Vorlesungsreihe Simulation betrieblicher Prozesse

Diskrete Simulationssysteme

Prof. Dr.-Ing. Thomas Wiedemann
email: wiedem@informatik.htw-dresden.de

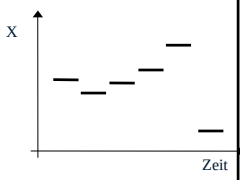
HOCHSCHULE FÜR TECHNIK UND WIRTSCHAFT DRESDEN (FH)
Fachbereich Informatik/Mathematik

Diskrete Simulationssysteme

- Software für die kontinuierliche Simulation
- Historische Entwicklung
- GPSS als bekanntester Vertreter älterer Systeme
- SLX als moderne Simulationssprache

Anforderungen an diskrete Simulationssysteme

- Simulationssystem muß algorithmische Beschreibung realer oder geplanter Prozesse mit diskreten Zeitverhalten (ereignisorientiert) erlauben



Schwerpunkte

- Abbildung von parallelen Aktionen und Wechselwirkungen einer großen Menge von Objekten
  - schnelle Umschaltung zwischen den einzelnen Prozessen
  - Effizientes Speichermanagement zur Verwaltung großer Objektmengen
- Modellierung von stochastischen Prozessen
  - Bereitstellung von Zufallszahlengeneratoren
  - Unterstützung der statistische Auswertung der Ergebnisse
- Animation und Visualisierung
  - einfache 2D-Animationen mit Statusanzeigen
  - aufwendige 3D-Animationen

Historische Entwicklung diskreter Simulationsysteme

- erste Simulationen diskreter Systeme mit universellen Programmiersprachen
- Basissprache der ersten Simulatoren war meist FORTRAN.
- FORTRAN war für damalige Verhältnisse sehr effizient und sehr gut für numerische Berechnungen in Wissenschaft und Technik geeignet
- Sehr gute Unterstützung der Zufallszahlengenerierung durch umfangreiche Mathematikpakete (Nachnutzung von Bibliotheken der kontinuierlichen Simulation)

Genereller Aufwand zur Implementierung war dabei jedoch noch höher als bei der kontinuierlichen Simulation

- keine leistungsfähige Funktionen zur Verwaltung dynamischer Objekte,
- Stark begrenzte Massenspeicher
- Geringe Rechengeschwindigkeiten stark begrenzt
- Hauptproblem Koroutinenfähigkeit - dadurch Standardprogrammiersprachen nicht direkt einsetzbar
- Lösung durch spezielle, mit Assemblerunterprogrammen erweiterte Subroutinen
- Die Subroutinensammlung bildeten sehr rasch die syntaktische und semantische Basis für spezialisierte Simulationssprachen (siehe Folgeabschnitt).

Die Simulationssprache GPSS

- Die Simulationssprache GPSS (General Purpose Simulation System) gehört zu den bekanntesten und verbreitetsten Sprachen für die diskrete Simulation.
- Die Ursachen dafür liegen zum einen bei der Unterstützung durch IBM als Hersteller und zum anderen in der auch heute noch relativ guten Funktionalität, Flexibilität und Leistungsfähigkeit. Obwohl das Grundprinzip der Sprache schon 1961 von IBM entwickelt wurde, werden auch heute sehr viele Modelle im Bereich von Bedien- und Warteschlangensystem mit aktuellen Versionen von GPSS entwickelt und Simulationen durchgeführt.
- GPSS wird als blockorientierte Sprache bezeichnet, da bewegliche Elemente durch ein Netz von einzelnen Blöcken laufen. Mit bestimmten Gruppen von Blöcken können die statischen Objekte des Modelles nachgebildet werden. Die Grundidee der Blöcke stammt dabei von den anfänglichen Blockdiagrammen ab, mit welcher die Strukturen der Modelle dargestellt wurden.
- In heutigen Versionen von GPSS werden bis zu 50 verschiedene Blöcke angeboten. Im Prinzip entspricht jedem Block ein fest programmiertes Unterprogramm. Die Parameter jedes Blockes können damit als Übergabeparameter für das Unterprogramm gesehen werden.
- GPSS hat bis Ende der 80er Jahre die diskrete Simulation entscheidend bestimmt und ist in vielen Büchern als Referenzsprache für Simulationsmodelle angegeben.

Sprachkomponenten von GPSS

- Die beweglichen Elemente in GPSS werden als Aktivatoren (engl. Transactions) bezeichnet. Programmtechnisch wird jeder Aktivator als ein Vektor von Werten dargestellt. Neben nutzer- und modellspezfischen Aktivatorparametern enthält jeder Aktivator auch Informationen zur Programmposition und zum Erzeugungs- und Austrittszeitpunkt.
- Die Aktivatoren bewegen sich durch die folgenden statischen Elemente, welche vor dem Start der Simulation definiert werden müssen:
  - **Generator (Generate) -**Generierungsblock, welcher nach einem vorgegebenen Zeitschema Aktivatoren generiert
  - **Einrichtung (Facility) -** Bedienungs- und Bearbeitungsstation, meist mit der Kapazität 1, wenn eine Einrichtung durch einen Aktivator belegt ist, werden keine weiteren Aktivatoren eingelassen
  - **Speicher (Storage) -** Speicherelement zur Aufnahme einer bestimmten Anzahl von Aktivatoren, gut geeignet zur Nachbildung von Lagerplätzen
  - **Warteschlange Queue) -** Warteschlange mit unendlicher Kapazität
  - **Test (Gate) -** Testblock zur Abprüfung von Bedingungen und zur bedingten Verzweigung oder Aufsplittung von Aktivatoren
  - **Terminator -** Block zur Vernichtung (Auskopplung) von Aktivatoren

Grundbefehle von GPSS

Den einzelnen Komponenten sind jeweils eine Reihe von Blöcken zugeordnet:

- **Generator**

  GENERATE A,B,C,D

  mit A,B-Generierungsabstand der Form A+/-B gleichverteilt, C-erster Zeitpunkt, D-maximale Anzahl, (B,C,D sind optional)

  Bsp.: GENERATE 100,10,50,40

  -> erzeugt ab Zeitpunkt 50 mit Abstand 100+/-10 Zeiteinheiten insgesamt 40 Aktivatoren

- **Einrichtung (in der Regl mit einer Kapazität = 1)**

  SEIZE ***Einrichtungsname*** versucht die Einrichtung zu betreten

  ADVANCE 10,2 verzögert zufällig um 10+/- Zeiteinheiten

  RELEASE ***Einrichtungsname*** verläßt die Einrichtung

- **Speicher**

  INITIAL SK=*Speichername,B* legt Speichergröße auf B Einheiten fest

  ENTER *Speichername* Belegung des Speichers mit einer Einheit

  ADVANCE 10 Verzögerung (Lagerzeit)

  LEAVE *Speichername* Aktivator verläßt den Speicher

Grundbefehle von GPSS II

Warteschlange mit unbegrenzter Kapazität

QUEUE *Warteschlangenname* Betreten einer Warteschlange

DEPART *Warteschlangenname* Verlassen der Warteschlange

Tests auf logische Bedingungen zum Verzweigen von Aktivatoren

TRANSFER A,B,C A -Wahrscheinlichkeit p,B-Verzweigung mit 1-p nach Blocknr. B , C- Sprungziel mit Wkt. p

Bsp.: TRANSFER 0.3,M1,M2 Verzweigung mit 30% Wkt. nach Marke M2 und 70% nach M1

LOGIC *Schaltername* setzt einen Schalter (R -aus S-ein I-invers)

GATE *Schaltername* prüft den Schalter und stoppt bei False

- Bsp.: LOGIC R, FLAG1
  GATE LR FLAG1

Terminator dient zum Freigeben (Vernichten von Aktivatoren)

- TERMINATE A vernichtet den eintreffenden Aktivator
  und verringert optional den Startzähler um A

GPSS-Befehle zur Simulationssteuerung

Block	Einsatz
SIMULATE	Startet die Übersetzung des GPSS-Quellprogramms
START A	Startet die Simulation und setzt den Startzähler auf A, Kombination mit Terminate erfolgt eine definierte Beendigung der Simulation
RESET	setzt alle statistischen Werte zurück, dient zur Unterdrückung der Einschwingvorgänge
REPORT	leitet eine Reihe von Anweisungen zur Druckausgabe ein
END	beendet das Programm

Typische Steuersequenz zur Unterdrückung von Einschwingprozessen

`SIMULATE`	Beginn des Steuerblocks
`START 100`	Startet Simulation mit 100 Aktivatoren (Einschwingen)
`RESET`	setzt die Statistiken zurück
`START 1000`	Startet eigentliche Simulation mit 1000 Aktivatoren
`END`	Ende des gesamten Simulationsprogramms

GPSS-Aktivatoren

GPSS-Aktivatoren können als ein Datenfeld aufgefaßt werden, dessen Attribute zur Modellierung der dynamischen Objekteigenschaften dienen:

Datenfelder :

- Laufende Nummer des Aktivators (zur eindeutigen Referenzierung)
- Erzeugungszeitpunkt (für statistische Zwecke)
- Prioritätsklasse für prioritätsabhängige Verzeigungen
- Nummer des aktuellen belegten Blockes
- Blockabgangszeit (geplanter Ereigniszeitpunkt)
- Freie Parameter für beliebige Anwendungen, unterteilt in verschiedene Datentypen (byte, integer, double), die Anzahl der Parameter kann durch den Anwender festgelegt werden
  - Mit ASSIGN kann der Inhalt modifiziert und durch Testbefehle kann der Inhalt jedes Aktivators getestet und wertabhängig verzweigt werden:

    ASSIGN 2+, 10, PH - addiert 10 zum zweiten Parameter

    TEST GE PH2, 100, AUSSCHUSS // verzeigt bei Par1>100 nach Ausschuss

Ein GPSS-Beispielprogramm

```
START   GENERATE  20        Erzeugung von Aktivatoren aller 20 Zeiteinheitem
        QUEUE     WS1       Eintritt in die Warteschlange vor Maschine 1
        SEIZE     MASCH1    Versuch auch Maschine 1 zu betreten
        DEPART    WS1       bei Erfolg wird Warteschlange verlassen
        ADVANCE   18,5      Verzögerung um 18+/- 5 ZE (Bearbeitung)
        QUEUE     WS2       Eintritt in die Warteschlange vor Maschine 2
        RELEASE   MASCH1    Maschine 1 verlassen
        SEIZE     MASCH2    Versuch Maschine 2 zu betreten
        DEPART    WS2       bei Erfolg wird Warteschlange 2 verlassen
        ADVANCE   19,5      Verzögerung um 19+/- 5 Zeiteinheiten
        RELEASE   MASCH2    Maschine 2 verlassen
        TERMINATE 1         Aktivator vernichten, Startzähler um 1 verringern
```

- Hinweis: Eine Besonderheit stellen die verschachtelten Aus- und Eintritts-anweisungen zur Modellierung von Blockierungen dar.

Quelltextbasierte, diskrete Simulationssysteme

- GPSS war lange Zeit der Quasistandard und hat die Entwicklung anderer diskreter Simulationssysteme stark beeinflusst,
- Ältere GPSS-Simulationssysteme sind mit einigen Nachteilen behaftet :
  - Quelltext muß teilweise noch im FORTRAN-Format an festen Positionen notiert werden
  - Rechenoperationen und Syntax unterscheiden sich stark von üblichen prozeduralen Sprachen (vgl. ASSIGN und TEST – Beispiele bei Aktivatorstruktur)
  - Probleme bei der Integration in moderne IT-Infrastrukturen (keine Datenbankschnittstellen, reine Kommandozeilenversionen
- Einziger großer Vorteil: durch die starke Optimierung der Laufzeitperformance sehr schnell im Vergleich zu heutigen Simulationssystemen (Faktor 10 bis 100)
- Dieser Performancevorteil wurde von einem der maßgeblichen Entwickler von GPSS, Jim Hendriksen in einer neuer Simulationssprache SLX auf eine zeitgemäßes Softwareniveau gebracht
- SLX kann auch ältere GPSS-Programme ausführen, ist aber sonst eine eigenständige und relativ moderne Programmiersprache
- Durch den zeitgemäßen Aufbau und das weite Spektrum an Funktionen, auch bis hin zur Ausführung von GPSS-Programmen, wird SLX in den Übungen verwendet.

Das Simulationssystem SLX

SLX im Überblick

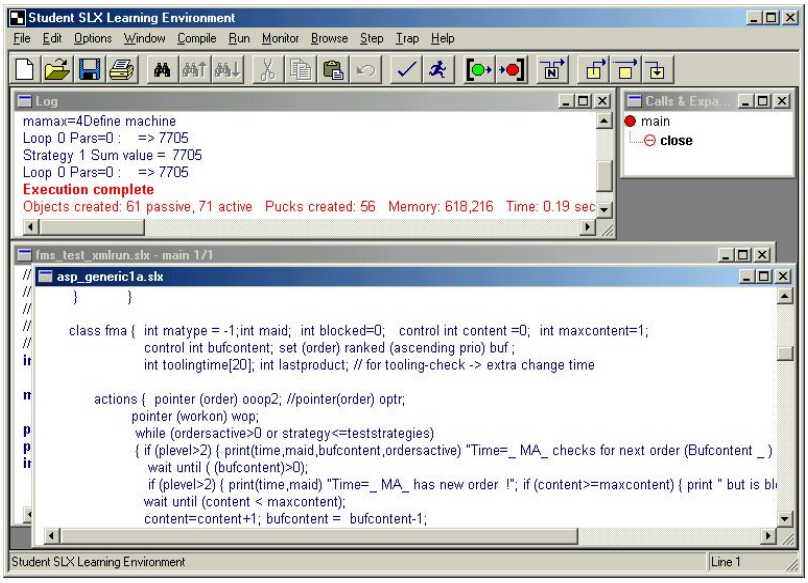
- SLX steht für **S**imulation **L**anguage with e**x**tensibilities
- Hersteller ist die Wolverine Software Corporation USA
- von der gleichen Firma stammen auch das leistungsfähigste GPSS-System GPSS/H und das Animationssystem PROOF
- alle Grundkonzepte der sehr effizienten Ereigniskalenderverwaltung von GPSS/H wurden übernommen
- weiterhin beruht SLX auch auf Konzepten aus SIMULA, C und C++
- SLX ist objektbasiert, aber nicht vollständig objektorientiert

Grundaufbau der Sprache

- alle nicht simulationsbezogenen Funktionen entsprechen C !
- Simulationsbezogene Funktionen verwenden sogenannte Pucks mit frei definierbarer Datenstruktur als dynamische Objekte
- größere Quelltexte können in Module gegliedert werden

Die Bedienoberfläche von SLX

- Dreigeteilt in Quelltext, Ausgabefenster und Debuggfenster



Nicht simulationsbezogene Funktionen von SLX

```
// Example
module basic
int i;

{ // Main proc
  procedure main()
  { int j;

    ...
    print "Hello\n";
  }
}
```

- Falls nachfolgend nicht anders gezeigt, gelten im Zweifelsfall immer die Regeln von C++
- Kommentare mit /* ... */ oder //
- Namen von Funktionen und Variablen beginnen mit Buchstaben und haben sonst eine beliebige Menge von Buchstaben, Ziffern und Unterstriche, alle Zeichen sind signifikant
- Globale Variabeln außerhalb von Prozeduren
- Lokale innerhalb von Prozeduren
- Anweisungen sind generell mit ; abzuschließen
- eine produzedur main() muss generell existieren, diese wird beim Start aufgerufen

SLX-Datentypen und Operatoren

		Anfangsinitialisierung
• int	- ganzzahlig 32 bit	0
• Float, double	- Gleitkomma (identische Speicherung mit 8 byte)	0.0
• boolean	- logische Zustände	FALSE
• string(anzahl)	- Zeichenketten	""

- pointer - Zeiger, zur Vermeidung von häufigen Pointerfehlern können SLX-Pointer im Gegensatz zu C nur auf Objekte verweisen
- Enumeration: typedef name **enum (wert1, wert2, wert3 .... Wertn)**
- Felder: datentyp name[dim1][dim2]...[dimN] Index läuft von 1 bis N !!!

Arithmetische Operatoren (analog zu C)

+ , - , * , \ , % (Modulodivisionsrest), auch Kombination mit Zuweisung wie +=

Relationale Vergleichsoperatoren und logische Operatoren

< , > <= , >= , == , != ! NOT && ||

Inkrement und Dekrement wie bei C a++ --c

SLX-Steuerkonstrukte

Die SLX-Steuerkonstrukte entsprechen denen von C.

- **If-then-else**

  If (boolescher Ausdruck) Einzelanweisung oder Block

  else Einzelanweisung oder Block ;

- **Fallunterscheidung:**

```
switch (Ganzahliger_Ausdruck)
  case Ausdruck : { Anweisung oder Block  break; }
```

- **FOR-Schleife**

  for (Anfangszuweisung; Laufbedingung; Laufzuweisung) Anweisung oder Block ;

  Bsp.: for (i=1; i< 20 ; i++) show(i);

- **WHILE –Schleife**

  While (boolescher Ausdruck) Einzelanweisung oder Block ;

- **DO WHILE –Schleife**

  do Einzelanweisung oder Block while (boolescher Ausdruck) ;

- **Endlos –Schleife**

  forever Einzelanweisung oder Block ;

SLX-Klassen

- Klassen und Objekte in SLX im Sinne der objektorientierten Programmierung
- allerdings keine Polymorphie (Laufzeitprobleme) und Vererbung

Definition:

```
[prefix] class Klassenname  [(parameter_für_Initialisierung ]
{     Attributdefinitionen;  //  datentyp name
      Methodendefinitionen;  //
}
```

- In SLX sind nur vordefinierte Methoden innerhalb von Klassen zulässig. Weitere Methoden sind als normale Prozeduren zu schreiben.

Vordefinierte Methoden mit vorbestimmten Aufrufstatus :

- initial - zur Initialisierung
- **actions - Anweisungen zur eigentlichen Simulationszeit (wichtigster Bereich)**
- clear - Löschen (z.B. zum Simulationsneustart)
- final - beim Beenden der Simulation
- report - zur Ausgabe von Informationen

SLX-Objekte

Bei vorhandener Klassendefinition können Objekte wie nachfolgend definiert werden:

- Statische Definition — Beispiel für Klasse class customer () { ...})

  klassenname objektname; **class person1;**

- Definition eines Pointers und dynamische Generierung mittels new

  pointer (customer) pcustomer;
  pcustomer = new customer( );

  Beispiel für den Aufruf der Objektmethoden

  report (* pcustomer); // ruft report-Methode auf
  clear (* pcustomer); // ruft clear -Methode auf
  destroy(* pcustomer); // zerstört und ruft final-Methode auf

Verwaltung von SLX-Objekten in SET´s

Im Sprachumfang von SLX existieren leistungsfähige Listenfunktionen (Sets) :

Definition eines Sets

```
set ( klassenname ) setname;  // homogenes Set einer Klasse
set ( * ) setname; // Set für beliebige Objekte
```

Zusätzlich Angabe eines Sortierkriteriums möglich:

a.) nach Reihenfolge

```
set ( klassenname ) ranked { FIFO | LIFO } setname;
```

b.) nach einem Schlüsselattribut

```
set ( klassenname ) ranked ( { ascending | descending } attr) setname;
```

Bsp.: set (customer) ranked FIFO waiting_line;

Arbeit mit Sets:

```
place pcustomer into waiting_line;  // Einordnen in ein Set
place pcustomer into waiting_line before pcust10;  // explizites Einordnen
remove pcustomer from waiting_line  // Entfernen aus Set (Objekt bleibt)
pcustomer = position (10) in waiting_line; // liefert Ref. auf 10. Element
for (pcustomer= each customer in waiting_line) {  ... }  // Iteration über Set
pcustomer = first customer in waiting_line; // Referenz auf erstes Objekt ... dto. Last
pcustomer = successor( pcustomer ) in waiting_line;  // nächstes Objekt
pcustomer = predecessor( pcustomer ) in waiting_line;  // vorheriges Objekt
```

Ein- und Ausgabe in SLX

Ausgabe auf das LOG-Fenster

- print "Demo von Textausgaben";
- print "text _ text " varname; // Wert wird an der Stell des _ eingefügt, es entscheidet allein die Reihenfolge der Variablen (kein %i oder ähnliches)
- Die Ausgabe in das Logfenster kann per Optionseinstellungen auch in einen Listingfile umgelenkt werden.

Eingabe von Daten:

- Über Tastatur (mit Dateihandler über "stdin")

```
Filedef keyboard name="stdin"
Read file = keyboard;
 prompt = "Please enter a number ";
 end = endinput; // muß weiter unten als Sprungmarke definiert sein
err = error_handler; // muß weiter unten als Sprungmarke definiert sein
I;  // liest diese Variable i ein
```

- Die Ausgabe in beliebige Dateien erfolgt analog zur Tastatureingabe mit der Anweisung write file = ...

Das SLX-Simulationskonzept

- SLX realisiert ein ereignisorientiertes, prozeßbasiertes Konzept,
- Modell besteht aus einer Menge sich gegenseitig beeinflussender Prozesse
- Simulationsuhr ist eine FLOAT-Größe und stellt sich auf den jeweils nächsten Ereigniszeitpunkt ein
- aktueller Wert der Modellzeit kann mit time abgefragt werden
- Bei den meisten diskreten Simulatoren kann ein dynamisches Objekt immer genau einmal als sich im Modell bewegende Einheit verwendet werden.
- Kann bei sich beeinflussenden, unterschiedlichen Typen von Faktoren problematisch sein (z.B. gleichzeitige Abhängigkeit von informatorischen und materiellen Größen)
- SLX erlaubt daher eine 1:N –Relation zwischen dem eigentlichen Modellobjekt und den Referenzen im Modell
- Die Referenzen werden als **PUCK** bezeichnet (wie im Eishockey)

Attribute der SLX-Pucks

Die Pucks sind selbst SLX-Objekte und verfügen über folgende Attribute:

- Name - Bezeichner des Pucks
  - **wird gebildet aus dem Namen der generierenden Prozedur oder Klasse**
  - **Instanznr. / lfd. Nummer innerhalb einer Instanz**
  - **Bsp.: main 1/1 ist der erste Puck, welcher automatisch von main() erzeugt wird**
  - **Die Bezeichner sind hilfreich bei der Unterscheidung von Pucks während des Debuggens.**
- poised - Verweis auf ausführbare SLX-Anweisung
- priority - Priorität (durch Anwender modifizierbar und wirkt sich auf Reihenfolge der Abarbeitung bei gleicher Zeit aus))
- mark_ time - Erzeugungszeitpunkt
- moving_time - nächster Bewegungszeitpunkt (falls bekannt)
- puck_state - Status des Pucks – eine der nachfolgenden Zustände:
  - **MOVING - ist aktiv und wartet auf Abarbeitung**
  - **SCHEDULED – ist um eine definierte Zeit verzögert**
  - **WAITING – wartet auf den Eintritt einer Bedingung**
  - **INTERRUPTED – unterbrochen durch spezielle Anweisung**
  - **TERMINATED – beendet, aber infolge existierende Referenzen noch nicht freigegeben**

Simulationsspezifische Anweisungen

- Definierte Verzögerungen

  advance ausdruck; // Ausdruck muß eine Zeit >0 ergeben (float)
- Unbestimmte Verzögerungen

  wait until (Bedingung) ; // wartet solange, bis Bedingung wahr ist

  Innerhalb von Bedingung muß mindestens eine Variable existieren, welche den SLX-Präfix control hat ! Bei einer Änderung dieser Variable wird die wait until – Anweisung automatisch neu berechnet.

  control weist den SLX-Compiler an, spezielle Verwaltungsstrukturen für diese Variable zwecks effizienter Auswertungen zu erzeugen!

  Bsp.; control int inhalt;

  ```
  wait until ( inhalt ==0); // wartet auf freien "Platz"
  inhalt++; advance 10; inhalt--;
  ```
- Totale Blockierung

  wait ; // Prozeß wird vollständig deaktiviert

  reactivate pcustomer; // Reaktivierung muß durch anderes Objekt erfolgen

Realisierung einfacher Modelle

Bei sehr einfachen Modellen kann direkt mit dem ersten Pucks und davon abgeleiteten "Kindern" gearbeitet werden:

```
procedure main()
{  print "Start Simulation";
   while (time < 10000)      // Der erste Puck dient zur Generierung der Modellobjekte
        fork  {  // erzeugt eigenständigen Kindsprozess
                 advance 50;            // zeitverbrauchende Aktion
                 wait until ( inhalt <1 ) ;
                  ...   // führe weitere Aktionen aus (auch erneutes Fork)
                terminate;  // vernichte Kindspuck
               }
       advance 100;  // wartet auf neuen Generierungszeitpunkt
}
```

Vorteil: relativ einfaches Programm

Nachteil: keine eigenen Objektattribute möglich

Modellierung mit Objekten

- Bei komplexeren Modellen reicht die einfache Aufspaltung des ersten Pucks nicht mehr aus. Eine klassenbasierte Modellierung ist dann notwendig.
- Voraussetzungen: Objektklasse mit der Methode actions

```
class customer  (int cusid ) {  int customerid;
                    initial   {  customerid= cusid;  }
                    actions {  print "Start customer _ " customerid ;
                               advance 100;
                              print "Stop customer _ "  customerid;
                               }
                  }
```

- Erzeugung des Objektes und Aufruf von actions mit activate **:**

```
activate   new customer();
```

oder mit extra Pointer für späteren Zugriff

```
pointer (customer) pcus1;
pcus1 = new customer();
activate pcus1;
```