Die Simulationssprache C-SIM

In diesem Bericht soll in die Technik des Simulationssystems C-SIM eingeführt werden. Damit soll ein Grundverständnis für die Simulationstechnik und die entsprechende Realisierung in klassischen Programmiersprachen gelegt werden. Um die folgenden Ausführungen verstehen zu können, sind Kenntnisse von C nötig.

Das Simulationssystem C-SIM erhält seine Funktionalität vorwiegend dadurch, daß Änderungen durch Umhängung von Zeigern realisiert werden, welches in der Regel eine sehr effiziente Prozedur ist, die darüber hinaus auch leicht für verschiedene Anwendungen erweitert werden kann. Dadurch ist dieses Simulationssystem effizienter und effektiver als andere Simulationskonzepte, die häufig große mehrdimensionale Felder verwalten müssen.

1 Prozesse, Phasen, Ereignisse

Ein Prozeß ist ein Modell eines (mehr oder minder stark abgeschirmten Systems). Ein System besitzt in der Regel einen (inneren) Zustand, eine definierte Schnittstelle nach außen, sowie ein bestimmtes Verhalten bei Einwirkung von Ereignissen. Dieses ist in einem Simulationssystem entsprechend nachzubilden.

Wir betrachten einen Prozeß als eine Menge von Variablen, die den internen statischen Zustand darstellen. Dieser Zustand bleibt über die gesamte Zeit der Simulation unverändert, es sei denn er wird explizit während des Simulationslaufs verändert. Außerdem besitzt ein Prozeß verschiedene Phasen, die den dynamischen Zustand beschreiben. Ein Prozeß führt in der Regel ein Programmstück aus, verweilt sodann eine gewisse Zeit, ehe er in einer anderen Phase ein weiteres Programmstück ausführt. Innerhalb eines solchen Programmstücks kann ein Prozeß eine Änderung des lokalen Zustands ausführen, oder auch Signale an andere Prozesse schicken.

Die Zeit ist hier eine virtuelle SystemZeit, die für das zeitliche Verhalten des realen Systems benötigt wird. Sie hat also nichts mit anderen Zeiten, z.B. der realen Zeit oder der Ausführungszeit auf einem Rechner, zu tun. Sie modelliert einfach nur das zeitliche Verhalten des realen Systems und kann dieses somit stark dehnen oder auch raffen.

Die Realisierung dieses Konzepts in verschiedenen Programmiersprachen ist unterschiedlich bequem. Wir betrachten hier nur C; in einer anderen Realisierung wurde dieses Konzept auch in Pascal erstellt.

1.1 Der Prozeßzustand

Der Zustand eines Prozesses soll durch eine Menge von Variablen dargestellt werden. Diese Variablen enthalten im wesentlichen Werte, z.B. Betriebsmittel wie Speicher, Tabellen für das Routing oder Zähler für Kommunikationsprozesse usw. Diese werden in der Programmiersprache C in der Regel in einem Datensatz zusammengefaßt, z.B.:

```
struct {  int Zaehler;
          int RoutingTable[10];
          float AverageServiceTime;
          bool ReducedPerformance;
          ...;
        } Server;
```

Ein solcher Datensatz muß für jeden Prozeß erzeugt werden. In der Regel unterscheidet man zwischen einem Prozeßtyp, der die wesentlichen Zustände umfaßt, und einer Instanz eines solchen Prozesses. Dabei können zu jedem Prozeßtyp beliebig viele Instanzen erzeugt werden, die jedoch verschiedene Systeme repräsentieren, also im Prinzip auch unterschiedlich deklariert werden könnten. Dennoch hat die Verwendung dieses Konzepts gewisse Vorteile, die später klarer werden. Statt:

```
struct ServerTyp { int Zaehler;
          ...;
        };
```

schreiben wir im folgenden:

```
Process(Server)
          int Zaehler;
          ...;
        };
```

Dadurch soll die Verwendung dieses Datensatzes als Prozeßdatensatz unmißverständlich klar gemacht werden. Es gibt in diesem Datensatz zum einen lokale Variable, die für die Prozeß-fort–schaltung, für den Status, die Funktionalität usw. verwendet werden, sowie lokale Variable, die vom Programmierer für den speziellen Prozeß benutzt werden, wie im Beispiel `Zaehler`.

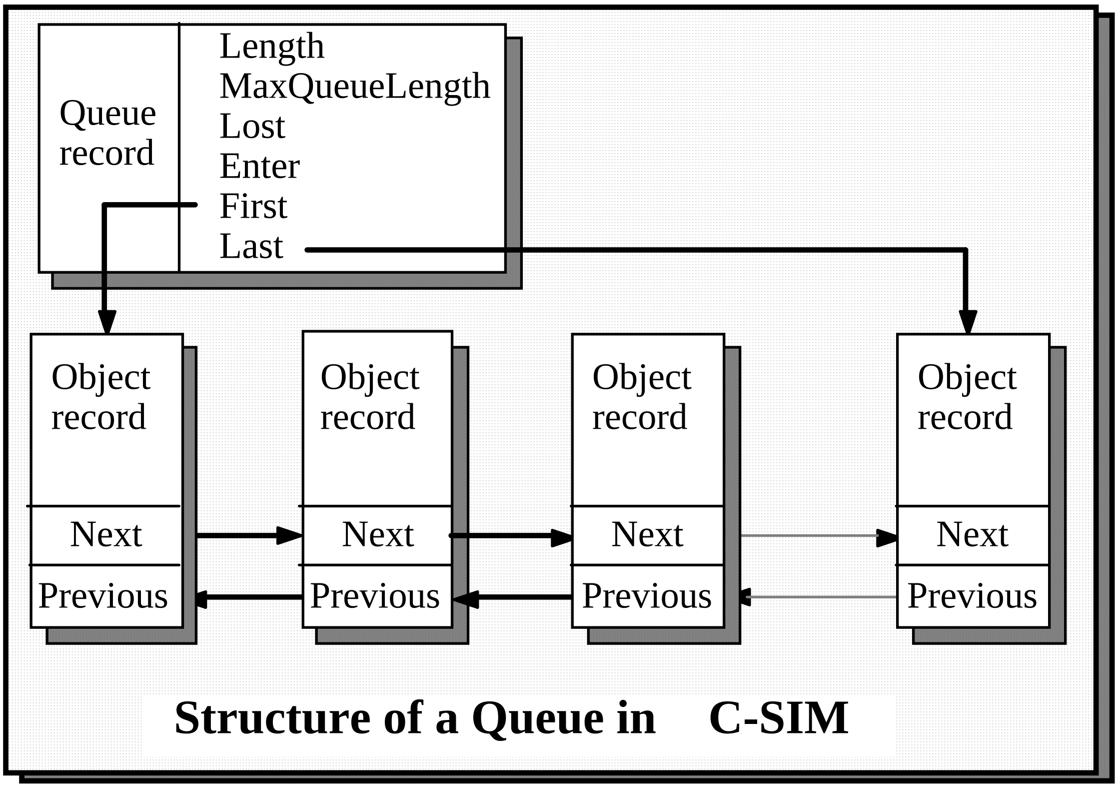
Im Prinzip können beliebig viele und verschiedene Datentypen als lokal zu einem Prozeß deklariert werden. Jedoch gibt es zumindest eine wesentliche Ausnahme: Der Auftrag (oder `Job`). Dieses ist gleichfalls ein Datensatz, für den jedoch eine andere spezielle Datenstruktur, die Warteschlange, definiert wird. Diese kann beliebig viele Aufträge speichern bzw. wieder herausgeben. Dazu gibt es mindestens zwei Funktionen:

```
Job = Seize(JobQueue);
```

und

```
Insert(Job, JobQueue);
```

Die Aufträge werden linear in einer Liste angeordnet, wobei die Funktion `Insert` einen Auftrag als letzten in die Warteschlange einfügt, und die Funktion `Seize` den ersten Auftrag aus der Warteschlange entfernt. Die Strategie ist also: FIFO (first in first out). Es ist jedoch auch möglich, andere Strategien, sowohl beim Einfügen als auch beim Herausholen, zu verwenden. Die Warteschlangen und Aufträge können durch die folgende Graphik veranschaulicht werden:



Structure of a Queue in C-SIM

Eine Warteschlange ist vom Typ `QueueType`, d.h. sie kann auf folgende Weise definiert werden:

```
QueueType Eingangswarteschlange, Ausgangswarteschlange;
```

Der Auftragstyp wird deklariert, indem geschrieben wird:

```
JobType { JobHeader;
          int Zugangszeit;
          ...
        }
```

Dabei sind nach dem `JobHeader`, der die Listenzeiger enthält, die lokalen Daten der Aufträge zu deklarieren.

Prozesse werden in einer Liste, der sogenannten Ereignisliste, geführt. Der Zeiger, der auf den ersten Prozeßdatensatz zeigt, wird als `ProcessList` geführt. Ein Prozeß kann jedoch in der Regel nicht mittels eines Befehls: `ProcessList->Zaehler` auf seine lokalen Variablen zugreifen, da die einzelnen Prozesse verschiedene Datentypen enthalten. Aus diesem Grunde muß ein etwas umständlicherer Zugriff über eine `union` in C durchgeführt werden. Um dieses vor dem Programmierer so weit es geht zu verbergen, soll im folgenden auf lokale Variable durch das Präfix `THIS` zugegriffen werden, also z.B.:

```
THIS Zaehler += 1;
```

Der Makrointerpreter wandelt dieses in einen Zugriff der folgenden Art um:

```
PT.ServerTyp -> Zaehler += 1;
```

was jedoch für den Programmierer in der Regel nicht von Bedeutung ist. Die erste Darstellung ist sicherlich sehr viel übersichtlicher und sollte daher auf jeden Fall verwendet werden.

1.2 Phasen

Ein Prozeß besteht neben Variablen, die seinen statischen Zustand speichern, aus Phasen, die im gewissen Sinne einen dynamischen Zustand darstellen. Eine Phase ist immer ein Programmstück, in welchem z.B. lokale oder globale Variablen verändert werden können, sowie eine Angabe, zu welcher (virtuellen System-) Zeit der Prozeß weiterarbeiten soll; gleichzeitig ist die nächste Phase anzugeben, welche zu diesem neuen Aktivierungszeitpunkt ausgeführt werden soll. Das Leben eines Prozesses besteht also darin, nacheinander bestimmte Phasen auszuführen, wobei zwischen zwei Phasen eine bestimmte Zeit vergeht.

Allerdings gibt es in der Regel mehr als einen Prozeß in einem Simulationsprogramm, und diese Prozesse üben untereinander eine Wechselwirkungen aus. Dadurch wird das dynamische Verhalten eines Systems sehr undurchsichtig, wenn man dieses statisch beschreiben wollte. Das Simulationsprogramm hat jedoch die Aufgabe, dieses Systemverhalten nur durch den Übergang von einer Phase in die nächste zu beschreiben, was sehr viel einfacher ist.

Nachdem der statische Zustand eines Prozesses mittels eines (statischen) Datensatzes in C-SIM beschrieben wird, soll eine Phase im wesentlichen durch ein Programmstück beschrieben werden, wobei der großen Anzahl von Phasen wegen diese in eine systematischen Struktur, der Verzweigungsanweisung, realisiert werden:

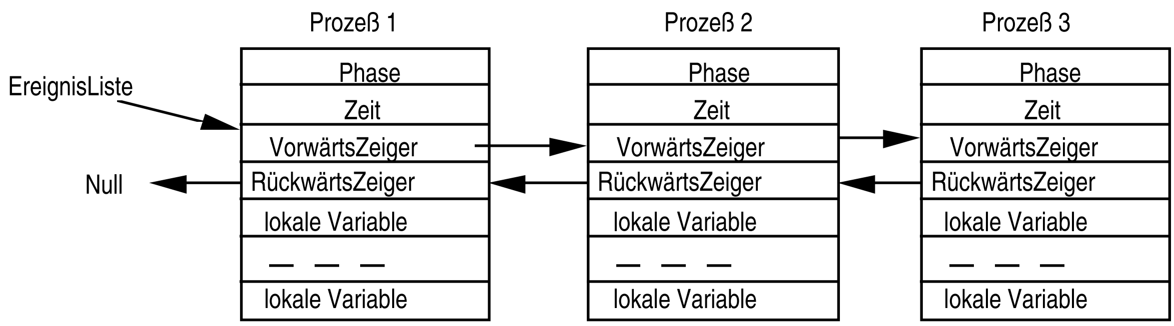
```
while(SystemZeit <= MaxSystemZeit)
{
   switch(EreignisListe -> Phase)
   { case 1:{ /* Führe Anweisungen aus */;
                break;
            }
     case 2:{ /* Führe Anweisungen aus */;
                break;
            }
     case 3:{ /* Führe Anweisungen aus */;
                break;
   }        }
   SortiereEreignisliste();
}
```

Solange die `SystemZeit` noch kleiner ist als eine vorgegebene maximale Zeit, solange wird die äußere `while`-Schleife durchlaufen. Damit ist die `SystemZeit` innerhalb der Schleife auf irgendeine Weise zu verändern. Außerdem muß es eine Liste von Prozessen geben, die hier als Ereignisliste bezeichnet wird. Nachdem der erste Prozeß dieser Liste bearbeitet wurde, wird die Ereignisliste neu sortiert und dann die Schleife durchlaufen, usw. Um die Feinheiten der Sprache zu zu kaschieren, verwenden wir die folgenden Notation:

```
Process Beispiel
   Phase 1
     /* Führe Anweisungen aus */
   Phase 2
     /* Führe Anweisungen aus */
   Phase 3
     /* Führe Anweisungen aus */
EndProcess
```

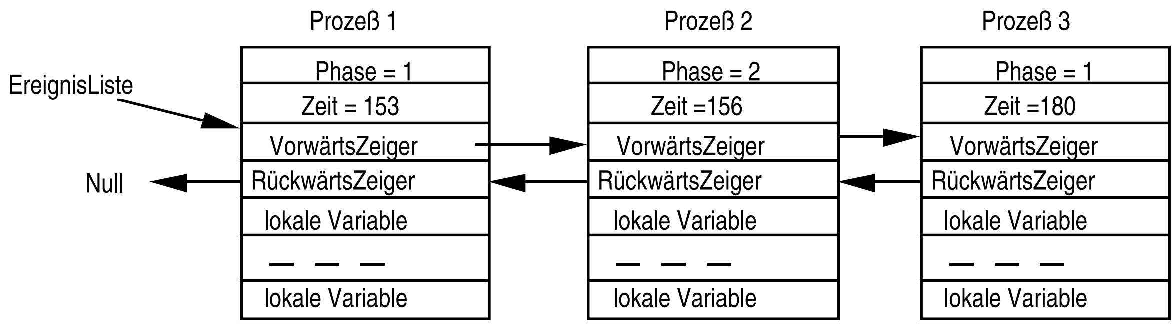
womit wir im wesentlichen des gleiche meinen. Allerdings muß jetzt noch geklärt werden, wie die einzelnen Phasen ausgewählt und die Ereignisliste sortiert wird.

Ein Prozeß besteht aus einem Datensatz mit lokalen Variablen, der darüber hinaus weitere Variable enthält, welche die als nächstes auszuführende Phase und den Zeitpunkt der Ausführung enthält. Da ein Prozeß in der Regel in einer Liste angeordnet ist, soll er außerdem Zeiger für die Verkettung enthalten. Somit besteht ein in der Ereignisliste eingehängter Prozeß (vorläufig) aus den folgenden Komponenten:



Dabei ist jeder Prozeß durch einen eigenen Datensatz repräsentiert. Verschiedene Instanzen eines Prozesses sind somit durch verschiedene Datensätze repräsentiert, wobei auch Prozesse vom gleichen Typ verschiedene Instanzen von Datensätzen haben. Häufig gibt es nur wenig verschiedene Prozeßtypen (z.B. einen Zugangs- und einen Bedienprozeßtyp), aber sehr viele Instanzen dieser Prozeßtypen (z.B. Server1, Server2,... und Zugang1, Zugan2 usw.).

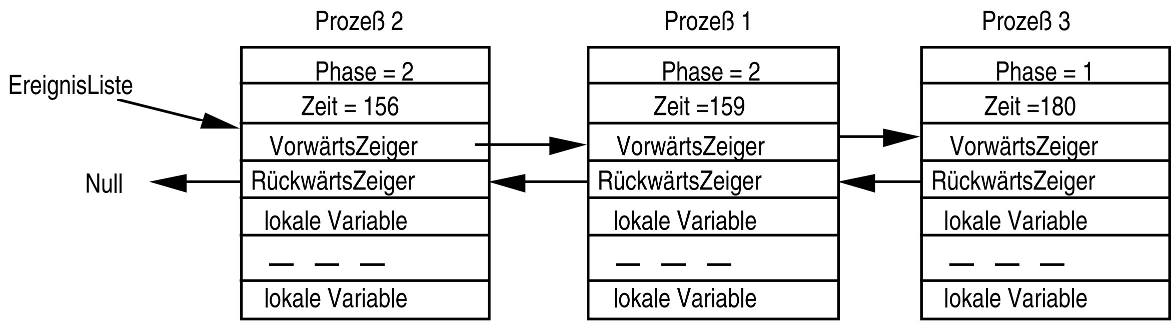
Als Beispiel wird angenommen, daß es für jeden Prozeß zwei Phaen (1 und 2) gibt, und daß zu einer Zeit 153 die jeweiligen Prozesse den folgenden Zustand haben:



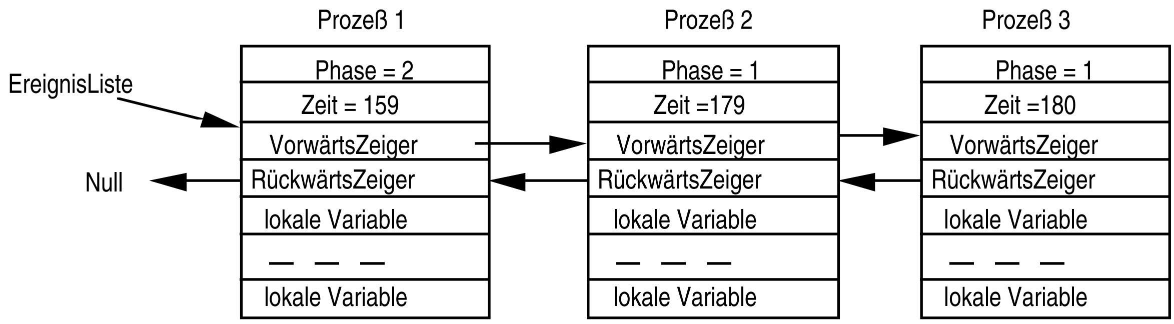
Man sieht, daß die Prozesse in aufsteigender Zeit sortiert sein müssen. Der 'erste' Prozeß hat somit die kleinste Zeit. Diese Zeit ist jetzt in jedem Falle gleich der aktuellen `Systemzeit`, d.h. der Prozeß bringt seine Zeit gewissermaßen mit. Der Prozeß 1 führt somit die erste Phase aus:

```
Process Beispiel
  Phase 1
    /* Tue etwas */
    THIS Phase = 2;
    THIS Zeit = SystemZeit + Zufall(1,10);
  Phase 2
    /* Tue etwas */
    THIS Phase = 1;
    THIS Zeit = SystemZeit + Zufall(0,50);
EndProcess
```

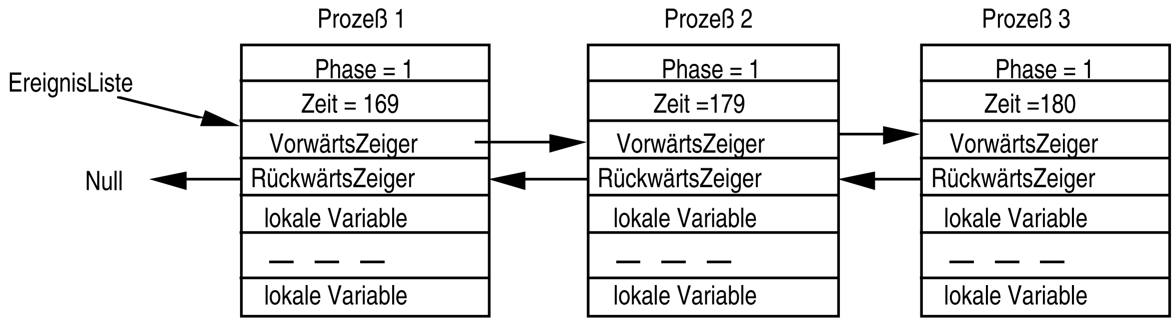
Die Zeit werde in Phase 1 zufällig aus dem Intervall [1,10] gezogen und betrage in diesem Falle `Zufall=6`. Die Nummer der Phase wird offensichtlich auf 2 gesetzt. Damit erhalten wir den folgenden Zustand der drei Prozesse:



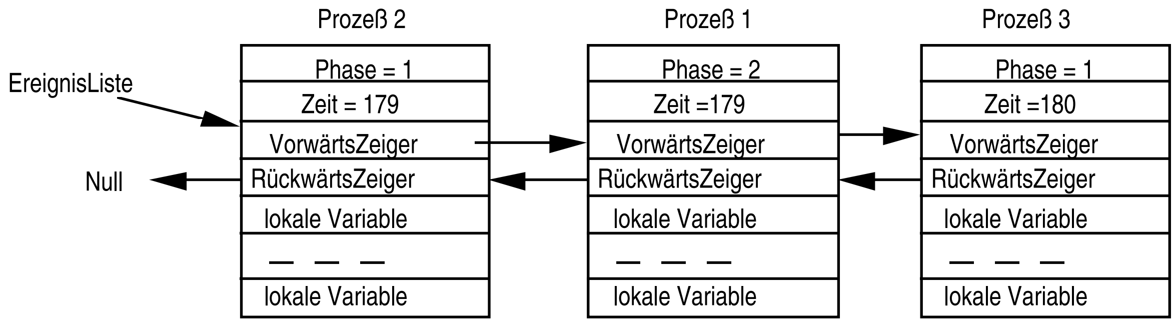
Es muß jetzt also der erste Prozeß hinter den zweitem umgehängt werden. Damit ist die neue `SystemZeit=156`. Jetzt wird die zweite Phase für den zweiten Prozeß ausgeführt. Dieser geht als nächstes in die Phase 1 und zieht dann zufällig eine Zeit, die hier sei: `Zufall=23`. Wir erhalten also den nächsten Zustand:



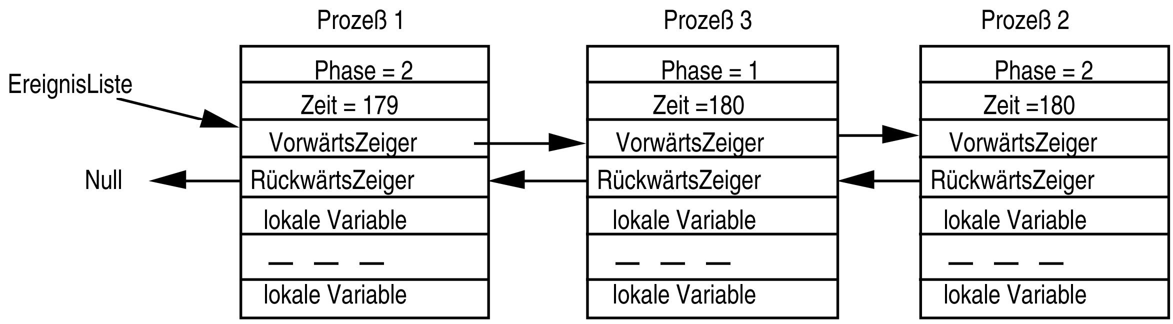
Bisher haben sich nur die ersten beiden Prozesse abgewechselt; der dritte kam noch gar nicht an die Reihe. Der er aber bis zum Zeitpunkt 180 ruhen soll, ist dieses durchaus beabsichtigt. Jetzt, d.h. zur `SystemZeit=159`, führt der erste Prozeß die Phase 2 aus, d.h. er wählt die Phae 1 und zieht die zufällige Zeit: `Zufall=10.` Da diese Zeit nur kleiner ist als die des nächsten Prozesses, bleibt der Prozeß 1 an erster Stelle, geht aber in die Phase 1 über:



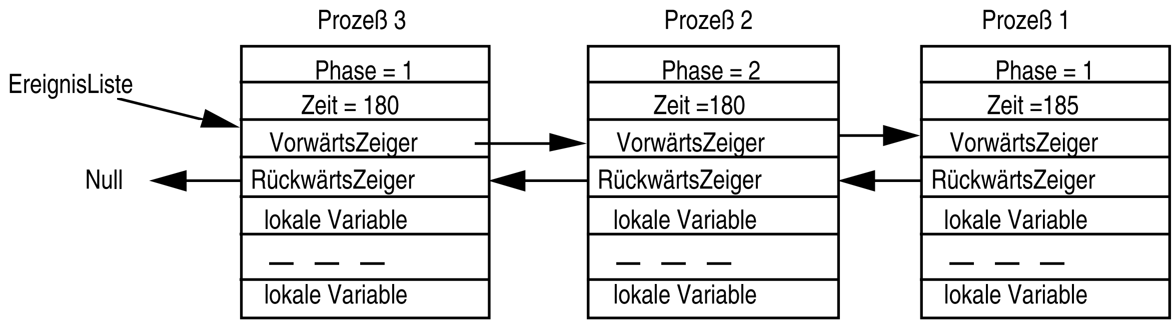
Zu diesem Zeitpunt, d.h. `SystemZeit=169`, wird der zweite Prozeß seine Phase 1 ausführen, wobei er in die Phase 2 wechselt und die Zufallszeit `Zufall=10` ziehe. Das bedeutet in diesem Falle, daß dieser Prozeß zur gleichen Zeit wie der Prozeß 2 aktiviert werden soll. Es ist jetzt festgelegt, daß sich der später zu diesem Zeitpunkt aktivierte Prozeß immer hinter allen früheren einordnet, so daß wir erhalten:



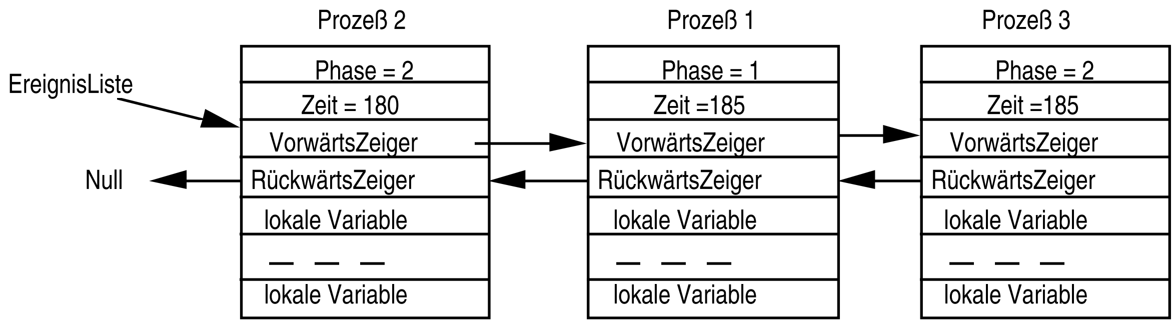
Jetzt führt der Prozeß 2 seine Phase zur `SystemZeit=169` aus. Er wechselt also in die Phase 2 und ziehe die zufällige Zeit `Zufall=1`. Damit hat er sich hinter den Prozeß 3 einzuordnen, d.h. wir erhalten den Zustand:



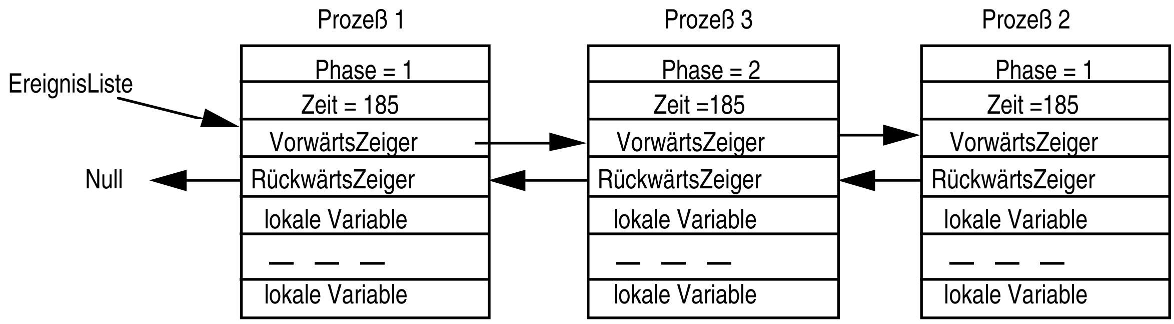
Prozeß 1 ziehe zur `SystemZeit=179` die Zufallszeit `Zufall=6`, so daß wir erhalten:



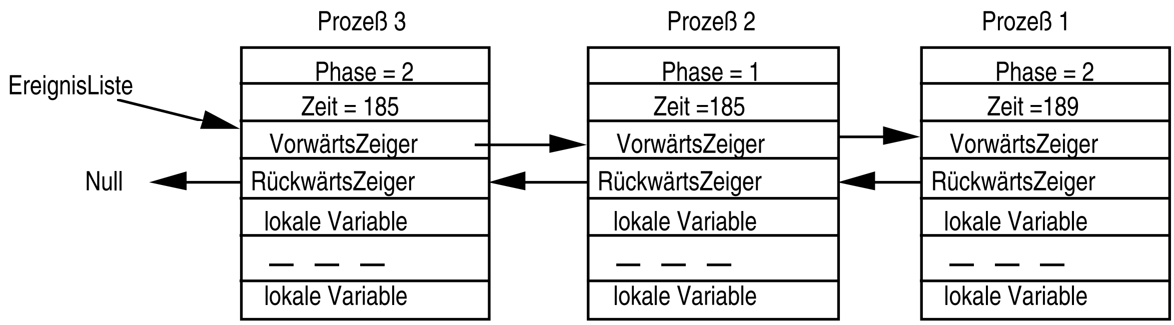
Nun wird endlich der Prozeß 3 einmal bedient. Er ziehe die Zufallszahl `Zufall=5`, so daß wir den Zustand:



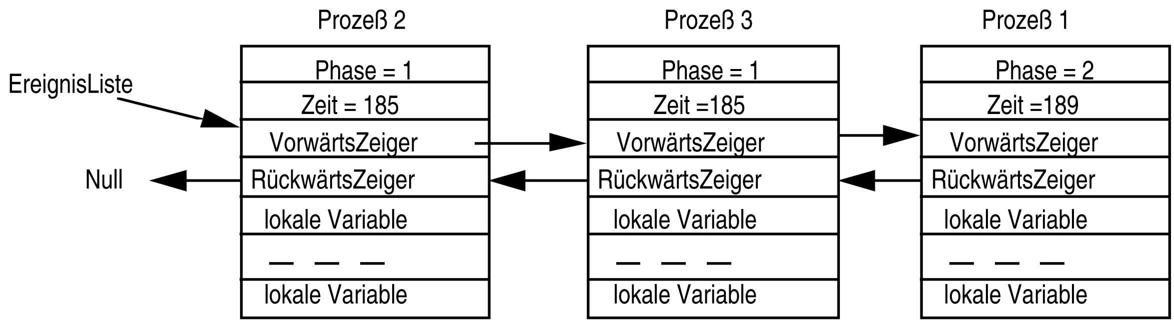
erhalten. Wieder mußte sich der Prozeß 3 ganz hinten anstellen. Ziehe jetzt der Prozeß 2 in der Phase 2 die Zeit `Zufall=5`, so erhalten wir:



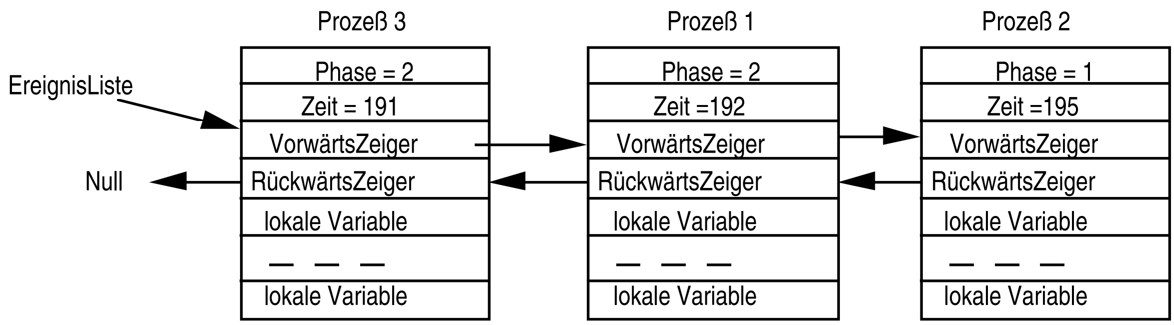
In diesem Zustand haben alle Prozesse die gleich Aktivierungszeit. Dennoch starten sie alle nacheinander, was durch die Regel bewirkt wird, daß ein später eingereihter gleichzeitiger Prozeß auch später zu starten ist. Zieht jetzt der Prozeß 1 die Zeit `Zufall=4`, so erhalten wir:



Als nächstes ziehe der Prozeß 3 in der Phase 2 die Zeit `Zufall=0`. Damit ist er hinter den letzten Prozeß, der ebenfalls zur Zeit 185 startet, einzuordnen, also hinter 2. Wir erhalten den Zustand:



Der Leser kann jetzt sicherlich das weitere Verhalten des Systems selbst beschreiben. Werden jetzt die zufälligen Zeiten: `Zufall=6, 6, 0, 3, 4` gezogen, so erhalten wir den Zustand (bitte erst nachrechnen, dann vergleichen):



Man sieht an diesem Beispiel, daß es sich um einen sehr komplizierten Vorgang handelt, obgleich jeder einzelne Prozeß nur ein sehr einfaches Verhalten zeigt. Die Tatsache, daß trotz des einfachen Verhaltens einzelner Prozesse das Verhalten des gesamten Systems schon nach wenigen Schritten kaum noch nachvollziehbar ist, zeigt die Mächtigkeit dieses Werkzeugs zur Modellierung realer Systeme.

In der Makro-Version wird eine solcher Prozeß überlicherweise mit Hilfe der folgenden Makros notiert:

```
#define Phase(N) case N:
#define NextAfter(PHASE,TIME)   {   PT.ProcessPtr->Phase = PHASE;\
                               PT.ProcessPtr->Time = TIME; break; }
#define RepeatAfter(TIME)        { {PT.ProcessPtr->Time = TIME; break; }
#define Next(PHASE)              { {PT.ProcessPtr->Phase = PHASE; break; }
```

Während das erste Makro offensichtlich eine einfache Übersetzung des `case`-Konstrukts von C darstellt, sind die anderen drei Makros für die Beschreibung der Weiterschaltung eines Prozesses sehr bequem zu verwenden. Man beachte, daß die Ausführung dieser Makros auf jeden Fall die augenblickliche Phase beendet (vorausgesetzt, die Makros werden nicht in einer Schleife, einem switch usw. ausgeführt!). Alternativ ist es daher auch sinnvoll, statt des `break` ein `goto` zu verwenden, z.B.:

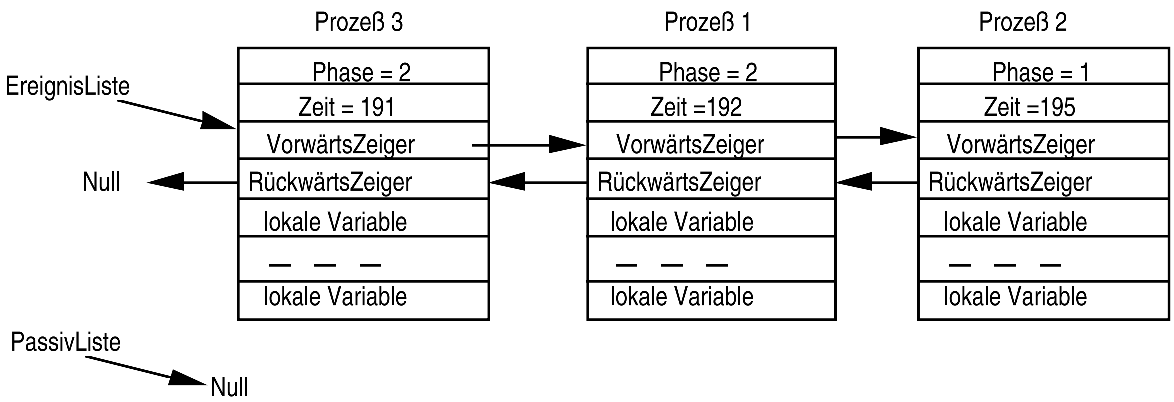
```
#define NextAfter(PHASE,TIME)   {   PT.ProcessPtr->Phase = PHASE;\
                              PT.ProcessPtr->Time = TIME; \
                              goto EndsimulationLabel; }
```

1.3 Activate, Passivate und Wait, Await

Im letzten Abschnitt wurden ablaufende Prozesse beschrieben; sie sind dadurch gekennzeichent, daß sie zu einer bestimmten Systemzeit wieder aktiviert werden, d.h. ihre nächste Phase ausführen. Daneben kann es aber auch vorkommen, daß ein Prozeß unter einer bestimmten Bedingung wieder aktiviert werden soll. Diese Bedingung wird in der Regel durch einen anderen Prozeß erzeugt; wir nehmen zunächst an, daß der andere Prozeß dem zu aktivierenden Prozeß explizit mitteilt, daß er weiterlaufen soll.

Die Funktionalität ist somit die folgende: Ein Prozeß 'passiviert' sich, wodurch er in eine eigene Warteschlange gehängt wird, die PassivQueue. Ein anderer Prozeß besitzt eine Referenz auf diesen Prozeß; unter bestimmten Umständen aktiviert dieser den passivierten Prozeß, so daß dieser weiterlaufen kann. Dazu wird er mit einer entsprechenden Zeit versehen in die Ereignisliste eingehängt.

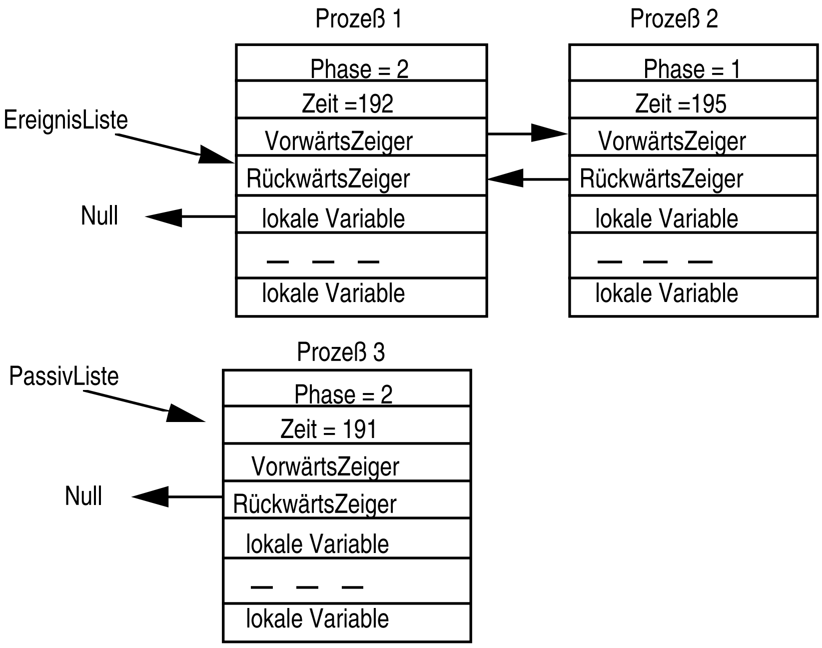
Als Beispiel sei der folgende Zustand gegeben:



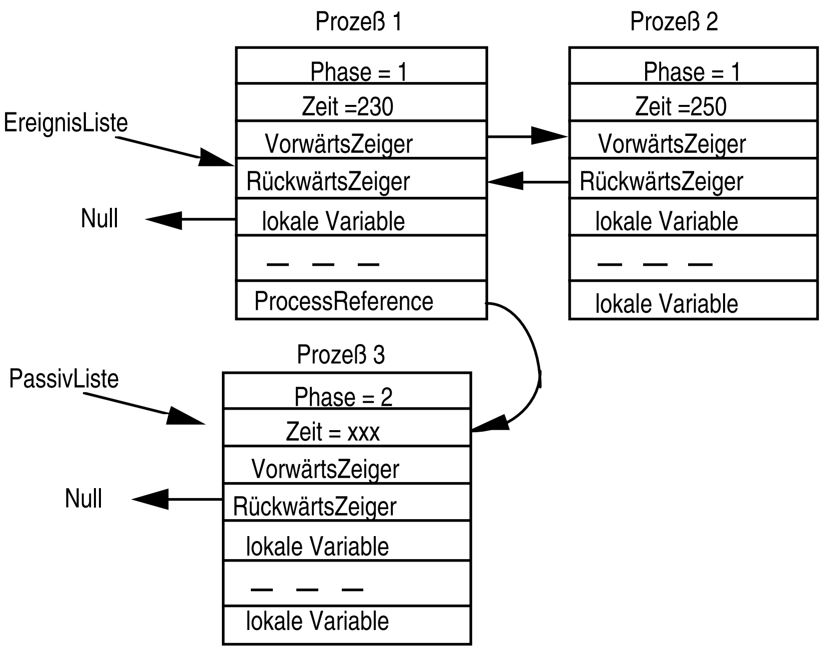
Hier ist noch kein Prozeß passiv. In der Phase 2 kann sich ein Prozeß passivieren, d.h. wir fügen den folgenden Befehl in unseren Beispielprozeß ein:

```
Process Beispiel
  Phase 1
    /* Tue etwas */
    if (Bedingung) then Activate(ProcessReference);
    THIS Phase = 2;
    THIS Zeit = SystemZeit + Zufall(1,10);
  Phase 2
    /* Tue etwas */
    if (Bedingung) then Passivate();
    THIS Phase = 1;
    THIS Zeit = SystemZeit + Zufall(0,50);
EndProcess
```

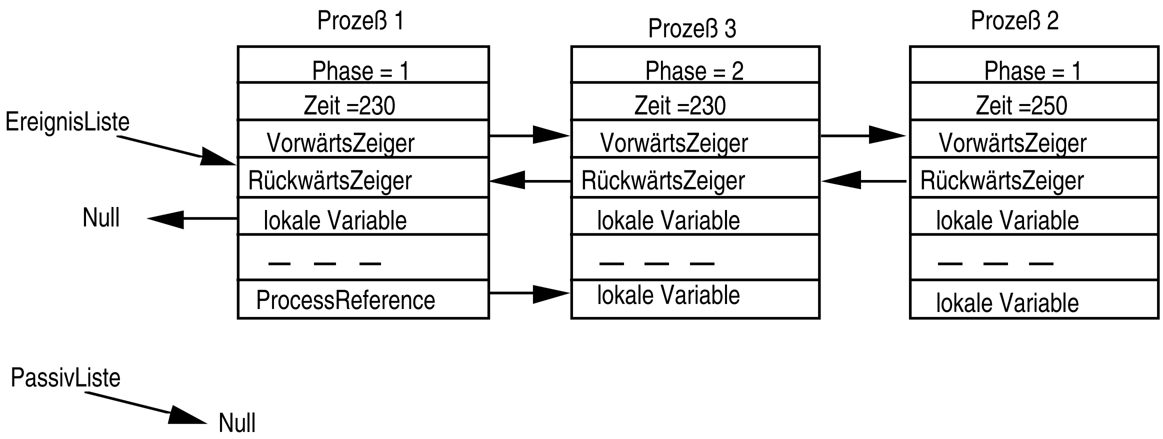
Ist die `{Bedingung)` wahr, so wird der Prozeß 3 passiviert und man erhätl den folgenden Zustand:



In diesem Zustand kann der dritte Prozeß also zunächst nicht mehr ablaufen. Sei nach einiger Zeit ein Zustand erreicht, der folgendermaßen aussieht:



und werde jetzt der Aktivierungsbefehl ausgeführt. Dazu ist es notwendig, daß der erste Prozeß in seiner lokalen Variablen `ProcessReference` eine Reference auf den dritten Prozeß enthält. Nach Ausführung des Befehls `Activate(ProcessReference)` erhalten wir den folgenden Zustand:



Man beachte, daß der aktivierte Prozeß hinter dem gerade aktiven Prozeß in die Warteschlange eingehängt wird. Dabei ist die nächste Aktivierungszeit dieses Prozesses gerade gleich der aktuellen Zeit, also der des Prozesses, der diesen aktiviert hat. Es gilt die Regel, daß ein aktivierter Prozeß stets hinter dem aktiven eingereiht wird, auch wenn mehrere Prozesse die gleiche Startzeit wie der aktive Prozeß haben.

Ein Prozeß kann sich somit in einem von mehreren Zuständen befinden. Er kann aktiviert sein, was bedeutet, daß er bei Erreichen einer bestimmten Systemzeit ausgeführt werden kann. Aktivierte Prozesse sind in der Ereignisliste eingereiht; der erste Prozeß in der Ereignisliste wird als ablaufend (oder auch aktiv) bezeichnet. Ein Prozeß kann aber auch passiviert sein, was bedeutet, daß er in der Passivliste eingereiht ist. Natürlich schließen sich diese beiden Zustände wechselseitig aus.

Ein passivierter Prozeß kann beim Übergang in den Zustand passiviert noch festlegen, welche Phase er bei der Aktivierung als erstes ausführen möchte. Außerdem kann bei aktivieren eines Prozesses diesem noch eine Startzeit mitgegeben werden, zu der er aktiviert werden soll. In diesem Falle wird der Prozeß hinter allen gleichzeitigen Prozessen in der Ereignislist eingefügt. Durch Angabe der Zeitdifferenz null kann ein Prozeß also auch hinter alle gleichzeitigen Prozessen an den Anfang in die Warteliste eingefügt werden.

Wir betrachten jetzt den Fall, daß ein Prozeß A einen Auftrag an einen anderen Prozeß B senden will. Der Prozeß B besitzt dazu eine Warteschlange: `EnterQueue`. Sobald der Prozeß B keinen Auftrag mehr bearbeiten kann, passiviert er sich:

```
Phase 3
   if (EnterQueue->Number == 0) Passivate();
   else THIS Job = Seize(EnterQueue);
   ...
Phase 4
```

Sobald Prozeß B die dritte Phase betritt überprüft er also, ob die Warteschlange `EnterQueue` leer ist; ist dieses der Fall, so passiviert sich dieser Prozeß. Wir er wieder aktiviert, so führt er die gleiche Phase ein weiteres Mal aus. Sollte sich dann ein Auftrag in seiner lokalen Warteschlange befinden, so kann der Prozeß B diesen Auftrag mit der lokalen Variable `Job` referenzieren und diese Phase weiter ausführen. Sollte die Warteschlange `EnterQueue` jedoch wieder leer sein, so passiviert sich dieser Prozeß aufs neue.

Ein anderer Prozeß, der einen Zeiger `ProcessReference` auf diesen Prozeß hat, kann diesen Prozeß wieder aktivieren, indem er den Befehl ausführt:

```
Insert(JobReference, ProcessReference->EnterQueue);
JobReference = Null;
Activate(ProcessReference);
```

Die erste Anweisung fügt den Auftrag, der durch den Zeiger `JobReference` referenziet wird, in die lokale Warteschlange des passivierten Prozesses ein. Die zweite Anweisung sorgt dafür, daß die lokale Reference auf die Auftrag gelöscht wird, damit nicht zwei verschiedene Zeiger in verschiedenen Prozessen auf den gleichen Auftrag zeigen. Der dritte Befehl aktiviert schließlich den passivierten Prozeß B wieder. Da dieser Vorgang in Wartenetzmodellen relativ häufig vorkommt, wurde dafür das sogenannte Enter-Konzept entwickelt. Es verwendet genau die gleichen Befehlsfolgen wie oben gezeigt, jedoch führt dafür Makros ein:

```
Await(THIS Job);
```

bzw.

```
Enter(JobReference, ProcessReference);
```

Der erste Befehl läßt den aufrufenden Prozeß also solange warten, bis sich ein Auftrag in seiner lokalen Warteschlange befindet. Der zweite Befehl sendet einen Auftrag `JobReference` an den Prozeß `ProcessReference`.

Eine Erweiterung erfährt dieser Befehl dadurch, daß man gleichzeitig alternativ auf das Ablaufen einer Zeit oder das Eintreffen eines Auftrags warten kann. Dieser Fall tritt z.B. in Kommunikationssystemen auf, in denen eine Paket abgesendet wird, und der sendende Prozeß eine bestimmte Zeitlang warten muß ehe eine Quittung eintrifft. Allerdings können Paket oder Quittung verloren gehen, so daß nach Ablaufen einer Zeit der sendende Prozeß wieder aktiv werden und eine entsprechende Aktion ausführen muß. Dazu eignet sich offensichtlich die folgende Befehlsfolge:

```
Phase 5
   if (ProcessQueue->EnterQueue != 0)          /* If Job in Queue: Proceed        */
     THIS Job = Seize(EnterQueue);             /* Get Job out of Queue; */
   else                                  /* But if no Job in Queue and    */
   { if (ProcessQueue->Status == Activ) /* First Time here, then  */
     { ProcessQueue->Status = Awaiting; /* Mark Process as Awaiting    */
       ProcessQueue->Zeit = ActivationTime;    /* Set next Activation Time      */
       break;                                  /* Finish this Phase             */
     } else THIS Job = Null;                   /* otherwise: Set job to Null    */
     ProcessQueue->Status == Activ;            /* Reset Status to Activ  */
   } ...                                 /* Proceed ...                */
```

Wir führen hierzu einen besonderen Standardbefehl ein, der heißen soll:

```
Phase 5
   AwaitUntil(THIS Job, ActivationTime);      /* If Job in Queue: Proceed      */
```

Offensichtlich wird hier die Wirkung erzielt, daß dieser Befehl die Ausführung des Prozesses solange unterbricht, bis entweder ein Auftrag in die lokale `EnterQueue` eingefügt wurde, oder bis die Zeit `ActivationTime` erreicht ist. Im letzten Fall steht auf der lokalen Variablen `Job` die Referenz: `Job==Null`, im ersten enthält diese Variable den Auftrag, der in die `EnterQueue` eingefügt wurde. Somit ist dieses zugleich das Kriterium, nach welchem entschieden werden kann, warum eine `AwaitUntil`-Anweisung verlassen wurde.

Es sei hier noch kurz die Funktion der `Activate`-Anweisung genauer beschrieben. Ein `Activate` auf einen passivierten Prozeß bewirkt die oben beschriebene Einfügung dieses Prozesses in die Ereigniswarteschlange. Ein `Activate` auf einen aktivierten Prozeß hat keinerlei Wirkung; der aktive Prozeß wird also auch nicht in der Ereigniswarteschlange verschoben. Ein `Activate` auf einen Prozeß im zustand $\boxed{\text{erwartend}}$ (also: `Awaiting`) hat die eben beschriebene Wirkung. Insbesondere ist zu beachten, daß dieser Prozeß beim `AwaitUntil` <u>nicht</u> in die Passivwarteschlange eingehängt wird, sondern in der Ereigniswarteschlange verbleibt, einsortiert zu der Zeit, zu der er spätestens wieder aktiv werden soll. Wird er daher früher aktiviert, so muß er jetzt weiter vorne in die Ereigniswarteschlange hinter dem ablaufenden Prozeß einsortiert werden, genau wie ein passiver Prozeß. Daher gilt also für das `Activate`:

```
Activate(ProcessRef) ...;
{ ...
   if ProcessRef->Status == Passiv || ProcessRef->Status == Awaiting)
   { Unlink(ProcessRef);
     LinkBehind(ProcessList, ProcessRef);
} }
```

Der erste Befehl `Unlink` entnimmt den Prozeß seiner aktuellen Liste (gleich welcher), und der zweite Befehl fügt ihn hinter dem aktiven Prozeß in der Ereignisliste ein.

1.4 Waiting

Wir haben bereits mehrere Zustände für einen Prozeß kennengelernt; deren englische Bezeichnung, wie sie im Programm verwendet wird, lauten: $\boxed{\texttt{Active}}$, $\boxed{\texttt{Passive}}$ oder $\boxed{\texttt{Awaiting}}$. Darüber hinaus gibt es weitere Zustände für einen Prozeß.

So kann ein Prozeß statt auf die Aktivierung durch einen anderen Prozeß auf eine beliebige Bedingung warten. Das bedeutet, daß jedesmal nach Änderung des Systemzustands der Prozeß aktiviert werden muß, die Bedingung überprüfen muß, und, falls diese noch immer nicht 'wahr' ist, muß er wieder in den entsprechenden Zustand übergehen. Dieser Zustand wird als $\boxed{\text{wartend}}$ bzw. englisch: $\boxed{\texttt{Waiting}}$ bezeichnet. Dazu gibt es eine weitere Warteschlange, die `WaitingList`, in die ein Prozeß nach Ausführung des Befehls `WaitFor(Bedingung)` eingehängt wird, wenn die Bedingung falsch ist. Als nächstes wird dann eine Phase eines aktivierten Prozesses ausgeführt. Da diese den Systemzustand geändert haben könnte, müssen dann wieder die wartenden Prozesse ausgeführt werden usw.

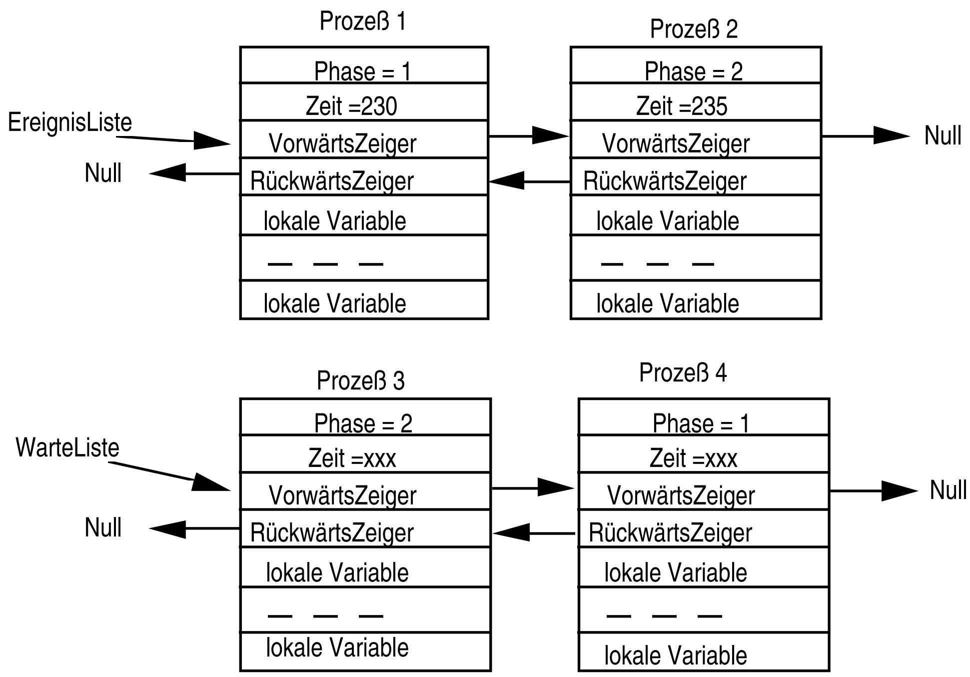
Um einen Prozeß auf eine Bedingung warten zu lassen, muß die folgende Anweisung ausgeführt werden:

```
Phase 7
   if ( ! Bedingung ) Waiting();
   ...
```

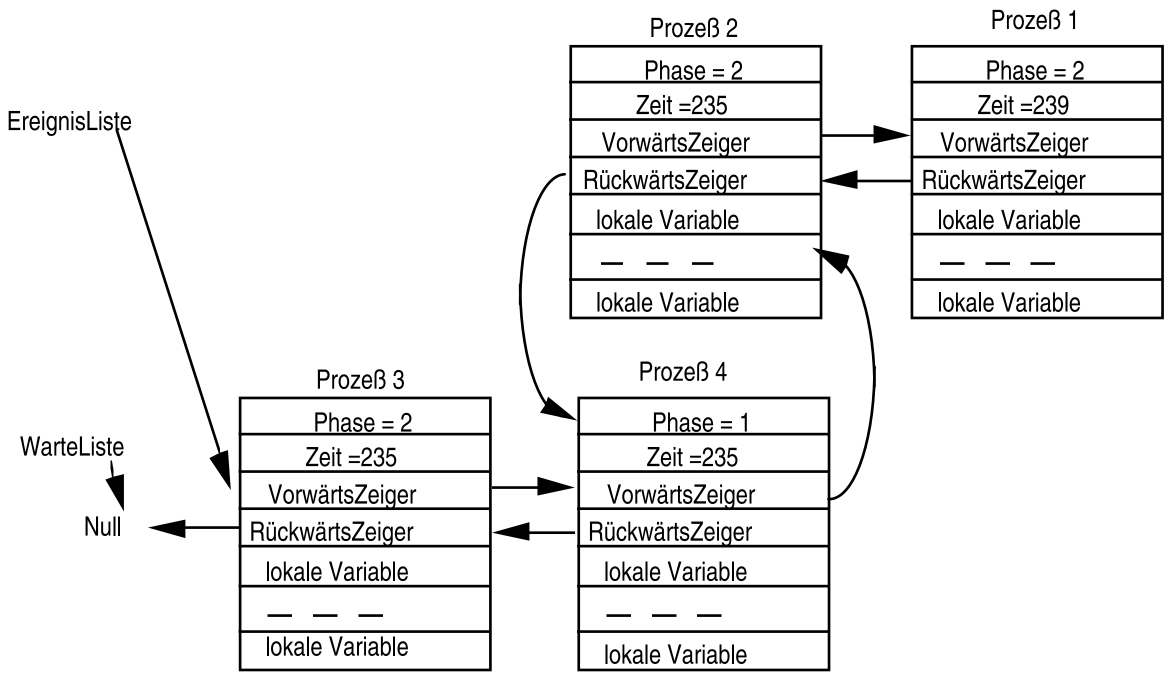
Ist die Bedingung falsch, so wird die Funktion `Waitng` aufgerufen. Diese hängt den aktiven Prozeß in eine besondere Warteschlange, die WarteListe. Ist die Bedingung jedoch wahr, so läuft der Prozeß so weiter wie in einer üblichen Phase. Statt der obigen Bedingung schreiben wir im folgenden auch:

```
Phase 7
   WaitFor(Bedingung );
   ...
```

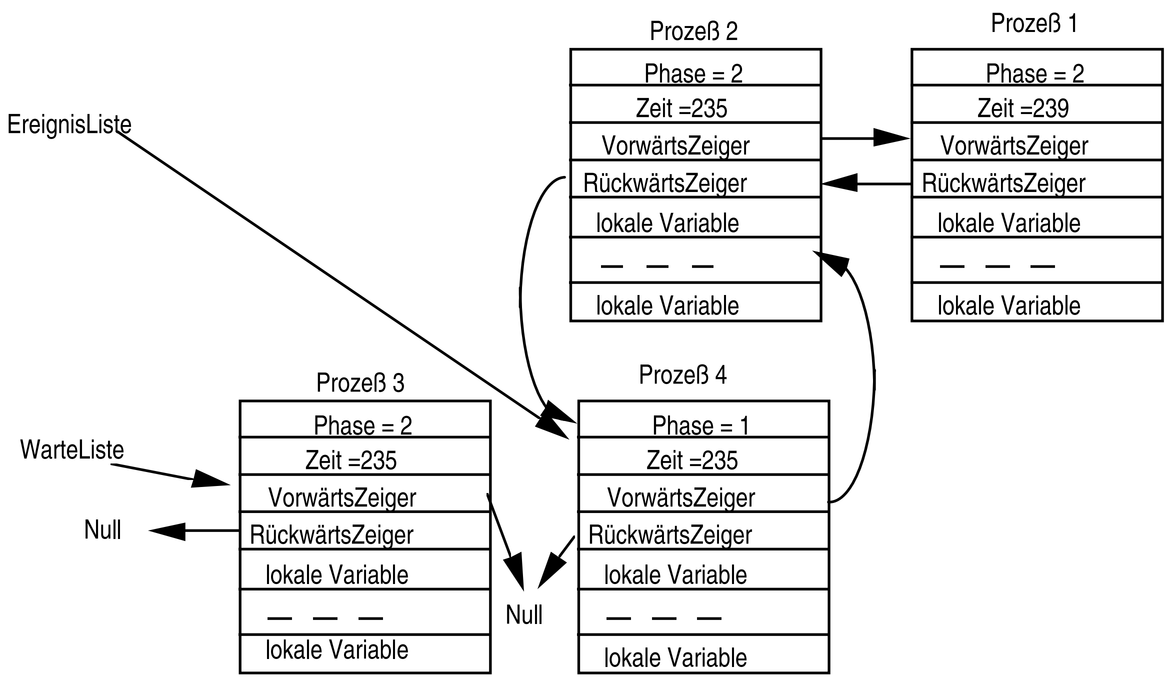
Im folgenden Beispiel seien zwei Prozesse wartend, zwei weitere Prozesse aktiv:



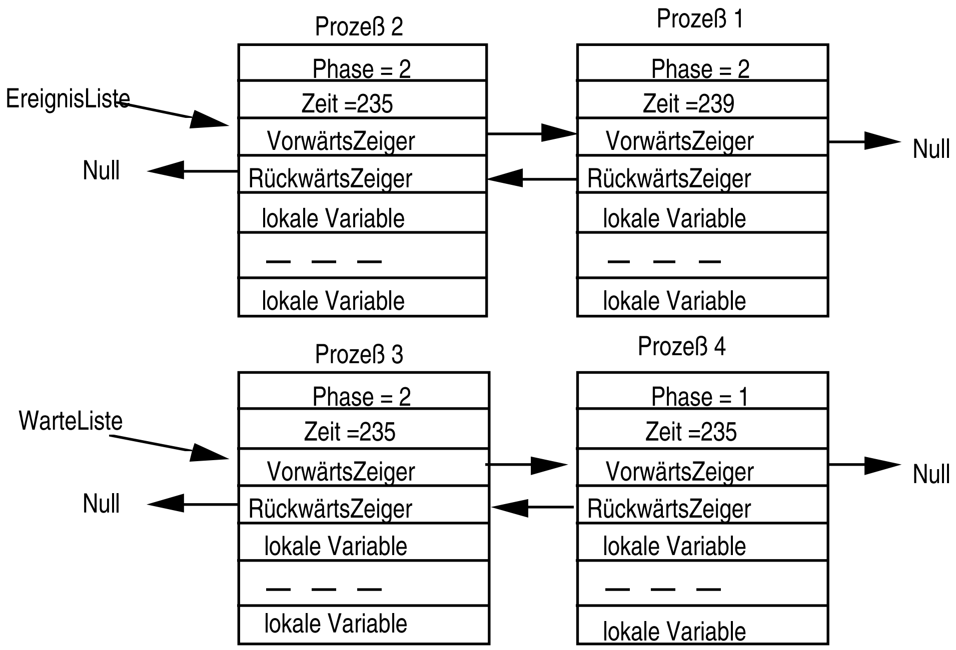
Wird der aktivierte Prozeß 1 ausgeführt, so erhalten wir den Zustand:



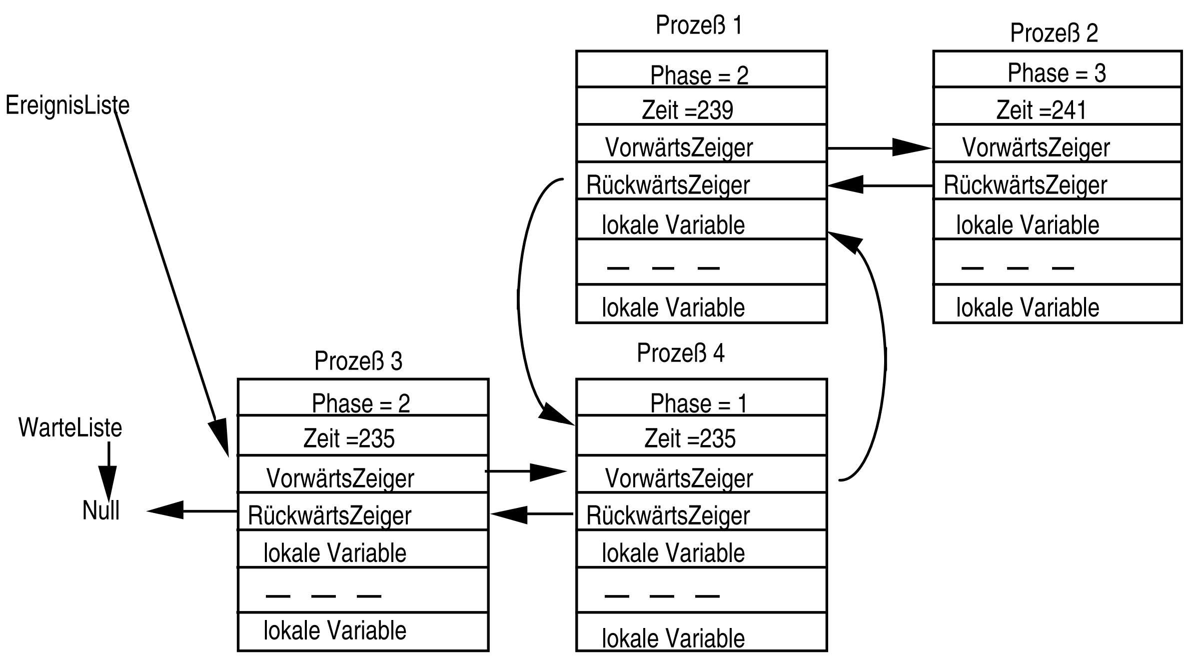
Wird jetzt der erste Prozeß 3 in der Ereignisliste ausgeführt und ist dessen Wartebedingung weiterhin falsch, so erhalten wir:



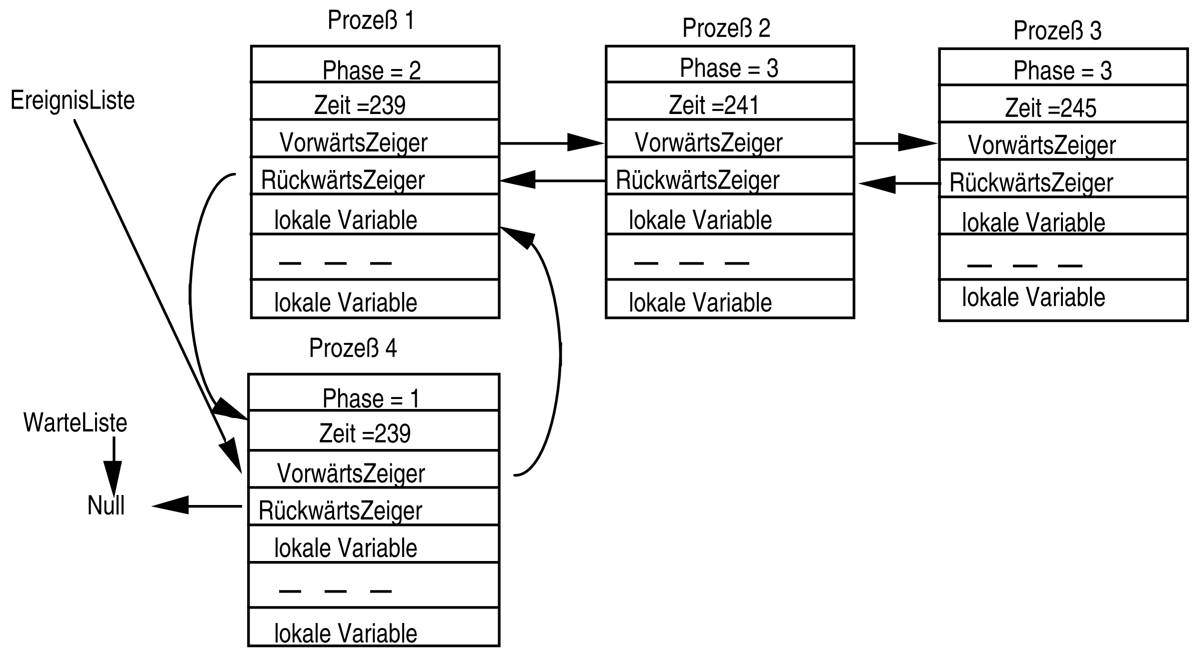
Werde jetzt der Prozeß 4 in der Ereignisliste ausgeführt und sei seine Wartebedingung ebenfalls falsch, so folgt:



und es kann der nächste aktive Prozeß ausgeführt werden. Nach dessen Ausführung haben wir wieder den Zustand:



Wieder müssen die beiden wartenden Prozesse ausgeführt werden, um die Bedingung zu testen, aufgrund derer sie warten. Werde jetzt der erste wartende Prozeß 3 ausgeführt und sei seine Bedingung wahr, d.h. er läuft seine Phase zu Ende. Dann erhalten wir den Zustand:



Dieser Prozeß wird also, solange er nicht wieder einen `WaitFor`-Befehl ausführt, wie eine normaler aktivierter Prozeß behandelt.

Man sieht an dieser Darstellung, daß für jede aktive Phase sämtliche wartenden Prozesse ausgeführt werden. Das ist natürlich sehr aufwendig, so daß diese Konstruktion eigentlich nur verwendet werden soll, wenn andere Kostruktionen nicht möglich sind. Allerdings ist dieses eine sehr elegante und sichere Lösung. Kommt es somit nicht primär auf die Laufzeit an, so ist dieses Konstrukt auf jeden Fall vorteilhaft.

1.5 Processor-Sharing

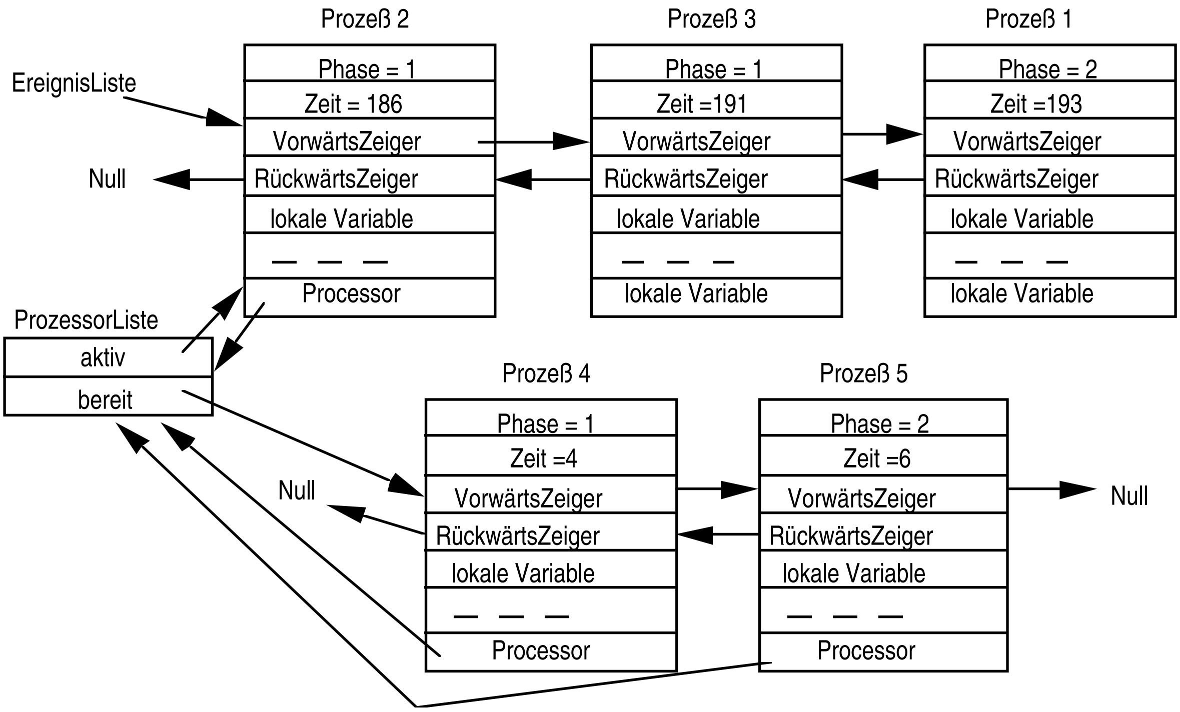
Bisher wurden Prozesse als selbstständige Einheiten betrachtet, die ein weitgehend eigenes Leben führen und deren zeitliches Verhalten durch andere Prozesse praktisch nicht beeinflußt werden kann. In manchen Anwendungen ist es jedoch sinnvoll, daß die Prozesse nicht unabhängig voneinander arbeiten, sondern daß mehrere Prozesse sich die Rechenleistung eines Prozessors teilen, wie im Falle eines Mehrprozeßsystems. Gibt es somit viele Prozesse, so wird jeder Prozeß nur langsamer vorankommen, bei wenigen Prozessen schneller.

1.5.1 Phasenorientiertes Processor-Sharing

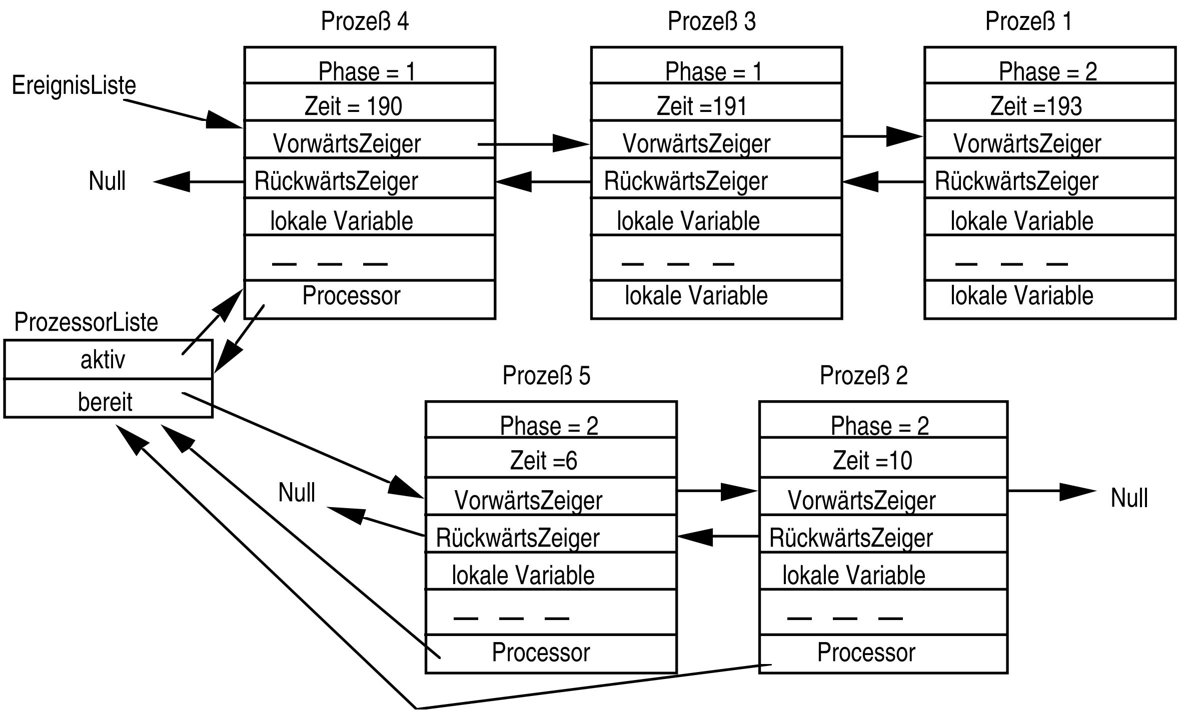
In der einfachsten Varianten dieser Konstruktion wird eine Prozeß mit dem Beginn der Ausführung seiner Phase solange warten, bis alle anderen Prozesse, die zufälligerweise vor ihm eingereiht wurden, ihre augenblicklichen Phasen abgearbeitet haben. Dieses kann folgendermaßen realisiert werden:

Alle Prozesse, die auf dem gleichen Prozessor laufen sollen, werden in einer Liste verwaltet, deren erstes Element der gerade vom Prozessor bediente Prozeß ist; alle anderen Prozesse befinden sich außerhalb der Ereignisliste in einer ProzessorListe. Sobald der erste Prozeß aktiv wird, wird zunächst der nächste Prozeß in der zugehörigen ProzessorListe entsprechend seiner Aktivierungszeit in die Ereignisliste eingehängt; sodann wird der aktive Prozeß bearbeitet. Wenn er in der nächsten Phase wieder von dem Prozessor bearbeitet werden soll, muß er sich in der zugehörigen ProzessorListe als letztes einreihen. Sobald der jetzt neu aktivierte Prozeß abgearbeitet wird, wiederholt sich dieser Vorgang analog. Ist die ProzessorListe leer, so kann sich ein zu aktivierender Prozeß direkt in die Ereignisliste einreihen, muß jedoch seine Zugehörigkeit zu der ProzessorListe kennzeichnen.

Um dieses zu realisieren, muß jeder Prozeß eine lokale Variable setzen, aus der die Zugehörigkeit des Prozesses zu einem bestimmten Prozessor hervorgeht. Da diese vom Programm geändert werden kann, kann ein Prozeß während seines Lebens nacheinander verschiedenen Prozessoren zugeordnet werden. Diese Variable enthalte (zumindest) zwei Zeiger auf Prozesse:

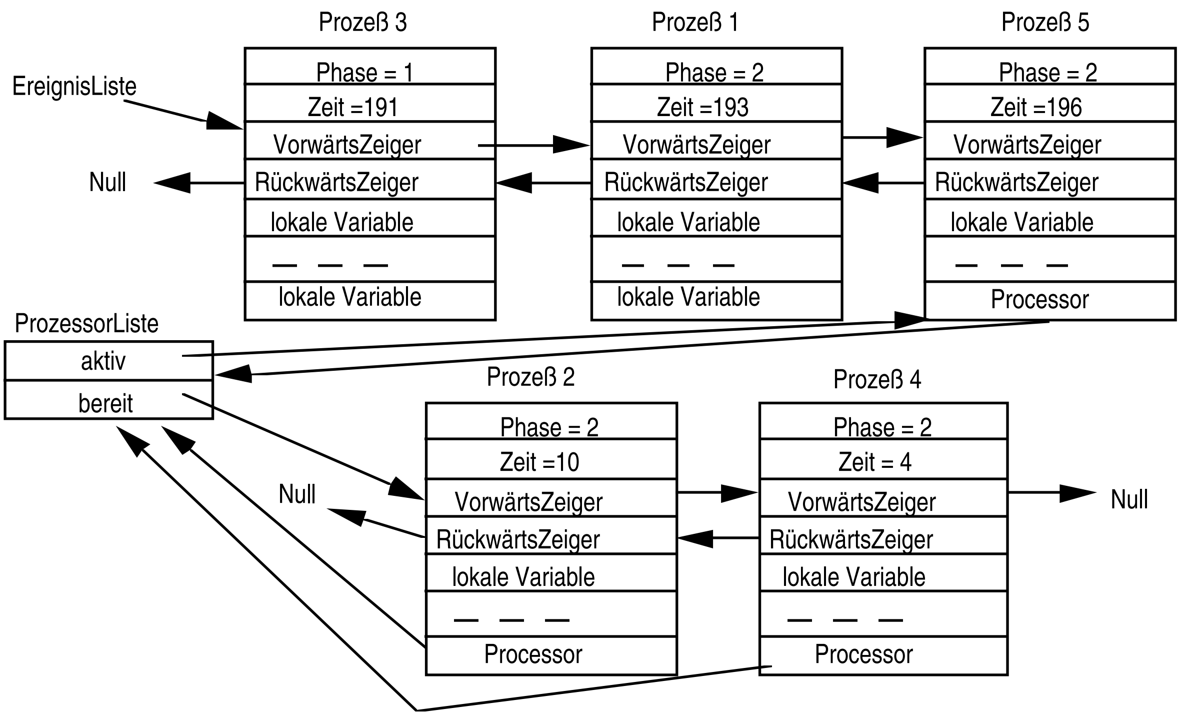


Die Prozesse 2, 4 und 5 befinden sich hier in der ProzessorListe, wobei Prozeß 2 gerade aktiviert und ablaufend ist; die anderen Prozesse aus der ProzessorListe sind bereit, abzulaufen, aber gerade nicht aktiviert. Die Zeiten in den Prozessen geben jetzt nicht die Aktivierungszeiten, sondern die Bearbeitungszeiten für die jeweiligen Phasen an. Prozeß 2 will wieder nach 10 Zeiteinheiten aktiviert werden. Es entsteht der folgende Zustand:



Da der Prozeß 4 nach 4 Zeiteinheiten aktiviert werden soll, ist diese relativ zur Startzeit, also zur Zeit 186 zu berechnen. Er wird somit als erster in die Ereignisliste eingefügt, während der

Prozeß 2 als letztes in die ProzessorListe eingefügt wird. Nachdem dessen Phase ausgeführt wird, wird er wieder in die ProzessorListe übernommen und der Prozeß 5 zur Zeit 190+6 in die EreignisListe einsortiert. Wir erhalten:



Sobald Prozeß 5 abgearbeitet ist, wird wieder Prozeß 2 in die EreignisListe übernommen, usw. Um dieses zu realisieren, werden die folgenden Programmstücke benötigt. Beim Eintritt in die Schleife ist zunächst der erste Prozeß aus der ProzessorList in die EreignisListe einzusortieren, nachdem dessen Aktivierungszeit bestimmt wurde; gibt es keinen weiteren Prozeß in der ProzessorListe, so ist zumindest `aktiv` auf `Null` zu setzen.

```
if (PT.ProcessPtr -> Processor != Null) /* Process is in ProcessorList */
{  if (PT.ProcessPtr -> Processor -> bereit != Null)
   { Ptr = Seize(PT.ProcessPtr -> Processor -> bereit);
     Ptr -> Time = SystemTime + Ptr -> Time;
     LinkFirst(Ptr, PT.ProcessPtr);
     SortProcess();
     Ptr -> Processor -> aktiv = Ptr;
   } else PT.ProcessPtr -> Processor -> aktiv = Null;
}
```

Beim Austritt aus der Schleife ist ein Prozess entweder in die ProzessorListe einzufügen, oder, falls kein aktivierter Prozeß aus dieser ProzessorListe exisitert, kann der Prozeß in der EreignisListe verbleiben:

```
  if (PT.ProcessPtr -> Processor != Null) /* Process is in ProcessorList */
  {  if (PT.ProcessPtr -> Processor -> aktiv != Null)
     { Ptr = PT.ProcessPtr;
       Ptr -> Time = Ptr -> Time - SystemTime;
       UnlinkFirst(PT.ProcessPtr);
       InsertLast(Ptr, Ptr -> Processor -> bereit);
     } else SortProcess();
  } else SortProcess();
```

Die Prozedur `SortProcess` wird also ausgeführt, wenn entweder der aktive Prozeß zu keiner ProzessorListe gehört (so daß die bisherige Funktionalität gewahrt bleibt), oder wenn der aktive Prozeß der einzige in der ProzessorListe ist; sie darf jedoch nicht ausgeführt werden, wenn es mehr als einen Prozeß in der ProzessorListe gibt.

Offenbar kann ein Prozeß aus einer ProzessorListe nur dann aktiviert werden, wenn ein anderer Prozeß aus der ProzessorListe aktiv ist. Wenn sich ein Prozeß passiviert, ist somit gleichzeitig eine Aktivierung des entsprechenden Prozesses in der ProzessorListe durchzuführen. Außerdem muß ein passiver Prozeß, der wieder aktiviert wird, mit der Zeit 0 in die ProzessorListe eingefügt werden, wenn es einen aktiven Prozeß aus dieser ProzessorListe gibt; ansonsten muß er in die EreignisListe eingefügt werden:

```
Activate(Process) struct ProcessorTyp Process;
{  if (PT.ProcessPtr -> Processor == Null /* Process not in ProcessorList*/
      || PT.ProcessPtr -> Processor -> aktiv == Null /* No activ Process */
   { /* Link Process behind activ Process in ProcessorList */ }
   else { Unlink(Process);
          Process -> Time = 0;
          InsertLast(Ptr, Ptr -> Processor -> bereit);
}      }
```

Man sieht, daß das Zusammenspiel mit anderen Funktionen u.U. zu einigen Komplikationen führt; auf jeden Fall muß dieses jeweils genau durchdacht werden.

1.5.2 Zeitscheibenorientiertes Processor-Sharing

Eine andere Möglichkeit, das Processor-Sharing zu realisieren ist zwar realistischer, aber deutlich aufwendiger. Denn in der Regel wird jeder Prozeß nur eine bestimmte **Zeitscheibe** (*time slice*) an Rechenzeit zugeteilt bekommen. Danach wird er unterbrochen und der nächste Prozeß kommt an die Reihe. Dieses kann natürlich auch mit diesem Simulationskonzept realisiert werden, wobei zu Beginn eines Schleifendurchlaufs zunächst der nächste aktive Prozeß gesucht werden muß. Dazu müssen alle Prozesse in allen Prozessoren in der gegebenen Reihenfolge um jeweils eine Zeitscheibe dekrementiert werden; gleichzeitig wird die Simulationszeit entsprechend weitergeschaltet. Sobald die Bedienzeit eines Prozeß in einem der Prozessoren kleiner oder gleich null wird, muß die SystemZeit entsprechend gesetzt und der Prozeß ausgeführt werden. Gleichzeitig sind natürlich auch die Prozesse in der EreignisListe entsprechend zu überwachen.

Da die Implementierung dieser Strategie, die auch noch um Prioritäten zwischen Prozessen erweitert werden kann, recht aufwendig ist, soll sie hier nicht im Detail implementiert werden.

1.5.3 Processor-Sharing mit eigenem Scheduler

Die bisher vorgeschlagenen Ansätze gehen jeweils davon aus, daß das System ein Prccessor-Sharing-Konzept zur Verfügung stellt. Dabei entscheidet das System, ob ein Zeitscheiben-, Unterbrechungs- Prioritätenkonzept usw verwendet wird. Das ist sehr speziell und sollte nach Möglichkeit vermieden werden, um die Flexibilität zu erhöhen. Daher soll hier ein allgemeineres Schedulingkonzept entwickelt werden.

Ein Scheduler ist ein eigener Prozeß, wie die anderen Prozesse in einem Simulationsprogramm auch. Er bearbeitet jedoch in der Regel nicht Aufträge, sondern Prozesse. Dazu besitzt er eine eigene Warteschlange, die **ScheduleListe**. Diese hat zwei Zeiger, `ScheduleFirst` und `ScheduleLast`, welche eine lineare Liste der geschedulten Prozesse enthält. Sobald diese Liste leer ist, erhält `ScheduleFirst` den Wert `Null` (=0), so daß der Scheduler selbst entscheiden kann, ob er noch weiterarbeiten kann oder pausieren muß.

Es gibt im wesentlichen zwei zusätzliche Befehle. Der erste Befehl wird von einem Prozeß aufgerufen, der durch einen Scheduler bearbeitet werden möchte. Er lautet:

```
ScheduleNextAfter(SchedulerName, PhaseName, ServiceTime)
```

Dieser Befehl

- setzt in dem aktuellen Prozeß die Phase auf `PhaseName`,
- setzt die Zeit auf die ServiceTime
- fügt den aktiven Prozeß in die Scheduleliste des Schedulerprozesses mit dem Namen SchedulerName ein
- aktiviert den Prozeß SchedulerName, falls dieser passiv ist; sonst läuft dieser unverändert weiter.

Der Scheduler ist ein Prozeß, welcher nach einer von ihm selbst gewählten Zeit wieder aktiv wird. Sobald dieses geschieht, kann er entweder einen weiteren Prozeß aus seiner eigenen Prozeßliste bearbeiten, oder er aktiviert einen solchen Prozeß durch `Deschedule()`. Letzteres bedeutet, daß der aktivierte Prozeß seine nächste Phase ausführt, da er in die Zeitliste eingefügt wurde. Ist die Phase dieses Prozesses beendet, so kann er sich entweder wieder in die Zeitliste einfügen oder er wird wieder von einem Scheduler bearbeitet.

Der Befehl `Deschedule()` wird nur vom Scheduler ausgeführt. Er fügt den ersten Prozeß der Scheduleliste, auf den also `ScheduleFirst` zeigt, in die Zeitliste ein. Die Startzeit des Prozesses wird auf die aktuelle `SystemTime` gesetzt, so daß dieser Prozeß als nächstes ausgeführt wird.

Mit diesen Mechanismen kann der Programmierer einen Scheduler konzipieren, der seine Ressourcen auf mehrere Prozesse aufteilt. Da dieses durch zwei einfache zusätzliche Befehle erreicht wird, ist dieses ein sehr effizientes Konzept. Eine Implementierung des phasenorientierten Schedulings könnte beispielsweise folgendermaßen aussehen.

```
  Phase(1)
    If ScheduleFirst == Null Then
      PASSIVATE;
    Else
      NextAfter(2, ScheduleFirst -> Time );
    IfEnd;
  Phase(2)
    Deschedule();
    NextAfter(1,0.0);
...
```

Um das zeitscheibenorientiere Scheduling zu verwenden, kann man folgendermaßen vorgehen:

```
  Phase(1)
    If ScheduleFirst == Null Then
      PASSIVATE;
    Else
      NextTimeSlice = MIN (ScheduleFirst -> Time , TimeSlice);
      ScheduleFirst -> Time -= NextTimeSlice ;
      NextAfter(2, NextTimeSlice );
    IfEnd;
  Phase(2)
    If ScheduleFirst -> Time == 0 Then
      Deschedule();
    Else
      ScheduleNext();
    IfEnd;
  NextAfter(1,0.0);
...
```

Hier ist ScheduleNext eine Funktion, die die ScheduleListe des aktuellen Prozesses umsortiert, indem sie den ersten Prozeß an die letzte Stelle bringt, und entsprechend den zweiten an die erste. Sie könnte etwa folgendermaßen implementiert sein:

```
ScheduleNext()
  { ProcessType proc;
    IF EventList -> ScheduleFirst -> next != Null Then
      /* Es gibt mehr als einen Prozeß in der Ereignisliste */
      EventList -> ScheduleLast -> next = EventList -> ScheduleFirst;
      EventList -> ScheduleLast = EventList -> ScheduleFirst;
      EventList -> ScheduleFirst = EventList -> ScheduleFirst -> next;
      EventList -> ScheduleLast -> next = Null;
    IfEnd,
  }
```

Darüber hinaus können auf diese Weise beliebig viele Scheduler erzeugt werden, so daß entsprechend flexibel auf unterschiedliche Anforderungen reagiert werden kann. Natürlich kann ein Prozeß immer nur einem Scheduler zugeordnet werden.

Durch diese Funktionalität wird auch das Werkstattsimulations-Konzept (Transaktionskonzept) direkt in C-SIM eingeführt, welches beispielsweise GPSS zugrundeliegt. Allerdings ist hier gleichzeitig auch die prozeßorientierte Simulation verfügbar, so daß dieses Konzept sehr viel allgemeiner ist (zumal der Scheduler keine Standardprozedur ist, sondern vom Programmierer selbst gestaltet werden kann).

2 Zeitkontinuierliche Simulation

Die bisher betrachteten Verfahren zur Simulation waren zeitdiskret in dem Sinne, daß jeweils zu einem Zeitpunkt ein Zustandswechsel stattfindet und dann wieder eine gewisse Zeit bis zum nächsten Zustandswechsel gewartet wird. Dabei ist diese Zeitspanne in der Regel im Sinne des Modells eine endliche Zeitspanne. In verschiedenen Anwendungen wären es aber nützlich, auch zeitkontinuierliche Simulation in dem Sinne zuzulassen, daß sich der (kontinuierliche) Zustand eines Modells in einem gewissen Zeitintervall kontinuierlich verändert. Dieses wird als zeitkontinuierliche Simulation bezeichnet. In diesem Abschnitt beschreiben wir die Möglichkeiten, die C-Sim bietet, dieses durchzuführen und welche Unterstützung C-Sim hierbei gewährt.

Werden zeitkontinuierliche Systeme beschrieben, so muß die Änderung des Zustands in der Zeit bekannt sein. Dieses wird häufig durch Differentialgleichungen ausgedrückt. Bekanntlich beschreibt

$$\lim_{\Delta x \to 0} \frac{y(x+\Delta x) - y(x)}{\Delta x} = \frac{dy}{dx} = y',$$

die relative Änderung des Funktionswerte y mit dem Definitionswert x, indem (geometrisch betrachtet) die Steigung der Funktion y im Punkte x angegeben wird. Man kann jetzt häufig die Eigenschaften einer Funktion vollständig beschreiben, indem nur die Änderungen eines Funktionswerts angegeben werden, und mit geeigneten Anfangsbedingungen läßt sich dann die Funktion y eindeutig ermitteln. Dieses Vorgehen wird als Lösen von Differentialgleichungen bezeichnet.

Eine numerische Lösung von Differentialgleichungen erhält man, indem man von den Anfangswerten ausgehend die Änderungen der Funktion in diskreten Schritten berechnet. Dieses Vorgehen kann natürlich sehr ungenau sein, wenn die Schritte nicht klein genug sind und die Berechnung des nächsten Wertes nicht sorgfältig durchgeführt wird. Beide Problematiken sollen hier kurz angerissen werden.

2.1 Das Eulersche Vorwärtsverfahren

Eine Differentialgleichung beschreibt eine Funktion

$$\begin{aligned} y\colon\ & \Re \to \Re \\ & t \mapsto y(t). \end{aligned}$$

Wir verwenden hier die (unabhängige) Variable t, um die Abhängigkeit von der Zeit deutlicher zu machen. Wird eine Differentialgleichung erster Ordnung mit Anfangswert $y(t_0) = y_0$ in die Form

$$\frac{dy}{dt} = f(t, y)$$

gebracht, so kann dieses interpretiert werden als würde die Ableitung dy/dt aus den Werten t und y ermittelt werden. Betrachten wir jetzt statt der Differentiale dy und dt endliche Differenzen Δy und Δt, so läßt sich mit der üblichen geometrischen Interpretation der nächste Punkt direkt berechnen durch die Formel

$$y(t+\Delta t) = y(t) + \Delta y_{12} = y(t) + \Delta t \cdot f(t, y(t)).$$

Das heißt, daß die Steigung in (t, y) berechnet und mit der Zeitdifferenz Δt multipliziert wird, um dann nach der üblichen geometrischen Interpretation auf die Differenz zum nächsten Punkt $y(t+\Delta t)$ zu kommen. Dieses sehr direkte Verfahren wird als „Eulersches Verfahren" oder „Eulersches Vorwärtsverfahren" oder auch als „Tangentenverfahren" bezeichnet. Es hat den Vorteil, sehr einfach zu sein. Man kann zeigen, daß der Fehler je Schritt durch Δt^2 beschränkt ist, während der Fehler über ein endliches Intervall mit mehreren Schritten durch Δt beschränkt ist; letzteres wird auch als „globaler Abbruchfehler" bezeichnet, während ersteres „lokaler Abbruchfehler" genannt wird.

2.2 Verbesserte Eulerformel

Eine Verbesserung dieses Verfahrens erhält man, indem man statt nur die Steigung im Punkt $(t, y(t))$ zu berücksichtigen auch noch die Steigung im Punkt $(t+\Delta t, y(t+\Delta t))$ berücksichtigt und in der Regel den Mittelwert aus beiden bildet. Man erhält

$$y(t+\Delta t) = y(t) + \Delta t \cdot \frac{f(t, y(t)) + f(t+\Delta t, y(t+\Delta t))}{2}.$$

Da der Wert $y(t+\Delta t)$ in der Regel nicht einfach bestimmt werden kann, wird für diesen der eulersche Wert verwende, also

$$y(t+\Delta t) = y(t) + \Delta t \cdot \frac{f(t, y(t)) + f(t+\Delta t, y(t) + \Delta t \cdot f(t, y(t)))}{2}.$$

Dieses wird als verbesserte Eulerformel oder als „Heumformel" bezeichnet. Ihr Fehler je Schritt ist durch Δt^3 beschränkt ist, während der globale Abbruchfehler durch Δt^2 beschränkt ist.

2.3 Runge-Kutta-Verfahren

Geht man davon aus, daß der Fehler dieser Verfahren durch die Verwendung einer Näherung der Steigung im Intervall $[t, t+\Delta t]$ entsteht, so müßte diese lediglich verbessert werden, um ein verbessertes Verfahren zu erhalten. Das Runge-Kutta-Verfahren verbessert dieses, indem die Steigungen im Intervall $[t, t+\Delta t]$ an mehreren Stellen berechnet und über diese mittels der sog. Simpson-Regel integriert wird, d.h.

$$y(t+\Delta t) = y(t) + \frac{\Delta t}{6} \cdot \left(k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4\right)$$

mit

$$k_1 = f\big(t, y(t)\big),$$
$$k_2 = f\big(t + \Delta t/2, y(t) + \Delta t \cdot k_1/2\big),$$
$$k_3 = f\big(t + \Delta t/2, y(t) + \Delta t \cdot k_2/2\big),$$
$$k_4 = f\big(t + \Delta t, y(t) + \Delta t \cdot k_3\big).$$

Dieses wird auch als Runge-Kutta-Verfahren vierter Ordnung bezeichnet, da vier Zwischenwerte berechnet werden (Es gibt entsprechend auch Runge-Kutta-Verfahren zweiter und dritter oder auch höherer als vierter Ordnung). Dieses Verfahren besitzt einen lokalen Abbruchfehler, der durch Δt^5 beschränkt ist, während der globale Abbruchfehler durch Δt^4 beschränkt ist.

2.4 Mehrschrittige Verfahren

Eine weitere Möglichkeit, die mittlere Steigung im Intervall $[t, t+\Delta t]$ genauer zu approximieren ist durch die numerische Integration nach Simpson gegeben.

2.4.1 Interpolation mit Polynomen höherer Ordnung

Sind drei Stützstellen einer Funktion g bekannt, so läßt sich die Funktion zunächst durch ein Polynom zweiten Grades approximieren und dann das Integral unter diesem Polynom bestimmen. Man erhält für das Polynom

$$g(x) = \frac{x^2}{2 \cdot \Delta^2} \cdot (u_0 - 2 \cdot u_1 + u_2) + \frac{x}{2 \cdot \Delta} \cdot (u_2 - u_0) + u_1$$

so daß folgt: $g(-\Delta) = u_0$, $g(0) = u_1$, und $g(\Delta) = u_2$. Integration von 0 bis Δ ergibt

$$\int_0^{\Delta} g(x)\, dx = \frac{\Delta}{12} \cdot (-u_0 + 8 \cdot u_1 + 5 \cdot u_2).$$

Wird dieses durch Δ geteilt, so erhalten wir für die mittlere Steigung im Bereich $[0, \Delta]$

$$\overline{u} = \frac{-u_0 + 8 \cdot u_1 + 5 \cdot u_2}{12}.$$

Summation von $\Delta \cdot \overline{u}$ and $\Delta \cdot \overline{u}$, wobei u_0 und u_2 vertauscht werden, ergibt

$$\overline{U} = \Delta \cdot \frac{-u_0 + 8 \cdot u_1 + 5 \cdot u_2}{12} + \Delta \cdot \frac{-u_2 + 8 \cdot u_1 + 5 \cdot u_0}{12} = \Delta \cdot \frac{u_0 + 4 \cdot u_1 + u_2}{3},$$

die Simpson-Regel. Dieses läßt sich verallgemeinern. Sind drei Punkte (x_0, y_0), (x_1, y_1), und (x_2, y_2) gegeben, wobei $x_0 < x_1 < x_2$, dann erhalten wir das Polynom zweiten Grades, wobei $g(x_0) = y_0, g(x_1) = y_1, g(x_2) = y_2$,

$$g(x)=\frac{(x-x_1)\cdot(x-x_2)}{(x_0-x_1)\cdot(x_0-x_2)}\cdot y_0+\frac{(x-x_0)\cdot(x-x_2)}{(x_1-x_0)\cdot(x_1-x_2)}\cdot y_1+\frac{(x-x_0)\cdot(x-x_1)}{(x_2-x_0)\cdot(x_2-x_1)}\cdot y_2 \cdot,$$

Wir können mit dieser Formel die Fläche im Intervall (x_1, x_2) oder das Minimum im Intervall (x_0, x_2) bestimmen, was für einige Anwendungen nützlich ist. Eine weitere Anwendung ist die Extrapolation zum Punkt Δ. Die Berechnung ist in der Regel sehr einfach, wenngleich etwas umständlich. Deshalb sollen hier nur die Ergebnisse angegeben werden.

Sei $(-\Delta, y_0)$, $(0, y_1)$, und (Δ, y_2) gegeben, so ist es nützlich, $(2\cdot\Delta, y_3)$ zu finden. Wir erhalten

$$g(2\cdot\Delta)=y_0+3\cdot(y_2-y_1).$$

Dieses kann als erste Approximation an den nächsten Funktionswert aufgefaßt werden.

Die Fläche im Intervall $[\Delta, 2\cdot\Delta]$ kann durch die folgende Formel berechnet werden:

$$\int_{\Delta}^{2\cdot\Delta} g(x)\,dx=\frac{\Delta}{12}\cdot(5\cdot y_0-16\cdot y_1+23\cdot y_2).$$

Mit den Abkürzungen

$$R_0=\frac{y_0}{(x_0-x_1)\cdot(x_0-x_2)}.\quad R_1=\frac{y_1}{(x_1-x_0)\cdot(x_1-x_2)},\quad R_2=\frac{y_2}{(x_2-x_0)\cdot(x_2-x_1)},$$

erhält man die Formel

$$g(x)=R_0\cdot(x-x_1)\cdot(x-x_2)+R_1\cdot(x-x_0)\cdot(x-x_2)+R_2\cdot(x-x_0)\cdot(x-x_1),$$

und einen Extremwert an der Stelle

$$x_{\min\max}=\frac{R_0\cdot(x_1+x_2)+R_1\cdot(x_0+x_2)+R_2\cdot(x_0+x_1)}{2\cdot(R_0+R_1+R_2)}.$$

Ähnlich wie für ein Polynom zweiten, läßt sich auch für eines dritten Grades h eine entsprechende Approximation finden, wenn vier Stützstellen bekannt sind. Für die Flächen in den jeweiligen Abschnitten erhalten wir

$$\int_{0}^{\Delta} h(x)\,dx=\frac{\Delta}{24}\cdot(9\cdot y_0+19\cdot y_1-5\cdot y_2+y_3).$$

$$\int_{\Delta}^{2\cdot\Delta} h(x)\,dx=\frac{\Delta}{24}\cdot(-y_0+13\cdot y_1+13\cdot y_2-y_3).$$

$$\int_{2\cdot\Delta}^{3\cdot\Delta} h(x)\,dx = \frac{\Delta}{24}\cdot\left(y_0 - 5\cdot y_1 + 19\cdot y_2 + 9\cdot y_3\right).$$

$$\int_{2\cdot\Delta}^{3\cdot\Delta} h(x)\,dx = \frac{\Delta}{24}\cdot\left(-9y_0 + 37\cdot y_1 - 59\cdot y_2 + 55\cdot y_3\right).$$

wobei die letzte Fläche außerhalb des eigentlichen Interpolationsintervalls liegt. Sie kann als Approximation an die nächste Steigung verwendet werden.

2.4.2 Adams-Bashfort-Prädiktor- und Korrektor-Formel

Aus diesen Überlegungen läßt sich jetzt unmittelbar ein weiteres Approximationsverfahren ermitteln. Sind vier Stützstellen $f(t-3\cdot\Delta t), f(t-2\cdot\Delta t), f(t-\Delta t), f(t)$ für die Steigung gegeben, so läßt sich der nächste Wert von y berechnen durch

$$y(t+\Delta t) = y(t) + \frac{\Delta t}{24}\cdot\left(-9\cdot f(t-3\cdot\Delta t) + 37\cdot f(t-2\cdot\Delta t) - 59\cdot f(t-\Delta t) + 55\cdot f(t)\right).$$

Dieses wird auch als „Adams-Bashfort-Prädiktor-Formel“ bezeichnet. Sind nur drei Stützstellen gegeben, so läßt sich die Formel verwenden

$$y(t+\Delta t) = y(t) + \frac{\Delta t}{12}\cdot\left(5\cdot f(t-2\cdot\Delta t) - 16\cdot f(t-\Delta t) + 23\cdot f(t)\right).$$

Sind schließlich nur zwei Stützstellen vorhanden, so kann nur linear approximiert werden,

$$y(t+\Delta t) = y(t) + \frac{\Delta t}{2}\cdot\left(-f(t-\Delta t) + 3\cdot f(t)\right).$$

Ist schließlich nur eine Stützstellen vorhanden, so erhalten wir das Euler-Verfahrens,

$$y(t+\Delta t) = y(t) + \Delta t\cdot f(t).$$

Da der Wert $y(t+\Delta t)$ durch Extrapolation berechnet wurde, kann natürlich eine etwas genauere Berechnung erreicht werden, indem das Polynom um eine Stelle verschoben wird, wobei der neue Wert von $f(t+\Delta t)$ mitbenutzt wird.

$$y_2(t+\Delta t) = y(t) + \frac{\Delta t}{24}\cdot\left(f(t-2\cdot\Delta t) - 5\cdot f(t-\Delta t) + 19\cdot f(t) + 9\cdot f(t+\Delta t)\right).$$

Sollten $y(t+\Delta t)$ und $y_2(t+\Delta t)$ zu stark divergieren, kann diese Formel noch einmal verwendet werden. Diese Formel wird als „Adams-Bashfort-Prädiktor-Korrektor-Formel“ bezeichnet. Für ein Polynom zweiten Grades folgt analog

$$y_2(t+\Delta t) = y(t) + \frac{\Delta t}{12}\cdot\left(-f(t-\Delta t) + 8\cdot f(t) + 5\cdot f(t+\Delta t)\right).$$

und für ein Polynom ersten Grades ist das arithmetische Mittel zu nehmen, d.h.

$$y_2(t+\Delta t) = y(t) + \frac{\Delta t}{2} \cdot \left(f(t) + f(t+\Delta t) \right).$$

Diese Formeln können beispielsweise verwendet werden, um die Formel für das Polynom ersten Grades zu verbessern. Nachdem vier Punkte gefunden wurden, kann dann das Polynom dritten Grades beliebig oft iteriert werden, um die anderen Punkte zu finden.

Das wichtigste Problem hierbei ist natürlich die Bestimmung von Anfangswerten. Dieses könnte mit den obigen Verfahren unterschiedlicher Polynome durchgeführt werden. Hier ist es u. U. möglich, mit zunächst kleineren Schrittweiten und ungenaueren Verfahren zu arbeiten, später dann die größeren Schrittweiten mit den genaueren Verfahren zu verwenden.

Das Verfahren nach Adams-Bashfort hat einen lokalen Fehler von Δt^5, wobei der konstante Faktor bei der Korrektorformel um den Faktor 15 kleiner ist als bei der Prädiktorformel. Es kann daher als relativ genau angesehen werden, wenn über die Charakteristik der Funktion y nichts weiter bekannt ist.

2.5 Unterstützung approximativer Lösungen durch C-SIM

C-Sim kann diese Verfahren natürlich unterstützen, indem es die Auswertung der Funktionen effizient standardisiert. Dieses ist jedoch nur bis zu einem gewissen Grad möglich., wenn die Auswertung der Funktion f nicht durch einen C-Funktionsaufruf erfolgen soll (was sicherlich sehr ineffizient wäre).

2.5.1 Runge-Kutta

Gehen wir davon aus, daß die Auswertung der Funktionen explizit erfolgen soll, so lassen sich die folgenden Makros einführen.

```
#define RungeKutta1(a, b, c) {double kt,ky,kb05,k1,k2,k3,k4; kt=a; ky=c;
#define RungeKutta2(a, b, c) kb05 = b*0.5; kt = a+kb05; ky = c + kb05*k1;
#define RungeKutta3(a, b, c) ky = c + kb05*k2;
#define RungeKutta4(a, b, c) kt = a+b; ky = c + b*k3;
#define RungeKutta(a, b, c) c + b/6.0*(k1+2*(k2+k3)+k4); }
```

Ein Aufruf zur Berechnung eines neuen Wertes wäre dann folgendermaßen möglich:

```
RungeKutta1(t, Dt, yt); k1= f(kt,ky);
RungeKutta2(t, Dt, yt); k2= f(kt,ky);
RungeKutta3(t, Dt, yt); k3= f(kt,ky);
RungeKutta4(t, Dt, yt); k4= f(kt,ky);
y2 = RungeKutta(t, Dt, yt);
```

Hiermit wäre dann jeweils genau der nächste Wert berechnet.

2.5.2 Adams-Bashfort-Prädiktor- und Korrektor-Formel

Für die Prädiktorformel erhalten wir

```
#define AdamsP4(a, b, c, d, t) ((-9*a+37*b-59*c+55*d)*t/24)
#define AdamsP3(a, b, c, t) ((5*a-16*b+23*c)*t/12)
#define AdamsP2(a, b, t) ((-a+3*b)*t/2)
#define AdamsP1(a, t) (a*t)
```

während für die Korrektorformel folgt

```
#define AdamsK4(a, b, c, d, t) ((a-5*b+19*c+9*d)*t/24)
#define AdamsK3(a, b, c, t) ((-a+8*b+5*c)*t/12)
#define AdamsK2(a, b, t) ((a+b)*t/2)
#define AdamsK1(a, t) (a*t)
```

sowie eine nützliche Hilfsfunktion

```
#define Shift(a, b, c, d, e) a=b; b=c; c=d; d=e;
```

Ein Aufruf zur Berechnung einer Folge von Werten wäre dann folgendermaßen möglich:

```
f0 = f(Dt,y0);
y1 = y1 + AdamsP2(f0, f1, Dt);
f1 = f(Dt,y1);
y2 = y1 + AdamsP2(f0, f1, Dt);
f2 = f(Dt,y2);
y2 = y1 + AdamsP3(f1, f2, Dt);
y3 = y2 + AdamsP3(f0, f1, f2, Dt);
f3 = f(Dt,y3);
y3 = y2 + AdamsP3(f1, f2, f3, Dt);
do { y4 = y3 + AdamsP4(f0, f1, f2, f3, Dt);
     f4 = f(Dt,y4);
     y4 = y3 + AdamsP4(f1, f2, f3, f4, Dt);
     ...
     f4 = f(Dt,y4);
     y4 = y3 + AdamsP4(f1, f2, f3, f4, Dt);
     Shift(f0, f1, f2, f3, f4);
  }
```

In vielen Fällen lassen sich die Anfangswerte genauer bestimmen, z.B. aus dem physikalischen Problem heraus. Weiß man beispielsweise, daß `y1=y3`, so folgt aus gegebenem `y2` und der Korrektorformel für Polynome zweiten Grades, daß

```
y3 = y2;
f3 = f(Dt,y3);
y3 = y2 + AdamsP3(f3, f2, f3, Dt);
...
f3 = f(Dt,y3);
y3 = y2 + AdamsP3(f3, f2, f3, Dt);
```

Dieses läßt sich mehrfach wiederholen, bis `y3` „stabil“ bleibt.

3 Hilfsprozeduren

Um ein solches komplexes Simulationssystem effektiv einsetzen zu können, müssen eine Reihe von Hilsprozeduren zur Verfügung gestellt werden. Diese sollen in diesem Kapitel eingeführt werden. Wir unterscheiden dabei solche für die Generierung von Werten, solche für die Darstellung von Werten während der Laufzeit, und solche für die Auswertung von Werten am Ende der Simulation.

Die erste Klasse umfaßt Zufallszahlengeneratoren und Funktionen zur Erzeugung statistischer Verteilungen. Darüber hinaus sollen zu Beginn der Simulation oder interaktiv während der Laufzeit Werte verändert werden können. Die zweite Klasse von Funktionen umfaßt solche zur graphischen Animation von Daten, und solche zur Ausgabe während der Simulation, in der Regel in einem besonderen Fenster. Die dritte Klasse schließlich bietet Funktionen zur Ermittlung statistischer Verteilungen bzw. Kenngrößen und zugehöriger Konfidenzintervalle an.

Da die Klasse von Funktionen sehr groß ist, wird ein Parameterkonzept eingeführt, welches nur jene Funktionen in das Simulationsprogramm einbindet, welche tatsächlich für das jeweilige Simulationsmodell benötigt werden. Dieses ließe sich im Prinzip natürlich auch mittels des Linkers durchführen, allerdings haben wir hiermit den Vorteil, daß gewisse Parameter ein- und ausgeschaltet werden können. Wird z.B. die Graphik im Programm verwendet, so läßt sich durch Ausschalten des `GraphikParameters` die Graphikausgabe verhindern, was das Programm sehr viel schneller macht. Ähnliches gilt für statistische Auswertung (die beim Testen des Programms in der Regel nicht benötigt wird) und für andere Funktionen.

3.1 Das Parameterkonzept von C-SIM

Ein Parameter hat die syntaktische Form:

```
<Klasse>Paramter
```

wobei `<Klasse>` für `Graphik`, `Moment`, `Process` und ähnlichem steht. Zusätzlich können zu einem Parameter noch Attribute treten, z.B.

```
MomentParamter Length = 1000;
```

wodurch die maximale Feldlänge bei der Erzeugung von Histogrammen zur Ergebnisauswertung auf 1000 gesetzt wird. In C-SIM muß syntaktisch geschrieben werden:

```
#define <Klasse>Paramter
```

und die Attribute explizit gesetzt werden.

3.2 Die Wertefunktionen von C-SIM

Wertefunktionen liefern einen Wert, der für die variable Simulation benötigt wird. Die Wertefunktion von C-SIM haben die syntaktische Form einer Funktion in C. Es gibt zur Zeit die Funktionen

```
const True 1; /* Wahrheitswert (Typ ist int) */
const False 0; /* Wahrheitswert (Typ ist int) */
typedef int bool; /* Wahrheitswerttyp (Typ ist int) */

double Random(); /* Liefert Zufallszahl zwischen 0 und 1: [0,1) */
bool Is_True(double p); /* Liefert mit Wahrscheinlichkeit p True */
int Equal(int low,up); /* Liefert int-Wert aus [low,up] */
double Equald(double low,up); /* Liefert double-Wert aus [low,up) */
double NegExp(double Mean); /* Liefert double-Wert mit
```

```
            negativ-exponentieller Verteilung mit Mittelwert Mean */
  double Normal(double Mean,StdDev); /* Liefert double-Wert mit
            normaler Verteilung mit Mittelwert Mean
            und Standardabweichung StdDev */
  double Erlang(double Mean,Steps); /* Liefert double-Wert mit
            Erlang-Verteilung mit Mittelwert Mean
            und Steps Stufen */
  int GeometricMean(double Mean); /* Liefert Anzahl der Schritte bei
            geometrischer Verteilung mit Mittelwert Mean */
  int GeometricProb(double p); /* Liefert Anzahl der Schritte bei
            geometrischer Verteilung mit Einzelwahrscheinlichkeit p */
```

Als Zufallszahlengenerator wird der bessere lineare Kongruenzzahlengenerator `drand48()` der Unix-Betriebssysteme verwendet. Der Anfangswert ist durch den Befehl: `srand48(<int>)` zu setzen.

3.3 Die Darstellungsfunktionen von C-SIM

Die Darstellungsfunktionen von C-SIM erfordern in der Regel die Bereitstellung von Datensätzen, welche dann auf einem Graphikfenster ausgegeben werden.. Es gibt zur Zeit Funktionen zur Darstellung der Struktur eines Modells, der Änderungen oder Übergänge zwischen Komponenten, der Darstellung von Zeitreihen oder von Verteilungen.

3.3.1 Struktur und Ablauf eines Simulationsmodells

Die Darstellung von Struktur und Ablauf eines Simulationsmodells wird in dem Handbuch X-SimView, Version 2.0 von Th. Ono-Tesfaye beschrieben und soll hier nicht wiederholt werden.

3.3.2 Zeitreihen (Time Series)

Um eine **Zeitreihe** darzustellen, müssen mindestens die Paramter

```
            #define GraphikParameter
            #define MomentParameter
            #define TimeSeriesParameter
```

gesetzt sein. Es gibt dann nur noch drei weitere Befehle. Mit:

```
            TimeSeriesType Zeitreihe;
```

wird ein Datensatz mit dem willkürlichen Namen `Zeitreihe` definiert, den der Programmierer zur Speicherung der Daten der Zeitreihe benötigt. Mit

```
            Zeitreihe = NewTimeSeries(Attribute, XText, YText, Length,
Funktion);
```

wird eine Instanz dieses Datensatzes erzeugt und das Fenster auf dem Bildschirm geöffnet. Die `Attribute` legen die Art des Diagramms fest. Es sind die folgenden Parameter möglich, die durch ein logische Oder '|' verknüpft werden müssen:

`BAR`	Werte werden durch Histogramme dargestellt
`SEPARATEBARS`	Zwischen zwei Blöcken wird eine Zwischenraum eingefügt
`BARVALUE`	An die Blöcke werden die Werte geschrieben
`DOTS`	Werte werden durch Punkte dargestellt

`LINE`	Werte werden durch Linien dargestellt
`DOTSLINE`	Werte werden durch mit Linien verbundene Punkte dargestellt
`HORIZGRID`	Es werden horizontale Linien gezeichnet
`VERTGRID`	Es werden vertikale Linien gezeichnet
`LOGSCALE`	Die Y-Achse wird logarithmisch aufgetragen

Die Parameter `XText, YText` sind Zeiger auf Texte oder Textkonstante, die im Diagramm gesetzt werden können. Die `Length` gibt die Anzahl der angezeigten Wert aus, und die `Funktion` ist

```
Funktion = HISTOGRAM: Zeitfolge wird als Histogramm benutzt
Funktion = TIMESERIES: Es wird eine Zeitfolge ausgegeben
Funktion = SEQUENCE: Bilde Mittelwert von Zeitfolge; gebe diesen aus
Funktion = FUNCTION: Bilde Mittelwert von Zeitfunktion; gebe diese aus
Funktion = k > 1: Es werden k Zeitfolgen ausgegeben
```

Der Unterschied zwischen einer Zeitfolge und einer Zeitfunktion wird unten erklärt. Mit

```
TimeSeries(Zeitreihe, Wert);
```

wird in den Datensatz ein `Wert` vom Typ `double` eingetragen und die neue Zeitfolge sofort ausgegeben. Gibt es mehr als eine Zeitfolge (`Funktion=k>1`), so kann durch

```
TimeSeriesIndex(Zeitreihe, Nummer, Wert);
```

der Wert der `Nummer`-ten Zeitfolge eingegeben werden. Ist `Nummer`=0, so werden alle Wert ausgegeben und der nächste Index bereitgestellt. Ist `Nummer`<0, so wird nur der nächste Index bereitgestellt, also keine Ausgabe gemacht. Die Zeitreihe kann beliebig lang werden, zeigt jedoch nur die letzten `Length` Werte an.

Soll nur eine Reihe von Werten ausgegeben werden, so kann dieses durch die Funktion -2 realisiert werden. Es gibt in:

```
TimeSeriesIndex(Zeitreihe, Nummer, Wert);
```

`Nummer` den Index an, und `Wert` den Wert an der entsprechenden Stelle. Ist `Nummer=0`, so wird eine Ausgabe gemacht.

3.3.3 Histogramme

Die Auswertung von Simulationsergebnissen muß mit statistisch einwandfreien Methoden durchgeführt werden. Um auch während der Simulation die Ergebnisse zwischenzeitlich kontrollieren zu können, müssen diese ausgegeben werden. Dazu kann ein Histogramm in einem eigenen Fenster dargestellt werden.

Um ein **Histogram** darzustellen, müssen mindestens die Paramter

```
#define GraphikParameter
#define MomentParameter
```

gesetzt sein. Es gibt dann nur noch drei weitere Befehle. Mit:

```
HistogramType HistogrammDaten;
```

wird ein Datensatz mit dem beliebigen Namen `HistogrammDaten` definiert, den der Programmierer zur Speicherung der Daten des Histogramms benötigt. Mit

```
HistogrammDaten = NewHistogramGraphik
      (Attribute, XText, YText, Unit, Length, Art, BatchLength);
```

wird eine Instanz dieses Datensatzes erzeugt und das Fenster auf dem Bildschirm geöffnet. Die `Attribute` legen die Art des Diagramms fest und sind im einzelnen dem Manuskript [Ono-Tesfaye] zu entnehmen. Die Parameter `XText`, `YText` sind Zeiger auf Texte, die im Diagramm gesetzt werden können. Die `Unit` gibt die Breite eines Feldes an, die `Length` gibt die Anzahl der aufgenommenen Wert an, und die `BatchLength` gibt an, nach wie vielen Werten ein Teiltest durchgeführt werden soll.

`Art` ist ein Parameter, der die Art der Ausgabe bestimmt. Es können folgende Werte angegeben werden:

$$\text{Art} \;::\; \left\{\begin{matrix}\text{SEQUENCE}\\ \text{FUNCTION}\end{matrix}\right\} \;|\; \left\{\begin{matrix}\text{ABSOLUTE}\\ \text{RELATIVE}\\ \text{DENSIT}\\ \text{DISTRIBUTIONY}\\ \text{COMPLEMENTARY}\end{matrix}\right\} \;|\; \text{NOOVERFLOW} \;|\; \text{CONFIDENCE}$$

`SEQUENCE` führt für eine Folge von Werten die Statistik durch; `FUNCTION`, für eine Funktion.

Um die Werte, die der Parameter Art annehmen kann, zu verstehen, muß ein wenig Statistik erklärt werden. Eine Dichtefunktion h(x) gibt an, wieviele Werte in dem Bereich $[x,x+\Delta x)$ liegen. Dabei kann die Anzahl dieser Werte einmal absolut angezeigt werden, zum anderen relativ zur Größe 1 (d.h. wieviel Prozent der Werte liegen in dem Bereich $[x,x+\Delta x)$), und zum dritten kann die Dichtefunktion angegeben werden, d.h. die Summe der Funktionswerte $h(x)*\Delta x$ über alle x ergibt den Normaliesierungswert 1. Dieses kann durch genau einen der drei Parameter `ABSOLUTE`, `RELATIVE` und `DENSITY` angegeben werden. Außerdem kann statt der Dichte auch die Verteilungsfunktion: $H(z)=\sum_{X<z} h(x)\cdot\Delta x$ angegeben werden, oder die komplementäre Verteilungsfunktion 1-H(X); dieses geschieht durch die Parameter `DISTRIBUTION` und `COMPLEMENTARY`. Es kann natürlich nur einer dieser fünf Parameter gesetzt werden. Er ist mit einem der ersten beiden durch ein logisches Oder '|' zu verknüpfen.

Der Parameter `NOOVERFLOW` verhindert die Ausgabe des Überlaufs in das Diagramm. Alle Werte, die größer als `Unit·Length` sind, werden im Feld mit dem Index `Length+1` kumuliert, während jene, die kleiner sind als 0 im Feld mit dem Index 0 gesammelt werden. Gibt es viele, evtl. nicht interessierende Werte in dem oberen Überlaufsbereich, so kann der Maßstab bei der Ausgabe stark verzerrt werden. Durch setzen von `NOOVERFLOW` kann dieses verhindert werden. Allerdings werden diese Werte intern natürlich berechnet, damit die Normalisierung korrekt durchgeführt werden kann. Bei der Interpretation der Ergebnisse ist darauf zu achten, daß der sichtbare Teil nicht mehr einer vollständigen Dichte, relativen Häufigkeit oder Verteilung entsprechen muß, da deren Darstellung am oberen Ende 'abgeschnitten' wurde.

Wahlweise kann auch noch `CONFIDENCE` angegeben werden um darauf hinzuweisen, daß durch die Funktion `HistogramOutput` Konfidenzintervalle der Dichte und der komplementären Verteilungsfunktion ausgegeben werden. Dadurch kann die Berechnung während des

Programmlaufs etwas verkürzt werden (dieser Parameter wird gegenwärtig nicht ausgewertet, d.h. es ist immer möglich, `HistogramOutput` zu berechnen).

Um einen `Wert` einzufügen, ist die Funktion

```
Histogram(HistogrammDaten, Wert);
```

aufzurufen. Zur Auswertung ist dann am Ende der Simulation die folgende Funktion aufzurufen:

```
HistogramOutput(HistogrammDaten);
```

Dadurch werden die Ergebnisse sowohl in einer besonderen Datei `'SimData<n>'` ausgegeben als auch in einem Graphikfenster als Verteilungsfunktion und Dichtefunktion angezeigt.

3.4 Die Auswertungsfunktionen von C-SIM

Die Auswertungsfunktionen von C-SIM werden nach statistischen Kriterien ausgewählt. Es gibt die Möglichkeit, Mittelwerte und Streuungen einzelner Größen, sowie Verteilungsfunktionen durch Histogramme aufzunehmen. Der Momentparamter ist in jedem Falle zusetzen.

3.4.1 Moment

Um einzelne zufällige Größen statistisch auszuwerten, muß mindestens der Paramter

```
#define MomentParameter
```

gesetzt sein. Mit:

```
MomentType MomentDaten;
```

wird ein Datensatz mit dem willkürlichen Namen `MomentDaten` definiert, den der Programmierer zur Speicherung der Daten der Zeitreihe benötigt. Mit

```
MomentDaten = NewMoment(Art);
```

wird eine Instanz dieses Datensatzes erzeugt. Die `Art` legt fest, ob es sich um eine Folge von Größen handelt oder um eine Funktion auf der Zeit:

Art = SEQUENCE: Folge von Werten: $\{w_i\}_{i=1\ldots A}$
Art = FUNCTION: Abbildung: $[0,T] \rightarrow \Re$

und sind im einzelnen dem Manuskript [Ono-Tesfaye] zu entnehmen. Die Parameter `XText`, `YText` sind Zeiger auf Texte, die im Diagramm gesetzt werden können. Die `Unit` gibt Breite eines Feldes an, die `Length` gibt die Anzahl der aufgenommenen Wert an, und die `BatchLength` gibt an, nach wievielen Werten ein Teiltest durchgeführt werden soll. Um einen `Wert` einzufügen, ist die Funktion

```
Moment(MomentDaten, Wert);
```

aufzurufen. Es läßt sich darüber hinaus mittels:

```
ClearMoment(MomentDaten);
```

ein Momentdatensatz stets löschen. Zur Auswertung ist dann am Ende der Simulation die folgende Funktion aufzurufen:

```
Moment1(MomentDaten); /* Liefert den Mittelwert der Daten */
Moment2(MomentDaten); /* Liefert das zweite Moment der Daten */
Variance(MomentDaten); /* Liefert die Varianz der Daten */
StandardDeviation(MomentDaten); /* Liefert die Standardabweichung
 der Daten */
SCV(MomentDaten); /* Liefert den quadrierten Variationskoeffizienten
 der Daten */
ConfidenceInterval(MomentDaten, ConfidenceValue); /* Liefert c, so daß
Moment1 - c < wahrer Mittelwert < Moment1 + c */
```

Die Werte haben alle den Typ `double`. Um eine Teilteststatistik durchzuführen, können noch die folgenden Parameter verändert werden (sie werden sonst automatisch gesetzt):

```
MomentDaten -> Max_Values_in_Batch   /* Anzahl von Werten je Teiltest*/
MomentDaten -> BatchNumber     /* falls negativ: Betrag ist
Anzahl der Vorläufe; sonst nicht
ändern (z.B. -2: zwei Vorläufe) */
```

3.4.2 Histogramme

Um ein **Histogram** auszuwerten, muß mindestens der Paramter

```
#define MomentParameter
```

gesetzt sein. Mit:

```
HistogramType HistogrammDaten;
```

wird ein Datensatz mit dem willkürlichen Namen `HistogrammDaten` definiert, den der Programmierer zur Speicherung der Daten der Zeitreihe benötigt. Mit

```
HistogrammDaten = NewHistogram(Art)
```

wird eine Instanz dieses Datensatzes erzeugt. Die Variable `Art` gibt an, welcher Typ von zufälliger Variable vorliegt:

$$\text{Art} \;::\; \left\{\begin{matrix}\text{SEQUENCE}\\ \text{FUNCTION}\end{matrix}\right\} \;\mid\; \left\{\begin{matrix}\text{ABSOLUTE}\\ \text{RELATIVE}\\ \text{DENSITY}\\ \text{COMPLEMENTARY}\\ \text{DISTRIBUTION}\end{matrix}\right\} \;\mid\; \text{CONFIDENCE}$$

SEQUENCE: Folge von Werte: $\{w_i\}_{i=1\ldots A}$
FUNCTION: Abbildung: $[0,T] \rightarrow \Re$

(Die anderen Parameter wurden im letzten Abschnitt beschrieben.) Um einen `Wert` einzufügen, ist die Funktion

```
Histogram(HistogrammDaten, (double) Wert);
```

aufzurufen. Zur Auswertung ist dann am Ende der Simulation die folgende Funktion aufzurufen:

```
HistogramOutput(HistogrammDaten);
```

Dadurch werden die Ergebnisse sowohl in einer besonderen Datei ausgegeben, als auch in einem Graphikfenster, falls `GraphikParameter` gesetzt ist.

Um eine Teilteststatistik durchzuführen, können noch die folgenden Parameter verändert werden (sie werden sonst automatisch gesetzt):

```
HistogrammDaten -> Max_Values_in_Batch   /* Anzahl von Werten
je Teiltest */
HistogrammDaten -> BatchNumber     /* falls negativ: Betrag ist
Anzahl der Vorläufe; sonst nicht
```

```
            ändern (z.B. -2: zwei Vorläufe) */
            HistogrammDaten -> TableLength   /* Anzahl der Werte im Histo-
gramm */
            HistogrammDaten -> TableScale   /* Skalierung (Breite je Ein-
trag) */
```

3.4.3 Stationen

Um eine **Station** auszuwerten, müssen mindestens die Paramter

```
#define MomentParameter
#define StationParameter
```

gesetzt sein. Mit:

```
StationType StationDaten;
```

wird ein Datensatz mit dem willkürlichen Namen `StationDaten` definiert, den der Programmierer zur Speicherung der Daten für die Station benötigt. Mit

```
StationDaten = NewStation();
```

wird eine Instanz dieses Datensatzes erzeugt. Beim Beginn der Messung muß die funktion

```
ClearStation(StationDaten);
```

aufgerufen werden. Um `Anzahl` Objekte in die Station einzufügen, ist die Funktion

```
StationIn(StationDaten, Anzahl);
```

aufzurufen. Werden Anzahl Objekte aus der Station entfernt, die genau Zeit Zeiteinheiten in der Station verbracht haben, so ist die Funktion:

```
StationOut(StationDaten, Anzahl, Zeit);
```

aufzurufen. Zur Auswertung ist dann am Ende der Simulation die folgende Funktion aufzurufen:

```
Moment1(StationDaten); /* Mittlere Anzahl von Objekten in der Station */
Moment2(StationDaten); /* 2. Moment Anzahl von Objekten in der Station */
SojournTime(StationDaten); /* mittl. Verweilzeit von Objekten
           in der Station */
Throughput(StationDaten); /* Durchsatz von Objekten durch die Station */
ServiceRate(StationDaten); /* Bedienrate von Objekten in der Station */
Utilization(StationDaten); /* Auslastung der Station */
Residual(StationDaten); /* restliche Füllung von Objekten in Station */
```

Das Simulationsprogramm C-SIM

Während im ersten Teil dieses Bericht beschrieben wurde, wie C-SIM logisch aufgebaut ist, soll hier dargestellt werden, wie C-SIM praktisch eingesetzt werden kann. Da C-SIM gegenwäratig nicht in einer Compilerversion vorliegt, sondern sämtliche Komponenten nur durch eine Makro-Substitution eingebunden werden, ist bei der Verwendung von C-SIM sehr sorgfältig vorzugehen.

4 Struktur eines C-SIM-Programms

Ein C-SIM Programm umfaßt eine Sammlung von Dateien, einige Einstellungen für die graphische Simulation und ein Programm, welches die Struktur des Simulationsmodells wiedergibt. Zusätzlich sind für die graphische Ausgabe einige Dateien zu installieren, die dem Bericht: X-SimView von Ono-Tesfaye entnommen werden können.

Es werden Dateien durch #include-Befehle eingebunden, wobei nur zwei Dateien eingebunden werden müssen (`CSIMFunctions.1`, `CSIMFunctions.2`), welche sich die anderen selbsttätig heraussuchen. Durch entsprechende Schalter (im Bericht als Parameter bezeichnet) können diese flexibel gesteuert werden.

Ein Programm wird in der Regel in der Directory übersetzt, in der ein Verweis auf die beiden oben genannten Dateien existiert. Da in CSIM bestimmte Abhängigkeiten existieren, wurde nicht das Unix-Library-System verwendet. Ein Programm hat somit den folgenden Aufbau:

```
/******* Parameter - Steuerung *********/

#define GraphikParameter     /* Fuer die Graphikausgabe, sonst auskommentieren */
/* #define TextParameter     /* Fuer die Textausgabe, sonst auskommentieren */
#define XSimViewParameter/* Fuer die grpahische Animation, sonst auskommentieren */
#define JobParameter /* Fuer die Simulation, Auftraege, sonst auskommentieren */
#define QueueParameter       /* Fuer die Simulation, Warteschlangen, sonst auskomm.*/
#define ProcessParameter     /* Fuer die Simulation, Prozesse, sonst auskommentieren */
#define StatistikParameter   /* Fuer die Statistik, sonst auskommentieren */
#define StationParameter     /* Fuer die Stationsstatistik, sonst auskommentieren */
#define TimeSeriesParameter/* Fuer die Zeitreihenausgabe, sonst auskommentieren */
/* #define EnterParameter    /* Fuer das Enterkonzept, sonst auskommentieren */

#include "simdateien/CSIMFunctions1.c"    /* Einbinden der Dateien: queue, process */

        JobData                 /* deklariert die zu den Paketen lokalen Variablen */
                TimeType ArrivalTime;
        };

        QueueData               /* Fuer die Warteschlange */
                int SVNumber;
        };

        ProcessData(Arrival)    /* deklariert die zum Arrivalprozess lokalen Variablen */
```

```
            int SVNumber;
            JobType Job;
        };
        ProcessData(Server)          /* deklariert die Serverprozess lokalen Variablen */
            int SVNumber;
            JobType Job;
        };
        ProcessData( ... )           /* deklariert die lokalen Variablen weiterer Prozesse*/
            ...
        };

#include "simdateien/CSIMFunctions2.c"    /* Einbinden der Dateien: queue, process */

        struct ArrivalTyp * ArrivalPtr;
        struct ServerTyp * ServerPtr;

        QueueType Queue;

        union ProcTag {        /* PT ist eine allgemeine Referenz auf alle Prozesse (nur intern!
*/
            struct ArrivalTyp * ArrivalPtr;
            struct ServerTyp * ServerPtr;
            struct ..Typ * .Ptr;                    /* weitere Prozesse */
            struct ProcType *ProcPtr;}    PT;

main()
{       srand(3377733);              /* beliebig wählbarer Anfang der Zufallsfolge */

        Queue = NewQueue(0);         /* Erzeugt eine gloable Warteschlange */

        /* Folgende Anweisungen erzeugen die aktiven Prozesse */
        AllocInitProcess(ArrivalTyp, ArrivalPtr, 0, 3, SystemTime, 0, ACTIVE);
        StartProcess(ArrivalPtr);

        AllocInitProcess(ServerTyp, ServerPtr, 10, 3, SystemTime, 0, ACTIVE);
        StartProcess(ServerPtr);

  /* Hier beginnt die Simulation (Simulationszeit, Ausgabe fuer SimZeit bei Graphik)*/
StartSimulation ( StateProb->BatchNumber < 500 , 1)

/* Zugangsprozess */
PROCESS(Arrival)
#define THIS PT.ArrivalPtr->                  /* nur für interne Zeigerverwaltung */

   PHASE(0)                                  /* Anfangspahse: Initialisierung */
        THIS Job = Null;
        NextAfter(1, 0 );

   PHASE(1)                         /* ERzeuge Auftrage; Sende diesen an Empfänger */
```

```
        THIS Job = NewJob();
        THIS Job -> ArrivalTime = SystemTime;
        Insert(THIS Job, Queue);
        RepeatAfter( 1.0 );
#undef THIS                             /* nur für interne Zeigerverwaltung */

/* Bedienprozess */
PROCESS(Server)
#define THIS PT.ServerPtr->             /* nur für interne Zeigerverwaltung */

   PHASE(10)                       /* Anfangspahse: Initialisierung */
        THIS Job = Null;
        NextAfter(11, 0.0 );

   PHASE(11)                       /* Warten auf Auftrag */
        if (Queue->Number == 0) PASSIVATE;
        THIS Job = Seize(Queue);
        NextAfter(12, NegExp(0.9) );    /* Verzögere, bis Auftrag fertiggestellt */

   PHASE(12)                       /* Auftrag ist fertiggestellt */
        DISPOSE( THIS Job );
        NextAfter(11, 0.0 );
#undef THIS                             /* nur für interne Zeigerverwaltung */

PROCESS(...)                            /* weitere Prozesse */
#define THIS PT....Ptr->                /* nur für interne Zeigerverwaltung */

   PHASE(20)                            /* Anfangsphase: Initialisierung */

  ...

  EndSimulation;

/* Hier erfolgt die Auswertung */
}
```

Die Dateien, die in C-SIM benötigt werden, werden abhängig von den Parametern durch die beiden Dateien CSIMFunctions1,2 eingebunden. Außerdem werden hier einige Konstante deklariert. Weitere Konstante, z.B. Einheiten, können in die Datei: "`definitions.c`" eingetragen werden.

Eine genauere Kenntnis der Dateistruktur ist im Prinzip nicht nötig, sollte aber evtl. anhand der mitgelieferten Dateien studiert werden.

4.1 Das Dateisystem

Legen wir die Modelldateien unter die Directory: `simulation`, so sind die folgenden Dateien für die Ausführung eines C-SIM-Programms nötig:

```
simulation/Modelldatei.c
simulation/csimdateien/
simulation/csimdateien/CSIMFunctions1.c
simulation/csimdateien/CSIMFunctions2.c
simulation/csimdateien/definitions.c
simulation/csimdateien/distribution.c
simulation/csimdateien/functions.c
simulation/csimdateien/moment.c
simulation/csimdateien/process.c
simulation/csimdateien/sim_var_queue.c
```

und für die graphische Einbindung wird außerdem benötigt:

```
.Xdefaults
simulation/libXSV.a
simulation/XSV
simulation/XSVS
simulation/XSV.bitmaps
simulation/XSimView.h
```

(.Xdefaults muß im home-Verzeichnis stehen.) Die csimdateien/ sollten in der Regel nicht verändert werden (wenngleich z.B. neue Defintionen in csimdateien/definitions.c eingetragen werden können). Die Dateien für die Graphik sollten ebenfalls nicht verändert werden. Sie sind als Unix-Libraries entsprechend durch Compilation zu erzeugen. Ein möglicher Make-File hierzu findet sich unter den Beispieldateien. Das Programm ist durch einen Befehl der Art:

```
gcc -o namename.c -lm libXSV.a
```

zu übersetzen.

4.2 Konstrukte zum zeitlichen Ablauf und zur Kommunikation

Da es vielen neuen Anwendern von C-SIM schwerfällt, die verschiedenen Möglichkeiten zur Kommunikation zu verstehen, sollten diese hier in knapper Form entsprechend der gegenwärtigen Syntax und Semantik zusammengefaßt werden.

Unter einer Referez verstehen wir im folgenden einen Zeiger auf ein Objekt, entweder einen Prozeß oder einen Auftrag.

Ein Prozeß ist entweder **aktiv** oder er ist **passiv**. Letzteres bedeutet, daß er sich selbst 'passiviert' hat (durch den Befehl: Passivate();). Ein passiver Prozeß kann von einem aktiven Prozeß durch den Befehl: Activate(<process-reference>); wieder aktiviert werden. Varianten hierzu sind: Activate_PT(<process-reference>,<typ-nummer>); um einen Prozeß eines bestimmten Typs zu aktivieren (lokale Variable ProcessTypeNumber) oder um alle passiven Prozesse zu aktivieren: Activate_all(); Ein Activate auf einen aktiven Prozeß hat keine Wirkung.

Ein Prozeß ist in Phasen gegliedert, die nacheinander mit einer gewissen zeitlichen Verzögerung ausgeführt werden. In der Regel wird am Ende einer Phase eine Zeit und die Nummer der nächsten auszuführenden Phase angegeben:

```
NextAfter(<nächste Phase>, <relative Zeit>);
```

Es kann jedoch auch sinnvoll sein, die gleiche Phase wiederholt auszuführen, was am effektivsten durch den Befehl:

```
RepeatAfter(<relative Zeit>);
```

realisiert wird. Soll eine Phase erst nach einer gewissen Zeit betreten werden, so ist es in manchen Situationen bequemer, die Verzögerung in diese Phase zu schreiben. Durch den Befehl

```
Delay(<relative-time>);
```

am Beginn einer Phase kann dieses realisiert werden.

Soll ein Auftrag (`Job`) an einen anderen Prozeß gesendet werden, so kann dieses sehr einfach durch das Enter-Konzept realisiert werden. Der Sender führt einfach den Befehl aus:

```
Enter(<Job-Reference>,<process-reference>);
```

und läuft dann weiter, bis seine Phase beendet ist. Der empfangende Prozeß besitzt standardmäßig eine lokale Warteschlange (`EnterQeueue`), welche den Auftrag aufnimmt. Der empfangende Prozeß kann entweder den Inhalt dieser Warteschlange abfragen (z.B. durch

```
if (THIS EnterQueue -> Number == 0) ...
```

oder durch eines der hierfür besonders entwickelten und meist effektiveren Konstrukte. Durch

```
Await(<Job-reference>);
```

wird der Prozeß solange verzögert, bis ein Auftrag in der `Enterqueue` eingetragen wurde; in diesem Fall enthält die Variable `<Job-reference>` einen Verweis auf diesen Auftrag (der sich nach diesem Befehl auch nicht mehr in der `EnterQueue` befindet). Da dieses durch passivieren des Prozesses geschieht, ist dieses sehr effizient.

Durch

```
AwaitUntil(<Job-reference>,<absolute-time>);
AwaitDelay(<Job-reference>,<relative-time>);
```

wird der Prozeß solange verzögert, bis entweder ein Auftrag in der `Enterqueue` eingetragen wurde oder alternativ der Zeitpunkt `absolute-time>` bzw. `<relative-time>+SystemTime` erreicht wurde (dann ist die `<Job-reference>==Null`). Da dieses durch passivieren des Prozesses geschieht, ist dieses sehr effizient. Dieses Konstrukt eignet sich offenbar besonders gut für die Zeitüberwachung, wenn z.B. auf die ausstehende Quittierung eines Pakets gewartet wird.

Man kann (und sollte) in diesem Fall die `EnterQueue` völlig transparent verwenden, d.h. nicht explizit darauf zugreifen. Dieses läßt sich mittels der vorgestellten Makros sehr viel effizienter und sauberer durchführen. Das Konzept ermöglicht also das asynchrone Senden von Signalen und Empfangen dieser Signale in einem Empfangsprozeß. Welche Elementarbefehle im einzelnen dazu benötigt werden kann zwar den Makro-Definitionen entnommen werden, interessiert aber für die Semantik dieser Konstrukte überhaupt nicht.

Eine Möglichkeit, eine beliebige Bedingung abzuprüfen bietet der `Wait`-Befehl. Da hier jedoch ein *busy-wait* ausgeführt wird, ist dieser Befehl sehr ineffektiv.

4.3 Einige Hinweise zu möglichen Fehlern:

Ein Programm ist insbesondere dann falsch, wenn die Phasen nicht richtig verwendet werden. Jede Phase benötigt eine eigene Nummer, die in der Regel aufsteigend geordnet ist (das ist sinnvoll, wenngleich nicht nötig). Außerdem müssen lokale Prozeßvariable mit dem Vorsatz THIS versehen werden (der eine entsprechende Referenz enthält). Dazu muß THIS mit `#define THIS PT.ServerPtr->` initialisert werden, wobei `Server` der Name des jeweiligen Prozesses ist, und vor der nächsten Initialisierung entinitialisiert werden: `#undef THIS`.

C-SIM verwendet sehr viele Zeigerstrukturen, die leicht verletzt werden können. Daher sollte man stets die Standardprozeduren verwenden, insbesondere bei der Verwaltung von Jobs. Existieren zwei Referenzen auf das gleiche Objekt, so ist es ein häufiger Fehler, eine zu ändern (z.B. durch `Dispose` freizugeben). Hier können Fehler in den Warteschlangen auftreten, die nur sehr schwierig zu verstehen sind. Auch dürfen die Köpfe der Warteschlangen (z.B. `Number`) nicht vom Programmierer verändert werden, sondern nur mittels der Standardfunktionen. Da dieses in C nicht unterdrückt werden kann, muß der Programmierer eben selbst darauf achten.

Die Einbindung von Graphik in die Programme können den Beispielprogrammen entnommen werden. Hier müssen zunächst die Dateien gemäß dem Handbuch über X-SIM-View angelegt werden.