Vorlesungsreihe Simulation betrieblicher Prozesse

Spezifische Technologien von SLX

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

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

Diskrete Simulation mit SLX

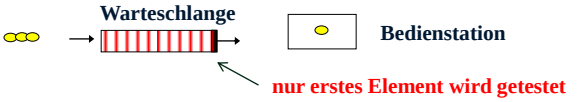
- Aktivitäts- vs. Prozessorientierte Simulation
  - Das Konzept der Control-Variabeln
- Modellzeit und Real-Time-Synchronisation
- Listenverwaltung
- Zufallszahlengenerierung
- Ergebnisanalyse
- Ausführung von GPSS-Modellen mit SLX
  - Das Prinzip der Compilererweiterung
  - Portierung / Ausführung von GPSS-Modellen
- Anbindung von SLX an andere IT-Systeme
  - Schnittstellen, Einbindung von DLL´s
  - Animation

Performancefragen bei abhängigen Ereignissen

- Durch den Systemkalender wird die korrekte Abarbeitung der unabhängigen Ereignisse garantiert. Die Betrachtung und programmtechnisch effiziente Bearbeitung der abhängigen Ereignisse ist dagegen wesentlich aufwendiger.
- Bei der klassischen ereignisorientierten Simulation wird nach der Abarbeitung aller unabhängigen Ereignisse eines Zeitpunktes **eine Prüfung aller abhängigen Ereignisse** durchgeführt. Bei einem Zutreffen der für die Abhängigkeit relevanten Bedingung wird das Ereignis ausgeführt.
- Typische Abhängigkeiten aus der Sicht der realen Systeme sind
  - **Das Warten auf die Auslösung** - damit das Ereignis stattfindet, müssen ein oder mehrere andere Ereignisse eingetreten sein.
  - **Blockieren** - der Zustand einer Modellgröße oder eines Prozesses wird bei Eintreten eines anderen Ereignisses in Entsprechung einer Unterbrechung oder eines Abbruches verändert (z.B. wird eine Maschine blockiert wenn das Ausgangslager voll ist; der Maschinenstatus ist damit zwar „Bearbeitung beendet“ aber die nächste Bearbeitung kann nicht beginnen, da daß Teil noch auf der Maschine ist)
- In der Praxis werden vielfach alle prinzipiell möglichen Ereignisse in der Kalenderliste geführt und es wird $T_i=\infty$ bei allen abhängigen Ereignissen eingetragen. Falls ein abhängiges Ereignis eintreten kann, wird die aktuelle Systemzeit eingesetzt und die zentrale Simulationssteuerung führt die zugehörigen Zustandsänderungen des Modells durch.

Effektivitätsverbesserungen bei abhängigen Ereignissen

- die relativ einfache Methodik der ereignisorientierten Zeitsteuerung wird ineffizient, wenn sehr große Modelle vorliegen und sehr viele abhängige Ereignisse getestet werden müssen (sehr großer Systemkalender) - besonders in großen Warteschlangen müssen die Bedingungen immer wieder (umsonst) getestet werden
- **Lösungsoption**: spezielle Behandlung von Warteschlangen (WS) oder anderen sortierten Listen von aktiven Simulationsobjekten
  - Ansatz: in einer FIFO (oder LIFO) kann nur das 1. oder letzte Objekt effektiv weiterbewegt werden -> damit Prüfung nur dieses Elements (alle anderen Objekte werden quasi bzgl. der Prüfung deaktiviert)



- Als parallele Option bietet sich die Extraktion und Aufbereitung der Bedingungen an. Die Konzentration auf die Abprüfung der Bedingungen für Aktivitäten läßt Raum für weitere Optimierungen innerhalb der Simulationssteuerung:
  - Bei einer Zerlegung komplexer Bedingungen Tests kann bei logischen UND-Verknüpfungen die Unterbedingung mit der geringsten Eintrittswahrscheinlichkeit an den Anfang gestellt werden.

```
If  ( Masch_Inhalt < Masch_Kapazität  &&  KeinAusfall )  ....{  Operation  }
```

häufig falsch (also zuerst) — meist true (also zuletzt)

Effektivitätsverbesserungen mit Callbacks

Weiterer Lösungsansatz :

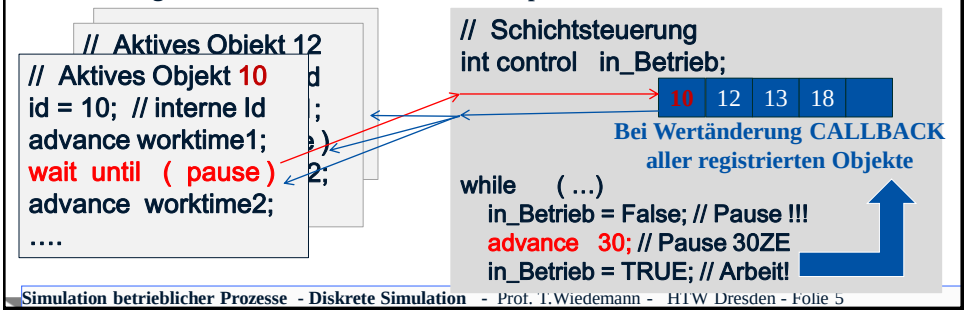
- Aufgrund der ausschließlichen Betrachtung von Ereignissen bei der ereignisorientierten Simulation können auch die **Bedingungen nur im Resultat von Ereignissen erfüllt** werden.

Schlussfolgerung und Lösung:

- Nach einem Negativtest einer Bedingung wird die **Abhängigkeit der Bedingung** in alle der Prüfung zugrunde liegende Zustandsvariablen eingetragen.
- Der Termin der nächsten Überprüfung der Bedingung auf **Unendlich** (also NIE) gesetzt.
- Kommt es bei einer der Entities zu einer Änderung der Zustandsvariablen wird bei dieser Änderung die Kalenderzeit der abhängigen Entities wieder auf die aktuelle Modellzeit gesetzt und die Bedingung damit wieder komplett geprüft.

Technische Lösung (am Beispiel von SLX – andere Systeme ähnlich)

- **Führung einer CALLBACK-Liste an einer speziellen Control-Variable**



Das Konzept der Control-Variabeln

- Definition bei allen Variablentypen mit Präfix control

```
control int anzahl;
```

- Der Compiler legt für Control-Variablen spezielle Verwaltungsstrukuren an, welche eine Relation zwischen einer Bedingung mit der Control-Variable und allen sonstigen Verwendungen der Control-Variable, bei denen eine Werteänderung möglich ist

Davon ausgehend gelten folgende Bedingungen und Sonderregeln:

- Eine wait until ( ) – Anweisung **MUSS** mindestens **EINE** control-Variable enthalten, diese sollte auch die control-Variable mit der häufigsten Änderungsfrequenz sein, es können auch mehrer Control-Var. Auftreten:
  - Bei `wait until ( anzaktiv < plaetze_aktiv) ;` // sollten beide Variable vom Typ control sein, wenn sich auch plaetze_aktiv ändern kann !
- Bei Schreiben des gleichen Wertes in eine Control-Var. tritt KEIN Callback ein !

Performancevergleich der Optimierungsoptionen

Beim Test aller Bedingungen zu jedem Zeitpunkt kommt es aufgrund des Verhältnisses

- Gesamtzahl aller Tests = Anzahl aller Modellzeitpunkte * Anzahl aller Bedingungen

und der etwa linearen Abhängigkeit der Zeitpunkte und Bedingungen von der Elementanzahl zu

Gesamtzahl aller Tests ~ (k_t * Anzahl Modellelemente) * (k_B * Anzahl Modellelemente)

und damit zu einem **quadratischen Wachstum des Rechenaufwandes** in Abhängigkeit von der Anzahl der Modellelemente (k_t und k_b sind modellabhängige Koeffizenten).

Bei der Optimierung der Zeitsteuerung durch die zuvor beschriebene Methode der Inaktivierung von Bedingungstests ändert sich das Verhältnis zu

Gesamtzahl ~ (k_{t2} * Anzahl Modellelemente) * (K_{B2} *Mittlere Zahl ausgelöster Ereignisse)

welches in der Regel zu einem etwa linearen Wachstum des Rechenaufwandes führt, auf jeden Fall aber günstiger als die allgemeine Methode ist.

- Diese relativ allgemeine Schätzung des Rechenaufwandes zeigt aber auch, daß bei der Beurteilung von Simulationssystemen die Frage des Verhaltens bei sehr großen Modellen nicht einfach von kleineren Modellen her extrapoliert werden kann.
- Je nach der (meist unbekannten) intern implementierten Zeitsteuerung kann das Wachstum der Rechenzeiten völlig unterschiedlichen Gesetzen unterliegen. (Zusätzlich können natürlich auch noch Probleme bei Speicherverwaltung und eventuell notwendigen Swapping auf die Platte zu völligen Einbrüchen bei der Performance führen).
- Besonders bei sehr großen Modellen, bei welcher das jeweilige Einzelereignis relativ selten eintritt, kann dies zu einer drastischen Reduzierung der Rechnerzeiten führen. Nachteilig ist natürlich der erhöhte Aufwand zur Verwaltung der Abhängigkeiten.

Prozeßorientierte Simulation

Eine weitere Option der Zeitsteuerung realisiert die prozeßorientierte Simulation :

- statt einen zentralen Zeitsteuerung wird die Zeitsteuerung durch die Modellelemente realisiert. Man sieht das Modell aus der Sicht der Elemente .
- Die Zeitsteuerung hat bei der prozessorientierten Simulation weniger steuernden und treibenden Charakter. Sie ist verantwortlich für die Synchronisation der einzelnen Prozesse. **Jeder Prozeß führt seine eigene Simulationszeit** und meldet den Wunsch nach Weiterschaltung an die zentrale Zeitsteuerung. Diese wertet alle ankommenden Anforderungen aus und gewährt der Anforderung mit dem kleinsten Zeitpunkt die Weiterschaltung. In Analogie zur Bedingungsprüfung werden bei der prozessorientierten Simulation auch überfällige Prozesse wieder aufgerufen.
- Der Aufbau prozeßorientierter Simulatoren erfordert bereits bei der Implementierung der Basisroutinen eine geeignete Programmiersprache, da die Prozeßunterprogramme als nichthierarchische Koroutinen mit beliebigen Ein- und Austrittspunkten codiert werden müssen. Dies ist z.B. mit Fortran oder C nicht möglich. Es entstanden daher speziell auf die Belange prozeßorientierter Simulation zugeschnittene Programmiersprachen wie SIMULA, und auch SLX !
- Der prozessorientierte Ansatz ist deutlich günstiger anwendbar bei der Beschreibung von Modellen mit sehr kompliziertem Verhalten der einzelnen dynamischen Elemente und **für verteilte Simulationssysteme.**

Das Simulationszeit bei SLX

- Die Simulationsuhr bei SLX ist vom Typ Gleitkomma (generell double)

Hintergrund /Diskussion

- Integer (Long) Variable wären effizienter, jedoch können nur einen begrenzten Bereich abdecken
- Gleitkommawerte sind breiter in der Darstellung (+/- 10exp+/-300)

Verwendung der Uhrzeit

- Abruf der aktuellen Modelluhrzeit mit time()
- Verzögerungen können auch direkt mit der Zeit definiert werden:
  - **wait until (time ==** *expr***);** // Warte bis t=expr (wie advance)
  - **wait until (time !=** *expr***) // *Warte auf Veränderung der Zeit !!***
  - **wait until (time >=** *expression***); //Warte auf t>=exp**
  - **wait until (time >** *expression***);** // zuerst Warten, danach nächster Zeitpunkt, welcher ermittelt wird

Synchronisation der Modellzeit mit der realen Welt

- Zweck: Einhaltung eines Zeitmaßstabs statt sprunghafter Wechsel
- besonders sinnvoll bei Animationen oder anderen, möglichst wirklichkeitsgetreuen Darstellungen
- Arbeit mit realtime() = Systemzeit und Sleep() – OS-Warteroutine

Beispielcode: **(nach „SLX-Handbuch. Schulze 2002“)**

```
base_time = real_time(); // Store the current real_time
forever
{ wait until(time != now); // wait for simulator clock change
        real_time_delay = time * 0.1- (real_time() - base_time);
        // real delta T -  simulated time unit  = real 1/10 second
        if (real_time_delay < 0.0)
          print (-real_time_delay * 1000.0)  "_ milliseconds behind!\n";
          else {  delay = real_time_delay * 1000;
                  Sleep(delay);  }// Windows time delay in milliseconds
        now = time;  print (time*0.1) "Elapsed Time [s]: ____._ \n";
}
```

Die Funktionsweise der SLX-Simulationsteuerung

Verwaltung aller dynamischen Elemente (Pucks) in 4 Listen :

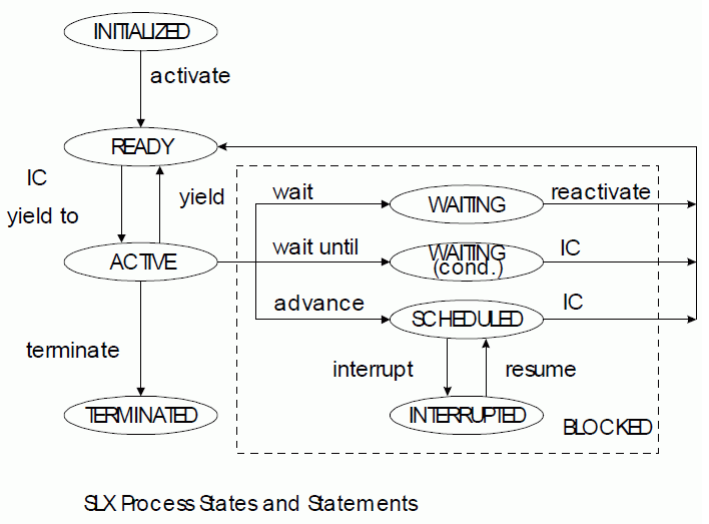
- Liste **Moving Pucks**:
  - enthält alle AKTUELL aktiven/bereiten Pucks, absteigend nach dem Wert des Puckattributs priority sortiert oder bei Gleichheit nach FIFO
- Liste **Scheduled Pucks**:
  - enthält alle Pucks mit BEKANNTER Aktivierungszeit, aufsteigend nach der Zeit, dann priority bzw. FIFO sortiert
- Liste **Waiting Pucks**:
  - enthält alle Pucks mit UNBEKANNTER Aktivierungszeit
- Liste **Interrupted Pucks**:
  - enthält alle Pucks, welche explizit BLOCKIERT sind
  - Sinnvoll zur temporären Auslagerung von Puckmengen (Performance!)

Die Ablage der Pucks auf diese Listen kann teilweise auch explizit (durch entsprechende Befehle) gesteuert werden (siehe Folgeseite)

Die Funktionsweise der SLX-Simulationsteuerung II

Gesamtüberblick und Steuerbefehle zur Listenverwaltung :

(IC steht für Internal Control – also die Simulationssteuerung selbst)



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
```


Zufallszahlengenerierung mit SLX

- SLX stellt die Basisklasse rn_stream zur Generierung von gleichverteilten Zufallszahlen in Entsprechung zur Methode von Greenberger bereit
- Die Klasse verfügt über folgende Attribute:
  - seed - aktueller Wert des Generators
  - start - Startwert des Zufallszahlengenerators (Default: 100.000)
  - count – Anzahl der bisher gelieferten Zufallszahlen
  - antithetic – Option für Generierung antithetischer Zufallszahlen der Form (1-X)
  - title - Name für die Ausgabe in Reports
  - histo[0..15] - intern für statistische Zwecke

- Die Deklaration und Initialisierung eines SLX-Zufallszahlengenerator erfolgt mit
  rn_stream *name* [**seed**= *Startwert*] [**antithetic**] [**title** = *Generatorname*]
  wobei die kursiven Texte durch modellspezifische Werte zu ersetzen sind.
- Bsp.: **rn_stream** Ankunft **seed**= *1000* **title** = Ankunftsintervall
- Alle rn_stream-Objekte werden in das System-Set rn_stream_set eingeordnet und stehen dort zentral zur Verfügung.

Erzeugung beliebiger Zufallzahlenverteilungen mit SLX

- Auf der Basis der gleichverteilten Zufallszahlengeneratoren kann eine große Anzahl mathematischer Verteilungsfunktionen generiert werden.
- Die Generierung einzelner Zufallszahlenwerte erfolgt mit einem Aufruf der Form:
  x = Name_Zufallszahlentyp (rn_stream Generatorname, Parameterliste **...)**
- Die Ausprägung der Parameterliste entspricht in der Regel in Typ und Anzahl der mathematischen Vorgabe.

Beispiele mit Syntax der Parameterliste :

- float rv_uniform(rn_stream r, double mean , double spread) // Gleichverteilung
- int rv_discrete_uniform(rn_stream r, int leftpoint , int rightpoint) //diskrete GV
- float rv_normal(rn_stream r, double xmean , double xstd) // Normalverteilung
- float rv_expo(rn_stream r, double mean) // Exponentialverteilung
- float rv_triangular(rn_stream r, double x1 , double x2 ,double x3) // Dreieckvert.
- float rv_gamma(rn_stream r, double alpha, double beta) // Gammavert.
- float frn(rn_stream r) - liefert direkt den generierten Wert von r aus [0,1]
- Beispiele zur Anwendung und weitere Verteilungen sind in der Datei rvtest.slx aufgeführt.

Beispiel zur Anwendung von Zufallszahlen

Verschiedene Zufallsprozesse in einem Modell sollten mit unabhängigen (also mit verschiedenen Startwerten initialisierten) Zufallszahlengeneratoren erzeugt werden.

```
// Variablendeklaration
rn_stream gen_ankunftsrate; double  ankunftsabstand = 100;
rn_stream gen_bedienrate;   double  bedienzeitmean = 50, bedienabw = 30;
...
// Modellierung der Ankünfte
   ankunft= rv_expo(gen_ankunftsrate,  ankunftsabstand ); // Exponentialvertl. ;
   advance (ankunft); // Warten bis zum Generieren
   //  Erzeugung eines neuen Objektes mit new .. activate / fork

        ... // in der Action-Methode des Objektes
        advance ( rv_normal(gen_bedienrate, bedienzeitmean ,bedienabw );
        ... //
```

Modellierung empirischer Verteilungen

- **SLX unterstützt die Realisierung empirischer Verteilungsfunktionen**
- **analoge Vorgehensweise für diskrete und kontinuierliche Verteilungen:**

Generierung über direkt vorgegebene Wertepaare

- **discrete_empirical** *Name_der_Vert* **(** *x1 y1 , x2 y2 , ... Xn yn* **);**
- **continuous_empirical** *Name_der_Vert* **(** *x1 y1 , x2 y2 , ... Xn yn* **);**
- **Die X-Werte müssen aufsteigend von 0 bis 1 definiert werden !**

Alternativ kann auch eine Datei mit den Werten vorgegeben werden:

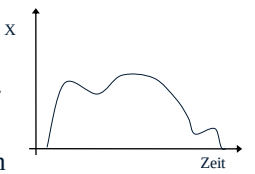
- **continuous_empirical** *Name_der_Vert* **file =** *Dateiname* **;**
- **Die Anweisungen generieren eine Funktion mit dem vorgegebenen Namen, welche als Parameter einen rn_stream erwartet:**
- **Procedure** *Name_der_Vert (rnstream r);*

Bsp.: **discrete_empirical** *Bedienzeit* **(** *0.1 6 , 0.3 10 , 0.6 15, 0.8 20 , 1* **);**

Erlaubt Aufruf mit : **z = BedienZeit(BedienzeitStream);**

Weitere Funktionen zru Modellierung zufälliger Prozesse

Kurvenbasierte Approximation



- zur noch besseren Modellierung von empirischen Verteilungsfunktionen verfügt SLX über die **Bezierfunktion**.
- Bezierfunktionen dienen auch im CAD-Bereich zur numerischen Definitionen von Kurven durch eine Anzahl von Stützpunkten.
- SLX-Funktion: rv_bezier(rn_stream r, bezierdat b);
  wobei b die Menge der Stützpunkte vorgibt (analog zu _empirical)

Hilfsfunktion zur Adaption von Verteilungsfunktionen an vorgegebene Grenzen:

- Häufig sind die praktischen Werte einer Zufallsgröße nur innerhalb von bestimmten Grenzen zulässig (z.B. sollten Normalverteilte Bedienzeiten > 0 sein)
- **random_input** t=rv_normal(rstream1, 10, 5) **accept (0, 40)**
  liefert nur die Werte im Intervall [0,40] als Ergebnisse der Normalverteilung
- weitere Einschränkungen der Gültigkeitsbereiche können natürlich auch mit den Standardbefehlen if-then und while() realisiert werden

Ergebnisdokumentation und –analyse

Bei der Verwendung zufälliger Modellgrößen sind zur korrekten Simulationsauswertung auch entsprechende Statistiken zu erstellen.

Typische Aufgaben und Vorgehensweisen

- Sammlung von Ergebnisdaten zu einer Größe
- Verdichtung der Daten im Zeitmaßstab bis hin zu Werten wie Mittelwert und Streuung
- Speicherung der Ergebnisdaten für mehrere Läufe
- Berechnung der statistischen Daten für alle Läufe

- In SLX und auch allen anderen modernen, diskreten Simulationssystemen werden diese Aufgabe meist durch Standardfunktionen unterstützt.

Definition von Einzelstatistiken

- **SLX stellt eine Klasse random_variable zur Sammlung von Statistiken für eine einzelne Größe bereit**
- **Definition einer random_variable :**

  random_variable [(time | weight)] rv_name

  [histogram start = startwert **width=**Breite **count =** AnzahlIntervalle]
- **[histogram ...] definiert ein optionales Histogramm.**
- Der optionale Parameter [(time | weight)] dient zur "Wichtung der Meßwerte.
  - Werte wie Bedienzeiten benötigen keine Wichtung.
  - Werden Wertveränderungen über die Zeit gemessen, z.B. die Anzahl der Objekte in einer Wartschlange, so müssen die Werte jeweils mit der Zeit gewichtet werden.
  - Der Anwender kann auch eigene Wichtungen, z.B. die Priorität eines Kunden oder Auftrags als Wichtung verwenden
- Jedes Objekt vom Typ random_variable wird in das Set random_variable_set aufgenommen. Die Methoden report und clear können darüber aufgerufen werden.

Beispiel zur Erzeugung eines Warteschlangenhistogramms :

```
random_variable  (time) Warteschlangenstatistik
histogram  start = 0  width = 100 count = 20 ;
```

Aufzeichnung von Einzelstatistiken

- **Bei vorhandener random_variable** rv_name **erfolgt die Aufzeichnung mit**

  tabulate rv_name = Wert [weight = Wichtung]

Bsp.: **tabulate** Warteschlangenstatistik = WS1_Warteschlangen_Laenge;

Der Abruf der statistischen Daten erfolgt mit :

- **x= sample_count(rv_name); // Anzahl der Meßwerte**
- **x= sample_min(rv_name); // Minimum der Meßwerte**
- **x= sample_max(rv_name); // Maximum der Meßwerte**
- **x= sample_sum(rv_name); // Summe der Meßwerte**
- **x= sample_mean(rv_name); // Mittelwert der Meßwerte**
- **x= sample_variance(rv_name); // Varianz der Meßwerte**
- **x= sample_stdev(rv_name); // Standardabweichung der Meßwerte**

Auswertung von Histogrammen

- **Bei vorhandenem Histogramm können außer der tabellarischen Ausgabe auch direkte Ergebniswerte ermittelt werden.**
- Innerhalb einer random_variable kann dazu mit dem pointer histo auf die Attribute des Histogramms zugegriffen werden:

  - das Feld double frequency [0... Class_count-1] liefert die Häufigkeiten

  - die Attribute lower_bound, upper_bound, class_width, class_count, count liefern Hilfsgrößen zur Durchführung von Berechnungen

Beispiel :

- **Bestimmung der Wahrscheinlichkeiten für eine bestimmte Wartezeit**
- **Berechnung der gesuchten Histogrammspalte durch**

  i = (Wartezeit-lowerbound) / class_width
- **Aufsummierung der Einzelhäufigkeiten bis zu i. Spalte**
- **Summe / count ergibt die gesuchte Wahrscheinlichkeit**
- **Achtung: Durch die diskretem Intervalle ist die Genauigkeit abhängig von der Anzahl der Histogrammklassen.**

Weitere Funktionen zur Simulationsauswertung in SLX

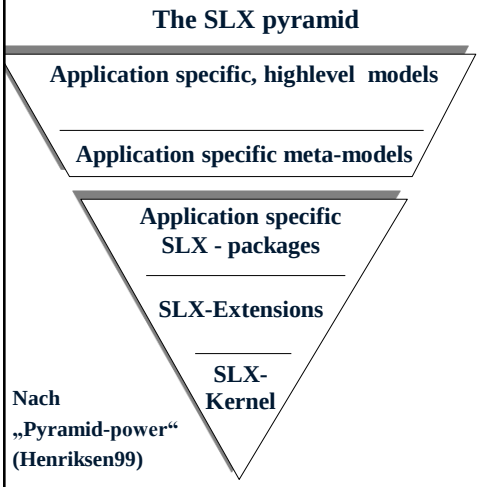
Zur Auswertung komplexer Experimente sind weitere Funktionen verfügbar:

- Schätzung von Konfidenzintervallen (build_mean_ci , report_mean_ci)
- Ein vorhandenes Konfidenzintervall kann auch zur Bestimmung der Ergebnisqualität verwendet werden:
  - Durchführung einer ersten Anzahl von Simulationsläufen
  - Bestimmung der Konfidenzintervalle für alle relevanten Größen
  - Falls das Minimum der Quotienten aller Konfidenzintervalle / Mittelwerte eine bestimmte Schranke (z.B. 0.05= 5% Irrtumswahrscheinlichkeit) unterschreitet, kann die Simulation beendet werden, ansonsten sind weitere Experimente durchzuführen!
- Vergleich von Systemvarianten eines Modells
  - Der Vergleich soll nur die Unterschiede bzgl. des geänderten Modells aufdecken !
  - Dazu sind die Zufallgeneratoren mit gleichen Anfangswerten zu starten (seed)
  - Ergebnisse der Simulation beider Modelle werden in Dateien geschrieben
  - Die SLX-Funktion build_common_main_ci(...) können dann zum Vergleich der Ergebnisdaten eingesetzt werden und liefern den Mittelwert der Differenzen.

Das Schichtenmodell des SLX-Compilers

Das Prinzip der Compilererweiterung

- SLX ist im Gegensatz zu anderen Simulationssprachen und auch universellen Programmiersprachen in **mehreren Schichten** aufgebaut



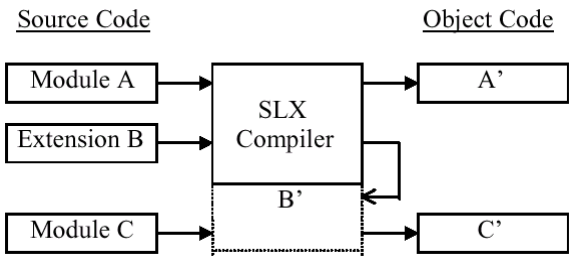
- Auf der höchsten Ebene kann eine völlig andere Syntax und Semantik verwendet werden.
- Das Metamodell definiert eine völlig neue Klasse von Modellen.
- Diese Ebene entspricht der bisherigen Modellierung.
- Die SLX-Extensions bauen aus den Kernel-befehlen komplexere Befehle zusammen (z.B. wird der print-Befehl durch Parsen der Parameter in eine Reihe von Kernelbefehlen überführt.
- Der SLX-Kernel stellt eine kleine Anzahl von Grundfunktionen bereit. (sichtbar über ...expansion)

Der erweiterbare SLX-Compiler

- Bei traditionellen grundlegenden Compilern ist der Befehlssatz in der Regel fixiert.
- Der SLX-Compiler kann durch spezielle Extensions um neue Grundregeln und Syntaxformen erweitert werden.
- Bei der nachfolgenden Übersetzung weiterer Module können die neuen Regeln dann bereits eingesetzt werden.

Vorteile:

- Ausgezeichnete Anpassung an fachspezifische Modellierungsanforderungen
- Effiziente Notation des Quelltexte



Ausführung von GPSS-Modellen mit SLX

- Ein Beispiel für eine Compilererweiterung stellt die GPSS-Erweiterung h5.slx für SLX dar. Nach dem import von h5.slx sind GPSS-Anweisungen verwendbar.
- Von Simulationsanwendern wurden auch schon vorhandene GPSS-Programme damit auf das modernere SLX-System portiert

```
import    <h5>
module barb13 {     facility     joe;
                    queue        joeq;
                    rn_stream Arrivals  seed=100000;
                    rn_stream Service   seed=200000;

 ...
                    {
                    enqueue  joeq;
                    seize    joe;
                    depart   joeq;
                    advance  rv_uniform(Service, 12.0, 18.0);
                    release  joe;
                    terminate;
                    }
...
```

Aufbau der SLX-Extensions

- Die SLX-Extensions bestehen aus einem Teil mit Standard-SLX-Code und einem Teil zum Parsen und Umsetzen des GPSS-Quelltextes auf die SLX-Funktionen.

```
// Definition des Speicherbetretens mit ENTER als SLX-Prozedur
procedure ENTER(inout storage s, float amount_needed)
{ forever  { if (s.remaining < amount_needed)
                    { wait list=s.change_in_too_full; // sleep
                      continue; // and try again
                    }
             if (s.availability != AVAIL)
             { wait list=s.change_in_availability; // sleep
               continue;  // and try again
             }

... //  Definition des neuen Befehls enter mit Parameterparsing
statement enter #storage [units=#units];
   definition { if (#units == "")
                    expand(#storage)   "ENTER(#, 1.0);\n";
                    else expand(#storage, #units)  "ENTER(#, #);\n";
              }
```


Anbindung von SLX an andere IT-Systeme

Die An- oder Einbindung von SLX in andere Systeme ist wie folgt möglich:

- universeller Datenaustausch über **formatierte Textdateien** (sehr schnell)
- Datenbankmodul für **ODBC-Datenbank** von der UNI-Magdeburg
- Universelles **DLL-Interface** (C-DLL´s können eingebunden werden und erlauben eine Erweiterung der Kernelfunktionen)

```
procedure Sleep(int milliseconds) dll = "kernel32";
```

- **HLA-Interface** vom Fraunhofer-Institut (HLA ist eine simulations-spezifische Schnittstelle analog zu CORBA, definiert vom amerikanischen Verteidigungsministerium)
- Kopplung mit dem **Optimierungssystem** ISSOP
- Die oben aufgeführten Kopplungsoptionen wurden zum Aufbau eines Kundenspezifischen Simulators genutzt. SLX ist dabei für den Endkunden nicht mehr direkt sichtbar. (Vorführung am Ende des Semesters)

Animation von SLX-Simulationsläufen

Visualisierung von SLX-Simulationsläufen

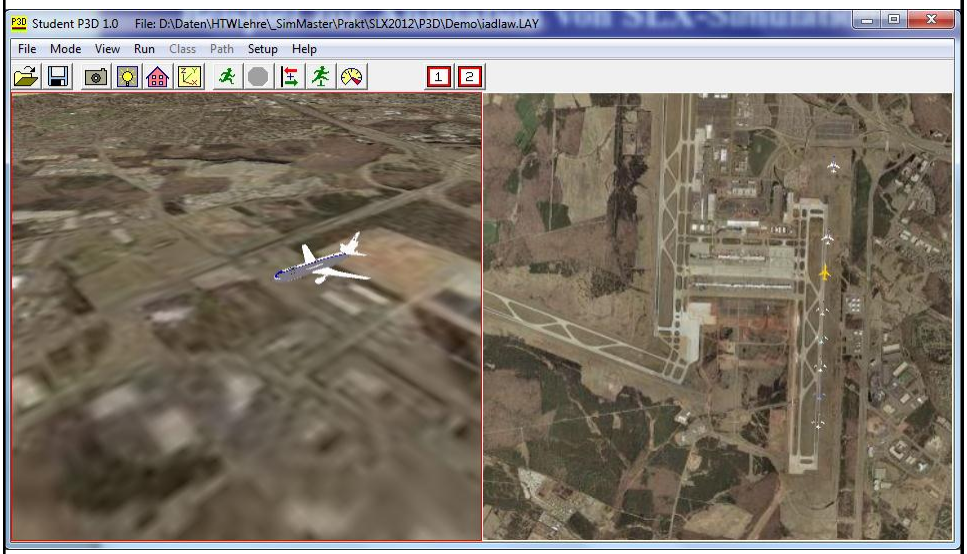
- SLX als Simulationssprache enthält keine direkt integrierte Animation.
- jedoch kann mit beliebigen Visualisierungstools oder mit dem von der gleichen Firma bereitgestellten Animationstool PROOF eine Animation angebunden werden

Allgemeines Konzept der Animationskopplung

- Erstellung von Layoutplänen (ggf. auch aus CAD-Daten) und auch mit Bitmaps als Hintergrundbild
- optional Vorab-Definition von Animationsmakros
  - zur Objektrepräsentation (z.B. Autos immer aus Kreisen/Vierecken bauen)
  - Zur Gruppenbildung auf Verfahrwegen (-> PATH-Definition)
  - zur Vorabdefinition von Kameras, Lichtquellen, Views etc.
- Tracefiles zur eigentlichen Bewegungsbeschreibung aus SLX (oder auch aus anderen Simulationstools heraus generiert) mit
  - Bewegungs- (Delta-X,Y,Z) und Eigenschaftenupdates (Farbe, Form…)
  - Stops und Viewänderungen

  (siehe auch arc.lay zur Erklärung der PROOF-Definitionen)

Beispiel zur Animation von SLX-Simulationsläufen



Allgemeine Bewertung von SLX

Vorteile

- SLX ist gegenwärtig das weltweit schnellste Simulationssystem (Performance teilweise bis zu 100 mal besser als bei anderen kommerziellen Produkten)
- Diese Eigenschaft ist insbesondere bei der Optimierung mit einer großen Anzahl von Einzelläufen relevant.
- Keine Grenzen bei der Flexibilität durch beliebige Syntaxkonstruktionen und Ergänung um externe DLL´s

Gewisse Nachteile liegen in der Natur der Quelltextprogrammierung

- Bei größeren Modellen schnell unübersichtlich – Ausweg durch automatisierte Codegenerierung analog zu Bausteinsystemen (Eigenentwicklung).
- Animation und grafische Ergebnisdarstellungen müssen explizit programmiert werden

Fazit:

- bei Abstrichen am Komfort sehr gutes Simulationssystem ohne Grenzen bei der Modellierung