Zusammenfassung: ODEMx

Robert Hilbrich

6. Oktober 2007

Inhaltsverzeichnis

1 Einführung

1.1 Simulationsbegriff

Computersimulation ist eine experimentelle Untersuchungsmethode von realen oder gedachten Systemen unter Verwendung von formalen Modellen, die als ausführbare Software das Verhalten dieser Modell in Hinblick auf das Untersuchungsziel näherungsweise nachbilden.

Überwiegend werden dynamische Systeme untersucht.

1.2 Zeitbegriffe

Realzeit, Modellzeit, Simulationsausführungszeit

Bei einer Echtzeitsimulation ist die Realzeit eines Vorgangs gleich der Ausführungszeit.

Zeitdiskrete Modelle werden in digitalen Simulationen modelliert, zeitkontinuierliche Modelle in analogen Simulationen.

1.3 Grundprinzipien von ODEMx

Basis ist eine *Prozessverwaltung in C++*. Es gibt dann einen Ereigniskalendar und Prozesse mit einer aktuellen Ereigniszeit. Eine virtuelle Funktion stellt den Lebenslauf eines Prozesses dar. Zusätzlich gibt es auch Scheduling und Synchronisationsmechanismen.

1.4 Prinzipielle Behandlung zeitkontinuierlicher Signale

1.4.1 Aufgaben des Simulators

Zunächst muß er die initialen Prozesse in den Terminkalendar laden (mindestens einen). Dann wird der erste Prozess mit der kleinsten Ereigniszeit und der höchsten Priorität gestartet. Dann wird seine virtuelle Lebenslauf-Methode aktiviert:

- diskrete Prozesse: Durchführung der Zustandsänderungen
  kontinuierliche Prozesse: Zeitverbrauch, erneutes Scheduling mit event. Prozesswechsel

  Oder

- Blocking (Entfernung aus dem Terminkalendar, Prozesswechsel)

  Oder

- Terminierung (Entfernung aus dem Terminkalendar, Prozesswechsel)

Nun folgt zyklisch der nächste Prozess im Terminkalendar.

Am Ende erfolgt die Auswertung bzw. Abbruch der Simulation

1.4.2 zeitkontinuierliche Zustandsänderungen

... durch die Bereitstellung numerischer Integrationsverfahren. Im Prinzip ist das eine *Anfangswertaufgabe*.

$$x(t_0) = x_0$$
$$x'(t) = f(x(t), t)$$
$$x(t_{k+1}) = x(t_k) + \int_{t_k}^{t_{k+1}} f(x(t), t)dt$$

Eine annähernde Lösung wird bereitgestellt mittels:

$$h = t_{k+1} - t_k$$
$$x(t_{k+1}) \approx x(t_k) + h \cdot x'(t_k)$$

Aber dies Näherungsverfahren hat schlechte Eigenschaften, daher nimmt OdemX das: *Runge-Kutta-Merson*-Verfahren mit fehlerabhängiger Schrittweitensteuerung.

Die Memberfunktion `integrate(abbruchzeit, zustandsbedingung)` ruft RKM auf!

Jeder kontinuierliche Prozess verfügt über:

- Zustandsattribut `state` (n-dimensionales Feld)
- erste Ableitung des Zustandsattributfeldes `rate`

  `state` und `rate` sind jreweils die Zustände und Ableitungen zum aktuellen Zeitpunkt!
- eine Funktion `derivatives` zur Codierung der Funktion `f` zur Berechnung von rate
- eine Integrationsmethode `void takeOneStep(h)`
  diese führt einen einzelnen Integrationsschritt aus (Berechnung von rate!) und spuckt auch ein `error`-Feld aus!

Zusätzlich gibt es nun auch noch eine globale Methode zur Umsetzung und Steuerung der numerischen Integeration. `int integrate(TimeEvent, ConditionEvent)`

Dies bedeutet, dass sie einen Integrationsschritt mit der Schrittweite h ausführt und dann die Norm vom Error Vektor bestimmt. Diese Norm wird mit dem vorgegebenen ErrorLimit verglichen.

Falls der Fehler wesentlich kleiner ist, dann kann ich das nächste Mal mit der doppelten Schrittweite weitermachen.

Falls der Fehler größer ist, dann wiederhole ich den Integrationsschritt, nun aber mit halbierter Schrittweite (vorher natürlich den ersten Schritt rückgängig machen!)

Sonst nächster Schritt mit unveränderter Schrittweite.

Natürlich besitzt integrate eine vorgebene Schrittweite, ein errorLimit und auch eine errorNorm (MaximumNorm).

1.4.3 Integration der diskreten Prozesse

Bisher wurden nur die kontinuierlichen Prozesse entsprechend RKM im Ereigniskalendar eingetragen. Diese Schrittweite muß nun aber noch an die Zeitpunkte von diskreten

Prozessen angepasst werden.

Die diskreten Prozesse haben eine *höhere Priorität* gegenüber kontinuierlichen Prozessen bei *Gleichzeitigkeit*.

Es kann aber auch zu **Synchronisationsproblemen** kommen, da es auch eine dynamische Abhängigkeit zwischen Prozessen geben kann. Es kann sich sogar um *Rückkopplungen* handeln.

2 Zeitkontinuierliche Systeme

2.1 Zustandsraum und Trajektorien

n Zustandsgrößen spannen einen n-dimensionalen Zustandsraum auf. Der Weg von einem Zustand zu einem anderen Zustand wird im Zustandsraum als Trjektorie bezeichnet (auch *Zustandsbahn*).

2.2 Wirkungsstrukturen

Es kann zwei Ursachen für Zustandsänderungen geben:

- Einwirkungen von außen und
- Einwirkungen von innen (bzw. das Verhalten der Objekte im System selbst)

Man trifft häufig die Grundannahme, dass die Umgebung rückkopplungsfrei ist. Dies bedeutet, dass die Einwirkungen auf das System unabhängig vom Systemverhalten sind.

Der Zustandsvektor eines deterministischen Systems (und damit jede Zustandsgröße) kann zu jeden Zeitpunkt eindeutig bestimmt werden, wenn

- die Anfangszustände für jede Zustandsgröße
- der Eingangsvektor im Zeitraum und
- die Zustandsfunktion für den Zeitraum

bekannt sind.

Trotzdem kann es zu chaotischem Verhalten kommen!

2.2.1 Die Rolle von Rückkopplungen

Anfangswert, Eingang und Zustandsänderung machen erst die Ableitung einer Zustandsentwicklung möglich! Systeme mit mindestens zwei Zustandsgrößen können bereits Rückkopplungen aufweisen (A verändert den Zufluß von B und umgekehrt). Dies kann zu *Schwingungen* führen, ohne dass das von außen erzwungen wurde.

Es kann bei inneren Rückkopplungen zu determiniertem chaotischen Verhalten kommen. (man kann nicht immer vorhersagbares Verhalten angeben, aber immernoch potentielle Verhaltensbereiche).

Typisches Beispiel für Rückkopplungen sind Räuber-Beute-Systeme:

- die Beute würde ohne Räuber allmählich anwachsen, bis die Nahrung nicht mehr ausreicht
- der Räuber würde ohne Beute verhungern

Also leben beide zusammen und beeinflussen gegenseitig ihre Populationen.

Gibt es hier Gleichgewichte? Ändern sich die Gleichgewichte bei Änderung der Startbedingungen?

Diese Fragestellungen kann man durch vollständige Zustandsraumanalyse beantworten. Hier werden Trajektorien für unterschiedliche Anfangswerte dargestellt. Manchmal gibt es unabhängig vom Startwert einen nicht-trivialen Gleichgewichtspunkt.

2.3 Stabilität zeitkontinuierlicher Systeme

2.3.1 Gleichgewichtspunkte

. . . im Zustandsraum sind natürliche Ruhepunkte eines Systems. Ihre Kenntnis ist für die Beurteilung des Systemverhaltens wichtig.

An diesen Stellen verschwinden die Ableitungen der Zustandsgrößen nach der Zeit, da sich ja nichts mehr verändert. Interessant sind insbesondere die Bewegungen eines Systems in der Umgebung seiner Gleichgewichtspunkte.

Nicht-lineare Systeme können mehrere Gleichgewichtspunkte besitzen. Welcher davon angenommen wird hängt von den Anfangswerten der Zustandsgrößen ab.

2.3.2 Stabile und Instabile Gleichgewichtspunkte

Beispiel: Pendel und Zustandsgröße Auslenkung.

Was ist aber ein Entscheidungskriterium für Stabilität und Instabilität von Gleichgewichtspunkten beim Pendel?

Stabil bedeutet, dass das System von selbst in den Gleichgewichtspunkt *zurückkehrt*. Ein instabiler Gleichgewichtspunkt dagegen bedeutet, dass das System bei Verlassen des Gleichgewichtspunktes nicht zurückkehrt, sondern meist in einen anderen Gleichgewichtszustand übergeht.

Es gibt auch mathematische Verfahren zur Bestimmung der Qualität von Gleichgewichtspunkten.

2.3.3 Wichtige Eigenschaften von Trajektorien

1. Der Verlauf einer Trajektorie ist durch den Anfangswert *eindeutig bestimmt*. Verschiedene Trajektorien schneiden sich i.A. nicht!

2. Eine Trajektorie kann auch nur aus einem Punkt bestehen, dann ist die Ableitung Null!

3. Alle Punkte, wo sich die Trajektorie nicht mehr bewegt heißen *stationäre Punkte* oder *Gleichgewichtspunkte*. Es kann aber auch Gleichgewichtspunkte geben, die durch das System gar nicht erst angenommen werden.

4. System können auch periodisches Verhalten zeigen. Dann sind Trajektorienmuster Kreis, Ellipsen oder andere geschlossene Kurvendarstellungen.

Ein System bewegt sich ausgehend von einem oder mehreren Anfangswerten auf gesonderten Bahnen auf einen Gleichgewichtspunkt zu, ohne diesen zu erreichen.

2.3.4 Beispiele für unterschiedliche Systeme

Ein Schwinger mit einem Grenzzyklus, es gibt eine optimale Schwingung zwischen Auslenkung in x-und-y-Richtung.

Rotationspendel, zunächst volle Rotation, danach stark gedämpft.

Der bistabile-Schwinger (Blattfeder zwischen Magneten) hat drei Gleichgewichtspunkte: In der Mitte und am jeweiligen Extremum. Die Mitte ist instabil, der Rand ist stabil. Die Bewegung endet in einem stabilen Gleichgewichtspunkt.

2.3.5 Typische Aufgabenstellungen bei der Systemuntersuchung

1. Bestimmung von Gleichgewichtspunkten
2. Ermittlung der Qualität von Gleichgewichtspunkten (Formen der Stabilität)
3. Bestimmung der Abhängigkeit von (1) und (2) von Systemparametern und möglichen Eingaben und Belastungen des Systems.

Nur einfache Systeme haben interessante *Punkte im Zustandsraum*. Oft kann man eher nur potentielle Bereiche im n-dimensionalen Zustandsraum angeben.

2.3.6 Chaotisches Systemverhalten

Stochastisch ist nicht chaotisch!

Sind Umwelteinflüsse und Anfangszustand determiniert, so ist ein *chaotisches Systemverhalten* dadurch gegeben, dass sich im Falle kleinster Änderungen in den Umwelteinflüssen oder Anfangswerten, die Systementwicklung dramatisch ändert.

Zum Beispiel streben dann die Trajektorien exponentiell beschleunigt auseinander.

Das künftige Verhalten ist damit *unbestimmbar*.

Daraus folgt, dass aus dem Systemverhalten für einen Anfangswert, nicht das Systemverhalten für einen benachbarten Anfangswert vorausgesagt werden kann. (Weder die Anzahl der Schleifen noch welcher Gleichgewichtspunkt angenommen wird.)

Häufig wird chaotisches Verhalten nur bei *bestimmten Parameterkonfigurationen* festgestellt.

Beispiel ist der chaotisches Bi-Stabile-Schwinger.

2.4 Betrachtung von Systemstrukturen

2.4.1 Grobe Einteilung

- Zeitdiskrete Systeme
- zeitkontinuierliche Systeme
  - nach Anzahl der Zustandsgrößen
  - ein System n-ter Ordnung hat n Zustandsgrößen

    Diese werden modelliert als n Differentialgleichungen 1. Ordnung (äquivalent zu einer DGL mit n Variablen).
- kombinierte System

2.4.2 Beispiele

Systeme 0. Ordnung (Speicherlose Systeme)

$$x = a \cdot sin^2 \omega t$$

Systeme 1. Ordnung (Exponentielles Wachstum)

Die Zustandsänderung hängt nur vom Zustand selbst ab. Es gibt keine äußere Erregung, aber Eigendynamik.

$$z' = (b - d) \cdot z = a \cdot z$$

Der Zustand bestimmt die Zustandsänderung (*Eigenrückkopplung*). Ist hierbei $a < 0$, so handelt es sich um einen exponentiellen Zerfall.

Logistisches Wachstum

Die Tragfähigkeit der Umgebung beeinflußt die Wachstumsrate!

$$z' = a \cdot z \cdot (1 - \frac{z}{k})$$

Im Prinzip ist das ne asymptotische Annäherung an ein Maximum.

Logistisches Wachstum mit konstanter Ernterate

Die Ernterate ist h. h ist ein *kritischer Parameter*, d.h. ab einem bestimmten Wert bricht die Population zusammen.

$$z' = a \cdot z \cdot (1 - \frac{z}{k}) - h$$

Exponentielle Verzögerung bzw. exponentielles Leck

Ist auch ein System 1. Ordnung:

$$z' = u(t) - a \cdot z$$

Die Verluste der Zustandsgröße sind proportional zum jeweiligen Bestand.

Beispiel ist eine Badewanne mit einem undichten Stopfen. Am Beginn hat man eine leere Wanne mit einem konstanten Zulauf, d.h. $u > a \cot z$. Die Wasserhöhe steigt, damit nehmen auch die Leckverluste zu, bis sie allmählich genau den Zulaufwert u erreichen. Dann ist die Wasserhöhe konstant.

Auch wenn sich $u(t)$ zeitlich verändert, so stellt sich der Zustand mit einer gewissen Verzögerung dennoch auf die jeweilige Umweltänderung ein.

Also kurz gesagt, je näher ich mich meinem Ziel annähere, um so langsamer nähere ich mich an. (Exponentiell!)

Systeme 2. Ordnung (Lineare Schwinger)

Dies ist eine typische Rückkopplung, denn bei über mindestens zwei Zustandsgrößen laufende Rückkopplungen sind immer Schwingungen zu erwarten.

In Abhängigkeit von Kopplungsstärke c und Dämpfung d zeigt das System in Gleichgewichtspunkten unterschiedliches Verhalten (stabil, instabil).

$$x' = y$$

$$y' = -c \cdot x - d \cdot y$$

Dieses lineare Verhalten kann zur Bewertung eines lokalen nicht-linearen Verhaltens herangezogen werden.

Nicht-Linearer Schwinger

Dies ist ein Räuber-Beute-System *ohne Kapazitätsbeschränkung.*

$$x' = a \cdot x - b \cdot x \cdot y$$

$$y' = c \cdot x \cdot y - d \cdot y$$

- a - Wachstumsrate der Beute
- a - Beuteverlust der Beute
- a - Beutegewinn der Räubers
- d - Atmungsrate des Räubers

Dies ist eine ungedämpfte Schwingung um den Gleichgewichtspunkt, dabei ist das Verschwinden der Beute unmöglich. Die Parameter bestimmen die Schwingungsfrequenz und die Anfangswerte die Amplitude. Kreisschwingungen, wie ein quergelegtes Ei.

Bistabiler Schwinger

$$x' = y$$

$$y' = x - x^3 - d \cdot y$$

d drückt die dämpfende Eigenkopplung einer Zustandsgröße aus. Ansonsten gibt es eine *lineare Kopplung* zwischen beiden Zustandsgrößen, wie beim linearen Schwinger. Zusätzlich aber noch eine kubische Kopplung zwischen beiden Größen mit negativem Vorzeichen.

Es gibt zwei stabile Gleichgewichtspunkte und einen instabilen. Wo man landet hängt von den Anfangsbedingungen ab. (Die Gleichgewichtspunkte werden aber auch angenommen!)

Chaotischer Bi-Stabiler-Schwinger

$$x' = y$$
$$y' = x - x^3 - d \cdot y + q \cdot cos(\omega t)$$

Der bistabile Schwinger wird zusätzlich durch eine zeitabhängige Winkelfunktion angeregt.

In gewissen Parameterbereichen ergibt sich ein chaotisches Verhalten. Die stabilen Gleichgewichte werden nicht angenommen!

2.4.3 Umformung in Zustandsgleichungen 1. Ordnung

Gewöhnliche Differential und Differenzengleichungen n-ter Ordnung lassen sich durfch Einführung neuer Zustandsgrößen immer in n Gleichungen 1. Ordnung transformieren. (odemX unterstützt nur Systeme 1. Ordnung)

2.5 Ermittlung von Gleichgewichtspunkten und Bestimmung ihrer Stabilität

Ziel ist die vollständige systematische Zustandsraumanalyse zur Bestimmung des Einflusses von Startwerten, Parametern und Eingabegrößen.

Für einfachere Systeme läßt sich das auch analytisch lösen.

2.5.1 Prinzip der analytischen Methode

Das DGL n-ter Ordnung in ein System 1. Ordnung der Dimension n umformen. Dann erfolgt die Berechnung der Gleichgewichtspunkte:

$z'(t) = f(z,t)$ mit der Bedingung $z'(t) = 0$ bringt zum Lösen des Gleichungssystems: $o = f(z,t)$.

Sei f linear, dann lösen von $0 = Az$. Dann sei z^* eine Lösung, also ein Gleichgewichtspunkt.

Die Bestimmung der Eigenwerte kann man dann auf das Systemverhalten in der Umgebung von z^* schlußfolgern.

Sind die Real-Anteile **sämtlicher Eigenwerte** < 0, so gilt *Systemstabilität*. Dies bedeutet, dass die Stabilität von linearen System nicht von der Systemumgebung und von den Anfangswerten abhängt.

Negative Realanteile heißen, dass sich das System ohne Schwingung in Richung des Gleichgewichtspunktes bewegt. Positive Realanteile heißen, dass das System instabil ist und eine Wegbewegung von den Gleichgewichtspunkten stattfindet.

Sind die Imaginär-Anteile nicht Null, so liegen Schwingungen vor!

Bei einer konjugiert komplexen, negativen Doppellösung liegen gedämpfte Schwingungen in Richtung des Gleichgewichtspunktes vor.

Ist der Real-Anteil gleich Null, so ist die Schwingung semi-stabil und schwingt um den Gleichgewichtspunkt.

Falls f nicht-linear ist, dann muß man das nicht-lineare-GLS lösen (Newton-Verfahren). Aber eine globale Systemverhaltensaussage gelingt nicht. Aber man kann in hinreichend kleinen Umgebungen von Gleichgewichtspunkten wieder Aussagen treffen. (Jakobi-Matrix)

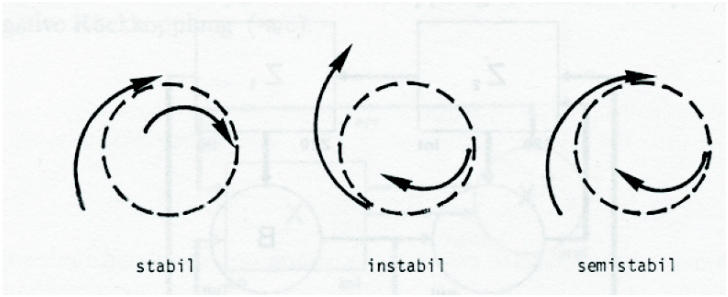
2.5.2 Stabilitätsanalyse nicht-linearer Systeme

Sie unterscheidet sich von der Stabilität linearer Systeme, denn sie ist abhängig vom Eingang $u(t)$. Das autonome System könnte stabil sein, aber die Eingangsbedingungen machen es instabil. Es kann auch stark von den Anfangswerten abhängen, da es mehrere Gleichgewichtspunkte mit unterschiedlichen Stabilitätseigenschaften gibt.

Also muß die Stabilitätseigenschaft *lokal gefasst* werden, ausgehend vom einem Anfangswert, der in in umittelbarer Nähe zum Gleichgewichtspunkt liegt.

In der Umgebung eines Gleichgewichtspunktes kann die Stabilitätsanalyse des nichtlinearen Systems durch eine Analyse eines linearen Systems als Approximation hinreichend erbracht werden.

Bewertung von Zustandsbahnen



Es sind auch mehrere Gleichgewichtspunkte möglich. Es können auch Trennflächen existieren, die Zustandsebenen in Bereiche mit unterschiedlichem Verhaltenscharakter einschließen.

Befindet sich dann ein System einmal auf einem bestimmten Bereich der Trennfläche, so besteht keine Möglichkeit mehr in freier Bewegung auf einen anderen Bereich zu kommen (bi-stabiler Schwinger).

Grenzzyklen sind geschlossene Zustandsbahnen, die nicht überschritten werden können. Sie teilen den Zustandsraum in Regionen mit unterschiedlichem Verhalten. Im mehrdimensionalen Raum nennt man diese Grenzflächen dann Torus (Plural: Tori).

Es gibt also drei Arten von *Attraktoren*: Gleichgewichtspunkte, Grenzlinien und Torus.

Unvorhersagbare Systeme haben zwar Attraktorflächen, in den sich der Zustand nach einer gewissen Zeit befinden muß, ohne dass jedoch sein Ort mit Sicherheit vorhergesagt werden kann.

3 Prozessmodellierung mit ODEMx

3.1 Entwurfsmaxime

Baut auf ODEM (Object Discrete Event Modelling) auf, nach dem Vorbild von Simula, C with Classes, ... Ziel ist eine leichte Bedienbarkeit mit einer automatischen Sammlung von Modellwerten (zur statistischen Auswertung) bei Bereitstellung von Zufallsgeneratoren, Synchronisationsverfahren und numerischen Integrationsverfahren.

3.1.1 Prinzip

Ein C++-Haupt-Programm erlaubt die pseudo-parallele Simulation mehrerer Systeme mit individueller Stack-Verwaltung ihrer jeweiligen Prozesse.

Die Prozesses eines Systems werden dynamisch verwaltet und in Abhängigkeit ihres jeweiligen Modellzeitverbrauchs alternieren ausgeführt. (Dazu gibt es einen Prozessterminkalendar je System)

Wir beschränken uns auf 1 System mit pseudo-parallelen Prozessen.

3.1.2 Modellklassen

Es gibt *zeitdiskrete Prozesse*. Trace ist dann eine Folge von Ereignissen. Die Ereignisaktionen operieren jeweils hauptsächlich nur über einen partiellen Systemzustand (die Prozess-Attribute).

Es gibt auch noch *zeitkontinuierliche Prozesse*. Hier wird der Trace approximiert durch eine sehr dichte Folge von Ereignissen. Nicht unbedingt äquidistante Schritte! Man kann hier auch einen Prozess abschalten, sobald eine Zustandsgröße unter einen Wert fällt.

3.2 Struktureller Aufbau von ODEMx

Man unterscheidet aktive Klassen und passive Klassen.

Es gibt die Module:

- `Utilties`
- `Random` für Zufallszahlen
- `Statistic` für statistische Analyse
- `Base` für die allgemeine Prozessverwaltung (enthält `continuous`, `process` und `Simulation`.
- `Coroutine` (nicht für direkte Nutzung)
- `Synchronisation` für die Prozesssynchronisation

In der Anwendung definiert man dann eine `DefaultSimulation: Simulation`, zeitkontinuierliche Prozesse die von `Continuous` erben oder normale Prozesse, die von `Process` erben.

3.3 Prozessverwaltung mit der Klasse Simulation

Aufgabe:

- Verwaltung der zustandsabhängigen Prozesslisten (insbesondere Prozessterminkalendar)
- Erfassung sämtlicher Modellelemente als Objekte von ODEMx-Klassen

Also die Bereitstellung von Kontextinformationen für die Simulation eines Systems.

Dabei kann man die *elementare Prozessverwaltung* nutzen (Nutzung von `DefaultSimulation`) oder eine *hierachische Prozessverwaltung* als nebenläufige Prozessverwaltung von Teilsystemen einbringen. (Mehrere nutzerdefinierte Simulation-Ableitungen).

3.3.1 Grundidee der elementaren Prozessverwaltung

Das Hauptprogramm und die Prozesse eines Simulationskontextes bilden ein Prozess-Ensemble.

Ein Prozess kann sich in verschiedenen Grundzuständen befinden: *generiert*, *aktiv*, *passiv*, *terminiert* und *vernichtet*. **Zu einem Zeitpunkt kann immer nur einer der aktivien Prozesse ausgeführt werden.** Andere aktive Prozesse sind während dieser Zeit *suspended*.

Der *Current-Prozess* ist der erste Eintrag an der Stelle mit der kleinsten Zeit.

Im Prozess-Termin-Kalendar gibt es einen Zeitstrahl und zu jedem Zeitpunkt *aktiviert Prozesse*, *passive Prozesse* und *terminierte Prozesse*.

3.3.2 Grundidee einer hierachischen Prozessverwaltung

Es gibt nun mehrere Simulationskontexte und das C++ Haupt-Programm ist nun der Mittler zwischen den Prozess-Systemen. Diese verschiedenen Prozessverwaltungen können nebenläufig agieren.

Das Hauptprogramm und die Prozesse aller Simulationskontexte bilden ein *hierachisches Prozess-System*.

3.3.3 Ein einfaches Beispiel

Objekte aktiver Klassen brauchen auch immer einen Ausführungskontext! Als Zeiger zu einem Simulation-Objekt!

```
class TimeTicker : public Process{
   public:
       TimeTicker(Simulation* sim) : Process(sim, "Ticker") {} /* Label mitgeben */

        virtual int main() {
             while(true) {
              holdFor(1.0);  /* Scheduling Operation */
              cout << ".";
             }
             return 0;
        }
```

```
};
```

Das Hauptprogramm könnte dann so aussehen:

```
int main(){
  TimerTicker* myTicker = new TimerTicker(getDefaultSimulation());
  myTicker->activate();

  for (int i=1; i<5; ++i) {
     getDefaultSimulation()->step();  /* nur ein Ereignis als Schritt */
     cout << getDefaultSimulation()->getTime();
  }
  getDefaultSimulation()->runUntil(13.0); /* Mehrere Ereignisse in Folge bis 13.0 */
```

Beginn der Simulationszeit ist 0.0. Dies ist auch der kleinste PTK-Eintragszeitpunkt. Der erste PTK Eintrag ist der Aktivierungszeitpunkt des current-Prozesses. Die Erzeugung und Aktivierungsvormerkung der initialen Prozesse läuft im Hauptprogramm ab.

`step()` aktiviert das Prozess-Ensemble, also immer den ersten PTK-Eintrag ausführen. Aktivierung bedeutet Wechsel von main zu MyTicker.

`activate()` - aktiviert einen Prozess, d.h. er wird in den Terminkalendar eingetragen.
`step()` - gibt die Steuerung vom Hauptprogramm zum Simulationskontext ab. Erst jetzt beginnt der Prozess zu arbeiten!

`holdFor()` - *Modellierungsmuster für zeitverbrauchende Zustandsübergänge*
Verzögerung des aktuellen Prozesses. Änderungen der Zustandsgrößen können somit Zeitverbrauch verbunden werden. Falls ein anderer Prozess damit Current wird, wird die Steuerung abgegeben.

3.3.4 Varianten der Simulationsausführung

- `step()` - *Einzelschrittausführung*, Rückkehr nach einer Ereignisrealisierung in `Current`
- `runUntil()` - *Zeitabhängige Ausführung*, Rückkehr nach Erreichen einer vorgegebenen Modellzeit
- `run()` - *Geschlossene Ausführung*, Rückkehr ins Hauptprogramm nach `run()`

- Implizite Beendigung (PTK ist leer, kein aktiver Prozess mehr)
- Explizite Beendigung (Ein Prozess ruft `exitSimulation`

Prozesse können auch mehrmals aktiviert werden, wenn der RunModus gewechselt wird.

3.3.5 DefaultSimulation

Ist eine vordefinierte Kontextklasse, von der es höchstens ein Objekt geben darf. Dabei ist `initSimulation()` einfach eine leere Funktion. Daher sind Prozesse und andere Objekte im Hauptprogramm zu erzeugen und zu initialisieren.

Per `getDefaultSimulation` erhält man einen Zeiger auf das DefaultSimulations-Objekt! Dies kann man nicht selber erstellen!

3.3.6 Nutzerdefinierte Simulationsklasse

Man kann Simulation auch in eine eigene Klasse ableiten. Dann darf davon aber auch nur ein Objekt erstellt werden.

Dann müssen auch die Funktionen `initSimulation()` zur Initialisierung einer Simulation und `exitSimulation()` bedacht werden.

4 ODEMx Modul BASE

4.1 Grundbegriffe und Einführung

Die mehrstufige Ableitung bis hin zum Prozess (Coroutine, Process, Continuous, Ingot) nennt man *Mehrstufige Prozessdefinition*.

Die Simulationsklasse steckt auch in Base drin! Sie stellt einen public-Zugang zum sicheren Prozess-Scheduling bereit und erbt von ganz vielen Klassen.

4.1.1 ExecutionList

Eine davon ist `ExecutionList`, die private Grundfunktionalität zur Verwaltung von Prozesslisten nach Zeit und Priorität zur Verfügung stellt. (PTK ist also von ExecutionList geerbt (Modul Utilities)).

Es gibt auch noch weitere Listen in Simulation: `created`, `runnable` (doppelt zu PTK), `idle`, `terminated`

ProcessQueue ist durch STL Container implementiert.

Prozesse sind in verschiedenen Queues nach Ereigniszeit und Priorität sortiert.

4.1.2 Prozessreihenfolge

Das Sortierungsschema für die Prozesse im PTK ist Ereigniszeit, dann Priorität und dann LIFO/FIFO.

Falls die geplante Zeit kleiner ist, als die aktuelle Zeit, wird einfach die Zwangsannahme der aktuellen Zeit gemacht.

In anderen Listen wird nur nach Priorität sortiert, denn da ist die Ereigniszeit unbekannt.

4.1.3 Zeitbezug

Modellzeitdatentyp wird durch die beiden Varianten von OdemX bestimmt. So kann die Simulationszeit entweder `int` oder `double` sein. `now` gibt die aktuelle Modellzeit zurück. (private Variable im Simulationskontext)

Der Zugriff erfolgt nur lesen mittels `getCurrentTime()` und `getSimulation()->getTime()`.

Die geplante Aktivierungszeit eines beliebigen Prozesses p im PTK erhält man durch `p->getExecutionTime()`.

Damit erhält man die folgenden semantischen Äquivalenzen:

- `now` ist semantisch gleich zu
- `getCurrentTime()` und zu
- `getSimulation()->getTime()` und ebenfalls zu
- `getCurrentProcess()->getExecutionTime()`

Funktionssignaturen

```
SimTime Process::getExecutionTime() const; /* 0.0 falls nicht im PTK eingetragen */
Process* Simulation::getCurrentProcess();
Simulation* getSimulation();
```

Für die verschiedenen Listen mit Prozessen gibt es auch die verschiedenen get-Methoden, entsprechend den möglichen Grundzuständen von Prozessen.

```
std::list<Process*>& Simulation::getCreatedProcesses()    { return created;    }
std::list<Process*>& Simulation::getRunnableProcesses()   { return runnable;   }
std::list<Process*>& Simulation::getIdleProcesses()       { return idle;       }
std::list<Process*>& Simulation::getTerminatedProcesses() { return terminates; }
```

Nun könnte man auf dieser STL-Liste mittels Iterator arbeiten:

```
typedef std::list<Process*> LIST;
for (LIST::iterator it = myList.begin(); it != myList.end(); ++it)
{
 *it->....
}
```

4.1.4 Prozess Scheduling

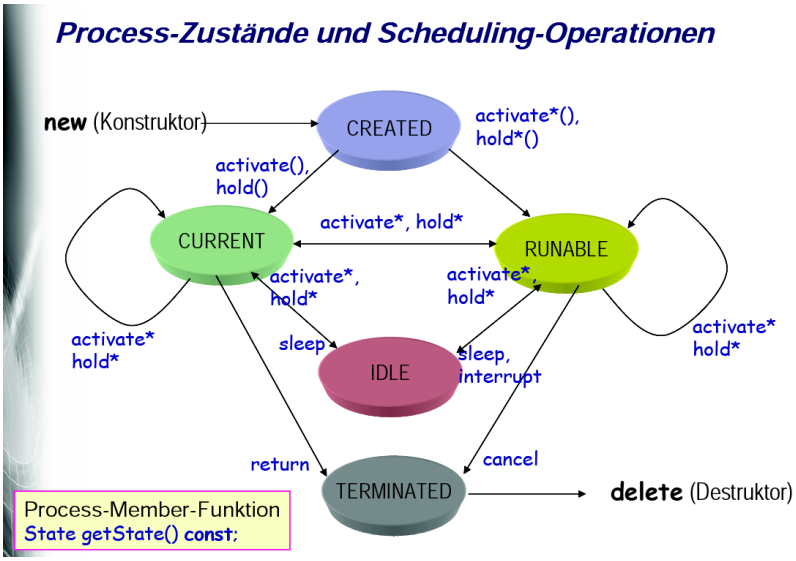
Grundstrategie

Ein Prozess kann in seinem gesamten Lebenslauf nur einem Simulationskontext zugeordnet werden und ist während seines Lebenslaufes in vier unterschiedlichen Zustandslisten seines Simulationskontextes erfasst.

Der Prozess kann sich aber pro Zeitpunkt nur in einer der vier Listen befinden! Ist er aktiv, dann ist er **zusätzlich** im PTK!

Aber, ein Prozess kann sich in maximal einer weiteren Nutzerwarteschlange (ProcessQueue) befinden.

Scheduling-Operationen



- *Prozessaktivierungen*

  `void activate()` - Sofortaktivierung, *falls aktueller Current keine höhere Priorität aufweist.* Eintrag im PTK mit NOW und nach LIFO

  `activateIn(t)` und `activateAt(t)` (falls $t < NOW \rightarrow t = NOW$) hier wird der Prozess einfach eingetragen, aber es findet kein Prozesswechsel statt (außer

wenn NOW und Prio größer sind)

Gilt aber nicht bei Aktivierung aus dem Hauptprogramm! Hier wird der neue Prozess einfach nur entsprechend seiner Priorität und Zeit eingeordnet, es findet aber kein Prozesswechsel statt. Das Hauptprogramm setzt die Ausführung fort.

`activateBefore(Process* p)`,`activateAfter(Process* p)` unmittelbarer Eintrag vor/nach p mit Zeit von p.
Falls $p = this$: leere Anweisung. Falls p $\notin$ PTK: Fehlermeldung.

- *Prozessverzögerungen*

  `hold()` - Eintrag zur aktuellen Ereigniszeit als **letzter** bei gleicher Priorität (FIFO)
  `holdFor(t)` und `holdUntil(t)` Eintrag zu relativen/absoluten Zeit t (sonst wie oben)

- *Prozessunterbrechnungen*

  `sleep()` - Entfernung aus dem PTK nach IDLE mit $t = 0.0$, dann Wechsel an 1. runnable-Prozess
  `interrupt()` - Unterbrechnung der hold/activate-Phase eines Prozesses und wird dann Current. Mittels `getIntterupter()` kann die Unterbrechung behandelt werden.
  `cancel()` - Prozessabbruch und Entfernung aus dem PTK nach TERMINATED

- *Interrupt-Routinen*

  `bool isInterrupted` - Abfrage des Interrupt Zustandes nach erfolgtem Interrupt
  `Process* getInterrupter` - Anzeige des Prozesses, der interrupt gerufen hat.
  `resetInterrup()` - löscht Interrupt Zustandseinträge

- *Prozess-Prioritäten* - je höher umso dringlicher

  `Priority getPriority`, `Priority setPriority(Priority newPrio)` - ziehen eventuell einen **Re-Scheduling** und dann ggf. einen **Prozesswechsel** nach sich!

- *Rückgabewerte* bei Terminierung mittels Return

  `bool hasReturned()`, `int getReturnValue()` (ohne Blockierung, also erst testen mit hasReturned!)

- *Prozess-Warteschlangen*

`getQueue` liefert Zeiger zur Warteschlange eines Prozesses
`getEnqueueTime` liefert Eintrittszeitin die aktuelle Warteschlange
`getDeqeueuTime` liefert die Zeit des Verlassens der Warteschlange

Memberfunktionen einer ProcessQueue (getTop, isEmpty, getLength, popQueue, remove, inSort)

4.1.5 Process Verhaltensfunktion

...ist einfach definiert:

```
virtual int main() = 0;
```

4.1.6 Anwendung der Funktionstypen

...`Selection` und `Condition`.

In Member-Funktionen von Process-Ableitungen werden diesen Funktionstypen durch konkrete Anwendungen festgelegt. Zum Beispiel zur Überprüfung von globalen Bedingungen (Condition) oder zur Überprüfung von Eigenschaften eines Prozesses (Selection). Das Ergebnis ist Boolean!

Der Aufruf dieser Memberfunktionen der Anwendung soll aber im Funktionskörper von Funktionen der ODEMx Bibliothek erfolgen, obwohl sie die konkreten Memberfunktionen nicht kennen. **Funktionsvariablen werden genutzt.**

4.2 Modul BASE: Continuous

Dies ist die Basisklasse für alle nutzerdefinierten, zeitkontinuierlichen Prozesse als zeitdiskrete Approximation (durch schrittweise numerische Integration bei Überwachung des Integrationsfehlers und übergeordnete Steuerung der Schrittweite unter Beachtung der Synchronisationsanforderungen).

`state[]` ist die eigentliche Zustandsgröße, `rate[]` ist die 1. Ableitung davon! (Als Vektor implementiert!)

Memberfunktionen

- `derivatives(t)` Modellierung von Zustandsgleichungen
- `takeAStep(h)` Einschritt-Integrationsverfahren
- `int integrate(time, condition)` Startet integrationsaktivität (zeitkontinuierliche Zustandsänderungen)
- `int main()` virtuelle Funktion zur nutzerdefinierten Modellierung des Prozessverhaltens

4.3 Modul BASE: HtmlTrace

Aufzeichnung von Simulationsereignissen mittels Trace-Clients. Es können Trace-Erzeuger definiert werden, die eigene Ereignisse ergänzen. Filter können die Menge der Aufzeichnung reduzieren.

4.4 Modul BASE: HtmlReport

Liefert aggregierte statistische Informationen einer Simulation oder einer Modellkomponente.

4.5 Modul BASE: ContuTrace

Dient der Beobachtung von Zustandsänderungen von zeitkontinuierlichen Prozessen! Ausgabe erfolgt als Textfile. Die 1:1 Beziehung zwischen dem ContuTracer und seinem Continuous Objekt muß durch den Nutzer garantiert werden.

Graphisch ist eine Ausgabe durch ODEMplot möglich!

5 ODEMx Modul RANDOM

5.1 Charakterisierung von Zufallsgrößen

Einflüsse der Systemumgebung oder Systemabläufe selbst unterliegen häufig dem Zufall. Die Simulation ist dann eine *Stichprobe* eines *statistischen Experimentes*.

Auf der Grundlage von vielen Stichproben können statistische Kennwertprofile abgeleitet werden.

Dabei ist es besonders wichtig, dass der Einfluß des Zufalls im Simulationsexperiment wiederholbar dargestellt werden kann.

Zufallsgrößen sind hier eigentlich Zahlen. Es gibt *diskrete Zufallsgrößen* und *stetige Zufallsgrößen*.

Eine Verteilungsfunktion := $F(x) = P(X < x)$ - der Wert von F ist an der Stelle x gleich der Wahrscheinlichkeit, dass X einen Wert unterhalb von x annimmt. Dabei durchläuft x alle Werte einer reelen Zahlengerade. Sie ist im Prinzip eine *kumulative Dichtefunktion*.

5.1.1 Diskrete Zufallsgrößen

Die *Poisson-Verteilung* beschreibt Ereignisregistrierungen in einem bestimmten Zeitbereich. Sie steht im Zusammenhang mit der Exponentialverteilung:

Sei X eine poisson-verteilte Zufallsgröße mit dem Erwartungswert λ, dann ist die Zwischenankunftszeit zweier aufeinanderfolgender Ereignisse exponential-verteilt mit dem Erwartungswert $\frac{1}{\lambda}$.

5.1.2 Stetige Zufallsgrößen

Bei einer stetigen Zufallsgröße X, nimmt X bei jedem Versuch einen zufällig bestimmten Wert an. Diese Werte genügen einer negativen Exponential-Verteilungsfunktion.

5.2 Klassen des Random-Moduls

DIST ist ABC aller Zufallszahlengeneratoren und bietet einen ganzzahligen Generator für (0,1)-gleichverteilte Zufallsgrößen.

Ableitungen von Dist transformieren diese (0,1)-Folge in verschiedene Verteilungsfunktionen.

`iDist` für diskete Zahlen und `rDist` für stetige Zufallszahlen.

Ein `DistContext`-Objekt stellt einen gemeinsamen Kontext für die verschiedenen Zufallszahlengeneratoren (Objekte von Dist-Ableitungen) dar. Mittels `getSample()` wird der nächste (0,1)-Zufallswert bestimmt.

Die Startwerte werden im DistContext durch `getNextSeed()` bereitgestellt. Es können auch per `setSeed()` eigene Urwerte im Context oder im Dist-Objekt gesetzt werden.

Es ist eine Protokollierung des Dist-Objektes möglich, um die Güte der Verteilungen zu analysieren.

```
HtmlReport* myReport;
Negexp* myRandom;
/* init ... */
myReport->addProducer(myRandom),
/* ... */
myReport->generateReport();
```

Nach der Simulation kann man einen Zufallsgenerator mittels `reset()` wieder zurücksetzen (Reset der Aufrufanzahl) wird dabei in der Reihenfolge durchgeführt:

1. reset für die Simulation (als DistContext Objekt)
2. reset für ein einzelnes Report-Objekt
3. reset für ein einzelnes Zufallszahlengenerator-Objekt

5.2.1 Ableitungen von rDist

- NEGEXP für Exponentialverteilung
- NORMAL für Normalverteilung
- UNIFORM für Gleichverteilung
- RCONST für konstanten Wert

Eine einheitliche Schnittstelle kann über einen *polymorphen* Zeiger realisiert werden.

5.3 Generatoren

5.3.1 Exponentialverteilung

Transformationsgenerator für exponential-verteilte Zufallswerte mit Erwartungswert a. Dichtefunktion $f(x) = ae^{-ax}$

```
Dist* arrival;
arrival = new Negexp(sim, "Ankunft", 6.0);
double d = arrival->sample();
arrival->report();
arrival->reset();
```

Es gibt hier veschiedene Formeln, wie man auf die Zufallszahlen kommt, wenn man eine (0,1)-Verteilung hat. Das nennt man dann *Transformation*.

5.3.2 Normalverteilung

Hier muß einfach als Parameter der Mittelwert und die Standardabweichung eingegeben werden.

5.3.3 Gleichverteilung

Hier muß dann das Intervall angegeben werden. Die Transformation zu einem Wert lautet: $x = a+(b-a)\cdot y$, wobei y der Wert aus der (0,1)-Folge ist.

5.3.4 Empirische Verteilung

Brauchen ein aufgezeichnetes Histogramm einer beobachteten Größe. Damit suchen wir die Häufigkeit der Werte in den einzelnen Klassen und bestimmen daraus die kumulative Häufigkeit. Es ergibt sich ein Polygon-Zug mit verschiedenen Stützstellen.

Hier wird einfach die 0-1-Folge als kumulative Wahrscheinlichkeit genommen und nach dem Zufallsgröße (x-Wert) gesucht.

5.3.5 Generatoren für Ja/Nein Entscheidungen

... bei einer angegebenen Wahrscheinlichkeit p für ja. $x = p > y$.

6 ODEMx Modul STATISTIK

6.1 Simulationsbericht

Report ist die Basisklasse für nutzerdefinierte Berichtsobjekte, sie verwaltet: *Tabellen*, *ReportProducer* und die *Berichtserzeugung*.

Jedes Report-Objekt stellt einen individuellen Bericht dar. Daher kann es auch mehrere Reports pro Simulation geben.

Mittels `addProducer, removeProducer` lassen sich die Tabelleninhalte eines Reports dynamisch verwalten.

Es lassen sich auch Profile von Modellgrößen anlegen (natürlich auch mit Mittelwert, StdAbw, Min, Max, ...)

`Tab` ist die Basisklasse für alle Tabellen. Hier gibt es verschiedene Möglichkeiten *Count*, *Sum, Tally, Accum (Histo)* und *Regress*.

6.2 Anwendung der Statistik-Klassen

Es gibt ein relativ einheitliches Nutzungsschema:

1. pro Variable ist ein Beobachter-Objekt zu konstruieren. Dieses übernimmt die explizite Erfassung der Variablenwerte

2. zu diskreten Zeitpunkten wird der Wert beobachtet `xobs->update(x)`

3. die Profile der Größen als Tabelleneinträge können generiert werden `xobs->report()`

4. Ausblenden von Einschwingphasen: `xobs->reset()`

6.3 Count

Braucht eine zeitdiskrete Variable von Typ int. Es fungiert als Zähler und bestimmt das Profil zu einem beliebigen Zeitpunkt.

Ausgabe ist der Stichprobenumfang (anzahl der Änderungen) und der aktuelle Wert.

Update macht hier ein `+=`.

6.4 Sum

Funktion ist die Summenbildung und die Bestimmung eines Profils.

Ergebnis ist wieder die Anzahl der Änderungen und der aktuelle Wert.

Bei einer Erhöhung um 15.3: `s->update(15.3)`

6.5 Tally

Erfasst Werteänderungen von Variablen. Das Profil ist unabhängig von der Dauer einer Wertbelegung. Man kriegt wieder Umfang, Min, Max, StdAb, Mittelwert

Jeder neue Wert wird geupdatet: `t->update(x)`

Die Berechnung der statistischen Größen erfolgt natürlich erst zur Reportzeit!

6.6 Accum

Braucht entweder eine zeitdiskrete oder zeitkontinuierliche Variable vom Typ double. Auch hier werden Werteänderungen erfasst und **ABER abhängig von der Dauer einer Wertebelegung!** Man kriegt auch wieder Anzahl, Min, Max, Mittelwert, StdAbw raus.

Für zeitdiskrete Aktualisierungen nimmt man update. Für zeitkontinuierliche würde man integrate nehmen, aber das ist noch nicht implementiert.

6.7 Histo

Ist wie Tally (also unabhängig von der Dauer der Wertebelegung, aber mit einer zusätzlichen Erfassung in vorgegebenen Werteklassen!

```
Hist *h = new Hist(sim, "", 1.0, 300.0, 25);
```

6.8 Regress

Braucht ein Paar zeitdiskreter Variablen vom Typ double, dann erfasst es die Werteänderungen des Paares und bestimmt die angenommene lineare Abhängigkeit für ein Intervall. Dies erfolgt wieder unabhängig von der zeitlichen Dauer der Wertebelegung. Ziel ist die Bestimmung eines Korrelationskoeffizienten (Maß für den Zusammenhang zwischen X und Y). (Maß der quadrierten Abstände muß minimal sein!)

7 ODEMx Modul SYNCHRONISATION

7.1 Einführung zu ProcessQueue

Diese ProcessQueues werden im Modul Synchronisation zur Verwaltung von schlafenden oder blockierten Prozessen verwendet (IDLE Zustand).

Mittels `awake` oder `awakeFirst` können blockierte Prozesse reaktiviert werden.

Die Synchronisation wird nun mit einer Ableitung von ProcessQueue realisiert (Queue), die zudem noch ein Statistikmodul mittels Accum enthält.

Bin und *Res* dienen der Prozess-Synchronisation bei einer Zuordnung geteilt genutzter Ressourcen. Ein aktiver Nutzerprozess blockiert, falls die Ressourcen in dieser Menge nicht zur Verfügung stehen. Ein blockierter Nutzerprozess wird fortgesetzt, falls die Ressourcen wieder vorhanden sind.

Intern werden Token verwendet, die die Ressourcen abstrakt modellieren.

Das Prinzip ist, dass ein Nutzerprozess in Konkurrenz eine Menge an Token entnimmt und sie nach gewisser Zeit wieder abgibt, aber nicht unbedingt an das gleiche Bin/Res-Objekt!

7.2 Klasse BIN

Bin erlaubt beliebig viele Token in der verwalteten Menge und gilt als Modell für einen unbegrenzten Puffer zwischen Produzenten und Konsumenten.

Dazu besitzt das BIN-Objekt zur Verwaltung von wartenden Prozessen ein privates `Queue`-Objekt. (`Queue* takeWait`). Die initiale Tokenanzahl wird per Konstruktor mitgeteilt.

Memberfunktion `take(n > 0)` läßt den Rufer warten, bis `n`-Token zur Verfügung stehen, aber Warteaktion ist durch einen weiteren Prozess unterbrechbar. (Return Value bei Interrupt ist 0.)

Memberfunktion `give(n > 0` gestattet die zeitlose Rückgabe von Token. Token müssen nicht demselben Bin-Objekt zurückgegeben werden!

```
unsigned int take(unsigned int n);
void give(unsigned int n);
unsigned int getTokenNumber() const { return tokenNumber; }
```

7.2.1 Implementation von take

Das Warten in `take` kann von einem anderen Manager-Prozess mittels `interrupt` unterbrochen werden. Damit wird auch der Rückgabewert auf Null gesetzt.

```
unsigned int take(unsigned int n) {

  /* Aktuellen Prozess in Warteschlange einreihen */
  takeWait.inSort(getCurrentProcess());

  if (n > tokenNumber or takeWait.getTop() != getCurrentProcess()) {
    ...
    while(n > tokenNumber or takeWait.getTop() != getCurrentProcess()) {
       getCurrentProcess()->sleep();

       /* Wenn ich im Warten unterbrochen wurde, dann 0 zurückgeben */
       if (getCurrentProcess()->isInterrupted()) {
            takeWait.remove(getCurrentProcess());
            return 0;
       }
    }
 }

 tokenNumber -= n;
 takeWait.remove(getCurrentProcess());
 awakeFirst(&takeWait); /* Aktivierung des nächsten Prozesses */
 return n;
}
```

7.2.2 Implementation von Give

```
void give(unsigned int n) {
   ...
   tokenNumber += n;
   ...
   awakeFirst(&takeWait);
}
```

7.2.3 Nutzung durch einen Process

```
Bin *service= new Bin ("service", 3);

class User: public Process {
  int no;
  User (int n), no(n), Process (...) { ...};

  int User::main() {
       service-> take(2);
       holdFor (5.0 * no);
       service-> give(2);
       ...
  }
 };
```

7.3 Klasse RES

RES beschränkt die maximale Tokenzahl und steht damit eher für ein Modell eines Parkplatzes mit beschränkten Stellflächen.

Dies Klasse besitzt ein privates Queue Objekt `acquireWait`. Per Konstruktor werden initiale und maximale Tokenzahl eingestellt.

Memberfunktion `acquire(n)` ist die unterbrechbare Warteaktion bis n Token verfügbar sind. Rückgabewert ist n, bei Interrupt 0.

Memberfunktion `release(n)` ist die zeitlose Rückgabe von Token. Wird die maximale Tokenzahl überschritten findet eine Fehlermeldung statt und die überflüssigen Token werden ignoriert. (Natürlich können die Token auch aus einem andern Bin/Res Objekt kommen.

Per `control(n)` kann dynamisch die Tokenzahl erhöht werden. Es findet eine Aktivierung des ersten wartenden Prozesses statt (`awakeFirst`).

`uncontrol(n)` stellt sicher, dass n Token überhaupt noch verfügbar sind, so ist ggf. der Rufer von uncontrol zu unterbrechen.

7.4 Fazit: BIN / RES

Die Synchronisation ist sehr primitiv und damit sehr interpretationsbedürftig. Die Token tragen keine Informationen, also eingeschränkte Ausdruckskraft. Dennoch ist eine time-out-überwachte Blockierung möglich. Wenigstens gibt es eine statistische Überwachungs- und Analysemöglichkeit.

7.5 Trace Prinzipien

Ziel ist die chronologische Erfassung der Ereignisse, obwohl weder die Objekt noch deren nutzerdefinierte Attribute bekannt sind (außer State-Vektor).

Dabei werden ODEMx Objekte TraceProducer und können Marken im Laufe der Zeit erzeugen. Eine solche Marke ist durch ihren MarkType zusätzlich bestimmt. Diese Markierungen werden dann einem TraceManager übergeben, der sie an alle registrierten TraceConsumer weiterleitet.

Dabei kann man noch einen Filter mittels `setFilter` für bestimmte Ereignisse setzen.

7.6 Report

Ein Report wird immer zu einem bestimmten Zeitpunkt erstellt und gibt dann Auskunft über Statistiken von registrierten Objekten. Dennoch ist das ReportFile nicht optimal, es wäre eine Statistik-Anpassung notwendig.

7.7 WaitQ

7.7.1 Einführung

Ziel ist die Modellierung der Erbringung einer *gemeinsamen Kooperationsleistung*. Dabei werden die Slave-Prozesse als Ressourcen für den Master-Prozess angesehen.

So bilden sich zeitweilige Kooperationsgemeinschaften, mit unterbrechbarem Warten, falls nicht ausreichend Kooperationspartner zur Verfügung stehen.

7.7.2 WaitQ-Eigenschaften

Eine Master Prozess lassen sich beliebig viele Client-Prozesse zuordnen. Der Master bestimmt alleine die Dauer der Kooperationsleistung und gibt die Clienten danach wieder frei.

Dabei realisiert der Master alleine die *Zustandsänderungen*, die mit der Kooperation verbunden sind (→ Zugriffsrechte sind nötig).

Beispiel: Für den Transport mit der Fähre ordnen sich die Autos für die Dauer des Transports der Fähre unter und werden erst nach Beendigung der Fahrt wieder freigegeben.

WaitQ besitzt zwei Queues: eine für die Master-Prozesse und eine für die Slave-Prozesse.

`bool wait()` - Aufrufer-Prozess wird zum Slave und wartet auf den Master-Prozess, falls momentan gerade keiner verfügbar ist. Wenn ja, wird der erste wartende Master-Prozess aktiviert. Der Rückgabewert liefert Auskunft über Aktivierung durch Master (true) oder Interrupt beim Warten (false).

`Process* coopt(selection)` - Aufrufer-Prozess wird zum Master und wartet auf Slave-Prozesse. Falls keiner da ist, wird blockiert. Wenn ein Slave-Prozess verfügbar ist, so wird der erste wartende Client zurückgegeben.

`Process* avail(selection)` - Liefert einfach nur den ersten wartenden Slave-Prozess, für den eine Bedingung gilt.

`selection` ist ein Function Pointer für eine Funktion, die einen Process* übergeben kriegt und einen Boolean-Wert zurückgibt. Diese Funktion ist als Member-Funktion einer Prozess-Ableitung bereitzustellen.

Wenn mittels coopt kein Slave ausgewählt werden konnte, so blockiert der Master, bis zum nächsten Wait eines Slaves. Eine Zustandsänderung, die zwischendurch passierte, wird vom Master nicht registriert.

Es gibt keine spezielle Funktion zum Reaktivieren von Clients! Man kann einfach `activate()` nehmen. Aber auf keinen Fall `interrupt`, denn sonst liefert wait false zurück!

7.8 CondQ

Grundlage hier für sind Ereignisse mit einer Zustandsbedigung, d.h. erst mit der Erfüllung der Bedingung soll das Ereignis eintreten.

Dabei geht man davon aus, dass es mehrere Prozesse im System gibt und sich der Zustand von diesen Prozessen mit der Zeit ändert.

ODEMx unterscheidet hier zwischen der Behandlung für zeitkontinuierliche und zeitdiskrete Prozesse. Bei kontinuierlichen Zustandsgrößen kann im Aufruf von integrate zusätzlich zum Zeitpunkt noch eine Condition mit übergeben werden (parameterlose Funktion mit Bool Returntyp). Wenn diese Condition true wird, wird integrate abgebrochen.

Zeitdiskrete Prozesse können dagegen die CondQ benutzen und hier `wait(Condition)` aufrufen. Auch hier wartet der Rufer bis die Condition erfüllt ist.

Innerhalb von CondQ gibt es auch noch eine `signal()` Funktion, die alle wartenden Prozesse weckt und ihre erneute Überprüfung der Condition anstößt. Sinnvollerweise sollten nur die Prozesse signal() rufen, die auch Zugang zu den Zuständen der wartenden Prozesse haben.

7.8.1 Implementation von wait()

```
bool Condq::wait (Condition cond) {
   processes.inSort (getCurrentProcess());
   // statistics
   SimTime t=env->getTime();

   while (!(getCurrentProcess()->*cond)() ) {
           getCurrentProcess()->sleep();
           if (getCurrentProcess()->isInterrupted()) {
```

```
                    processes. remove(getCurrentProcess());
                    return false;
            }
    }
    processes.remove(getCurrentProcess());

    // statistics
    t = env->getTime() - t;
    sumWaitTime += t;
    if (t==0) zeros++;
    users++;
            ...
    return true;
}
```

7.8.2 Implementation von signal()

```
void Condq::signal() {
    // trace
    getTrace()->mark( this, markSignal, getCurrentProcess());
    // observer
    ...
    // statistics
    signals++;
    if (processes.isEmpty())
          return;
    // test conditions
    awakeAll(&processes);
}
```

8 Synchronisation von zeitdiskreten und zeitkontinuierlichen Prozessen

8.1 Scheduling von zeitkontinuierlichen Prozessen

Im Prinzip wie diskrete Prozesse mit einem `holdFor(h)`, wobei `h` die Schrittweite ist.

Aus dem `integrate(30)` wird einfach eine Schleife in der eine private t-Variable schrittweise hochgezählt wird und ein holdFor für die Schrittweite gemacht wird. Natürlich wird auch die Integration von t bis t+h vollzogen.

Die numerischen Integrationsverfahren unterscheiden sich durch die Güte ihres *Diskretisierungsfehlers.*

Je kleiner die Schrittweite ist, umso größer die Rundungsfehler. Je größer die Schrittweite ist, umso größer ist der Diskretisierungsfehler. Daher gibt es ein optimales Intervall für die Schrittweite, die ja eh dynamisch angepasst wird.

8.2 Merkmale von Continuous Processen

8.2.1 Lebensabschnitte

Es gibt zeitdiskrete Änderungen der Zustandsattribute, wobei die modell-zeitliche Dauer durch Scheduling Operationen bestimmt ist.

Und es zeitkontinuierliche Änderungen spezieller Zustandsattribute (state-Vektoren), wobei die modellzeitliche-Dauer ihrer Lebensabschnitte einzig durch die `integrate`-Realisierungen bestimmt sind.

8.2.2 initialer Lebensabschnittstyp

Kontinuierliche Prozesse werden immer in einem zeitdiskreten Lebensabschnitt gestartet, bei Festlegung von Anfangswerten und Verfahrensparametern.

8.2.3 Synchronisation

kontinuierliche Prozesse können sich in zeitdiskreten an den Objekten res, bin, waitq, condq mit andern Prozessen synchronisieren. Wenn ihre Partner kontinuous sind, dann sind sie auch gerade im diskreten Lebensabschnitt.

8.2.4 Unterbrechung

Zeitkontinuierliche Abschnitte können per integrate unterbrochen werden.

8.2.5 Dynamische Strukturänderung

Während integrate können andere Prozesse im diskreten Lebensabschnitt die Integrate Parameter ändern.

8.2.6 Kopplung

Prozesse können unterbrechbar permanent miteinander gekoppelt sein.

8.3 Prinzipien der Synchronisation

Bei Gleichzeitigkeit werden diskrete Prozesse gegenüber kontinuierlichen Prozessen priorisiert ausgeführt. (Defaults: diskret = 0, kont. ¡ 0)

Die Schrittweite von kont. Prozessen passt sich den Ereigniszeitpunkten von den diskreten Prozessen an, von denen sie abhängig sind. (`addpeer()`, `delpeer()`) Kein zeitliches Überholen möglich.

Abhängige zeitkontinuierliche Prozesse können nicht perfekt synchronisiert werden. Lösung: ein Gesamtprozess mit einem State-Vektor.