischenspeicherung im Zusammenhang mit paging reservieren sollte. Es ergab sich aber, daß dann die übrigen beiden nicht ausreichen.

Schließlich wurde ein Lauf mit sehr unrealistischem Job-mix mit exzessiver Belastung der Datenübertragungswege gemacht. Dabei zeigte sich, daß die Datenübertragungsrate sich hierbei jedenfalls unterhalb 10^6 byte/sec bewegte, so daß es genügte, diese Kanäle in der Halbduplexausführung (mit getrenntem Speicherzugriff für Datenkanäle und CPU) wesentlich billigeren Simplexausführung zu installieren.

c) Grundaufgaben, die sich bei dieser Simulation stellen

Nachdem wir im Abschnitt b) die Arbeit mit einem Simulationsprogramm kennen gelernt haben, wollen wir uns nun der Frage zuwenden, was für Anforderungen sich daraus an das Simulationssystem ergeben.

c1) Modell für den internen Ablauf

In einer Variablen namens

UHR (variable)

wird jeweils der Zeitpunkt eingetragen, bis zu dem die Zeit im simulierten System fortgeschritten ist.

Als Elemente (entities) des Simulationssystems treten Dinge auf wie z.B. CPU, Eingabekanal von der Trommel (EKvT), Ausgabekanal auf die Trommel (AKnT), denen neben anderen Eigenschaften auch eine Ereigniszeit zugeordnet ist, die den Zeitpunkt angibt, zu dem der dort gerade laufende Vorgang abgeschlossen ist und ein neuer Vorgang in Angriff genommen werden kann:

Elemente mit Ereigniszeit (z.B. CPU, EKvT, AKnT, etc.)

Man kann sich diese Elemente in einer speziellen Liste (ZEITLISTE) eingetragen denken. Wenn nun die Simulation zum nächsten Zeitpunkt im simulierten System fortschreiten soll, zu dem wieder eine neue Situation eintritt, auf die man mit geeigneten Simulationsmaßnahmen zu reagieren hat, so braucht

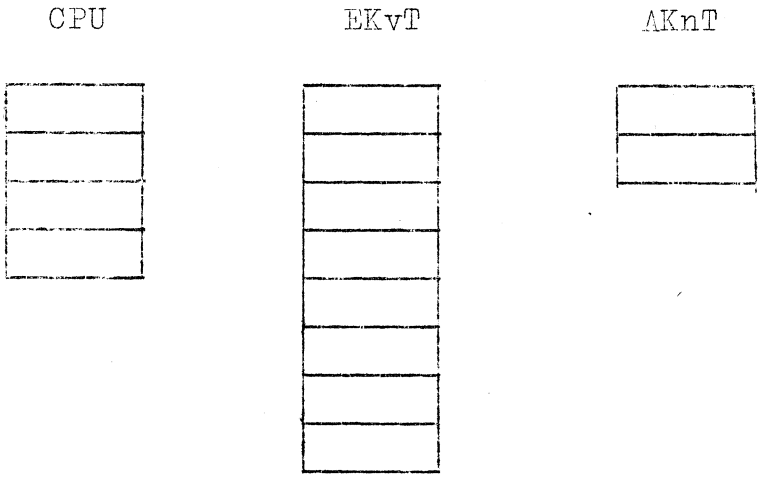
nur die ZEITLISTE nach dem Element mit der kleinsten Ereigniszeit durchmustert zu werden. Die dabei ermittelte Ereigniszeit wird als neue Zeit in die Variable UHR eingetragen und die entsprechenden Simulationsmaßnahmen werden getroffen, wie z.B. die Simulation des nächsten Datentransfers einzuleiten, wenn dann gerade ein Datenkanal frei geworden ist.

Als Elemente (entities) des Simulationssystems treten daneben auch Dinge auf, wie z.B. ein Programmstück, oder ein Datensatz, denen zwar keine Ereigniszeit zugeschrieben wird, wohl aber Eigenschaften wie Länge, Lage auf der Trommel, und bei geeignet dimensionierten Programmstücken der Zeitbedarf bei Ausführung in der CPU

> Elemente ohne Ereigniszeit (z.B. Programmstückchen, Datensätze, etc.)

Ein einfacher Ablauf innerhalb des Simulationssystems ließe sich nun folgendermaßen vorstellen.

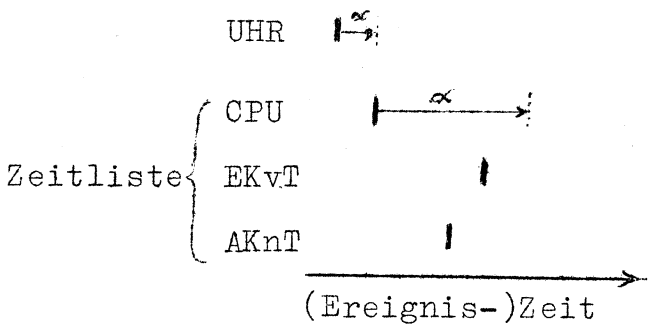
Zur CPU und den Datenübertragungskanälen EKvT, AKnT gehöre jeweils eine Warteschlange der als nächstes zu bearbeitenden Elemente (Programmstücke, Daten etc.).



Nun erfolgt:

α) Durchmustern der ZEITLISTE

In der Zeitliste sei das Element mit der kleinsten Ereigniszeit die CPU. Diese Zeit wird als neuer Zeitpunkt im simulierten System, bis zu dem die Simulation bereits fortgeschritten ist, in die Variable UHR eingetragen.



Zu diesem Zeitpunkt also wird die CPU frei für einen neuen Auftrag. Man holt also das nächste Element aus der CPU-Warteschlange, wobei es aus dieser entfernt wird. Sein zu erwartender CPU-Zeitbedarf (Eigenschaft dieses Programmelements) wird mit der UHR-Zeit addiert als neue CPU-Ereigniszeit eingetragen (vgl. Abb.). Falls das Programmelement zu einer Verwaltungsroutine gehört, so könnte man diesen Zeitbedarf noch in die Variable hinein addieren, in der der CPU-Verwaltungszeitbedarf berechnet werden soll.

Das Programmstück verlange nun z.B., daß ein anderes Programmelement von der Trommel geholt und anschließend gerechnet werden soll. Dann muß also das neu verlangte Programmelement noch in die EKvT-Warteschlange eingetragen werden.

Nun ist alles getan, zu dem die CPU-Freigabe Anlaß gab und es erfolgt erneut

β) Durchmustern der ZEITLISTE.

Jetzt findet man in der Zeitliste z.B. AKnT als Element mit kleinster Ereigniszeit, die wieder nach UHR gebracht wird. Nun kann das nächste Element aus der zu AKnT gehörigen Warteschlange auf die Trommel gebracht werden.

Aus den Eigenschaften des zu übertragenden Elements (Länge, Trommellage) in Verbindung mit der momentanen Trommelstellung (abhängig von UHR) wird die Übertragungszeit berechnet und auf die AKnT-Ereigniszeit addiert, die dann also den Zeit-

punkt angibt, zu dem der Übertragungsvorgang beendet sein wird und AKnT wieder frei für eine weitere Übertragung wird. Das zu übertragende Element wird natürlich nun aus der Warteschlange entfernt.

Nun ist alles getan, zu dem die AKnT-Freigabe Anlaß gab und es erfolgt erneut

γ) Durchmustern der ZEITLISTE

Jetzt findet man in der Zeitliste z.B. EKvT als Element mit kleinster Ereigniszeit, die wieder nach UHR gebracht wird.

Das Element, dessen Übertragung in den Kernspeicher nun beendet ist, stehe noch als erstes Element der EKvT-Warteschlange zur Verfügung. Seinen Eigenschaften entnehme man z.B., daß es ein Programmstück war, das nun gerechnet werden soll. Dann hat man es nun also in die CPU-Warteschlange einzutragen und aus der EKvT-Warteschlange zu entfernen.

Nun hat man die Simulation der Übertragung des nächsten Elementes der EKvT-Warteschlange in Angriff zu nehmen. Aus seinen Eigenschaften entnehme man die zur Berechnung der Übertragungszeit benötigten Daten (Länge, Trommellage), addiere diese auf die EKvT-Ereigniszeit, belasse das Element aber noch zur weiteren Bearbeitung nach Ankunft im Kernspeicher in der EKvT-Warteschlange.

Nun ist alles getan, zu dem die EKvT-Freigabe Anlaß gab und es folgt erneut

Durchmustern der ZEITLISTE etc.

Zusammenfassend können wir also sagen: Es werden ELEMENTE eingeführt, denen gewisse EIGENSCHAFTEN zugeschrieben werden. Diese Elemente werden in verschiedene LISTEN (Warteschlangen, Zeitliste) eingetragen, wobei die ZEITLISTE der Elemente mit Ereigniszeit wegen deren häufiger vollständiger Durchmusterung eine gewisse Sonderrolle spielt. Aus allem ergibt sich, daß eine Simulationssprache also jedenfalls derartige Listenverarbeitung erleichtern sollte.

c2) Wahrscheinlichkeitsverteilungen

Bei der Simulation eines Betriebssystems muß man an verschiedenen Stellen aus einer gegebenen Anzahl von Dingen, denen jeweils eine Häufigkeit ihres Auftretens zugeordnet ist, nach Zufallsgesetzen eins auswählen.

Beispielsweise wurde die Job-Reihenfolge bei gegebenem Job-mix (= Häufigkeitsverteilung für die einzelnen Jobs) so bestimmt. Will man die Trommelstellung nicht jeweils mitrechnen, so kann man die Trommelzugriffszeit ebenso statistisch einsetzen. Ferner ist meist die Multiplikation etc. Operandenabhängig; nachdem aber bei der Simulation die Operanden im allgemeinen nicht bekannt sind, kann man auch hier statistische Werte annehmen.

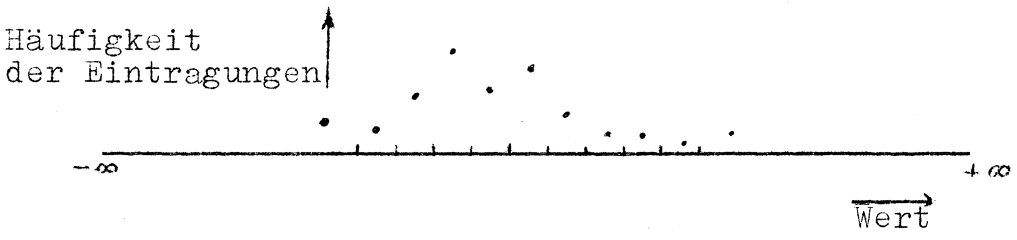
Daher muß eine Simulationssprache Möglichkeiten bieten, bequem Wahrscheinlichkeitsverteilungen vorzugeben und danach eine statistische Auswahl vorzunehmen.

c3) Histogramme

Bei der Arbeit mit dem Simulationssystem für ein Betriebssystem haben wir wichtige Aufschlüsse aus Angaben über die Länge der Warteschlangen etc. gewonnen. Man hat also zu verschieden Zeiten jeweils eine derartige Größe zu kontrollieren und die dabei festgestellten Werte zu sammeln.

Die Sammlung der Werte einer Größe zu verschiedenen Zeitpunkten erfolgt jeweils in einem dieser Größe zugeordneten Histogramm, wobei eine Intervalleinteilung des infrage kommenden Wertebereiches vorgegeben ist, und bei "Eintragung" eines Wertes in dieses Histogramm lediglich festgestellt wird, in welchem dieser Intervalle der Wert liegt. Nun wird mitgezählt, wieviele Eintragungen jeweils für die einzelnen Intervalle erfolgt sind.

Als Ergebnis liefert das Histogramm daher eine Häufigkeitsverteilung für die Eintragungungen über die Intervalle des Histogramms.



Im folgenden betrachten wir nun eine besonders einfache unter den allgemeinen Simulationssprachen im einzelnen, um einen Überblick über die wichtigsten für Simulationsprobleme bereitzustellenden Hilfsmittel zu gewinnen.

3. Allgemeine Simulationssprachen

Die allgemeinen Simulationssprachen SIMON und CSL, auf die wir im folgenden näher eingehen wollen, sind auf der ATLAS-Anlage sowie für die ICT 1900-Serie implementiert und stellen Erweiterungen eingeführter Programmiersprachen wie ALGOL bzw. FORTRAN dar. (vgl I.C.T. CONTROL AND SIMULATION MANUAL.)

3.1 SIMON

(P.R. HILLS 1964/65 Bristol College of Science and Technology)

Die Simulationssprache SIMON besteht aus normalem ALGOL 60, für das geeignete CODE-Prozeduren bereit gestellt werden, die den speziellen Bedürfnissen der Simulation Rechnung tragen. Und zwar handelt es sich um folgende CODE-Prozeduren.

a) Elemente (entities)

ENTITY (ent, kennr) ist eine eigentliche Prozedur und ist erklärt für ein Feld ent und einen Ausdruck kennr vom Typ integer

integer array ent [o:n]

Der Aufruf dieser Prozedur führt ent [o] als "Element" ein und ent [1], ent [2], ..., ent [n] als dessen n "Eigenschaften". Repräsentiere ent [o] also z.B. ein Schiff, so könnte ent [1] also z.B. in geeigneter ganzzahliger Verschlüsselung angeben, ob es im Hafen, auf See, oder untergegangen ist, ent [3] seine Tonnage wiedergeben, etc.

Ferner wird dem Element ent [o] die durch kennr bestimmte ganze Zahl zugeordnet, die mit Hilfe der später zu behandelnden Prozedur REFNUM abgefragt werden kann.

GROUP ENTITY (gent, anz, kennr) ist eine eigentliche Prozedur und ist erklärt für ein Feld gent und Ausdrücke anz, kennr vom Typ integer

integer array gent [1:anz, o:n]

Der Aufruf dieser Prozedur führt die Größen gent [1, o], gent[2, o], ... gent [anz, o] jeweils als Element ein, wobei jeweils zum Element

gent [i, o] ($1 \leq i \leq$ anz) die Eigenschaften
gent [i, k] ($1 \leq k \leq n$)

gehören. Man sagt, die Elemente g [i, o] gehören zur "Elementgruppe" gent.

Allen Elementen dieser Elementgruppe wird ferner die gleich durch kennr bestimmte Zahl zugeordnet (die wieder mittels REFNUM abgefragt werden kann).

b) Listen

Die folgenden Prozeduren dienen der Listenverarbeitung, wobei nach a) eingeführte "Elemente" als Listenelemente in Frage kommen.

SET (set) ist eine eigentliche Prozedur, ist erklärt für eine integer Variable set und führt diese als Liste (bzw. Warteschlange) ein.

ADD FIRST (ent) TO: (set)
ADD LAST (ent) TO: (set)

ist jeweils eine eigentliche Prozedur, ist erklärt für ein Element (entity) ent und eine Liste set und fügt das Element ent als erstes bzw. letztes Element in die Liste set ein.

BEHEAD (set)
BETAIL (set)

ist jeweils eine eigentliche Prozedur, ist erklärt für eine Liste set und entfernt das erste bzw. letzte Element aus der Liste set.

DELETE (ent) FROM: (set)
DELETE (set1) FROM: (set2)

ist jeweils eine eigentliche Prozedur, ist erklärt für ein Element ent und eine Liste set bzw. zwei Listen set1 und set 2 und bewirkt das Entfernen des Elements ent bzw. der Elemente die auch set1 angehören aus set bzw. set2.

ROTATE (set, n)

ist eine eigentliche Prozedur, ist erklärt für eine Liste set und einen Ausdruck n vom Typ integer und bewirkt, daß n mal jeweils das erste Element aus set vorn weggenommen und hinten wieder angefügt wird.

HEAD OF (set)
TAIL OF (set)

ist jeweils eine Funktionsprozedur vom Typ integer, ist erklärt für eine Liste set und liefert als Funktionswert das erste bzw. letzte Element der Liste set, die selbst unverändert bleibt.

SIZE OF (set)

ist eine Funktionsprozedur vom Typ integer, ist erklärt für eine Liste set und liefert als Funktionswert die Anzahl der Elemente in der Liste set

MEMNUM (gent)

ist eine Funktionsprozedur vom Typ integer, ist erklärt für ein Element gent aus einer Elementgruppe und liefert als Funktionswert den Gruppenindex des Elements gent, d.h. den ersten index, der das Element innerhalb der Elementgruppe bestimmt.

Das ist von Bedeutung, wenn gent nicht direkt, sondern zB. implizite durch HEAD OF (set) gegeben ist.

REFNUM (ent) ist eine Funktionsprozedur vom Typ integer, ist erklärt für ein Element ent und liefert als Wert die Kennummer des Elements ent, die diesem mittels ENTITY oder GROUPENTITY zugeordnet wurde.

Das ist von Bedeutung, wenn ent implizite gegeben ist. Oft verwendet man dann die Kennnummer für den Sprung in einem SWITCH.

c) Ereigniszeit

Besitzt ein Element als Eigenschaft eine Ereigniszeit, so spielt diese gegenüber anderen Eigenschaften eine Sonderrolle, vermutlich auch wegen der Notwendigkeit, die Zeitliste häufig nach der kleinsten Ereigniszeit zu durchmustern. Das äußert sich in SIMON darin, daß die Ereigniszeit nicht direkt zugänglich ist, sondern nur über bestimmte Prozeduren.

SETTIME (ent) TO: (Wert) ist eine eigentliche Prozedur, ist erklärt für ein Element ent und einen Ausdruck Wert vom Typ integer, und weist Wert dem Element ent als Ereigniszeit zu.

TIMEVALUE (ent) ist eine Funktionsprozedur vom Typ integer, ist erklärt für ein Element ent und liefert als Funktionswert die Ereigniszeit des Elements ent.

SCAN (set) FOR: (member) WITH: (leasttime) ist eine eigentliche Prozedur und ist erklärt für eine Liste set und zwei integer Variablen member und leasttime als Resultatparameter. Alle Elemente der Liste set müssen Ereigniszeiten zugewiesen bekommen haben.
Dann wird die Liste set nach dem Element mit kleinster Ereigniszeit durchmustert; der Wert dieser kleinsten Ereigniszeit wird der Variablen leasttime und das betreffende Element selbst der Variablen member zugewiesen.

d) Zufallsverteilungen

Die folgenden Prozeduren dienen der Erzeugung von Zufallszahlen und der Eingabe und Berücksichtigung vorgegebener Häufigkeitsverteilungen.

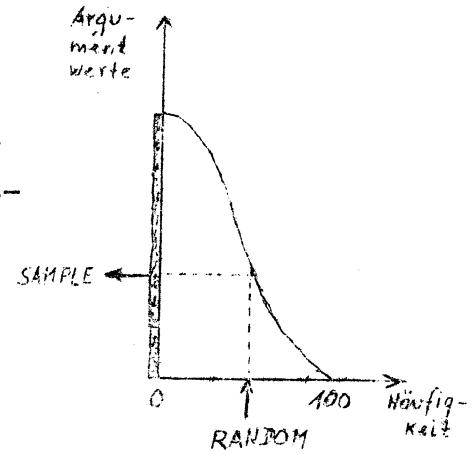
RANDOM (n) ist eine Funktionsprozedur vom Typ integer, ist erklärt für einen Ausdruck n vom Typ integer im Bereich $0 \leq n \leq 10$ und liefert als Funktionswert eine Zufallszahl aus dem (gleichverteilten) Bereich 0(1)99 der ganzen Zahlen.
Falls $1 \leq n \leq 10$, so wird die Zufallszahl aus einem der zehn mit dem Compiler gegebenen Zufallsgeneratoren Nr n entnommen. Verschiedene Generatoren sind zweckmäßig, wenn gegenseitige Unabhängigkeit von Zufallszahlen sichergestellt werden soll.

Falls n = o, so wird beim Aufruf von RANDOM die nächste Zahl vom Eingabemedium eingelesen und als Zufallswert geliefert.

DISTRI (dname, Kennz) ist eine eigentliche Prozedur, und ist erklärt für einen Ausdruck vom Typ integer, dessen Wert im Bereich $0 \leq \text{Kennz} \leq 10$ liegt, und ein Feld integer array dname [0 : 20].

Beim Aufruf dieser Prozedur wird der Name dname des Feldes als Häufigkeitsverteilung eingeführt und dieser die Kennzahl Kennz zugeordnet, die zu Kontrollzwecken beim Einlesen verwendet wird und außerdem bei der Berücksichtigung dieser Häufigkeitsverteilung in der Prozedur SAMPLE verwendet wird.
Außerdem werden beim Aufruf von DISTRI 21 Zahlen vom Eingabemedium gelesen, die die betreffende Häufigkeitsverteilung festlegen. Die erste dieser Zahlen muß

mit der Kennzahl Kennz übereinstimmen, um Irrtümer in der Reihenfolge auf dem Datenstreifen auszuschalten. Die folgenden zehn Zahlenpaare enthalten jeweils zu einem Argumentwert (aus dem Bereich 0(1)99) die zugehörige Häufigkeit als Summenkurve, so daß der erste dieser Häufigkeitswerte 0, der letzte 100 sein muß. Bei der Auswertung dieser Häufigkeitsverteilung durch SAMPLE wird zunächst mittels RANDOM ein gleichverteilter Zufallswert aus 0(1)99 ermittelt, festgestellt zu welchem Argumentwert dieser Wert als Häufigkeit in der Häufigkeitsverteilung gehört und dieser Argumentwert schließlich als Zufallswert hergenommen.



Wenn man die Häufigkeiten von Zahlenpaar zu Zahlenpaar linear in Schritten von je 11 anwachsen läßt, so wird jeder eingetragene Argumentwert im Mittel in $^1/_{10}$ der Fälle auftreten. Der gleiche Argumentwert darf auch mehrfach eingetragen sein, so daß er entsprechend häufiger auftreten wird.

SAMPLE (dname)

ist eine Funktionsprozedur vom Typ integer, ist erklärt für eine Häufigkeitsverteilung dname, und liefert als Funktionswert einen Zufallswert im Bereich 0(1)99 unter Berücksichtigung der Häufigkeitsverteilung dname. Hierfür wird als Hilfsprozedur intern RANDOM aufgerufen und zwar mit der Kennzahl der Häufigkeitsverteilung dname als Argument.

e) Histogramme

HISTOGRAM (hname, untergrenze, intervall) ist eine eigentliche Prozedur und ist erklärt für Ausdrücke untergrenze, intervall vom Typ integer und ein Feld integer array hname [0 : 20].

Damit wird hname als Histogramm eingeführt. Dieses Histogramm besitzt 11 Intervalle, auf die die Eintragungen verteilt werden. Das erste erstreckt sich von $-\infty$ bis untergrenze, daran anschließend folgen 9 Intervalle der Breite intervall, das letzte erstreckt sich daran anschließend bis nach $+\infty$. Dabei wird jeweils die linke (untere) Intervallbegrenzung mit zum Intervall gerechnet.

ADDTO (hname, wert) ist eine eigentliche Prozedur, und ist erklärt für einen Ausdruck wert vom Typ integer sowie ein Histogramm hname.

Beim Aufruf dieser Prozedur wird wert in das Histogramm "eingetragen", indem die dort gegebene Häufigkeit der Eintragungen im (dem Wert entsprechenden) Intervall um 1 erhöht wird.

WRITEDOWN (hname, string) ist eine eigentliche Prozedur, die erklärt ist für ein Histogramm hname und einen String "string". Beim Aufruf dieser Prozedur wird auf dem Ausgabemedium die im Histogramm akkumulierte Häufigkeitsverteilung der Eintragungen in die verschiedenen Intervalle unter Voranstellung des string als Überschrift in folgender Form gedruckt

string
g_1 g_2 g_3 g_4 g_5 g_6 g_7 g_8 g_9 g_{10}
n_0 n_1 n_2 n_3 n_4 n_5 n_6 n_7 n_8 n_9 n_{10}
MEAN α
VARIANCE β

Dabei sind die g_i die Intervallgrenzen und n_i ist jeweils die Häufigkeit der Eintragungen in das mit g_i beginnende Intervall ($g_0 = -\infty$). Außerdem wird noch der Mittelwert α und die Streuung β ausgedruckt.

f) Programmstruktur

Vor dem Aufruf der oben beschriebenen SIMON-Prozeduren muß als erstes die CODE-Prozedur

PLANTMASTER

aufgerufen werden, die die internen Verweisungslisten (Masterlists) bereitstellt, in denen dann Elemente und Listen eingetragen werden können.

Der Datenstreifen für ein SIMON-Programm muß stets mit einer ganzen Zahl SSS beginnen, die der Benutzer aus

der Gesamtzahl E der im Programm verwendeten Elemente
der Zahl S der Listen
der Summe M aus den maximal in den Listen jeweils auftreten den Elemente

nach

$$SSS = 2E + S + M$$

errechnen muß und aus der der für die Verweisungslisten benötigte Speicherplatz berechnet werden kann.

Nach den einmaligen Voreinstellungen der Parameter läßt sich der Wiederholungsteil eines Simulationsprogramms im allgemeinen in drei Phasen zerlegen:

A Durchsuchen der Zeitliste nach dem Element mit kleinster Ereigniszeit und fortschalten der Simulationszeit auf den gefundenen Wert.

B Vornahme solcher Zustandsänderungen im Simulationssystem, die direkte Folge der Zeitfortschaltung sind.

C Vornahme weiterer Zustandsänderungen, die sich zunächst aus den Zustandsänderungen der Phase B ergeben. Diese können ihrerseits weitere Zustandsänderungen nach sich ziehen, die durch wiederholte Iteration der Phase C vorgenommen werden, bis schließlich die Iteration der Phase C stationär wird. Dann wird wieder mit Phase A begonnen.

Das Programm endet schließlich mit dem Ausdrucken der gefundenen Resultate. Wir erläutern dies im folgenden Abschnitt an einem Beispiel.

g) Beispiel: Friseurladen mit zwei Friseuren

Den Umgang mit dieser Simulationssprache wollen wir nun am Problem der Simulation eines Friseurladens mit zwei Friseuren demonstrieren.

Was für Ereignisse haben wir nun dabei zu erwarten?

g1) Neuer Kunde kommt

Bei der Durchmusterung der Zeitliste hat sich also als Element mit kleinster Ereigniszeit ein Element NEUER KUNDE[o] ergeben und die Simulationszeit (UHR) hat die Ereigniszeit seiner erwarteten Ankunft erreicht. Das Element NEUER KUNDE[o] muß nun also in die Liste WARTESCHLANGE hinten eingetragen und aus der Zeitliste entfernt werden.

Aus statistischen Gegebenheiten entnehmen wir nun die zu erwartende Ankunftszeit eines NÄCHSTEN KUNDEN[o] und reihen diesen als Element (mit der betreffenden Ereigniszeit versehen) in die Zeitliste ein.

An den Kunden selbst interessieren uns in diesem Modell weitere, die Kunden unterscheidende Eigenschaften nicht. Außerdem ist es möglich, das gleiche Element mehrfach in eine Warteschlange einzutragen. Daher werden wir nicht jedem Kunden ein besonderes Element zubilligen, sondern nur ein Element KUNDE [o] einführen und mit der Ankunftszeit des nächsten Kunden als Ereigniszeit versehen.

Dann brauchen wir dabei nicht Streichung und Neueintrag in der Zeitliste vornehmen, sondern alles was zu tun bleibt ist

α) das Element KUNDE [o] in die Liste WARTESCHLANGE eintragen

β) dem Element KUNDE [o] eine neue Ereigniszeit zuweisen.

Die Berechnung der neuen Ereigniszeit entsprechend β) erfolgt statistisch. Wir geben also mittels DISTRI nebenstehende Häufigkeitsverteilung für die Ankunftszeit-intervalle der Kunden ein, nennen sie ANKUNFT und ordnen ihr z.B. die Kennzahl 1 zu.

```
  1
0 ; 0
0 ; 11
1 ; 22
1 ; 33
1 ; 44
2 ; 55
2 ; 66
3 ; 77
3 ; 88
4 ; 100
```

Wir können nun das betreffende Programmstück schreiben, berücksichtigen aber noch, daß ein Kunde unverrichteter Dinge wieder fortgeht, wenn die Warteschlange bei seiner Ankunft mehr als 3 Kunden enthält.

Dann hat man also:

```
if SIZEOF (WARTESCHLANGE) greater 3
      then WRITE ('('KUNDE GEHT WIEDER')')
      else ADDLAST (KUNDE [o], WARTESCHLANGE);
SETTIME (KUNDE [o], SAMPLE (ANKUNFT) + UHR)
```

Das setzt voraus, daß eingangs die Verteilung ANKUNFT mittels DISTRI(ANKUNFT, 1) eingelesen wurde, daß KUNDE [o] z.B. mittels ENTITY(KUNDE, 1) als Element und WARTESCHLANGE

mittels SET(WARTESCHLANGE) als Liste eingeführt wurde, und weiter, daß vorher die Vereinbarungen

```
integer WARTESCHLANGE, UHR;
integer array KUNDE [0:0], ANKUNFT [0:20]
```

getroffen wurden.

g2) Friseur A bzw. B ist fertig

Wenn das Ereignis eintritt, daß z.B. Friseur A fertig ist, so hat also A FRISEUR[0] als Element (mit kleinster Ereigniszeit) in der Zeitliste gestanden, und die Simulationszeit UHR hat soeben die Ereigniszeit seines erwarteten Fertigwerdens erreicht. Wir haben ihn dann in eine Liste FREIE FRISEURE einzutragen und aus der Zeitliste zu entfernen.

Das analoge Vorgehen findet statt, wenn Friseur B fertig geworden ist.

g3) Listenausgleich zwischen WARTESCHLANGE und FREIE FRISEURE

Während die unter g1) und g2) gegebenen Ereignisse nach der Einteilung von F zur Phase B des Simulationsprogrammes gehören, in der solche Zustandsänderungen vorgenommen werden, die direkte Folge der Zeitfortschaltung sind, gehört die folgende Zustandsänderung zur Phase C_1 in der Zustandsänderungen behandelt werden, die als Folge von anderen Zustandsänderungen notwendig werden.

Falls nämlich als Folge von Phase B die Listen FREIE FRISEURE und WARTESCHLANGE beide mit Elementen besetzt sind, so muß nun der WARTESCHLANGE der nächste Kunde entnommen werden und dem nächsten Friseur aus der Liste FREIE FRISEURE zugeführt werden, wobei dieser aus der Liste entfernt werden, eine neue Ereigniszeit seines erwarteten Fertigwerdens zugewiesen erhalten und in die ZEITLISTE eingetragen werden muß.

Falls dann immer noch beide Listen FREIE FRISEURE und WARTESCHLANGE besetzt sind, muß der Vorgang nochmals wiederholt werden.

Zur Phase B des Simulationsprogramms (Ziffern g1 und g2) ist vielleicht noch nachzutragen, was geschieht, wenn zwei Elemente der ZEITLISTE die gleiche Ereigniszeit besitzen. Die Durchmusterung der ZEITLISTE wird dann, wenn diese Ereigniszeit die kleinste ist, nur eins dieser Elemente liefern, das andere bleibt in der Zeitliste stehen. Wenn nun die Verarbeitung des ersten dieser Elemente abgeschlossen ist, so wird die nächste Durchmusterung der Zeitliste eben nochmals die gleiche kleinste Ereigniszeit mit dem anderen zugehörigen Element liefern und UHR erhält nochmals den gleichen Wert zugewiesen.

g4) Das Simulationsprogramm

Im folgenden geben wir nun das, abgesehen vom äußersten Block, in dem die Codeprozeduren vereinbart werden, vollständige Simulationsprogramm für den Friseurladen mit zwei Friseuren wieder.

```
comment  die vorangehenden Codeprozedurvereinbarungen haben
         wir nicht aufgeführt. Daran schließt sich das folgende
         Programm;

PLANTMASTER;
begin comment Vereinbarungen;
        integer UHR, ZEITPUNKT, s, ELEMENT, r,
                WARTESCHLANGE, FREIE FRISEURE, ZEITLISTE;
        integer array KUNDE, A FRISEUR, B FRISEUR [o:o],
                      ANKUNFT, A FRISIERZEIT, B FRISIER-
                      ZEIT, WARTENDE [o:2o];

    comment Uberschrift für die Resultate;
    WRITE ('('<= UHRZEIT   WARTESCHLANGENLÄNGE <==')');
    comment Definition von Elementen  Listen, Histogramm
            und Verteilungen;
            ENTITY (KUNDE,1);
            ENTITY (A FRISEUR,2);
            ENTITY (B FRISEUR,3);
```

```
SET (FREIE FRISEURE);
SET (WARTESCHLANGE);
SET (ZEITLISTE);

DISTRI (ANKUNFT, 1),
DISTRI (A FRISIERZEIT, 2);
DISTRI (B FRISIERZEIT, 3);

HISTOGRAM (WARTENDE, 1, 1);

comment  Anfangsvorbelegung;

ADDFIRST (B FRISEUR[o], FREIE FRISEURE);
ADDFIRST (A FRISEUR[o], FREIE FRISEURE);
SETTIME (KUNDE[o], SAMPLE (ANKUNFT));
ADDFIRST (KUNDE[o], ZEITLISTE);
UHR := o;

comment  Beginn des Simulationszyklus;

phase A: SCAN(ZEITLISTE)FOR:(ELEMENT)WITH:(ZEITPUNKT);

phase B: UHR:=ZEITPUNKT;

         if ELEMENT equal KUNDE[o]
            then begin if SIZEOF(WARTESCHLANGE) less 4
                          then ADDLAST(KUNDE[o], WARTESCHLANGE)
                          else WRITE('('< = KUNDE GEHT WIEDER< = ')');
                       SETTIME(KUNDE[o], SAMPLE(ANKUNFT)+UHR);
                       goto phase C
                 end
          else begin   ADDFIRST(ELEMENT, FREIE FRISEURE);
                       DELETE(ELEMENT, ZEITLISTE);
                       goto phase C
                 end;

phase C: if SIZEOF(WARTESCHLANGE) greater o
            and SIZEOF(FREIE FRISEURE) greater o
            then begin BEHEAD(WARTESCHLANGE);
                       r:=HEADOF(FREIE FRISEURE);
                       if requal AFRISEUR[o]
                         then s:=SAMPLE (AFRISIERZEIT)
                         else s:=SAMPLE (BFRISIERZEIT);
```

```
                    SETTIME (r, s+UHR);
                    ADDFIRST (r, ZEITLISTE);
                    goto phase C
                end;
comment Nun ist Phase C fertig. Es folgt noch Kontrolldruck, Histo-
        grammeintragung und Entscheidung über Fortsetzung der
        Simulation;
PRINT (UHR, SIZEOF(WARTESCHLANGE));
ADDTO (WARTENDE, SIZEOF(WARTESCHLANGE));
if UHR less 34 then goto phase A;
WRITEDOWN(WARTENDE, '('HISTOGRAMM DER WARTESCHLANGE')');
end
```

g5) Die Eingabedaten zu diesem Programm

Gemäß f haben wir zunächst SSS zu errechnen. In diesem Programm gibt es E=3 Elemente und S=3 Listen. Die maximale Elementzahl je Liste ist für die ZEITLISTE gleich 3, für die Liste FREIE FRISEURE gleich 2 und für die WARTESCHLANGE gleich 4, so daß sich M=9 und schließlich SSS=18 ergibt. Der Datenstreifen hat dann der Reihe nach SSS und die Verteilungen ANKUNFT, AFRISIERZEIT, BFRISIERZEIT zu enthalten und lautet

```
18;
 1;  0,  0;  0, 11;  1, 22;  1, 33;  1,  44;
     2, 55;  2, 66;  3, 77;  3, 88;  4, 100;
 2;  5,  0;  8, 11;  9, 22;  9, 33; 10,  44;
    10, 55; 10, 66; 10, 77; 15, 88; 18, 100;
 3;  6,  0;  6, 11;  7, 22;  9, 33; 10,  44;
    10, 55; 11, 66; 12, 77; 13, 88; 20, 100;
```

g6) Die Resultate dieses Programms

UHRZEIT	WARTESCHLANGENLÄNGE
0	0
1	1
4	2
5	1
6	0
7	1
7	2
9	3
10	4
11	3
12	2
12	3
15	4
KUNDE GEHT WIEDER	
19	4
19	3
19	2
20	3
23	4
KUNDE GEHT WIEDER	
23	4
KUNDE GEHT WIEDER	
23	4
KUNDE GEHT WIEDER	
24	4
KUNDE GEHT WIEDER	
26	4
29	3
29	4
31	3
33	4
KUNDE GEHT WIEDER	
34	4

HISTOGRAMM DER WARTESCHLANGE

1	2	3	4	5	6	7	8	9	10	11
3	3	4	7	11	0	0	0	0	0	0	0

MEAN 2.714
VARIANCE 1.847

3.2. MOBULA

Über die Simulationssprache MOBULA liegt mir ein Bericht von März/Juni 67 vor, der keine Angabe des Verfassers oder Publikationsortes enthält, noch einen Hinweis darauf, ob diese Sprache irgendwo implementiert ist.

Ich möchte diese Sprache im Anschluß an SIMON kurz streifen, da es sich hierbei ebenfalls um eine Erweiterung von ALGOL handelt, die allerdings sehr viel weiter geht wie SIMON und ganz neue Sprachelemente einführt. Mein Ziel hierbei ist, einen ganz oberflächlichen Eindruck daran zu vermitteln, in welche Richtung diese Erweiterungen gehen. Mir selbst scheint MOBULA von der logischen Struktur her sehr viel befriedigender aufgebaut zu sein als SIMON, ist aber eine eigene, relativ komplizierte Sprache.

Beispiel: Trifft man in MOBULA z.B. die folgenden Vereinbarungen

`class soldiers[1500];` hierdurch werden die Elemente soldiers[1] bis soldiers[1500] sowie die Klasse soldiers eingeführt.

`set soldiers: married, sick;` hierdurch werden Listen married und sick eingeführt, die mit Elementen soldiers[i] besetzt werden dürfen.

```
feature income;
integer marriage allowance;
name    man;
```

so kann man später etwa folgende Laufanweisung schreiben

```
for man over married unless man in sick do
    begin  income(man):=income(man)+marriage allowance;
```

Beispiel: Die Syntax ist in Backus-Notation beschrieben, wofür folgende Beispiele gegeben seien:

⟨member⟩ :=⟨class⟩/⟨set⟩/⟨queue⟩/⟨entity⟩/null
⟨index expression⟩:= ⟨arithmetic expression⟩/head/tail

Beispiel: Daß MOBULA neben größeren Möglichkeiten zur Formulierung auch recht unübersichtliche Notationen bietet, entnehme man diesem Beispiel. Trifft man nämlich die folgenden Vereinbarungen

```
class soldiers[1500];
```
eine Klasse soldiers und Elemente soldiers[1] bis soldiers [1500] werden hierdurch eingeführt.

```
set class soldiers: platoons[36];
```
eine Klasse platoons und Listen platoons[1] bis platoons[36] werden hierdurch eingeführt, wobei die Listen mit Elementen soldiers[i] zu besetzen sind.

```
set class platoons: squads[12];
```
eine Klasse squads und Listen squads[1] bis squads[12] werden hierdurch eingeführt, wobei die Listen für die Listenelemente platoons[i] zugelassen sind.

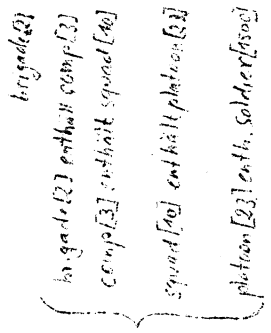
```
set class squads: companies[4];

set class companies: brigades[3];
```

dann kann später im Programm die folgende Größe hingeschrieben werden

Brigades [[[[[2]3]10]23]1500]

die entweder den "Wert" soldier[1500] oder den Wert null (Wortsymbol) hat, je nachdem ob die mit den eckigen Klammern implizierten Bedingungen erfüllt sind oder nicht. Diese Bedingungen veranschaulichen wir an folgendem Bild:

Brigades [[[[[2]3]10]23]1500]



Bedingungen

Die angeführten Beispiele mögen genügen, um einen flüchtigen Eindruck von der Art dieser Sprache zu vermitteln, und wir wollen nicht tiefer eindringen.

3.3 CSL

Die Simulationssprache CSL wurde entwickelt von IBM United Kingdom Ltd. zusammen mit ESSO PETROLEUM Comp. und ist implementiert für die ICT-1900Serie und ATLAS. Die mir vorliegende Beschreibung stammt von ICT und ist im Oktober 1966 erschienen.

CSL ist eine Erweiterung von FORTRAN und der implementierte CSL-Compiler stellt FORTRAN-Programme her. Gegenüber FORTRAN besitzt CSL unter anderem eine große Zahl zusätzlicher reservierter Namen und viele weitere Besonderheiten wie z.B. zusammengesetzte Anweisungen, die durch gleichmäßiges Einrücken der betreffenden Teil-Anweisungen als zusammengehörig ausgewiesen werden müssen.

Wir haben früher SIMON behandelt als Beispiel einer besonders einfach strukturierten Simulationssprache, um an ihr die wesentlichen Begriffsbildungen und Grundoperationen und deren Zusammenwirken bei der Simulation an einem einfachen Beispiel kennen zu lernen. (Außerdem zeichnet sich SIMON dadurch aus, daß es allein durch ALGOL-Code Prozeduren realisiert werden kann.) Wir wollen nun auch auf CSL näher eingehen, da hier insbesondere die Zeitbehandlung anders vorgenommen ist als bei SIMON und mehr Grundoperationen für die Listenbehandlung bereitgestellt werden. Mein Ziel ist dabei, durch die Betrachtung der Unterschiede zwischen SIMON und CSL die Gemeinsamkeiten als Charakteristika der Simulation um so deutlicher hervortreten zu lassen und aufzuzeigen, wo die Erbauer von CSL hauptsächlich den Mehraufwand zu investieren für sinnvoll erachteten.

a) Zeitbehandlung und Programmablauf

ACTIVITIES Dieses statement kennzeichnet die Stelle des Programms, an der jeweils der Zeitzyklus beginnt (Phase B). Die Programmdurchführung erfolgt dann in der Regel jeweils bis zum letzten END, worauf automatisch die Gesamtheit aller Ereigniszeiten (Zeitzellen) durchmustert wird. die Zeitfortschaltung vorgenommen und bei ACTIVITIES wieder begonnen wird.

Die Ereigniszeiten der Elemente geben hier nicht wie bei SIMON Zeitpunkte der Simulationszeit an, sondern Zeitdifferenzen bis zum fällig werden. Die Zeitfortschaltung besteht dann in der Durchmusterung aller Zeitzellen nach dem kleinsten Wert $\geq$ o. Dieser Wert wird automatisch auf die Standardvariable CLOCK (=Simulationszeit) addiert und außerdem von allen Zeitzellen abgezogen, so daß momentan fällige Ereigniszeiten stets den Wert o haben. Das hat zur Folge, daß jeweils für die infrage kommenden Elemente abgefragt werden muß, ob die Ereignis-

zeit den Wert o hat, was durch Bereitstellung geeigneter Durchmusterungsroutinen ermöglicht wird.

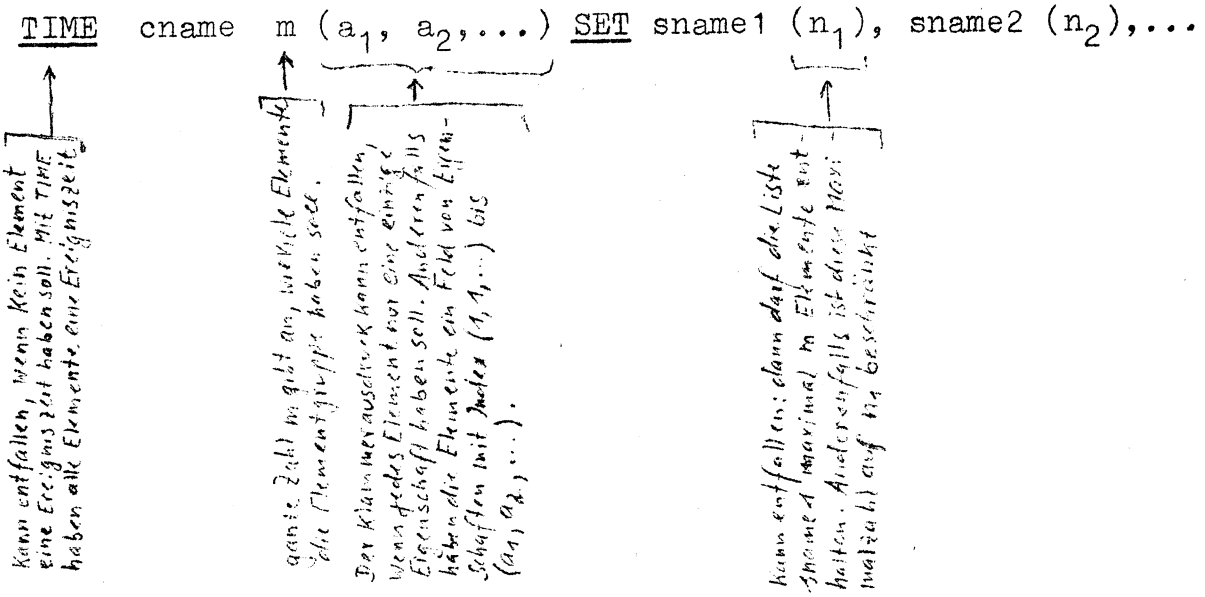
RECYCLE — Dies Statement bewirkt die Vormerkung, daß beim nächsten Zeitzyklus keine Zeitfortschaltung vorgenommen werden soll. Werden also beim durchlaufen des Zeitzyklus nach einer Zeitfortschaltung (Phase B) Zustandsänderungen vorgenommen, die ihrerseits weitere Zustandsänderungen nach sich ziehen, die infolge der Reihenfolge der Programmniederschrift Wiederholung des Zeitzyklus ohne Zeitfortschaltung notwendig machen (Phase C), so wird dies durch das Statement RECYCLE ermöglicht.

EXIT — Dies Statement wird erreicht, wenn die Simulation abgebrochen werden soll, und bewirkt das dynamische Programmende durch einen Sprung in das Betriebssystem.

b) Elementgruppen (classes) und Listen (sets)

In CSL werden nicht einzelne Elemente, sondern immer gleich ganze Elementgruppen (classes) eingeführt. Listen können allein in Verbindung mit einer Elementgruppe eingeführt werden und dürfen ausschließlich mit Elementen aus der zugehörigen Elementgruppe besetzt werden. Hierin und in der zusätzlichen Forderung, daß ein Element nicht mehrfach in eine Liste eingetragen werden darf, liegt eine starke Einschränkung gegenüber den Möglichkeiten von SIMON (vgl. WARTESCHLANGE im Friseur-Beispiel), die jedoch offenbar von den CSL-Erfindern nicht als wesentlich empfunden wurde, und der an anderer Stelle viel weitergehende Möglichkeiten gegenüber stehen.

Zur Einführung einer Elementgruppe und Listen für diese Elemente dient das statement



Mit diesem statement werden die Elemente cname.M mit $1 \leq M \leq m$ eingeführt, wobei M auch eine Variable sein darf. Jedes Element besitzt dann ein Feld von Eigenschaften cname.M (l_1, l_2, ...) mit $1 \leq l_i \leq a_i$. Falls in obigem statement das Wort TIME nicht entfällt, besitzt jedes Element außerdem als Eigenschaft eine Ereigniszeit T.cname.M, die in dieser Form unmittelbar in Ausdrücken und bei Wertzuweisungen zugänglich ist und allein an dem vorangestellten Buchstaben T (als Ereigniszeit) erkannt wird.

Die dem obigen statement entsprechenden Prozeduren in SIMON wären ENTITY (ent, kennr), GROUP ENTITY (gent, anz, kennr) und SET (set).

c) Einfache Listenmanipulationen

Einigen statements zur Listenmanipulation, wie z.B. das Eintragen eines Elements in eine Liste, läßt sich ein Wahrheitswert true bzw. false zuordnen, je nach dem ob die Anweisung ausführbar war oder nicht. Beispielsweise ist die Anweisung zum Eintrag eines Elements in eine Liste dann erfolglos, wenn das

Element schon in der Liste enthalten ist. Daher haben solche statements gleichzeitig den Charakter von Bedingungen, deren Wahrheitswert mittels alternativer Sprungangaben abgefragt werden kann (Erfolgsprüfung).

Diese Sprungangabe kann entfallen. Dann werden automatisch geeignete Sprungziele eingesetzt und zwar im Erfolgsfall zur nächsten Anweisung, im Mißerfolgsfall zum nächsten BEGIN-statement (bzw. zum Ende des Zeitzyklus falls kein weiteres BEGIN folgt). Solche BEGIN statements können zu diesem Zweck ins Programm eingestreut werden und gliedern die Folge der Anweisungen des Programms in Teilfolgen, die man ihrerseits als Aktivitäten (activities) bezeichnet.

Eine solche Sprungangabe (zur Erfolgsprüfung) hat die Gestalt

$$m_1 @ m_2$$

und bewirkt bei Erfolg den Sprung zur Marke m_1, andernfalls zur Marke m_2. Entweder m_1 oder m_2 oder die ganze Sprungangabe kann entfallen. Fehlt m_1, so wird automatisch statt dessen als Sprungziel das nächste statement eingesetzt; fehlt m_2, so wird automatisch statt dessen als Sprungziel das nächste BEGIN-statement bzw. das letzte END eingesetzt.

Neben den später zu behandelnden komplizierteren (FOR, FIND, TESTCHAIN-Compound) statements stehen folgende einfacheren statements zur Listenmanipulation zur Verfügung.

ent HEAD set $m_1 @ m_2$
ent TAIL set $m_1 @ m_2$

bewirkt das bedingte Eintragen des Elements ent am Anfang bzw. am Ende der Liste set. Bedingung: ent ist nicht schon in set eingetragen. Wie bereits oben generell vorausgeschickt, müssen ent und set zur gleichen Elementgruppe (class) gehören.
[dem entspricht in SIMON:
ADDFIRST(ent)TO:(set) bzw.
ADDLAST(ent)TO:(set)]

HEAD s_1 GAINS s_2
TAIL s_1 GAINS s_2

(ohne Erfolgskontrolle) In der Liste s_1 werden vorn bzw. hinten alle Elemente der Liste s_2 eingetragen, die nicht schon in s_1 sind.
[hat keine Entsprechung in SIMON]

ent FROM set $m_1 \textcircled{@} m_2$

Das Element ent wird bedingt aus der Liste set entfernt. Bedingung: ent ist in set.
[Entsprechend DELETE(ent)FROM:(SET) in SIMON]

s_1 LOSES s_2

(ohne Erfolgskontrolle) Diejenigen Elemente werden aus der Liste s_1 entfernt, die zugleich in der Liste s_2 sind.
[Entsprechend DELETE(s_1)FROM:(s_2) in SIMON]

ZERO s_1, s_2, ...

(ohne Erfolgskontrolle) Leermachen der Listen s_1, s_2, ... Hierbei brauchen nicht alle aufgeführten Listen zur gleichen Elementgruppe gehören.
[in SIMON realisierbar durch DELETE(s_1)FROM:(s_1)]

LOAD s_1, s_2, ...

(ohne Erfolgskontrolle) die Listen s_1, s_2, ... werden jeweils mit allen Elementen der jeweils zugehörigen Elementgruppe (class) besetzt (Voraussetzung ist, daß die Listen groß genug vereinbart wurden). Dementsprechend dürfen s_1, s_2, ... verschiedenen Elementgruppen zugehören.
[hat keine Entsprechung in SIMON]

s_1 CONVERSE s_2

(ohne Erfolgskontrolle) Die Liste s_1 wird neu besetzt mit allen Elementen der Elementgruppe (class), die nicht in der Liste s_2 stehen.
[hat keine Entsprechung in SIMON]

d) Hauptunterschiede zwischen CSL und SIMON

Weil in CSL Listen nur jeweils für eine bestimmte Elementgruppe vereinbart werden können, erübrigen sich Äquivalente zu den SIMON-Prozeduren REFNUM(ent) und MEMNUM(gent), denn bei der Durchmusterung einer Liste ist die Elementgruppe sowieso von vornherein bestimmt und die Durchmusterungsroutinen von CSL liefern daher automatisch als Resultat den Gruppenindex.

Die SIMON-Prozeduren BEHEAD(set) und BETAIL(set) haben keine Entsprechung in CSL, denn einerseits hat die Reihenfolge der Listenelemente hier keine so große Bedeutung, nachdem jedes Element nur einmal in jeder Liste auftreten darf, und andererseits kann man mittels der sehr komfortablen Durchmusterungsroutinen in CSL leicht das gesuchte Element ent feststellen und dann mittels entFROMset entfernen.

Diese komfortablen CSL-Durchmusterungsroutinen wie FOR und FIND, die wir in den folgenden Abschnitten behandeln werden, bieten viel weitergehende Möglichkeiten als die SIMON-Prozeduren HEADOF(set), TAILOF(set), SIZEOF(set) sowie SCAN(set)FOR:(member)WITH:(leasttime), und erübrigen auch ein Analogon zu ROTATE(set, n), das in SIMON in Verbindung mit HEADOF(set) neben der Durchmusterungsroutine SCAN die einzige Möglichkeit zur Listendurchmusterung bietet.

Diese CSL-Durchmusterungsroutinen gestatten es Bedingungen anzugeben, die die bei der Durchmusterung in Betracht zu ziehenden Elemente erfüllen müssen, und Ausdrücke anzugeben, die bei dem gesuchten Element einen Maximal- bzw. Minimalwert liefern. Ferner sind FOR-Schleifen über set-Elemente möglich. Diese Komfort-Ausstattung geht weit über das Maß dessen hinaus, das erforderlich wäre, um den Unterschied in der Ereigniszeitbehandlung gegenüber SIMON auszugleichen, der das Auffinden aller Elemente mit Ereigniszeit 0 notwendig macht.

Schließlich entfällt auch die Notwendigkeit für ein Analogon zu den SIMON-Prozeduren TIMEVALUE(ent) und SETTIME(ent)TO:(value), da die Ereigniszeit T.ent in CSL wie eine gewöhnliche Variable zugänglich ist.

Was Histogramme und Wahrscheinlichkeitsverteilungen anlangt, so ist der Unterschied zu SIMON, abgesehen von der Möglichkeit, verschiedene Standardzufallsverteilungen unmittelbar zu verwenden, nicht sehr groß.

Als Hauptunterschiede von CSL gegenüber SIMON hat man somit:

1. Listen sind jeweils nur für bestimmte Elementgruppen zugelassen.
2. Ein Element kann nicht mehrfach in eine Liste eingetragen werden.
3. Die Ereigniszeiten werden als Zeitdifferenzen zur Simulationszeit CLOCK geführt und wie diese am Ende des Zeitzyklus automatisch fortgeschaltet.
4. Die Listenbehandlungsroutinen in CSL bieten einen ganz beträchtlich größeren Komfort, als es in SIMON der Fall ist.
5. CSL besitzt viele eigene Sprachelemente und kann nicht wie SIMON durch Erstellung einiger CODE-Prozeduren implementiert werden.

In den folgenden Abschnitten betrachten wir nun die über SIMON hinausgehenden Möglichkeiten, die CSL zur Listenverarbeitung bietet.

e) FOR-Compound-Statements (zusammengesetzte FOR-Anweisungen)

Als Laufanweisung kennt FORTRAN nur das DO-statement
DO marke var = m_1, m_2, m_3, wobei marke die letzte zum Wiederholungsteil gehörige Anweisung markiert und die Laufvariable var die Werte von m_1 bis m_2 in Schritten von m_3 durchläuft.

Das FOR-statement gibt es in zwei Varianten. Die erste Variante leistet das gleiche wie das DO-statement:

FOR var = m_1, m_2, m_3
]

Die Laufvariable var durchläuft die Werte m_1 bis m_2 in Schritten von m_3. Im Unterschied zum DO-statement werden hier jedoch die zum Wiederholungsteil gehörigen Anweisungen durch gleichmäßiges Einrücken kenntlich gemacht, was durch das Zeichen ( ]) angedeutet sei.

Die Klammerung von zusammengesetzten Anweisungen durch gleichmäßiges Einrücken spielt auch im Folgenden eine wichtige Rolle. Verschachtelung ist hierbei zulässig.

Die für die Listenbehandlung wichtige Form des FOR-statements lautet

FOR var = set
]

Die Laufvariable var durchläuft hierbei die Gruppenindizes derjenigen Elemente, die in der Liste set aufgeführt sind. (In den Anweisungen des Wiederholungsteils können dann Größen wie cname.var(i) oder T.cname.var auftreten)

f) Bedingungen

In den folgenden Durchmusterungsroutinen spielen Bedingungen eine Rolle. Auf die zusammengesetzten Bedingungsketten werden wir in Abschnitt i) eingehen. Hier seien einige einfache Bedingungen angeführt, die in CSL zu den üblichen Bedingungen wie Relationen etc. hinzukommen:

ent IN set — hat den Wert TRUE, falls das Element ent in der Liste set steht.

ent NOTIN set — hat den Wert TRUE, falls das Element ent nicht in der Liste set steht.

s_1 EQUALS s_2 — hat den Wert TRUE, falls die Liste s_1 genau die gleichen Elemente wie die Liste s_2 enthält, ohne Beachtung ihrer Reihenfolge.

s_1 WITHIN s_2 — hat den Wert TRUE, falls alle Elemente der Liste s_1 auch in der Liste s_2 enthalten sind.

s_1, s_2,... EMPTY — hat den Wert TRUE, falls keine der Listen s_1, s_2,... ein Element enthält.

Fügt man einer solchen Bedingung wie in Abschnitt c) eine Sprungangabe $m_1 \textcircled{\alpha} m_2$ bei, so wird sie zum bedingten Sprung. Man kann diese Bedingungen ohne Sprungangabe ebenso als bedingte Sprünge mit vom Übersetzer automatisch eingesetzten Sprungadressen auffassen, was bei der späteren Behandlung von Bedingungsketten deutlich wird.

g) FIND-Compound-Statements (zusammengesetzte Durchmusterungsanweisungen

Die FIND-statements werden in Verbindung mit sogenannten Bedingungsketten verwendet, auf deren Aufbau wir in Abschnitt i) eingehen. Eine Bedingungskette wird aufgebaut aus Bedingungen und ist selbst eine Bedingung, die den Wahrheitswert TRUE oder FALSE haben kann. Eine Bedingungskette kann auch leer sein und hat dann den Wert TRUE. Die eine Bedingungskette bildenden Bedingungen werden wieder durch gleichmäßiges Einrücken zusammengefaßt.

Das FIND-statement dient der Durchmusterung von Listen und liefert ein Element aus der Liste mit bestimmten Eigenschaften. Mittels einer Sprungangabe kann die Erfolgsprüfung vorgenommen werden, ob ein solches Element in der Liste gefunden wurde (vgl. Ziff. c). Das FIND-statement hat die Gestalt

FIND variable setname criterion $m_1 \textcircled{\sim} m_2$

] (Bedingungskette)

criterion steht für eins der Worte ANY, FIRST, LAST, MAX(arth expr), MIN(arith expr) und gibt an, ob unter den die Bedingungskette erfüllenden Listenelementen ein beliebiges, das erste, das letzte, oder eins für das ein arithmetischer Ausdruck den größten bzw. kleinsten Wert annimmt, genommen werden soll.

Im einzelnen durchläuft variable den Gruppenindex aller Elemente der Liste setname Beachtet werden nur solche Elemente, für die die Bedingungskette den Wert TRUE annimmt. Unter diesen Elementen wird sodann unter Berücksichtigung von criterion eins ausgewählt und abschließend variable mit dessen Gruppenindex als Resultatwert besetzt.

Je nach criterion geschieht dabei folgendes

ANY — Aus den beachteten (Bedingungskette!) Elementen wird ein zufälliges ausgewählt. Eine Standardzufallsfolge wird hierfür herangezogen. Soll eine andere durch den Wert stream näher gekennzeichnete Zufallsauswahl herangezogen werden (vgl. Ziffer g) so schreibt man an Stelle von ANY nun ANY(stream).

FIRST — Aus den beachteten Elementen wird das erste in der Liste auftretende ausgewählt [wenn die Bedingungskette leer ist, so entspricht das der SIMON-Prozedur HEADOF(set)].

LAST — Aus den beachteten Elementen wird das letzte in der Liste auftretende ausgewählt [ist die Bedingungskette leer, so entspricht dies der SIMON-Prozedur TAILOF(set)].

MAX(arith expr) — Für die beachteten Elemente wird der arithmetische Ausdruck arith expr berechnet und auf integer gerundet. Sodann wird unter den Elementen, für die sich hierbei der Maximalwert ergeben hat, das letzte in der Liste auftretende ausgewählt.

MIN(arith expr) — Analog MAX wird hier ein Element ausgewählt, für das arith expr nach Integerrundung den Minimalwert annimmt.
[mit MIN(T.cname.variable) hat man den Gruppenindex variable des Elements mit kleinster Ereigniszeit, was also der SIMON-Prozedur SCAN(set)FOR:(member)WITH:(leasttime) entspricht]

h) Testchain-Compound-Statements (zusammengesetzte Bedingungskettenanweisungen) und Listenumordnung

Bedingungsketten können analog dem FIND-statement noch mit folgenden weiteren "Titel"-Anweisungen verarbeitet werden (wobei die zu der betreffenden Titelanweisung gehörige Bedingungskette unter derselben folgt und wieder gleichmäßig eingerückt wird).

CHAIN m_1 (L) m_2
(Bedingungskette)

Prüfung der Bedingungskette: Sprung nach m_1, wenn ihr Wert TRUE ist, andernfalls Sprung nach m_2. Das in c) über die Sprungangabe gesagte gilt auch hier.

ALL variable set m_1 (A) m_2
(Bedingungskette)

Prüft, ob die Bedingungskette stets erfüllt ist, wenn die variable alle Gruppenindices der Elemente in der Liste set durchläuft; ist dies der Fall (TRUE), so erfolgt Sprung nach m_1 (vgl. wieder c).

EXISTS (integer expr) variable set m_1 (E) m_2
(Bedingungskette)

Prüft analog ALL, ob die Bedingungskette für mindestens so viele Elemente der Liste set erfüllt ist, wie der Ausdruck int.expr angibt.

UNIQUE (int.expr) variable set m_1 (U) m_2
(Bedingungskette)

Prüft analog EXISTS, ob die Bedingungskette für genau so viele Elemente der Liste set erfüllt ist, wie int.expr angibt.

COUNT variable set
(Bedingungskette)

(keine Sprungangabe!) Zunächst durchläuft variable die Gruppenindices der Elemente in der Liste set und es wird mitgezählt, für wieviele der Elemente die Bedingungskette den Wert TRUE hatte. Abschließend wird variable die Anzahl dieser Elemente als Resultatwert zugewiesen. [Falls die Bedingungskette leer ist, so hat man das Äquivalent zu der SIMON-Prozedur SIZEOF(set)]

SUM (int.expr) variable=set (keine Sprungangabe!) Zunächst
] (Bedingungskette)
durchläuft variable die Gruppenindices der Elemente in der Liste set. Für diejenigen Elemente, für die die Bedingungskette erfüllt ist, wird der Ausdruck int expr (vom Typ integer) errechnet und aufsummiert. Schließlich wird variable als Resultatwert das Ergebnis dieser Summation zugewiesen.

In der Form
SUM(int.expr)variable=m_1, m_2, m_3
ohne Bedingungskette kann die Summe über alle Werte des int expr berechnet werden, wenn variable von m_1 in Schritten von m_3 bis m_2 läuft. Abschließend kommt das Resultat wieder nach variable.

RANK variable set (arith expr) (keine Bedingungskette und keine Sprungangabe!) Zunächst durchläuft variable die Gruppenindices der Elemente in der Liste set. Jeweils der Ausdruck arith expr wird errechnet und nach dessen Wert in abnehmender Größe die Elemente in der Liste set in neuer Reihenfolge angeordnet.

SPLIT variable s_1 INTO s_2 ELSE s_3
] (Bedingungskette)
Die Listennamen s_1, s_2, s_3 müssen für die gleiche Elementgruppe erklärt sein. Dann wird zunächst s_2 und s_3 leergemacht, sodann durchläuft variable die Gruppenindices der Elemente in s_1. Ergibt die Bedingungskette dabei

jeweils für ein Element den Wert TRUE, so wird es in s_2 eingetra en, anderenfalls in s_3.

Die Teile INTOs_2 bzw. ELSEs_3 können auch entfallen; dann wird der betreffende Teil der SPLIT-Anweisung unterdrückt.

Ersetzt man INTOs_2 durch INTO HEADs_2 bzw. INTO TAILs_2 so wird s_2 zu Beginn der SPLIT-Anweisung nicht leergemacht, sondern die betreffenden Elemente aus s_1 werden vorn bzw. hinten zusätzlich zu den bereits vorhandenen Element in s_2 eingetragen, sofern sie nicht bereits eingetragen waren. Analoges gilt bei Ersetzung von ELSEs_3 durch ELSE HEADs_3 bzw. ELSE TAILs_3.

Als spezielle Bedingung innerhalb der Bedingungskette ist beim SPLIT-statement hier

QUALIFY (Bedingungskette 2)

zugelassen. Dann werden die Elemente aus s_1, für die die Bedingungskette 2 FALSE ist, übergangen und weder in s_2 noch in s_3 eingetragen.

i) Bedingungsketten

Als "Bedingungen", aus denen Bedingungsketten aufgebaut werden, können grundsätzlich fast alle unbedingten oder bedingten Anweisungen herangezogen werden, sofern man bei ihnen keine Sprungangabe macht (vgl. Ziff. c).

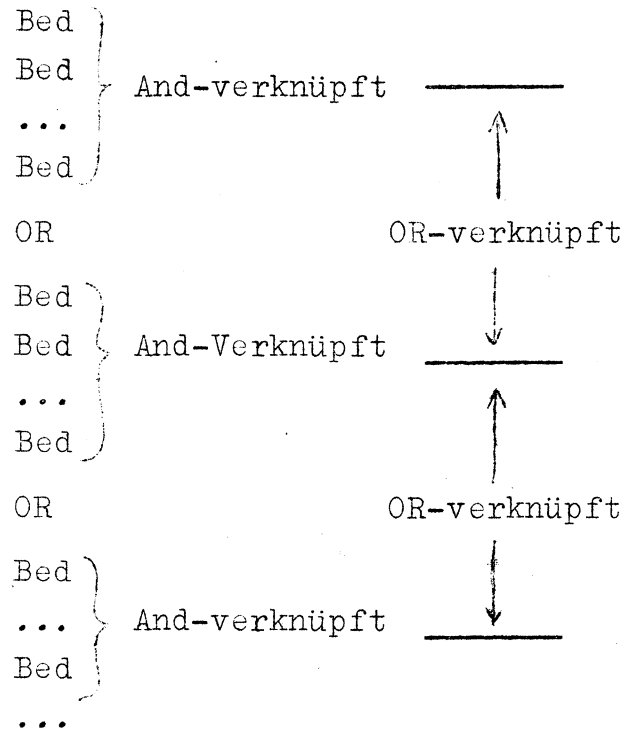
Solche Anweisungen werden bei ihrer Berücksichtigung als "Bedingung" normal vollzogen, und ihnen, wenn z.B. keine Erfolgsprüfung und also kein Wahrheitswert für sie vorgesehen ist, der Wahrheitswert TRUE zugeordnet. Dabei ist jedoch zu beachten, daß die Prüfung einer AND-verknüpften Bedingungsfolge schon beim ersten FALSE abgebrochen wird, weil der Wahrheitswert der Folge dann bereits feststeht.

Als Bedingungen kommen also neben Relationen insbesondere die in f) genannten in Frage, aber auch die in e), g) und h) genannten compound-statements unter Weglassung der Sprungangabe, wobei bis zu 20fache Verschachtelung der zusammengesetzten Bedingungen ineinander zulässig ist.

Schreibt man nun eine Folge solcher Bedingungen gleichmäßig eingerückt untereinander, so gelten sie als and-verknüpft. Schreibt man zwischen jeweils zwei solche in sich and-verknüpften Bedingungsteilfolgen das Wort

OR

so gelten diese Teilfolgen untereinander als or-verknüpft. Das so entstehende Gebilde ist eine Bedingungskette, dem als Resultat ein Wahrheitswert zukommt.



j) Zufallsverteilungen

In den im folgenden aufgeführten Zufallsfunktionsprozeduren kommt jeweils ein Parameter stream vom Typ integer vor, der mit einer Variablen besetzt sein muß, die einen ganzzahligen Wert > 0 haben muß. Bei jedem Aufruf einer Zufallsfunktion wird diesem Parameter ein neuer Wert zugewiesen, während der alte intern beim Erzeugen der Zufallszahl mit verwendet wird. Die erhaltene Zufallszahl selbst kann Variablen vom Typ real oder integer zugewiesen werden.

RANDOM (stream, range) ist eine Funktionsprozedur. Range ist ein Ausdruck vom Typ integer. Als Funktionswert stellt sich eine Zufallszahl im Bereich $1 \leq$ RANDOM $\leq$ range bei rechteckiger Häufigkeitsverteilung ein.

DEVIATE (stream, deviation, mean) ist eine Funktionsprozedur; deviation, mean sind Ausdrücke vom Typ integer. Als Funktionswert stellt sich eine Zufallszahl aus einer Häufigkeits-Normalverteilung ein, deren Mittelwert = mean und deren Streuung = deviation ist.

NEGEXP (stream, mean) ist eine Funktionsprozedur, und mean ein Ausdruck vom Typ integer. Als Funktionswert stellt sich eine Zufallszahl aus einer negativen Exponentialverteilung ein, deren Mittelwert = mean ist.

POISSON (stream, mean) ist eine Funktionsprozedur, und mean ein Ausdruck vom Typ integer. Als Funktionswert stellt sich eine Zufallszahl aus einer POISSON-Verteilung mit Mittelwert = mean ein.

Soll eine vom Programmierer vorzugebende Häufigkeitsverteilung bei der Erzeugung einer Zufallszahl zugrunde gelegt werden, wobei m Werte a_t und die jeweils zugehörige Häufigkeit n_t des Auftretens von a_t angegeben werden sollen, so muß zunächst ein Feld

ARRAY aname (2, m+1)

vereinbart sein und dieses wie üblich mit der zu verwendenden

Häufigkeitsverteilung vorbesetzt werden, so daß die einzelnen Feldkomponenten wie folgt besetzt sind:

$\sum_{t=1}^{m} n_t$	n_1	n_2	n_3		n_m
m	a_1	a_2	a_3		a_m

n_t = Häufigkeit des Auftretens für den Wert a_t

(Hierfür wird die gewöhnliche Einleseroutine verwendet, wobei das Einlesen spaltenweise erfolgt)
Danach kann man das Statement schreiben:

DIST aname1, aname 2, ... Durch dieses Statement werden die Feldnamen aname1, aname2, ... als Verteilungsnamen eingeführt und die dort gespeicherten Häufigkeitsverteilungen umgerechnet in Summenkurven (wobei sich also die Besetzung dieser Felder ändert!)

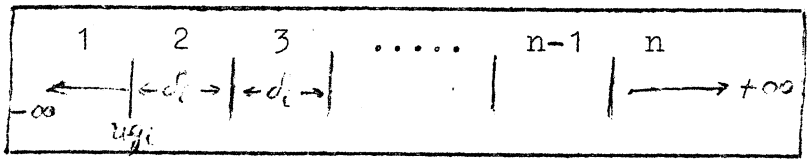
SAMPLE (index, dist1, stream1, dist2, stream2, ...)
ist eine Funktionsprozedur; $dist_i$ sind mittels DIST als Verteilungsnamen eingeführte Feldnamen, $stream_i$ sind die jeweils zugehörigen (eingangs beschriebenen) Parameter stream; index ist ein Ausdruck vom Typ integer.

Als Funktionswert stellt sich eine Zufallszahl aus derjenigen Verteilung $dist_i$ mit dem Parameter $stream_i$ ein, die durch i = index bestimmt ist.

k) Histogramme

HIST hname1 (n_1, ug_1, δ_1), hname2 (n_2, g_2, δ_2), ...
führt Histogramme mit dem Namen $hname_i$ ein. n_i, ug_i, $_i$ sind Ausdrücke vom Typ integer.

Jeweils das Histogramm $hname_i$ besitzt dabei n_i Intervalle; das erste dieser Intervalle erstreckt sich von $-\infty$ bis ug_i, dann folgen n-2 Intervalle der einheitlichen Intervallbreite δ_i, an deren letztes sich das Intervall bis $+\infty$ anschließt.



ADD int expr, hname — trägt den Wert des Ausdruckes int expr (vom Typ integer) in das Histogramm hname ein.

OUTPUT hname1, hname2, ... — Druckt auf dem Ausgabemedium die Histogramme hname1, hname2, ... jeweils in der folgenden Gestalt

m_1		TO a_2
m_2	a_3	TO a_4
m_3	a_5	TO a_6
.....		
m_n	a_2	TO

dabei bedeutet jeweils m_ν $a_{2\nu-1}$ TO $a_{2\nu}$ daß m_ν Eintragungen im Bereich von $a_{2\nu-1}$ bis $a_{2\nu}$ registriert wurden (dabei gilt $a_{2\nu+1} = a_{2\nu} + 1$)

Überschriften etc. müssen extra mittels WRITE ausgegeben werden.

CLEAR hname1, hname2, ... — entleert die Histogramme hname1, hname2, ...

YIELD (hname, range) — ist eine Funktionsprozedur vom Typ integer; range ist ein Ausdruck vom Typ integer.

Als Funktionswert stellt sich die Anzahl der bisherigen Eintragungen in dasjenige Intervall des Histogramms hname ein, dem der Wert range angehört.

1) Wahlschalterabhängiger Kontrolldruck

CHECK (name1, name2, . .) druckt bei entsprechender Wahlschalterstellung die aufgeführten Variablen in folgender Form aus:

```
name1 EQUALS Wert
name2 EQUALS Wert
.................
```

4. Schlußbemerkung

An der sehr einfachen Simulationssprache SIMON, die aus ALGOL durch Hinzufügen einiger CODE-Prozeduren entsteht, haben wir gesehen, was eine Simulationssprache als solche kennzeichnet. Bei der etwas weiter ausgebauten Simulationssprache CSL haben wir gesehen, daß der Mehraufwand dabei praktisch ausschließlich den Listenbehandlungsroutinen zugute kam.

Neben den hier behandelten Simulationssprachen haben die folgenden noch größere Bedeutung: SIMSCRIPT, GPSS, SIMULA67. Wieder ist bei ihnen ein wesentliches Merkmal die Einführung von Listenstrukturen als Datentypen. Darüber hinaus gestattet SIMULA67 nicht nur z.B. Wartesohlangen als Listen zu etablieren, sondern implizite können auch die zu verschiedenen Zeitpunkten verschiedenen Zuständen der einzelnen Variablen jeweils als zu Listen zusammengefaßt aufgefaßt werden. Über die zeitlichen Veränderungen der einzelnen Variablen wird automatisch Buch geführt und so kann ein komplizierterer Simulationsprozess später nach verschiedenen, sich im Laufe

der Auswertung neu ergebenden Gesichtspunkten, ausgewertet werden ohne daß eine nochmalige Wiederholung des Simulationsvorgangs erforderlich ist.

Die verwendete Simulationssprache ist jedoch nur ein Hilfsmittel für das Problem der Simulation eines Betriebssystems. Wir sind hauptsächlich darum auf die Simulationssprachen so weit eingegangen, um Einblick in die Struktur der Simulation als solcher zu gewinnen. Das eigentliche Problem bei der Simulation eines Betriebssystems ist jedoch nicht die verwendete Simulationssprache, sondern die Gewinnung eines geeigneten Simulationsmodells.

Beispielsweise ist auch die Frage, wie man Engpässe lokalisiert und was genau man unter dem Begriff "Engpaß" zu verstehen hat, letzten Endes ungeklärt. Das naive Vorgehen aus Ziffer 2 hat zwar zu Verbesserungen geführt und das Problem erahnen lassen, ist jedoch recht unbefriedigend. Da Engpässe Job-mix abhängig sind, kann man die Warteschlangen bzw. die betreffenden Wartezeiten nur jeweils für einen bestimmten Job-mix betrachten. Dabei werden aber auch vor einem Engpaß die Warteschlangen nicht beliebig anwachsen, sondern es wird sich eine in etwa stationäre Warteschlangenlänge einstellen: Wenn nämlich z.B. die Speicherverdrängung zu langsam geht, so wird der Speicher zu langsam frei für neue Transferaufträge in dem Speicher, auf die die CPU wartet. Die Folge ist, daß dann auch entsprechend weniger Verdrängungsaufträge gegeben werden.

Um zu einer Bewertung der Engpässe zu kommen, könnte man eine Art Gradientenmethode versuchen, indem man für jeden Engpaß isoliert eine beschleunigte Verarbeitungszeit annimmt und die Auswirkung auf den Durchsatz des Systems an Aufträgen untersucht. Jedoch muß diese Methode jedenfalls dann versagen, wenn nur gleichzeitige Beschleunigung von zwei ("gekoppelten") Engpässen den Durchsatz erhöht.

Der Begriff "Engpaß" wird weiter beleuchtet durch die Bemerkung, daß es auch erwünschte Engpässe gibt, wie z.B. den bei der Eingabe, wenn mehr Aufträge gerechnet werden konnten, als vorliegen. Manche Engpässe sind legitim, wie der durch die Leistungsfähigkeit des Druckers bestimmte, wenn der Jobmix ausschließlich Druckaufträge gibt.

Im Anschluß an den "Verwaltungsengpaß" aus Ziffer 2, bei dem die CPU hauptsächlich mit Verwaltungsroutinen "verstopft" wurde, erhebt sich die Frage, welche Programmteile hat man in diesem Sinne zur Verwaltung zu rechnen; und wie findet man innerhalb der Verwaltungsroutinen diejenigen heraus, deren Beschleunigung am meisten Gewinn für den Durchsatz bringt.

Abschließend wollen wir noch kurz auf den praktischen Anwendungsfall für Betriebssystemsimulation eingehen, der zumeist darin besteht, daß ein Betriebssystem entwickelt werden soll und dafür Entscheidungen über alternative Strategien bei einzelnen Teilaufgaben zu treffen sind, die zu einem späteren Zeitpunkt nicht mehr geändert werden können.

Das bedeutet, daß unter starkem Zeitdruck ein Modell für die Simulation dieses Teilkomplexes entwickelt werden muß, was dadurch erschwert ist, daß für große Teile des restlichen Systems die Planung noch nicht abgeschlossen ist. Der mit der Entwicklung des Simulationsmodells Beauftragte wird dann eine Reihe von "Eingangsparametern" für sein Simulationsteilmodell isolieren können, die das Simulationsergebnis stark mitbestimmen und wird nun seinerseits dem beauftragenden Systemprogrammierer nahelegen, sich vordringlich mit der Entscheidung über diese Eingangsparameter zu befassen, bevor die ursprüngliche Fragestellung mittels Simulation geklärt werden kann.

So werden sich Systemprogrammierung und Simulationsaufgaben wechselseitig bedingen und eine enge Zusammenarbeit zwischen den Arbeitsgruppen für Systemprogrammierung und Simulation erforderlich machen.