SLX 2.0

Objektorientierte Modellierung, Spezifikation und Implementierung II

Überblick

- Simulation Language with Extensibility 2.0
- Sprachumfang entspr. Teilmenge von C++ & ODEMx

```
pointer(X) x = new X();
x->i = 2;
```

- verzichtet auf komplizierte Sprachkonstrukte (z.B. Zeigerarithmetik, Zeiger auf Zeiger, Vielzahl an Containertypen aus STL)

```
set(X) xs;
place x into xs;
```

- fügt neue Schlüsselwörter für Simulationskonstrukte hinzu (z.B. advance, wait until, forever)

```
class X {
    control int i;
    actions {
        wait until i == 2;
        advance 10;
    }
}
```

Überblick

- Intra-Prozess-Parallelität beschreibbar (vgl. mit mehreren Steuerflüssen innerhalb einer ODEMx-Process-main)

```
actions {
      fork
      { }
      parent
      { }
}
```

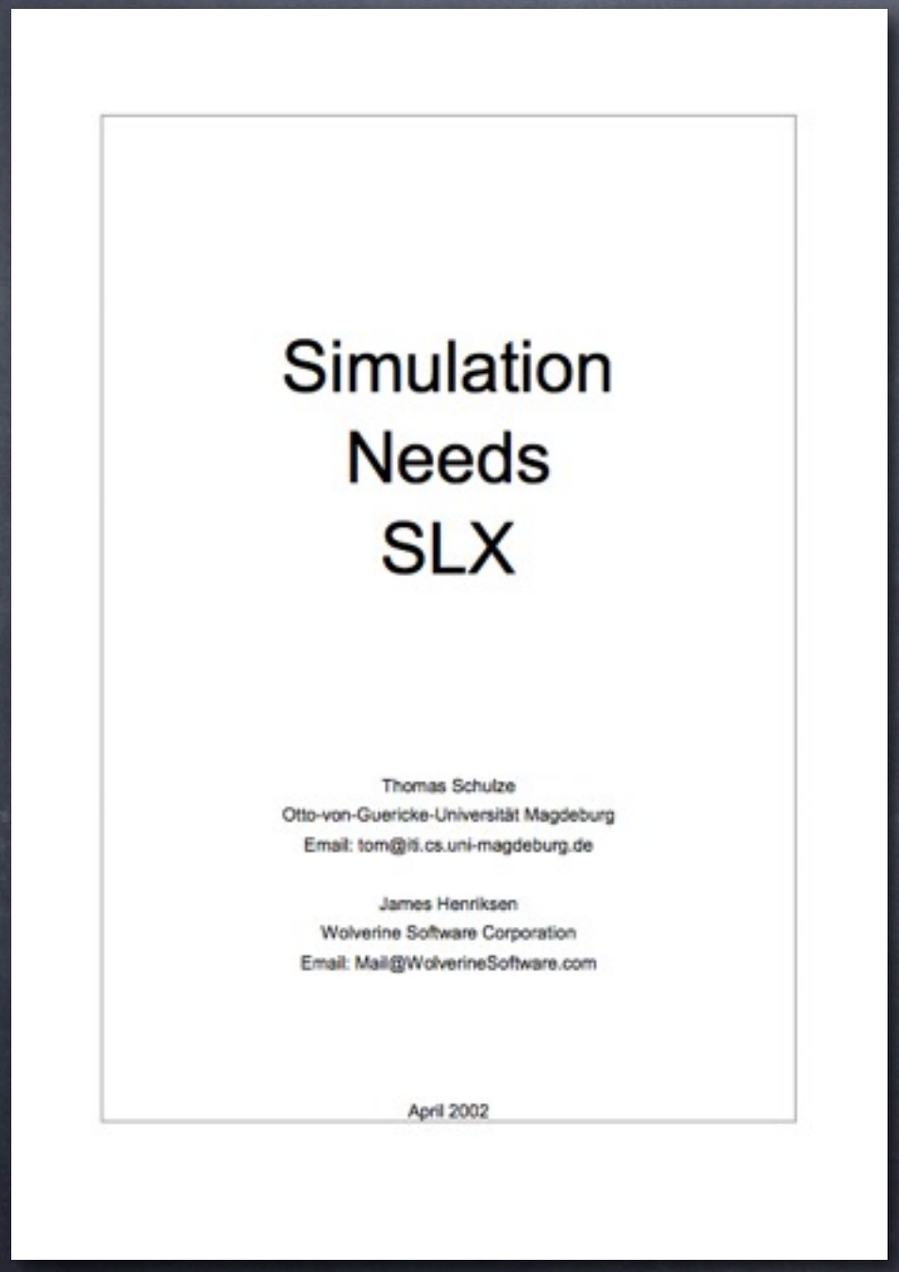
- Sprache erweiterbar um neue anwendungsspez. Konstrukte (ähnlich zu C-Makros, aber mit mehr Möglichkeiten)
  - Bsp.: GPSS-Konstrukte, die auf SLX-Kernkonstr. abgebildet werden

```
statement aw n=#p1
definition { }

procedure main() {
      aw n=2;
}
```

- Animationen sind mit PROOF beschreibbar

Überblick



Simulation
Needs
SLX

Thomas Schulze
Otto-von-Guericke-Universität Magdeburg
Email: tom@iti.cs.uni-magdeburg.de

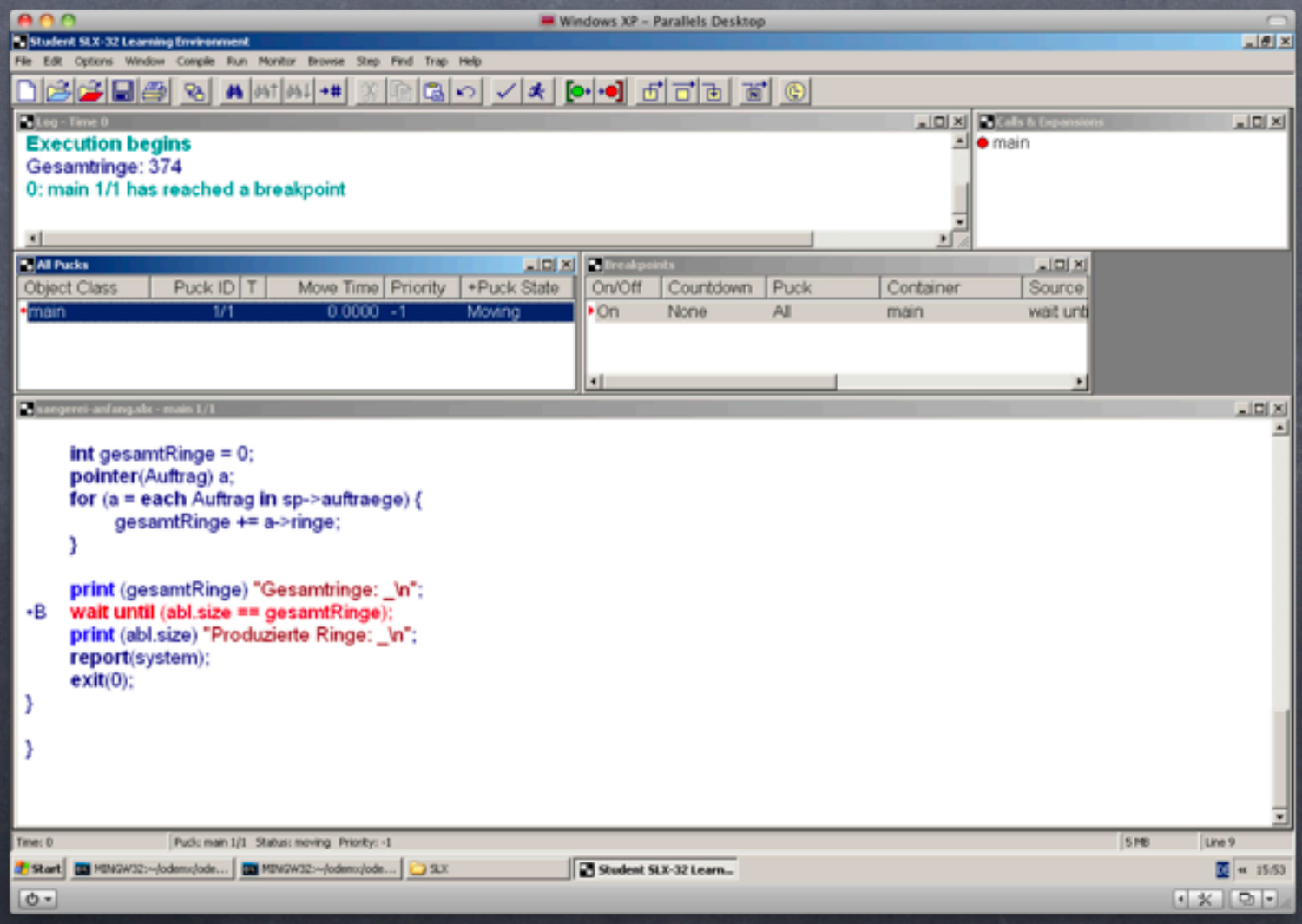
James Henriksen
Wolverine Software Corporation
Email: Mail@WolverineSoftware.com

April 2002

- Literatur:
  „Simulation Needs SLX“
  (SLX-Handbuch in deutscher Sprache)
  http://isgwww.cs.uni-magdeburg.de/pelo/sa/SimulationNeedsSLX.pdf

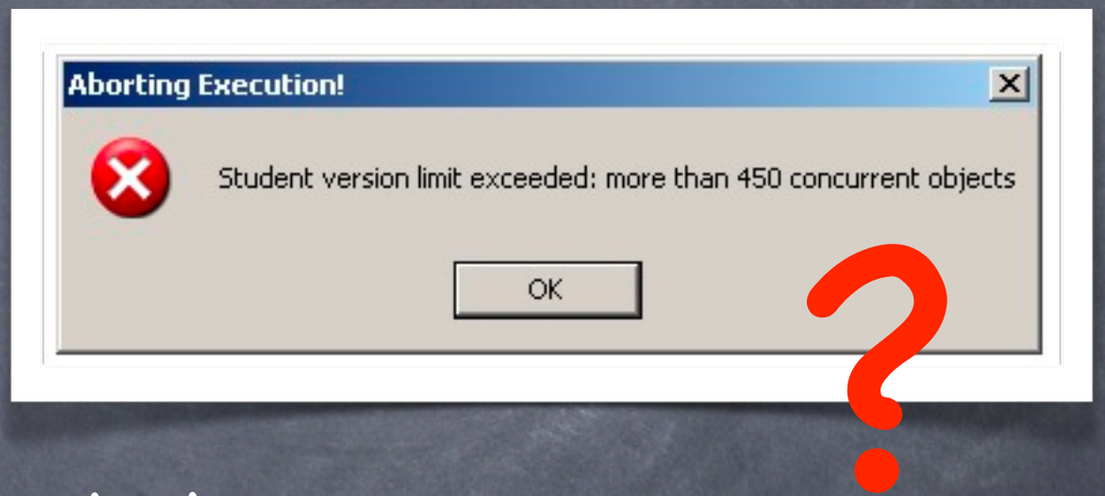
Überblick

- Windows-basierte IDE mit Editor & Debugger
  http://www.wolverinesoftware.com/



Überblick

- Studentenversion ist eingeschränkt



- max. 450 Objekte
  (gleichzeitig aktiv - egal ob
  aktive oder passive Objekte)
- und max. 500 Pucks
  (für jeden Prozess ex. mind. 1 Puck)

Überblick

- Semantik als Abb. nach C
  (Details nicht verfügbar)

Themen

- Typ-System
- Prozedurale Modellierung
  (Kontrollstrukturen, Prozeduren)
- Objektorientierte Modellierung
  (Klasse, Objekt, Vererbung)
- Verhaltensmodellierung
- Spracherweiterungen beschreiben
- Animationsmodellierung

Typ-System

- Vordefinierte Datentypen
- Felder
- Aufzählungstypen
- Zeichenketten
- Klassen
- Objektzeiger
- Mengentypen
- Sim-Typen

Datentypen

```
int i;
int j = 1;
```

- int (32-bit)
- float = double (64-bit)
- boolean (TRUE oder FALSE)
- immer initialisiert
  int: 0, float/double: 0.0, boolean: FALSE
- keine Zeiger erlaubt
- Operatoren aus C übernommen
  + - * / % < <= > >= == != = += -= ++ -- !
  NOT && || << >> ~ & ^ |

Felder

```
int d = 2;
int f[d];
f[2] = 3;
```

- n-dimensional
- Definition:
  <Typ> <Feldname>[d1][d2]...[dn]
- Typ muss Datentyp oder Objektzeiger sein
- indizierung von 1 bis n
  (nicht wie in C von 0 bis n-1)

Felder

```
int d = 2;
int f[d];
f[2] = 3;
```

- Dimension muss in Def. angegeben werden (Variablen sind zulässig)
- Dimension darf nur bei Verwendung als Prozedur-Parameter entfallen

```
procedure p(int arg[*]) {}
```

- Angabe einer initialen Belegung für jede Dimension möglich (sonst Vorbelegung mit Standardwert)

```
int g[2][2] = { {1,2}, {3} };
```

Aufzählungen

```
// C
#include <stdio.h>
enum Color {
    red, green, blue
};
int main() {
    Color c = green;
    int i = c;
    printf("%d",i);
}
```

```
procedure main() {
    Color c = blue;
    c = first Color; // red
    c = predecessor(c); // NONE
    int i = c;
•• Semantic error: an int is required
here; "c" is a Color
}
```

- Definition:
  type <Name> enum {
  <Wert1>, <Werte2>, …
  }
- Standardwert:
  NONE-Literal
- keine Verwendung von Enum-Wert als Int-Wert erlaubt
- Vordef. Funktionen:
  first, last, successor, predecessor
- Werte eines Enum-Typs per for-Konstrukt iterierbar

```
type Color enum {
    red, green, blue
};
```

Zeichenketten

```
string(5) h = "hello";
```

```
string(3) h2 = "hello"; // hel
```

```
string(5) h3 = h2 cat "lo"; // hello
```

```
string(1) a = ascii(65); // a = “A“
```

```
int i = length(h3);
```

```
procedure p(string(*) arg) {}
```

- Definition:
  string(<Länge>) <Name>
- Standard-Wert:
  “” (leere Zeichenkette)
- Vordef. Funktionen:
  cat, substring, length,
  ascii (ascii-nach-string-Konv.),
  str_ivalue (string-nach-int-Konv.)
- keine dynamischen Größenänderungen
  - Länge muss angegeben werden
    (nur Integer-Konstanten erlaubt)
  - Länge darf nur bei Verwendung als Prozedur-Parameter entfallen

Klassen

```
class X {
    int i;

    procedure p() { }

    actions { }
}
```

- Definition:
  [<Prefix>] class <Name>
  [(<Constructor_Parameter>)] {
    <Attribute>;
    …
    <Procedure>;
    …
    <Property>;
    …
  }
- Klasse definiert Attribute und Prozeduren und kann vordef. Properties enthalten (actions, initial, final, clear, report)

initial (Konstruktor)

```
class X (int j) {
    int i;

    initial {
        i = j;
    }
}

class Y (int j) {
    int i = j;
}

class Z (int i) {
    int i;
•• Semantic error: "i" has been
previously defined in the current
scope
}
```

- Parameterangaben hinter dem Klassennamen
- Konstruktion in initial-Property
- höchstens ein Konstruktor

Klassen

- Sichtbarkeit von Attributen und Prozeduren:
  - public (Standardfall)
  - oder private
- Anders als in anderen Sprachen
  - z.B. C++
    - class: private
    - struct/union: public

```
class X {
    int i; // public
    private int j;

    procedure p() { }
    private procedure p() { }
}
```

Objekte

```
class X (int j) {
    int i;
    Y y;

    initial {
        i = j;
    }
}

class Y { }

X hx(1);

procedure main() {
    X lx(1);
}
```

- existieren (wie in C) entweder
  - lokal auf dem Stack
    - automatische Vernichtung bei Verlassen des Gültigkeitsbereiches
  - oder global auf dem Heap
- als
  - globale oder lokale Variable
  - oder Objektattribute (enthalten-in)

Objektzeiger

```
class X (int j) {
    int i;

    initial {
        i = j;
    }
}

X hx(1);

procedure main() {
    pointer(X) px = &hx;
    px = new X(2);
}
```

- Definition für best. Objekte:
  pointer(<Klasse>) <Name>
- Definition für bel. Objekte:
  pointer(*) <Name>
- Typ muss eine Klasse sein
  (insbes. keine Zeiger auf Zeiger und keine Zeiger auf Werte von Datentypen)
- Wert entspr. Objektreferenz
- Standardwert: NULL-Literal

Objektvernichtung

- automatisch
  - bei Verlassen des Sichtbarkeitsbereiches
  - über Referenzzählung
- manuell
  - destroy-Funktion

```
class Y {
        int i;
        final {
                print "Y destructor\n";
        }
}

class X (int j) {
        int i = j;
        pointer(Y) y;
        initial {
                y = new Y();
        }
        final {
                print(i) "X(_) destructor\n";
        }
}
```

```
pointer(X) x1 = new X(1);
pointer(X) x2 = new X(2);

print "before x1 = x2\n";
x1 = x2;
print "after x1 = x2\n";

destroy x1;
```

```
Execution begins
before x1 = x2
X(1         ) destructor
Y destructor
after x1 = x2
X(2         ) destructor
Y destructor
Execution complete
```

Objektzeiger

- Zeiger-Operatoren (übernommen aus C):
  - Zeiger auf das Objekt x
    `&x`
  - Objekt am Zeiger y
    `*y`
  - Wert des Attributes i, wobei x ein Objekt ist
    `x.i`
  - Wert des Attributes i, wobei y ein Zeiger ist
    `y->i`

Objektzeiger

```
class X (int j) {
    int i;

    initial {
        i = j;
    }
}

class Y {
    int m;
}

procedure main() {
    pointer(*) u = new X(1);
    int k = u->m;
•• Execution error at time 0:
"u" points to an object of class X,
which has no "m"
}
```

- Zugriff auf nicht vorhandene Attribute über Universal-Zeiger?
- Kompilierfehler falls keine Klasse das Attribut definiert
- Laufzeitfehler sonst

Objektzeiger

- Dynamische Typ-Bestimmung:
  - a) Typ eines Ausdrucks:
    type(<Ausdruck>)
  - b) Typ einer Klasse:
    type <Klasse>
- Ergebnis:
  Typ-Objekt

```
pointer(*) u = new X(1);
if (type(*u) == type X) {
     // u is of type X
}
```

```
if (type (TRUE) == type boolean)
     print "bool";

if (type (1+3) == type int)
     print "int";
```

Eigenartiger SLX-Bug

```
passive class A {
    int i = 2;
}
```

```
advance 2;
pointer(A) a;
a = new A();
int i = a->i;
```

✘

•• Execution error at time 0: NULL pointer reference

```
pointer(A) a;
a = new A();
int i;
i = a->i;
```

✔

```
pointer(A) a = new A();
int i = a->i;
```

✔

Mengentyp(en)

```
set(X) xs;
set(X) ranked LIFO xsl;

class X {
    int a;
    int b;
}

set(X) ranked (ascending a,
    descending b) s;
```

- einziger Mengentyp:
  geordnete Menge von Objektzeigern
- Definition (homogenes Set):
  set(<Klasse>) <Name>
- Definition (universelles Set):
  set(*) <Name>
- Sortierung einstellbar:
  - FIFO (Standard),
  - LIFO,
  - oder aufsteigend/absteigend
    nach Attributen (nur homogene Sets)

- set(...) ranked FIFO <Name>
- set(...) ranked LIFO <Name>
- set(...) ranked (ascending
  a1, ...) <Name>
- set(...) ranked (descending
  d1, ...) <Name>

set-Operationen

```
set(X) xs;
set(X) ranked LIFO xsl;

pointer(X) x1 = new X();

place x1 into xs;
int i = position(x1) in xs;
remove x1 from xs;

empty xs;
empty set xs;
```

- Einfügen
  place <Pointer> into <Set>
- Einfügen an Position
  place <Pointer> into <Set> after <Pointer>
- Entfernen
  remove <Pointer> from <Set>
- Leeren
  empty [set] <Set>
- Position ermitteln
  int <Position> = position(<Pointer>) in <Set>

set-Operationen

```
set(X) xs;
set(X) ranked LIFO xsl;

pointer(X) x1 = new X();
place x1 into xs;

pointer(X) x;
for (x = each X in xs) { }
for (x = each object in xs) { }
```

- Iterieren
  - nur Objekte einer best. Klasse:
    for (<Pointer> = each <Klasse> in <Set>) {...}
  - alle Objekte in der Menge:
    for (<Pointer> = each object in <Set>) {...}

set-Operationen

```
class X (int j) {
    int i;

    initial {
        i = j;
    }
}

class Y {
    int m = 2;
}
```

```
set(*) s;
pointer(*) p;

p = new X(1);
place p into s;
p = new Y();
place p into s;

pointer(Y) fy;
fy = first Y in s;
```

- Iterieren
  - erstes/letztes Objekt
    - einer best. Klasse
      first <Klasse> in <Set>
      last <Klasse> in <Set>
    - einer beliebigen Klasse
      first object in <Set>
      last object in <Set>
  - Nachfolger/Vorgänger eines Objektes
    - beliebiger Klasse
      successor(<Pointer>) in <Set>
      predecessor(<Pointer>) in <Set>
    - bestimmter Klasse
      <Klasse> successor(<Pointer>) in <Set>
      <Klasse> predecessor(<Pointer>) in <Set>

set-Operationen

- Enthalten/Nicht enthalten
  <Set> contains <Pointer>
  <Pointer> is_in <Set>
  <Pointer> is_not_in <Set>

- Anzahl Objekte
  <Set>.size

Sim-Typen

- Zufallszahlengeneratoren (weitere später)
  - Definition:
    rn_stream <Name>
  - Bereitstellung stetig gleichverteilter Zufallszahlen im Intervall [0,1]
  - globale Funktion zur Erzeugung eines neuen Zufallswertes:
    frn(<rn_stream_Variable>)
  - Verbreitete Verteilungsfunktionen als Transformationsfunktionen bereits vordefiniert
    - Bsp. Exponentialverteiltung
      rv_expo(<rn_stream_Variable, <Erwartungswert>)

Prozeduren

- Programmeintrittspunkt
  - procedure main () { }
  - procedure main (int argc, string(*) argv) { }
- Programmflussbeschreibung
  - Verzweigung
    if-then-else, switch-case
  - Wiederholung
    for, while, do-while,
    continue, break
  - Zusätzlich
    forever (= while(true)), goto

```
procedure main() {
Lbl:    if (TRUE) { } else { }
        while (TRUE) { }
        forever {
            break;
        }
        goto Lbl;
}
```

Prozeduren

- existieren als globale Prozeduren und als Objektprozeduren (auch Methoden)
- Prozedur-Definition:
  procedure <Name> (<Parameter_Def>, ...) [returning <Typ>] {
    ...
  }
- Parameter-Definition:
  [<Richtung>] <Typ> <Name>
- Richtung:
  in (Standard), out, inout

Parameter

```
procedure p(in int i) {
     i = 2;
•• Semantic error: "i" is an IN argument;
it cannot be modified
}
```

- Richtung: in
  - nur Lesen ist erlaubt

Parameter

```
class X (int j) {
    int i;

    initial {
        i = j;
    }
}
```

- Richtung: inout
  - Lesen & Schreiben ist erlaubt
- Parameter werden per Referenz übergeben!

```
procedure p(inout int i, inout X x) {
    i = 2;
    x.i = 2;
}

procedure main() {
    int i = 1;
    pointer(X) x = new X(1);
    p(i, *x);
    // i == 2 und x.i == 2
}
```

Parameter

```
class X (int j) {
    int i;

    initial {
        i = j;
    }
}
```

- Richtung: out
  - nach Manual ist nur Schreiben erlaubt, aber auch Lesen ist möglich
- Unterschied zu inout unklar
- Parameter werden per Referenz übergeben!

```
procedure p(out int i) {
    // i == 1
    int j = i; // lesender Zugriff
    i = 2;
}

procedure main() {
    int i = 1;
    p(i);
    // i == 2
}
```

Prozeduren

- Prozedur-Definition
- a) global
- b) als Objektmethode

Unterschiede zu C

- Variablen sind immer initialisiert
- nur Zeiger auf Objekte sind erlaubt
- keine Funktions-Prototypen erforderlich
- keine eingebetteten Zuweisungen erlaubt
  if ((a = b) == c)
- kein Komma-Operator vorhanden
  a = b, ++c;
- kein „x ? y : z“

Objektorientierung

- Version 1.0 >> objektbasiert
  - keine Vererbung
  - nur Objekt-Komposition möglich
- Version 2.0 >> objektorientiert
  - Einfachvererbung zwischen Klassen, Mehrfachvererbung zwischen Interfaces (ähnlich wie in Java)

>> siehe Teil 2

Verhaltensmodellierung

- aktive & passive Objekte
- Prozesslebenslauf
- Puck
- wait until
- wait & reactivate

Aktive & passive Objekte

```
passive class Y {
    int i;
}

class X {
    Y y;
    actions { }
}

procedure main() {
    X x; // Erzeugung
    activate &x; // Aktivierung
}
```

- Objekte von Klassen (class) sind aktiv
  - besitzen Lebenslaufbeschreibung (actions-Property)
  - actions z.B. Wertzuweisung, Warten auf Zustands- oder Zeitereignis
  - Verhaltenausführung beginnt erst mit Aktivierung (activate <Pointer>)
  - ME als Zeiger auf das aktuelle Objekt
- Def. passiver Objekte mit Schlüsselwort passive (kein actions-Property)

Aktive & passive Objekte

```
class X {
    actions {
        pointer(puck) p;
        p = ACTIVE;
    }
}
```

- spezieller Typ für die Verwaltung der Ausführungspositionen im Lebenslauf eines aktiven Objektes: puck
- Laufzeitkonstrukt - ähnlich GPSS-Transaktion oder Instruction Pointer
- z.B. bei Warten auf Zustands-ereignis:
  aktuelle Position (puck) in Warteliste speichern, spätere Reaktivierung

Lebenslaufbeschreibung

- GPSS: viele versch. Blocktypen
- SLX: Reduktion auf Basis-Simulations-Primitive >>

new, activate, terminate	Erzeugung, Aktivierung und Vernichtung von Objekten
advance	Modellierung von Zeitverbrauch
wait until (...)	Warten bis Zustandsbedingung erfüllt
wait	Warten auf Reaktivierung durch andere Prozessinstanz
reactivate	Reaktivierung einer wartenden Prozessinstanz
interrupt, resume, yield, yield to	Unterbrechung einer anderen Prozessinstanz inkl. Fortsetzung

>> Beschreibung von GPSS-Blöcken mit Basis-Primitiven

Beschreibung von GPSS-Blöcken

GPSS:

- FACILITY
- STORAGE
- USER CHAIN
- LOGIC SWITCH
- TEST
- GATE
- ASSEMBLE
- GATHER
- MATCH

Gemeinsames Basisverhalten:

- Warten auf das Vorliegen einer Zustandsbedingung
- Änderung von Zustandsgrößen

<u>Beispiel STORAGE:</u>

- SEIZE
  - Warte bis Facility frei und verfügbar ist
  - Setze frei-Zustand der Facility auf „belegt“
- RELEASE
  - Setze frei-Zustand der Facility auf „frei“

Strukturbeschreibung:

- Passive Klassen

wait until

x2 : X x3 : X

Puck-Listen-Eintrag

```
passive class F {
        control boolean busy;
        control boolean avail = TRUE;
        procedure seize() {
            wait until not busy && avail;
            busy = TRUE;
        }
        procedure release() {
            busy = FALSE;
        }
}
```

x2 : X

x3 : X

Puck-Position

x1 : X

```
F f;

class X {
        actions {
            f.seize();
            advance 5;
            f.release();
        }
}
```

- Puck wartet bis Zustandsbedingung (ZB) erfüllt ist
- ZB erfüllt: nächste Anweisung
- ZB nicht erfüllt:
  - für jede Zustandsvariable (ZV) einer ZB gibt es eine Puck-Liste
  - Puck wird in allen ZV-Puck-Listen der ZB eingeordnet (vgl. Retry-Chains in GPSS)
  - Erneute Überprüfung der ZB sobald sich eine ZV ändert
- ZVs für ZBs müssen explizit mit control deklariert werden

Ende

Teil 1